



UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

**Aus dem Institut für IT-Sicherheit
der Universität zu Lübeck
Direktor: Prof. Dr.-Ing. Thomas Eisenbarth**

Software Defenses against CPU Side-Channels

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck

Aus der Sektion Informatik / Technik

vorgelegt von
Jan Wichelmann
aus Vechta

Lübeck, 2024

1. Berichterstatter: Prof. Dr.-Ing. Thomas Eisenbarth

2. Berichterstatter: Prof. Dr. Daniel Gruss

Tag der mündlichen Prüfung: 02. August 2024

Zum Druck genehmigt. Lübeck, den 14. August 2024

Abstract

Side-channel attacks on CPUs allow crossing architectural security barriers which were deemed impenetrable by software. Even trusted execution environments (TEEs), which seek strong hardware-based isolation of applications running on an untrusted system, have been subject to attacks which managed to extract the code and data inside them. Attack techniques range from tracking memory access patterns via cache attacks to single-stepping of TEEs, deducing secrets from ciphertext changes in memory, and even invoking transient behavior that yields state which should not be accessible by software at all.

We begin this thesis with an overview over the various side-channel attack techniques and highlight their capabilities. We then take a defender's perspective, aiming to deploy side-channel resistant software. The most common class of side-channel vulnerabilities in software concerns memory access pattern leakages, where the attacker can infer secrets by observing secret-dependent control flow and memory accesses. Absent effective hardware countermeasures against side-channels, the standard solution for such leakages is constant-time code, which exhibits access patterns that are independent from secrets. However, writing such code is complex and error-prone, prompting research into automatic leakage detection. To date, there are many different proposals for finding side-channel leakages in software, with varying focus on soundness, capabilities and scalability. We survey the existing research and propose a new framework *Microwalk*, which aims to unite the features of the different techniques in a single practically usable tool.

Progressing forward, we discuss side-channel defenses which do not need as much manual intervention by the developer and comprise both hardware and software approaches. We propose *Cipherfix*, which automatically rewrites binaries to be resistant to ciphertext side-channel attacks, an attack method allowing to break constant-time implementations on TEEs that use deterministic memory encryption schemes. We conclude with *Obelix*, a dynamic obfuscation engine that addresses many classes of side-channels in a single drop-in solution, including memory access pattern leakages, single-stepping and ciphertext side-channels.

Kurzfassung

Seitenkanalangriffe auf CPUs überwinden architekturelle Sicherheitsbarrieren, die für Software als undurchdringlich galten. Selbst Trusted Execution Environments (TEEs), die eine starke hardwarebasierte Isolation von Anwendungen auf einem nicht vertrauenswürdigen System anstreben, waren Ziel von Angriffen, denen es gelang, Code and Daten aus diesen Umgebungen zu extrahieren. Angriffstechniken reichen hierbei vom Mitschneiden von Speicherzugriffsmustern mittels Cache-Angriffen bis hin zum Single-Stepping von TEEs, Ableiten von Geheimnissen aus Chiffretextänderungen im Speicher und sogar Verursachen von transientem Verhalten, das zu für Software unerreichbaren Systemzuständen führt.

Wir beginnen diese Arbeit mit einem Überblick über die verschiedenen Seitenkanal-Angriffstechniken und ihre Fähigkeiten. Anschließend nehmen wir die Perspektive eines Verteidigers ein, der seitenkanalresistente Software entwickeln und einsetzen möchte. Die häufigste Klasse von Seitenkanal-Schwachstellen sind Lecks von Speicherzugriffsmustern, aus denen der Angreifer durch Beobachtung von geheimnisabhängigen Kontrollflüssen und Speicherzugriffen auf Geheimnisse schließen kann. Mangels effektiver Hardware-Gegenmaßnahmen gegen solche Schwachstellen besteht die Standardlösung in constant-time Code, dessen Zugriffsmuster unabhängig von geheimen Eingaben sind. Das Schreiben solchen Codes ist jedoch komplex und fehleranfällig, was Forschung zur automatischen Schwachstellenerkennung veranlasst hat. Inzwischen gibt es viele verschiedene Vorschläge zur Erkennung von Seitenkanal-Schwachstellen in Software, mit variierendem Schwerpunkt auf Korrektheit, Fähigkeiten oder Skalierbarkeit. Wir geben einen Überblick über die bestehende Forschung und schlagen ein neues Framework mit dem Namen Microwalk vor, welches zum Ziel hat, die Merkmale der verschiedenen Techniken in einem praktisch nutzbaren Werkzeug zu vereinen.

Im Anschluss diskutieren wir Seitenkanal-Gegenmaßnahmen, die weniger manuelle Eingriffe seitens des Entwicklers benötigen und sowohl software- als auch hardwarebasierte Ansätze umfassen. Wir schlagen Cipherfix vor, das Binärdateien automatisch umschreibt, um diese gegen Chiffretext-Seitenkanalangriffe resistent zu machen. Chiffretext-Seitenkanäle ermöglichen es, constant-time Implementierungen zu brechen, wenn diese auf TEEs mit deterministischer Speicherverschlüsselung laufen. Zum Abschluss stellen wir Obelix vor, das mittels dynamischer Obfuscation viele Klassen von

Seitenkanalangriffen in einer einzigen direkt einsetzbaren Lösung behebt, insbesondere Speicherzugriffsmuster-Lecks, Single-Stepping und Chiffretext-Seitenkanäle.

Acknowledgements

First of all, I would like to thank my advisor Thomas for offering me this opportunity and mentoring my research. The combination of seemingly endless patience, approachability, humor and will to let me set my own priorities contributed to a great work environment.

I would also like to thank my many colleagues, students and collaborators, some of whom I have even learned to call friends. Thank you for the fun time, for enduring my odd puns and machine learning jokes, and of course for many insightful discussions. Specifically, thank you Luca, for our numerous collaborations when designing SEV attacks and for still wanting to work with me even after I repeatedly rewrote half of the paper; Anja Rabich for doing the Obelix measurements an x-th time and re-rendering the plots an y-th time; Florian for the help with repetitive paper evaluations; Jonas for asking the right innocent question at the right random time, reviving our stuck debugging efforts more than once; Ines for her advice and help when organizing teaching and navigating administrative depths; Anja Köhl for reliably helping in the Cybersecurity course over many years; and Anna, Paula, Johannes, Pajam and Tim for some very fun lunch breaks and evenings with interesting discussions over unexpected topics.

I can say with certainty that this work would not have turned out the way it did if it weren't for Anna, who has been a constant throughout my entire PhD time. I sometimes (but only sometimes) miss our months-long debugging sessions after we had the brilliant idea of implementing our own binary rewriting engine ("almost done, just this one last bug"), an endeavor which only marked the beginning of some fantastic teamwork. Thank you Anna for many productive discussions, for our seamless joint paper writing, for proof-reading this thesis, and for your support and optimism whenever things looked not as bright for a moment.

Thank you Noemi for introducing me to some of my favorite series and for being available for so many entertaining activities during our study and subsequent PhD phase, providing a welcome break whenever things got stressful.

I could also always rely on my family throughout highs and lows during my study and PhD, providing advice and help whenever needed, which I am very grateful for.

Finally, I have to express my deep gratitude to my better half, Sonja, for still being with me after ten years of “will be there in at most 5 minutes, just quickly finishing this piece of code!”, for listening to all those rants about whatever went wrong that day, and for nudging me to finally complete this thesis. Thank you for your unending patience and support.

Contents

I	Software Defenses against CPU Side-Channels	1
1	Introduction	3
1.1	Main Contributions	5
1.1.1	Individual Publications	6
1.2	Other Contributions	9
1.3	Outline	12
2	Background	13
2.1	System Architecture	13
2.1.1	Operating System and Address Spaces	14
2.1.2	Virtualization	15
2.1.3	Caches	16
2.1.4	Execution Engine	18
2.2	Trusted Execution Environments	22
2.2.1	Process-Level: Intel SGX	22
2.2.2	Virtual Machine Protection: AMD SEV	22
2.3	Software Instrumentation	24
2.3.1	Static Instrumentation	24
2.3.2	Dynamic Instrumentation	25
2.4	Code Analysis	25
2.4.1	Symbolic Execution	26
2.4.2	Taint Analysis	26
2.4.3	Fuzzing	27
3	State of the Art	29
3.1	Side-Channel Leakage in the CPU	29
3.1.1	Cache Attacks	30
3.1.2	Page Fault Controlled Channel	34
3.1.3	Single-Stepping	35
3.1.4	Transient Execution	36

3.1.5	Value-Based Leakages	39
3.1.6	Other Attacks and Leakages	40
3.2	Software Leakage Analysis	42
3.2.1	Static Approaches	43
3.2.2	Dynamic Approaches	47
3.2.3	Comparison	50
3.2.4	Discussion	54
3.2.5	Other Vulnerability Types	54
3.3	Side-Channel Countermeasures	56
3.3.1	Cache Attacks	56
3.3.2	Single-Stepping	62
3.3.3	Transient Execution	63
3.3.4	Value-Based Leakages	66
4	Conclusion	69
	References	71
 II Publications		99
5	MicroWalk	101
6	Microwalk-CI	137
7	MAMBO-V	177
8	Cipherfix	205
9	Obelix	247
 III Appendix		291
	About the Author	293

Part I

Software Defenses against CPU Side-Channels

Introduction

The broad availability of high-bandwidth internet access leads to a centralization of IT services: Instead of maintaining their own data centers, many companies and agencies rent resources on third-party cloud systems. Moving workloads into the cloud avoids the costly acquisition and maintenance of own hardware, and allows quick scaling of applications under an increased load.

However, by moving potentially sensitive computations into the cloud, the customer effectively entrusts the cloud provider with maintaining confidentiality of their applications and isolating them from the applications of other customers. A standard measure to keep customers running on the same hardware from interfering with each other is *virtualization*: Every customer's code runs in a separate *virtual machine* (VM), which is assigned a share of the system's resources and behaves like a native bare-metal system. VMs are managed by the hypervisor (i.e., cloud provider) and cannot access each other's code and data.

If the owner of sensitive code and data does *not* trust the cloud provider, virtualization becomes insufficient, as the hypervisor has full access to all code and data on the given system. Instead, the customer may choose to use a *trusted execution environment* (TEE), which adds an additional layer of hardware-enforced protection that even resists privileged attacks from the hypervisor itself. While VMs can be freely modified by the hypervisor, TEEs encrypt the application's memory and do not allow any other user to read or manipulate it.

Unfortunately, it was shown countless times that the isolation guarantees from VMs and TEEs do not hold in practice. While direct, *architectural* attacks were quickly patched in subsequent processor generations [160, 161, 250, 252], there is another attack class that is much harder to mitigate: *Side-channel attacks*. A side-channel attack does not directly read sensitive information, but *observes* the execution behavior of a program to infer secrets *indirectly*. This reaches from very simple attacks like measuring the total execution time of a program that compares passwords and aborts upon the first difference (thus leaking the amount of correct characters) to complex attacks that exploit a certain behavior of the *hardware* that the program runs on. The latter class, so-called *microarchitectural* side-channel attacks, exploit optimizations that were introduced by

the processor vendors and are transparent to the application developer, like caching [30, 171] or out-of-order execution [144]. TEEs with their powerful attacker model come with another set of attacks, which allow the hypervisor to single-step code [45] or infer secrets from ciphertext changes in memory [136, 138].

The indirect nature of these attacks makes it difficult to implement effective hardware countermeasures without sacrificing performance, and many such proposals have eventually been shown to be vulnerable to an improved version of the same attack. However, there are mitigations that address at least *some* of the most relevant attack classes and that can be implemented by software developers. The arguably most common type of side-channel leakage are *secret-dependent operations*, also called *memory access pattern leakages*. Common examples are executing an `if` statement depending on the value of a secret key bit, or using a key byte as index for an array access. Both can be observed by an attacker monitoring memory accesses, e.g., via a cache attack. While the developer of a sensitive application (e.g., cryptographic software) should not be responsible for addressing security issues stemming from unrelated hardware optimizations, they can adapt their software in a way that it becomes resistant against such side-channels.

To remove memory access pattern leakages (and thus prevent all attacks exploiting them), the developer systematically replaces secret-dependent operations in their program by *linearized* ones, such that the program always accesses the same code and data, independent from its secret input. Such programs are called *constant-time*, as they exhibit the same observable execution behavior for any input. Writing constant-time code is challenging and prone to missing subtle leakages which may be exploitable nonetheless [16, 208, 239]. As standard bug-finding tools and sanitizers are not capable of finding side-channel leakages, specialized tools and approaches were developed that aim to help the developer identify missed leakages in their code and patch them appropriately. Methods vary from static analysis and proof generation to dynamic analysis and fuzzing [78].

Other research goes one step further and designs tools that *automate* code hardening through means like static instrumentation. For example, it is possible to automatically linearize code, allowing developers to write leaking, but better maintainable code, that is later transformed by the compiler. Such automated hardening tools can also be created for other side-channel attack classes. Several software-only and software/hardware co-design approaches have been proposed for preventing speculative execution attacks [49, 62], TEE single-stepping [55, 59, 168, 246] and ciphertext side-channels [244, 246].

1.1 Main Contributions

In this thesis, we design new side-channel leakage analysis techniques and build automated hardening tools for software running in insecure TEEs.

We advance the state of the art of *software defenses against CPU side-channels* by:

- **Designing a practical dynamic side-channel leakage analysis framework.** With Microwalk, we build a new software leakage analysis framework targeted at practical use in day-to-day development. We propose new efficient leakage analysis algorithms based on comparing execution traces generated through dynamic binary instrumentation (DBI). The framework satisfies three key objectives of practical leakage analysis: *Localization* and *quantification* of side-channel leakages, and *efficiency* of the analysis method. We design a modular and extensible framework and devise templates for easy inclusion in Continuous Integration (CI) pipelines. We successfully use Microwalk to uncover many previously unknown side-channel vulnerabilities, which includes leakages in closed-source libraries like Microsoft CNG, that is shipped with every Windows system but received little scrutiny beforehand.
- **Bringing dynamic leakage analysis to scripting languages and non-x86 platforms.** We show that side-channel leakage analysis algorithms designed for evaluating binaries can also be applied to programming languages directly. For this, we employ source-based static instrumentation to insert trace generation code at relevant source locations. With a custom preprocessor we convert those traces into the generic trace format of Microwalk, allowing us to reuse its existing analysis modules. Using the extended framework, we conduct the first broad leakage analysis of popular JavaScript cryptographic libraries from NPM, identifying many vulnerabilities.

Finally, we demonstrate the first side-channel leakage analysis toolchain for the RISC-V platform. We port the ARM-based DBI framework MAMBO to RISC-V, allowing us to generate native execution traces on RISC-V, which we can subsequently analyze with Microwalk. We hope that the early availability of leakage analysis tooling on RISC-V supports the development of native cryptographic code that is resistant to side-channels from day one.

- **Designing an automated software hardening scheme against ciphertext side-channels.** As ciphertext side-channels are specific to TEEs, existing software does not deploy any suitable countermeasures and, due to their impact on code maintainability, is also not likely to deploy them in the future. Starting from a taint tracking tool that collects memory loads and stores that access secret data and are potentially vulnerable against ciphertext side-channel attacks, we build an

automatic software hardening framework based on rewriting existing binaries. We complement each memory write with additional instructions which generate a new random mask and add this mask to the written value, leading to fresh ciphertexts. We evaluate different methods for sampling sufficiently good randomness and for keeping track of the secrecy of a given memory location. We apply our toolchain to several popular cryptographic implementations and show that it can protect them against ciphertext side-channel attacks with a reasonable runtime overhead.

- **Exploring a catch-all drop-in software countermeasure against a wide spectrum of TEE-related attacks.** We survey existing software countermeasures against the high variety of side-channel attacks against TEEs, and find that these countermeasures only cover few attack vectors and often require expert knowledge to apply. With the objective of “restoring trust in TEEs” we look for an automated solution that can protect against *all* relevant classes of attacks, and explore the suitability of a rather exotic approach: Full program obfuscation. We show how an existing obfuscation tool can be adapted to protect against timing and controlled channel attacks, single-stepping and ciphertext side-channels. For this, we analyze the practical capabilities of single-stepping attackers on Intel SGX and AMD SEV, and design an algorithm that generates corresponding indistinguishable code blocks. In our evaluation we find that, while the resulting tool comes with a high performance overhead, it can reliably protect entire cryptographic implementations against attackers trying to fingerprint code or extract secret data.

1.1.1 Individual Publications

In the following, we summarize the individual publications included in this thesis, which make up the main contributions. The respective full text is given in Part II.

Microwalk: A Framework for Finding Side Channels in Binaries. To avert side-channel vulnerabilities caused by the microarchitecture a program is executed on, programmers need to employ software-based mitigation techniques. One such technique is constant-time code, i.e., code that exhibits the same control flow and memory access patterns independent from a secret input. There are various ways to verify whether a given program is constant-time, but to this point research mostly focused on static analysis (which tends to be slow and hard to deploy in practice) and on imprecise or highly manual methods. We introduced a new fast dynamic leakage analysis method based on DBI and mutual information analysis: First, we generated a number of random secret inputs and collected corresponding execution traces. We then compared those traces with a focus on the observed memory access patterns. Any difference between two traces is directly linked to a secret-dependent computation, which is potentially

exploitable by a side-channel attacker. As the approach tends to find many potential vulnerabilities in a program, we assigned them a severity score, which is the mutual information between the secret input and the resulting trace.

We implemented the workflow in a new modular and extensible framework, *Microwalk*. With *Microwalk*, we showed that our leakage analysis technique can efficiently find side-channel leakages in cryptographic libraries, and we uncovered previously unknown leakages in the closed-source Microsoft CNG and Intel IPP cryptographic libraries.

The paper was published at *ACSAC 2018* in collaboration with Daniel Moghimi, Thomas Eisenbarth, and Berk Sunar [243]. The full text is in Chapter 5.

Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. While side-channel leakage analysis of software is a well-studied problem, the results never found wide practical adoption, mostly since the research artifacts were hardly usable, had bad performance, or required in-depth knowledge of the underlying methods. For these reasons, as was shown by a study in 2022 [118], developers of cryptographic libraries still heavily relied on occasional vulnerability disclosures by researchers and, in some cases, on manual code analysis through tools like *ctgrind* [133].

To address these challenges, we developed a new dynamic leakage analysis algorithm, which inserts the execution traces into a trie-like data structure, allowing precise localization and quantification of potential leakages in linear time. As mutual information alone gives misleading results in some cases, we evaluated alternative leakage measurement methods. The result is a single scalar leakage score which is shown to the user, so they can quickly prioritize the reported leakages. In another contribution, we abstracted *Microwalk*'s trace generation stage to support arbitrary programming languages, and showed how one can generate suitable execution traces even for script languages like JavaScript.

Finally, we showed how one can design a general-purpose analysis template for the extended *Microwalk* framework, which abstracts away most configuration complexity and allows a developer to easily integrate side-channel leakage analysis into their existing Continuous Integration workflow. The Docker images, configuration templates and example repositories were made available at GitHub [154]. We used the new toolchain to do the first thorough leakage analysis of popular cryptographic libraries on NPM, and uncovered many vulnerabilities.

The paper was published at *ACM CCS 2022* in collaboration with Florian Sieck, Anna Pätschke and Thomas Eisenbarth [247]. The full text is in Chapter 6.

MAMBO-V: Dynamic Side-Channel Leakage Analysis on RISC-V. Most existing leakage analysis tools target source code or x86 binaries. While analyzing source code can find source-level leakages, it misses vulnerabilities introduced by compiler optimizations. Binary approaches catch such issues, but they are inherently bound to a certain architecture. It is desirable to have leakage analysis support on other platforms as well, especially on the upcoming RISC-V architecture which is growing quickly, to avoid repeating the many security issues from x86 and ARM. As no DBI tool for RISC-V was available, we created a port of the ARM-based DBI tool MAMBO, naming it MAMBO-V. We used MAMBO-V to generate Microwalk-compatible execution traces for RISC-V binaries, allowing us to conduct the first side-channel leakage analysis of native RISC-V cryptographic code. We identified several vulnerabilities, mostly caused by insecure fallback implementations in popular libraries which do not yet offer assembly implementations optimized for RISC-V.

The paper was published at *DIMVA 2023* in collaboration with Christopher Peredy, Florian Sieck, Anna Pättschke and Thomas Eisenbarth [245]. The full text is in Chapter 7.

Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. Ciphertext side-channels are a rather young attack class, targeting deterministic memory encryption in TEEs (in this case, AMD SEV). They exploit the fact that a fixed plaintext always results in the same ciphertexts at a certain memory address, allowing the attacker to detect repeating plaintexts by keeping track of observed ciphertexts. They can then use this information to break constant-time cryptographic implementations. As hardware-based countermeasures are not available, software hardening is necessary. We solved this problem through masking, where we added a random value each time data was written to memory. When reading the data, the mask was subtracted again.

As simply masking all data was not efficient, we prepended an analysis step, where we used dynamic taint tracking to find all locations where secret data was processed, and did precise context-aware tracking of stack variables and heap allocations to find where those secrets were stored. We then implemented a static binary instrumentation tool, which rewrote the relevant memory loads and stores to use masking. Due to our binary approach, we could harden a program across library boundaries without requiring recompilation. We showed that the resulting performance overhead was tolerable, and discussed possible trade-offs between performance and security, like the bookkeeping of secrecy information and the choice of the random number generator for the masks.

The paper was published at *USENIX Security 2023* in collaboration with Anna Pättschke, Luca Wilke und Thomas Eisenbarth [244]. The full text is in Chapter 8.

Obelix: Mitigating Side-Channels through Dynamic Obfuscation. Over time many different side-channel attacks were demonstrated against TEEs. Some were addressed through hardware updates, but many must be averted by software countermeasures. However, the existing countermeasures only focus on single attack classes, are incompatible with each other, or can only protect the secret data within the TEE, but not hide the code itself. We designed a drop-in mitigation that can prevent a wide range of side-channel attacks and manages to protect both code and data. Our approach was based on dynamic obfuscation, where code and data were stored in oblivious memory and fetched block by block. To protect against single-stepping, we built the blocks in a way that they are indistinguishable for a high-resolution attacker, and verified this through precise measurements on Intel SGX and AMD SEV. While the obfuscation-based approach led to a high performance overhead in our proof-of-concept implementation, it is suitable to thoroughly protect small or asynchronously executed programs against all relevant side-channels. Due to the modular design of *Obelix*, it can be extended to also protect against transient execution and fault injection attacks.

The paper was published at *IEEE S&P 2024* in collaboration with Anja Rabich, Anna Pättschke and Thomas Eisenbarth [246]. The full text is in Chapter 9.

1.2 Other Contributions

Apart from the main contributions outlined previously, the author has also contributed to other results which are not included in this thesis. We briefly summarize them in the following.

MemJam: A False Dependency Attack against Constant-Time Crypto Implementations. Most side-channel attacks target structures like the translation look-aside buffer (TLB) or the data and instruction caches. While they were successfully used to break cryptographic implementations, they are limited to a spatial locality of the size of a page or cache line. With MemJam, we demonstrated a new side-channel attack with intra-cache line resolution, that exploits false memory read-after-write dependencies between two hyperthreads. We showed that MemJam can be used to break several cryptographic primitives in the Intel IPP library, which was specifically hardened against cache attacks.

The paper was published in the *Springer IJPP 2019* journal in collaboration with Ahmad Moghimi, Thomas Eisenbarth and Berk Sunar [157].

SEVurity: No Security Without Integrity – Breaking Integrity-Free Memory Encryption with Minimal Assumptions. We analyzed the memory encryption of AMD SEV on Zen 1 and Zen 2 platforms. We found that they use an XEX encryption mode with an address-dependent tweak, which we were able to reverse engineer. We then showed how one can construct an initial encryption oracle solely by moving around existing ciphertext blocks in memory. This way, we were able to build a multi-stage exploit which finally allowed us to inject arbitrary code and data into the protected VM, without relying on I/O operations as did previous attacks. As a response to this and other attacks, AMD introduced the SEV-SNP extension, which fixed our attack on Zen 3 by preventing the hypervisor from writing to encrypted memory.

The paper was published at *IEEE S&P 2020* in collaboration with Luca Wilke, Mathias Morbitzer and Thomas Eisenbarth [250].

undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation.

AMD SEV offers a remote attestation feature which computes a so-called *measurement* of the initial VM state to prove to the VM’s owner that the correct bootloader was placed in encrypted memory. To generate the corresponding hash, the hypervisor sequentially calls a dedicated API in the trusted co-processor, passing the code and data which should be stored in encrypted memory. However, we discovered that the measurement did not include the *address* of the data, so we could arbitrarily reorder it in memory, with the measurement not changing as long as we called the API in the right order. This way, the owner of the VM would believe that the bootloader was loaded correctly, and would supply their secret disk encryption key for booting the system. We were able to reorder the code in such a way that we could inject small amounts of data into the VM, which after several steps resulted in a full encryption oracle and control over its execution.

The paper was published at *WOOT 2021* in collaboration with Luca Wilke, Florian Sieck and Thomas Eisenbarth [252].

Util::Lookup: Exploiting key decoding in cryptographic libraries. We used Microwalk to conduct a leakage analysis of key decoding functions, which were often ignored during side-channel analysis. We found that many libraries implement table-based Base64 decoding, leaking the secret key through the sequence of accessed indexes. However, this is difficult to exploit using conventional cache attacks due to the small amount of leakage (around one bit per Base64 character) and high performance of the implementation. We significantly increased the practically achievable attack precision by combining single-stepping with a cache attack, allowing us to accurately measure each table lookup and then reconstruct the private key. Our attack was aided by a software mitigation recently proposed by Intel to counter Load Value Injection (LVI) attacks, showing that countermeasures against one class of attacks may actually amplify another.

The paper was published at *ACM CCS 2021* in collaboration with Florian Sieck, Sebastian Berndt and Thomas Eisenbarth [208].

A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. While AMD SEV-SNP includes measures to prevent the hypervisor from moving ciphertexts (which we exploited in our SEVurity and undeSErVed trust papers), it still relies on a deterministic memory encryption. In this work, we systematically analyzed the attack surface exposed by deterministic ciphertexts, and identified two main attack primitives: The dictionary attack, where a certain variable only has few possible values, leading to few distinguishable ciphertexts; and the collision attack, where the attacker only observes whether a variable *changed*, allowing them to break common constant-time primitives like conditional swaps. We showed that ciphertext side-channel attacks can break constant-time cryptographic implementations, and proposed and discussed several potential mitigation approaches.

The paper was published at *IEEE S&P 2022* in collaboration with Mengyuan Li, Luca Wilke, Thomas Eisenbarth, Radu Teodorescu and Yinqian Zhang [136].

ASAP: Algorithm Substitution Attacks on Cryptographic Protocols. In an algorithm substitution attack (ASA), an adversary modifies the shipped implementation of a cryptographic primitive to embed secrets in publicly sent data, which allows them, for example, to leak secret keys. We analyzed the applicability of ASAs on the protocol level, and found that commonly desired properties like forward secrecy and post-compromise security directly enable ASAs. We showed that we can easily hide private keys in nonces used by the TLS and WireGuard protocols, while the Signal protocol is much more robust due to its double ratchet structure.

The paper was published at *ACM AsiaCCS 2022* in collaboration with Sebastian Berndt, Claudius Pott, Tim-Henrik Traving and Thomas Eisenbarth [29].

Help, my Signal has bad Device! We analyzed the post-compromise security of Signal when the user's identity key is leaked by an attacker. While existing conversations remain secure under the core Signal protocol, this does not hold for the extensions which allow using Signal on multiple devices. Indeed, we showed that an attacker who possesses the identity key can stealthily register secondary devices under the user's account, allowing them to receive and send all future messages involving the compromised user. We proposed short-term and long-term countermeasures, concluding that the device registration workflow of Signal must be replaced by a more robust protocol to fully avert our attack.

The paper was published at *DIMVA 2021* in collaboration with Sebastian Berndt, Claudius Pott and Thomas Eisenbarth [242].

SEV-Step: A Single-Stepping Framework for AMD-SEV. Single-stepping is a powerful attack primitive against TEEs. As the name suggests, it allows the malicious hypervisor to step the protected code with instruction granularity, by programming a timer that interrupts the enclave after precisely one instruction. *SGX-Step* [45] offers a software framework that aids in building such attacks on Intel SGX. We showed that single-stepping is also possible on AMD SEV, and designed a framework called *SEV-Step* that offers the necessary infrastructure for single-stepping alongside further attack tools like page-fault tracking and cache attacks. We demonstrated SEV-Step by conducting an end-to-end cache attack against the cryptographic code contained in the Linux operating system, allowing us to obtain the volume key of a LUKS2-encrypted disk. In addition, we conducted precise measurements indicating that *Nemesis*-style [44] instruction latency measurement attacks also apply to AMD SEV.

The paper will appear at *CHES 2024* and was written in collaboration with Luca Wilke, Anja Rabich and Thomas Eisenbarth [251].

1.3 Outline

This thesis is structured in two parts. In the first part, after providing fundamental background on system architecture and microarchitecture, code analysis and software instrumentation in Chapter 2, we discuss the state of the art in side-channel attacks and defenses in Chapter 3. More precisely, we first summarize the many different classes of side-channel attacks and typical software-level leakages (Section 3.1). Then, we survey automated software leakage detection techniques and discuss the advantages and drawbacks of each approach (Section 3.2). Finally, we give an overview over manual and automated side-channel defenses, comparing both hardware- and software-oriented proposals (Section 3.3). In Chapter 4 we conclude the first part of this thesis and give an outlook on open questions and future research.

The second part then follows with the publications making up the main contributions of this thesis, in their original peer-reviewed text.

Background

In this chapter, we give some background necessary for following the contributions of this thesis. For the side-channel attacks and defenses, we provide an overview over common system architecture and trusted execution environments. In addition, as we later discuss software leakage analysis and automated code hardening in-depth, we need a general understanding of software instrumentation and code analysis.

2.1 System Architecture

System architecture can be viewed on different levels. At the highest level, most modern computers follow the von Neumann architecture model: A central processing unit (CPU) is connected with main memory and some input/output devices via a number of buses. Software usually relies on this *architectural* view of the system. This thesis mostly focuses on the next two lower levels, which are often referred to as *microarchitecture*: The organization of a CPU into separate physical cores with some shared resources like the last-level cache and memory bus, and the execution engine of a physical core itself. Software typically is oblivious to these layers, though it may contain optimizations to maximize utilization of microarchitectural components.

We start with the architectural view, which involves processes, the operating system, address spaces and virtualization. Afterward, we discuss caches and, finally, a physical core's microarchitecture and the out-of-order and speculative execution optimizations, which are subject to many side-channel attacks.

The following only gives a rough overview over the concepts needed for understanding side-channel attacks, and focuses on the x86-64 architecture. For a more detailed introduction into system architecture, we refer the reader to books like *Computer Architecture – A Quantitative Approach* [98] and *Operating Systems: Three Easy Pieces* [18], which also form the foundation of this section.

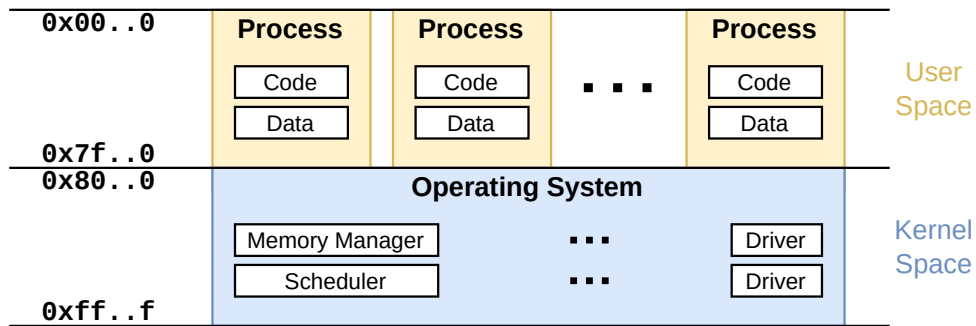


Figure 2.1: Memory layout of modern operating systems. Each process has its own virtual address space, comprising the entire encodable address range. The lower half of the address space (*user space*) belongs to the process itself, and may contain code, data, the heap, the stack and libraries the application depends on. The higher half (*kernel space*) is used by the operating system and is inaccessible by user code. Due to the isolated virtual address spaces, processes can only access their own code and data.

2.1.1 Operating System and Address Spaces

Contrary to embedded devices, general-purpose computers often run many processes in parallel. This requires careful isolation to keep a defective or compromised process from blocking resources, destabilizing the system, or interacting with data from other processes. As a solution, CPUs generally enforce a privilege model, where the (trusted) operating system (OS) runs with the highest privilege and manages many processes running at the lowest privilege level. Depending on the architecture, device drivers may occupy intermediate privilege levels or run with the same privileges as the operating system kernel. The processes themselves are isolated via *virtual address spaces* (Figure 2.1). This means that they can't access physical memory directly, but all accesses are to virtual (linear) addresses which are translated into physical addresses by the processor's memory management unit (MMU). Besides the security benefits, this separation also greatly simplifies software development, as the physical properties of the system are abstracted away and a process only sees its own contiguous, linear address space.

To enable efficient address translation, the virtual-to-physical address mappings are stored as a tree structure, where each node is a table that encodes several bits of the virtual address (9 bits per level on x86). The smallest translation unit is referred to as *page*, which typically has a size of 4096 bytes (4 kB). Consequently, the leafs of the translation tree are called *page tables*. As is common in the side-channel field, we use the term synonymous to the entire tree structure in this thesis, and do not distinguish between the different tree levels.

Page tables are managed by the kernel, which can also define for each page table entry whether the page or sub tree is privileged (*user/supervisor access bit*), whether it is currently

present in physical memory or was swapped out by the OS (*present* bit), or whether the page is writable (*read/write* bit). Other notable page table bits are the *accessed* bit, which is set by the CPU when the page is accessed by a program, and the *no-execute* bit, which prevents the page's data from being interpreted as executable instructions. Common operating systems partition the virtual space into two halves, where the lower half (0x00 . . 0-0x7f . . f) belongs to the process and the higher half (0x80 . . 0-0xff . . f) maps the kernel. By putting the kernel into the higher half of every process's virtual address space (protected by the *user/supervisor access* bit), system calls become very efficient, as the current address space does not need to be changed during user/kernel mode context switches.

When the process tries to access a virtual address that is not mapped in the current process (or the *present* bit of the corresponding page table entry is not set), a *page fault* exception is raised. This redirects execution to an interrupt handler in the OS, which receives the requested virtual address and may then decide whether the page is made available and the process is resumed afterward, or whether the process is terminated due to an illegal access.

2.1.2 Virtualization

In some cases, isolated processes may not suffice, as the user wants to virtualize an entire operating system. A common application for virtualization are cloud services, which host several customers on the same hardware to improve utilization and thus save costs. From the customer's view, their *virtual machine* (VM) looks and behaves like it runs on physical hardware.

As software-emulation through tools like QEMU [28, 182] is inefficient, modern CPUs provide hardware support for virtualization through extensions like Intel VT-x [105, Vol. 3, Ch. 24] or AMD SVM [10, Vol. 2, Ch. 15]. These introduce a new virtualized execution mode. The host operating system becomes the *hypervisor* (Figure 2.2). While the hypervisor may forward certain functionality to the VM, most accesses to system functionality must go through the hypervisor. This commonly happens through interrupts which transfer execution from the VM to the hypervisor.

A notable feature is memory virtualization through *nested paging*, which is hardware support for two layers of page tables: The mapping of host physical addresses (HPAs) to guest physical addresses (GPAs), which is managed by the hypervisor, and the mapping of GPAs to guest virtual addresses, which is done by the operating system inside the VM. When a process running inside the VM tries to access memory, the hardware first conducts a page walk in the VM's page table, and after resolving the GPA does a page walk in the host's page table to find the actual physical address. If a GPA is not mapped,

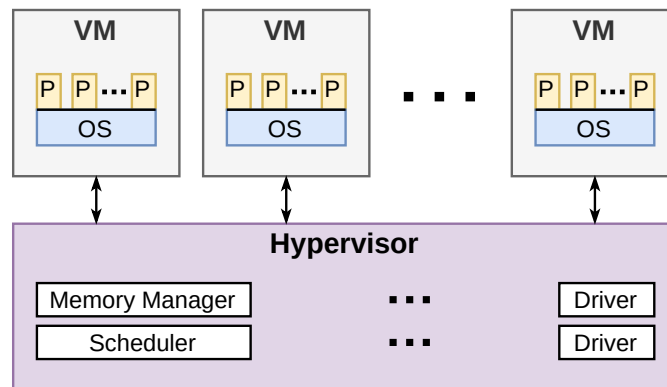


Figure 2.2: Hardware-assisted virtualization. The host operating system becomes the *hypervisor*, which mostly handles the same tasks as a normal OS, but instead of processes manages one or more virtual machines (VMs). Each VM contains a full operating system itself, which in turn runs its own processes. The hypervisor controls the VMs, handles interrupts and forwards input/output.

the processor raises a nested page fault exception (NPF), which exits the VM and must be handled by the hypervisor. After the page was resolved, the VM is resumed.

2.1.3 Caches

With growing throughput and parallelism of CPUs, main memory has become a major bottleneck. Both due to the lower frequency and physical distance of main memory, the CPU can process data much faster than it can be retrieved and written back. To address this, CPUs store the current working set in locations that are smaller and more close to the execution units, so-called *caches*. The key observation behind caches is that data is often accessed in a spatially and temporally local manner, i.e., it is likely that an access targets a similar address as a recent access. On common processors, all data (and code) that is used by a CPU core must pass through the cache. If the data is already present there, it can be directly processed. This is called *cache hit*. On the other hand, if the data is not in the cache and must be retrieved from main memory, this is referred to as *cache miss* and comes with a performance penalty.

Eviction Policies. As the cache is much smaller than main memory (ranging from kilobytes to few megabytes), the processor must carefully manage its use. The most important component is the *eviction policy*, which controls which address is evicted from the cache to make room for another address. Common policies are (pseudo) *least recently used* (LRU) and random replacement. As the names indicate, (P)LRU removes the address that has been used the longest time ago, while random replacement simply picks a random one. Evicted data is written back into main memory.

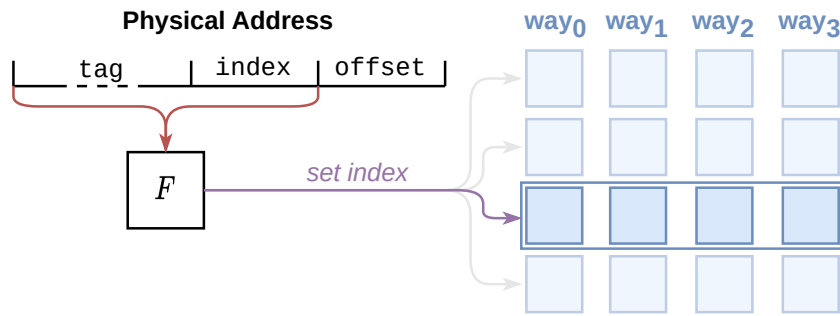


Figure 2.3: A set-associative cache with 4 ways. The tag and index parts of a cache line's physical address are converted into a set index through an arbitrary function F . The set index points to the cache set of that address, which consists of the cells where the cache line can be stored in (one for each way).

Cache Layout. To improve efficiency, the entire memory is partitioned into so-called *cache lines*, which usually amount to 64 bytes of data that are treated as a single unit. The physical address of a cached cache line is referred to as its *tag*. There are different methods for assigning cache lines to cache cells.

A simple layout is the directly mapped cache, where each cache line can end up in a single defined location which depends on its physical address. If a cache line is accessed that is mapped to the same location, the former one is evicted. On the other side, a *fully associative* cache allows any address to end up anywhere in the cache. While the former minimizes hardware overhead for tag comparison, parts of the cache may end up unused. The latter makes use of the full cache, but has a high hardware cost for comparing the cache tags.

A middle ground between both approaches is the *n-way set-associative cache* (Figure 2.3), which divides the cache into a number of *ways* and *sets*. Each cache set consists of a cell from each way, so the size of each set equals the number of ways n . A mapping function translates the physical address of a cache line into a *set index*. The cache line can only be stored in this cache set, but may end up in any cache way. This method minimizes hardware cost while maximizing cache utilization.

In the L1 and L2 caches (see below) of x86 CPUs the set index often matches bit 11 to bit 6 of the address, which are shared between virtual and physical addresses and are thus quickly available without having to wait for address translation. The last-level cache often has larger set counts, and the set index is computed through an arbitrary (hash-)function.

Cache Levels. As the performance of a cache correlates with its size and distance to the execution units, CPUs employ several cache levels, leading to a cache hierarchy

(Figure 2.4). The closest and smallest cache is the L1 (level 1) cache, which is often divided into L1d and L1i for data and instructions, respectively. The L1 is private to a physical core and directly connected to the core's front end and execution engine, serving all memory accesses. The next layer, L2, is usually also core-private and contains data evicted from the L1. Finally, many CPUs have a third layer, the *last-level cache* (LLC, or L3), that is shared between all cores and has the largest size. When a cache line is evicted, it is pushed into the next layer, so data from L2 is moved into the LLC, and the LLC evicts into main memory.

Caches may be *inclusive*, *non-inclusive* or *exclusive*. In an inclusive cache hierarchy, a cache line must be present in all lower cache levels. For example, if a cache line resides in the L1 cache, it must exist in L2 and LLC as well; data in L2 must be present in the LLC. Consequently, if a cache line is evicted from a low cache level (e.g., another physical core fills the LLC), it is also removed from all higher cache levels. Non-inclusive caches do not have this restriction. If a cache line is contained in an exclusive cache, it cannot be present in other cache levels.

If the same address is kept in different cache levels and cores, inconsistencies and stale data may occur. To avoid that, the processor enforces *cache coherence*, which guarantees that all cores always have the latest architectural view of cached data. A simple approach to that is *bus snooping*, where all cores use a shared bus to keep track of cached addresses. As this involves broadcasting and thus does not scale well with an increasing number of cores, recent architectures use *cache directories* instead. Cache directories map cache line addresses (tags) to the cores they are cached in, and store additional bits like the *dirty* state of the respective cache lines, allowing any core to efficiently query the state of a given cache line [258].

Other Caches. The data and instruction caches are not the only caches in modern processors. For example, as page walks are quite expensive, virtual-to-physical address mappings are cached in the *translation look-aside buffer* (TLB). Contrary to the data and instruction caches, the TLB is architecturally visible and must be explicitly invalidated by the kernel when changing page mappings. Many modern CPUs also feature μ OP caches, which store decoded instructions.

2.1.4 Execution Engine

Each CPU core consists of caches, a *front end* and a *back end* (Figure 2.5). The front end pulls the instructions from the L1i cache, decodes them into μ OPs and passes those to the back end for execution. The back end, also called *execution engine*, inserts the μ OPs into the scheduler, from where they are picked up by the respective execution units. The

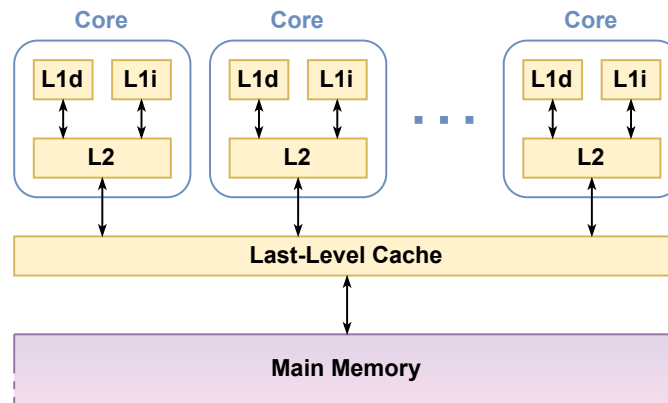


Figure 2.4: Cache hierarchy. Each physical core has its private L1 and L2 caches, where the L1 cache is divided into a data (L1d) and an instruction (L1i) cache. All cores share a common last-level cache (LLC). Data traverses all cache levels on its way to the CPU's execution engine. If a cache line gets evicted from the LLC, it is written back into main memory.

execution units carry out operations like arithmetic and memory accesses, for which they communicate with the L1d cache. The execution is pipelined to maximize front end and back end utilization, i.e., the next instruction is fetched while the previous one is decoded, and its predecessor is executed, and so on. After all μ OPs of an instruction are executed, the instruction is *retired* (or *committed*), i.e., its result becomes architecturally visible.

If a processor implements *simultaneous multi-threading* (SMT), each physical core is divided into two *logical* cores, which may have individual front ends, but share the same back end. This way, two processes can execute in parallel on the same physical core, maximizing utilization of the execution units in the back end at little additional hardware or power cost.

Out-of-order Execution. Traditional in-order execution is susceptible to stalls due to long-running operations, e.g., when an instruction waits for a non-cached memory access. To avoid idling while waiting for the instruction to complete, the processor may bring forward subsequent instructions which do not depend on the result of the stalled one. These out-of-order instructions traverse the front end and back end as usual, but wait in the reorder buffer for retirement until all previous instructions have retired. This means that future instructions may already affect the microarchitectural state, but they are not yet architecturally visible. A notable technique used for out-of-order execution is register renaming, which separates logical registers from physical registers and thus solves false dependencies which would otherwise prevent reordering.

Speculative Execution. Another cause for stalls are branch instructions which have to wait for an earlier instruction to complete in order to determine the branch target. To solve this, CPUs have heuristics that *guess* the target and then continue executing instructions *speculatively*. If the guess later turns out correct, the CPU has already executed many instructions that it now only needs to retire, improving overall performance. If the guess turns out wrong, the CPU has to rollback all speculatively executed instructions, which however is less expensive than always waiting for all branches to be retired. On Intel processors, the most relevant microarchitectural structure is the *branch prediction unit* (BPU). The BPU relies on information from the *pattern history table* (PHT), which tracks the historic outcomes of conditional branch instructions, the *branch target buffer* (BTB), which contains recent jump targets, and the *return stack buffer* (RSB), which caches return addresses.

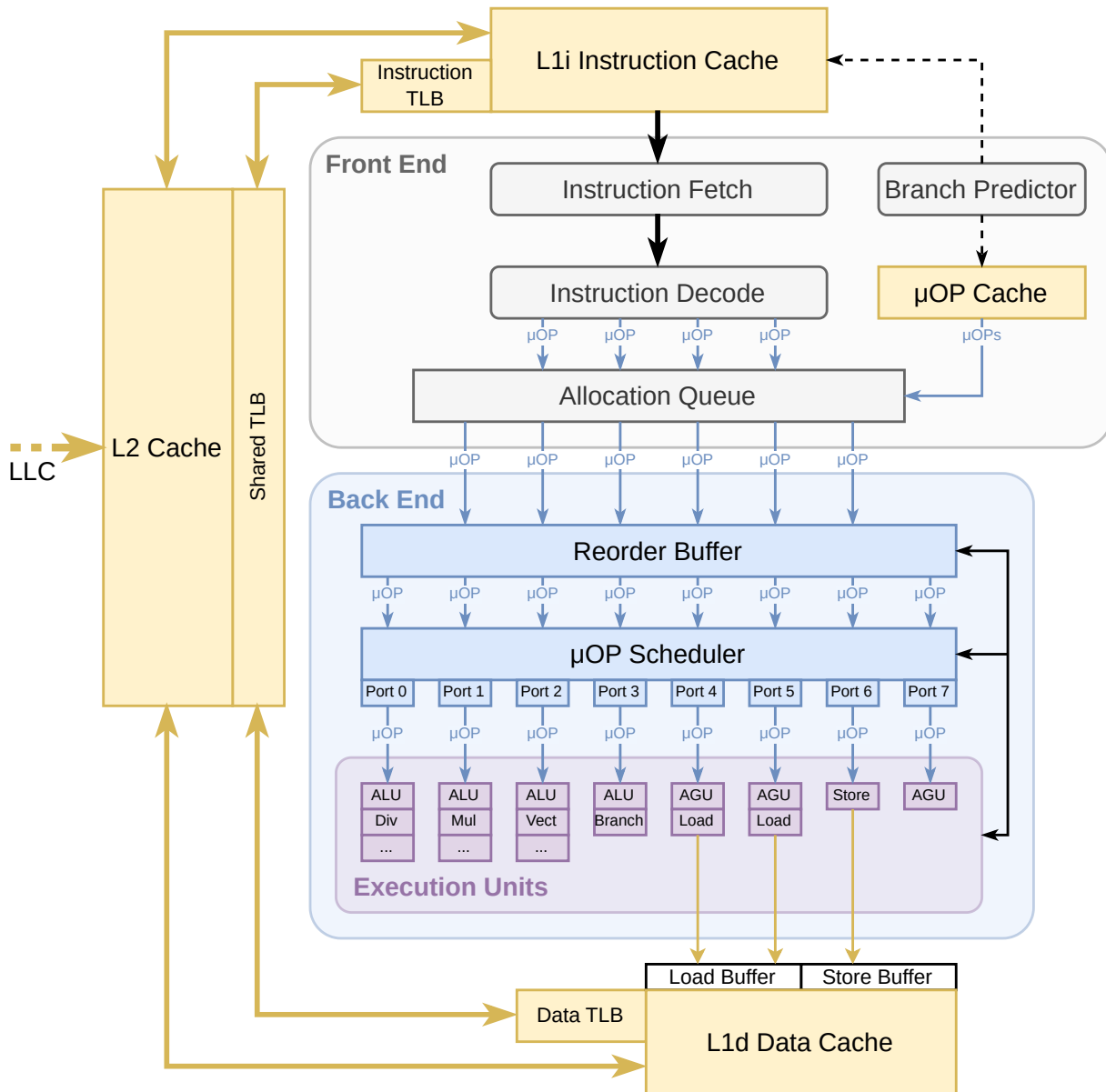


Figure 2.5: Illustration of a CPU core from the Intel Skylake Client microarchitecture, adapted and simplified from [211]. The front end fetches instructions from the instruction cache and decodes them into a series of μ OPs. The μ OPs are forwarded to the back end, which schedules them for execution. Actual execution is done by execution units, which are attached to execution ports. Each execution unit supports a fixed set of operations, like general-purpose arithmetic (ALU, arithmetic logic unit), vectorized arithmetic, address computations (AGU, address generation unit), or loads/stores.

2.2 Trusted Execution Environments

Usually, the local operating system or hypervisor is trusted, i.e., applications can freely handle sensitive data without taking particular precautions. However, this scenario not always holds – for example, a program may be running on the servers of a cloud provider, who may be compromised. Since the OS or the hypervisor have full access to the address space of the respective process or VM, they can arbitrarily manipulate its state or extract sensitive data. Trusted execution environments (TEEs) address this problem by providing hardware-assisted isolation, separating sensitive code and data into a so-called *enclave*. Common features of TEEs are encryption of memory and execution state, (remote) attestation of the enclave’s initial state, and measures to prevent the attacker from tampering with the enclave’s execution. There are two TEE types: Protection of processes and of whole VMs. We briefly describe both with reference to their most notable implementation.

2.2.1 Process-Level: Intel SGX

Intel *Software Guard Extensions* (SGX) [105, Vol. 3, Ch. 34] [106] divides a process into a trusted and untrusted part (Figure 2.6). The trusted part, invoked by the untrusted code via an *ECALL*, runs inside the SGX enclave and communicates with the untrusted code via *OCALLs*. SGX reserves a fixed part of the system’s physical memory and prevents all non-enclave accesses to this region. Most of the reserved memory is taken up by the *enclave page cache*, which holds the actual memory pages assignable to the enclave. SGX guarantees freshness and integrity of the encrypted data, even when it is temporarily swapped out of the EPC [56]. When leaving the enclave due to an interrupt, the current execution state is written into the *state save area* (SSA), which also resides on an EPC page.

Recently Intel introduced *Total Memory Encryption - Multi Key* (TME-MK) [111], which moves encryption into the memory controller and supports page-granular assignment of encryption keys. This change allows to arbitrarily expand the EPC, but does not support freshness and only protects integrity with respect to software-based attacks [113].

2.2.2 Virtual Machine Protection: AMD SEV

Whole-VM protection has received increased attention in recent years, as it would allow users to run their sensitive workloads in the cloud, without actually having to trust the provider to properly safeguard their data. The first scheme in this context is AMD’s *Secure Encrypted Virtualization* (SEV) [10, Vol. 2, Sec. 15.34], which offers drop-in protection

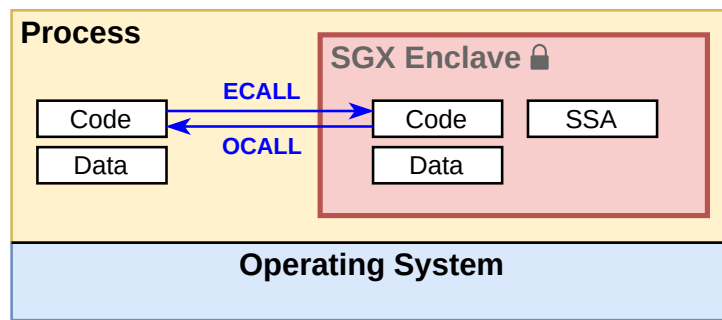


Figure 2.6: Process with SGX enclave. The enclave is isolated from the code and data of the parent process and may only be entered via ECALLs. The code inside the enclave has its own protected heap and stack, which are not accessible by the untrusted code outside the enclave. When the enclave is exited (e.g., due to an interrupt or an OCALL), the current execution state is written to the state save area (SSA). The enclave’s memory is encrypted and inaccessible by any other system component.

for VMs with all typical TEE features like remote attestation and memory encryption. SEV was iterated several times following attacks [41, 160, 161, 250, 252], first with the introduction of *Encrypted State* (SEV-ES), which added encryption to the VM’s *save area* (VMSA) during context switches, and *Secure Nested Paging* (SEV-SNP) [9], which prevents the hypervisor from modifying the VM’s memory and its page mappings.

To allow the hypervisor to setup and manage VMs, AMD introduced the *secure processor*, which holds the encryption keys and guarantees correct execution of remote attestation. The VM encryption keys are attached to the VM’s *address-space identifier* (ASID) and are only loaded when the VM is executing. If the VM wants to access certain privileged resources or receives an interrupt that it cannot handle, all execution state is written into the VMSA and control is transferred to the hypervisor. The hypervisor may then write result data into a dedicated shared page and resume the VM.

SEV-SNP uses a tweaked Xor-Encrypt-Xor (XEX) memory encryption scheme, where the tweak is computed from the physical address. Contrary to the original Intel SGX, AMD SEV’s memory encryption is deterministic, i.e., the ciphertext of a given encryption block solely depends on the plaintext, the secret key and the physical address. Additionally, even with SEV-SNP, the hypervisor can read the VM’s ciphertext.

Another upcoming VM TEE is Intel’s *Trusted Domain Extensions* (TDX) [108], which is similar to SEV in principle, but builds on top of TME-MK and provides stronger software-side isolation by routing all VM/hypervisor interactions through the dedicated TDX module, which forms another layer. Finally, ARM is currently working on its *Confidential Computing Architecture* (CCA) [17], which is stated to offer equivalent functionality.

2.3 Software Instrumentation

Instrumentation is the act of automatically inserting additional code into a program without modifying its source code [11]. The inserted code is invoked at runtime and may extract or modify program state. Applications of instrumentation are dynamic program analysis, for example to find out-of-bounds accesses or to trace information flow, or automatic deployment of defenses against certain attacks. There are two flavors of instrumentation: *Static instrumentation*, which permanently embeds the new code into the program binary, and *dynamic instrumentation*, which modifies the program's machine code while it is executed.

2.3.1 Static Instrumentation

Most commonly, static instrumentation is done through compiler extensions, which insert additional passes into the optimization pipeline. Depending on the goal, the passes may modify the compiler's intermediate representation (IR) or directly the generated machine code. A famous example for compiler-level instrumentation is AddressSanitizer (ASan) [1], which finds typical memory errors like out-of-bounds accesses and use-after-free bugs.

A different approach is *binary rewriting* or *static binary instrumentation* (SBI), which modifies an existing binary [26, 64, 134, 163, 244, 253]. Correct binary rewriting is notoriously difficult, as it needs to move existing machine code, adjust absolute offsets and avoid breaking jump tables. Additionally, binary formats like ELF are not intended for later modification, requiring great care when expanding segments to hold more code. Compiler-based approaches do not have these limitations, and IR-based instrumentation can even freely use variables without worrying about register allocation. Hence, compiler-level instrumentation is typically more stable, can use more source-level information, and achieves better performance.

However, as a consequence, compiler-level instrumentation often changes the machine code and stack layout, reducing the applicability of code analysis results to the uninstrumented program. Binary rewriting takes the program as it is, so it accurately captures its real runtime behavior. In addition, binary rewriting also covers pre-compiled dependencies, avoiding complex rebuilds of system libraries.

Thus, in summary, the choice between binary rewriting and compiler-level instrumentation depends on the desired application.

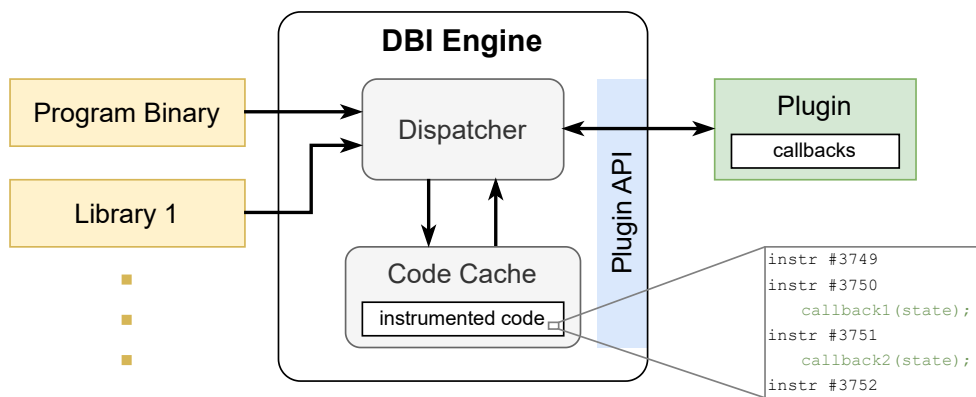


Figure 2.7: Design of a typical DBI framework. The main parts of a DBI engine are the dispatcher and the code cache. The dispatcher loads the code from the input binaries and transforms it by inserting calls to callback functions provided by a plugin. The instrumented code is cached and executed when needed. After executing a sequence of instrumented code, control is returned to the dispatcher, which fetches and instruments the next code sequence.

2.3.2 Dynamic Instrumentation

If persistent instrumentation is not needed, *dynamic binary instrumentation* (DBI) offers the stability of compiler-level instrumentation combined with the transparency of binary rewriting. For this, a DBI framework like Intel Pin [176], DynamoRIO [69], Valgrind [165, 222] or MAMBO [82, 149] loads the program binary and rewrites the machine code in memory to invoke the DBI engine at defined locations (Figure 2.7). The user provides a plugin that registers instrumentation callbacks, which instruct the DBI engine which transformations are desired. These transformations may themselves insert callbacks into the program code, allowing to, e.g., log all memory accesses.

A major advantage of DBI is the availability of runtime information and the high control of the process layout in memory, which avoids much of the complexity of binary rewriting. The code is usually instrumented on-the-fly, and invalid or unknown paths are caught by the DBI engine. Most DBI engines aim for full behavioral transparency, i.e., the program shows the same architectural behavior as without instrumentation. This way, the user gets an accurate view of the program’s execution.

2.4 Code Analysis

Analysis of program code and behavior plays a large role in security research. Applications range from classic bug finding over information flow tracking to automated patching. Depending on the desired result, one may pick static approaches which prove a certain program property (e.g., absence of out-of-bounds errors), or dynamic approaches,

which evaluate the program for a set of inputs. In the following, we introduce three common code analysis primitives: Symbolic execution, taint analysis and fuzzing.

2.4.1 Symbolic Execution

Instead of running the program with concrete inputs, a symbolic execution engine assumes *symbolic* inputs and gradually builds a set of constraints representing the entire execution [11, 20, 123]. The engine then invokes a *solver* to test these constraints (and optionally some security property) for satisfiability. Symbolic execution allows to *prove* a security property over all possible inputs, making it a valuable tool for code analysis. However, due to its generality it also suffers from issues limiting its practical applicability: Path explosion leads to exponential growth of formulas and drastically reduces analysis performance. Additionally, there can be difficulties when mapping between the real system and the symbolic expression, e.g., when dealing with raw pointers and freely addressable memory, which may require a simplified model.

A weaker version of symbolic execution is *dynamic* (or *concolic*) symbolic execution [80], where the program is executed with concrete inputs and the engine only builds a formula for the observed execution path. While it sacrifices the ability to generate definite proofs, dynamic symbolic execution is a valuable tool for finding test cases and vulnerabilities that comes with better performance than fully static symbolic execution.

There are many different symbolic execution tools, often rooted within the security research community; common examples are angr [13, 206], KLEE [46, 124] and SymCC [177, 219].

2.4.2 Taint Analysis

Taint analysis [11, 199] allows to analyze the flow of data through a program. In the *taint source*, every variable of interest is assigned a label (taint), which is propagated throughout the program's execution until it encounters a *taint sink*. Labels can be attached to registers, status flags and memory locations. When such a labeled value is used as an operand, the result is labeled as well.

Dynamic taint analysis (DTA) is implemented with instrumentation, i.e., it analyzes the program for concrete inputs, though symbolic taint analysis is also possible. The performance and accuracy of taint analysis depend on the analysis granularity (e.g., bit-level vs. byte-level) and the prevalence of undertainting/overtainting due to under/overapproximations in the label propagation logic.

2.4.3 Fuzzing

Software testing often suffers from insufficient coverage, i.e., execution paths are missed by the available input set. *Fuzzing* [150, 155] supplements these handcrafted inputs with automatically generated ones, trying to maximize program coverage.

There are many different types of fuzzers and fuzzing methods, which are selected depending on the model and goal. For example, a fuzzer may try to satisfy some constraint, or simply crash the program. Black-box fuzzers do not get any internal program information and can only guess the next input, while white-box fuzzers use techniques like dynamic symbolic execution to directly generate promising inputs. Grey-box fuzzers like AFL [135] take a middle ground, receiving some information about the program and the observed execution path to make an educated guess for the next input. Fuzzers have been successfully used to uncover many vulnerabilities and fix them prior to their exploitation, making fuzzers an essential tool for developing secure software.

State of the Art

In this chapter, we discuss the state of the art in side-channel attacks and defenses. We first summarize and classify side-channel vulnerabilities and the associated attack techniques in Section 3.1. In Section 3.2, we survey and evaluate tools for finding memory access pattern leakages in software, addressing a broad class of side-channel vulnerabilities. Finally, in Section 3.3, we dive into generic software/hardware mitigations for different kinds of side-channel leakages.

3.1 Side-Channel Leakage in the CPU

Contrary to “classic” data exfiltration attacks like buffer overflow exploits, CPU side-channel attacks do not seek *direct, architectural* access to secret data, but they measure and manipulate execution behavior to infer secrets *indirectly*. The field of side-channel attacks is very broad, spanning a multitude of attack techniques and threat models. In the following, we first introduce the most common type of side-channel leakage, memory access patterns. Then, we give an overview over the various attack techniques and discuss their respective capabilities.

Memory Access Pattern Leakage. The majority of side-channel exploits, especially of cryptographic implementations, is based on observing memory access patterns. These observations concern code and data with a granularity ranging from memory pages to individual words. The temporal resolution varies between attacks, from imprecise measurements covering an entire execution of a cryptographic primitive to precise pinpointing of a single access. Memory access pattern attacks are tightly linked to classical timing attacks [126], since control flow variations (e.g., loop iteration counts) affect the total execution time of the program, and the latency of memory accesses depends on whether data is cached or not.

Regardless of the attack technique, the vulnerability of a program to memory access pattern analysis boils down to its use of secret-dependent memory accesses and branches: If an accessed memory location is correlated with a secret input, the attacker may recover

and even attacks over the network [130]. Cache attacks are also a central part of transient execution attacks as a covert channel for extracting the transiently accessed data.

In the following, we discuss the impact of cache attacks on cryptography, the influence of the targeted cache level, and several common attack techniques and their advantages and drawbacks.

Cache Attacks and Cryptography. Cache attacks have arguably received the highest scrutiny due to being able to break insufficiently hardened implementations of many cryptographic primitives. For example, AES implementations often use large lookup tables which combine several round operations, speeding up encryption and decryption significantly. These so-called *T-tables* have been subject to attacks across many libraries, as tracing the accessed table indexes allows to quickly recover the key [15, 30, 92, 115, 151, 156, 166, 171, 215, 251]. Similarly, sliding window exponentiation, an optimization used by many ElGamal and RSA implementations, often relies on a lookup table with precomputed multipliers. Due to their size, the entries can cover multiple cache lines each, making accesses to them easily detectable by an attacker [100, 146]. Nevertheless, we also showed that, given sufficient temporal resolution, cache attacks can even be used to target very small and dense lookup tables, for example those used for Base64 decoding [208]. We achieved this by combining a cache attack with single-stepping (Section 3.1.3) to accurately measure specific table accesses, and by exploiting the high redundancy of the PEM RSA storage format we were able to reconstruct an RSA private key.

Aside from lookup tables, cache attacks also work against secret-dependent control flow. A prime example is the square-and-multiply algorithm for RSA (and analogously, double-and-add for elliptic curves), which executes the multiplication depending on the current secret bit. An attacker able to detect whether the multiplication was executed can directly reconstruct the secret exponent [259].

As a reaction to those attacks, most libraries have moved to constant-time implementations or employ countermeasures like blinding. However, these must be applied with a high amount of care, and even minimal remaining leakage might still be exploited, as shown by Aranha et al., who managed to break a hardened ECDSA implementation which leaked the second most-significant bit of the nonce [16]. See Section 3.3.1 for a deeper discussion of countermeasures.

Cache Levels. Any cache level can be subject to an attack, where each has advantages and disadvantages. Attacks against the L1 and L2 caches often enjoy a high temporal resolution due to their small size, which speeds up generation and application of the eviction sets needed for some attack techniques. However, such attacks generally require

the attacker to run on the same CPU core as the victim, which is easier to achieve in TEE scenarios than in the generic cloud setting. On the other side, attacks against the LLC (L3) cache can run on an arbitrary physical core, as the cache is shared between all cores. As the LLC is much larger and often has a more complex structure (e.g., cache slices [116, 152]) than lower cache levels, attacks tend to be slower and somewhat less precise [66, 181]. However, this drawback can be compensated through sophisticated attack techniques and a higher amount of measurements.

Cache Attack Techniques. There are many different cache attack techniques, but they fall generally into three groups [77]: *Evict+Time*, *Flush+Reload* and *Prime+Probe*.

In the *Evict+Time* attack, the attacker evicts the target cache line from the cache. Subsequently, they call the victim and measure its execution time. If the time is high, the victim accessed the evicted cache line and encountered a cache miss. If the time is low, the victim did not access the address. Given a sufficient amount of measurements, the attacker is able to correlate the measured times with secret bits [171]. Note that the attack is *synchronous*: The attacker must be able to precisely invoke the victim and measure its execution. This holds, for example, for network services like SSH and TLS, where the attacker can arbitrarily initiate connections.

If this is not an option, the attacker can resort to *asynchronous* attacks. One of those is *Flush+Reload* [259], which applies to settings where the victim and the attacker share a physical memory page. This is by far not unrealistic [85, 139, 203]: For example, operating systems may choose to map physical memory containing a standard library into multiple processes at once, avoiding to hold the same read-only data in physical memory multiple times. If any process modifies such a shared page, it is copied (copy-on-write). From an architectural point of view, this so-called *memory deduplication* is a valid and secure optimization. A hypervisor may do the same for virtual machines, by merging pages which hold the same data. To conduct a *Flush+Reload* attack, the attacker first flushes the shared cache line back into main memory using a dedicated instruction (e.g., `clflush` on x86). After waiting for the victim to access the flushed address (e.g., a part of a lookup table), the attacker reloads the address and measures the time needed to do so. If the access hits, the victim accessed the address as well. *Flush+Reload* is very fast and precise, as removing the target address from cache is efficient, and due to targeting a specific physical address there are few false positives. A variant of the attack is *Flush+Flush* [89], which replaces the reload step with another flush. The latency of the flush depends on whether the address is cached or not, so it yields the same information as a reload, but without generating suspicious cache misses. This makes the attack stealthier in presence of detection mechanisms.

The most generic attack method is *Prime+Probe* [171, 175], which is asynchronous and

does not require shared memory, as it relies on cache contention. In an initial step, the attacker generates an *eviction set* for reliably filling an entire cache set with their own data. The attack then consists of two steps: First the attacker *primes* the cache set using their eviction set, which evicts the target address from the cache. After waiting for the victim to make its accesses, the attacker *probes* the cache set: They access each address from their eviction set, and measure the time needed to do so. If they observe a cache miss, a part of the eviction set must have been evicted from the cache, which is likely due to the victim accessing the target address in the meantime. Due to the indirect approach (compared to *Flush+Reload*'s direct one), the attack is more prone to false positives and more complex to carry out. Some interfering factors are cache slicing [100, 116, 152], non-inclusive caches [142, 258] and replacement policies [37, 84, 142], but all of these can be circumvented (or even exploited), making *Prime+Probe* the most used attack class due to its few preconditions and high flexibility.

A notable combination of *Flush+Reload* and *Prime+Probe* is *Evict+Reload* [90, 258]. The method also needs shared memory, but evicts the target address through an eviction set. Hence, it does not require a special cache flushing instruction, which is not available on all systems.

While most cache attacks use time measurements to distinguish hits and misses, this is not the only approach. The *Prime+Abort* [66] attack uses Intel TSX [105, Vol. 1, Ch. 16], an ISA extension that offers hardware-supported transactional memory. The attacker starts a TSX transaction and then primes the cache with their eviction set. Then, they wait for the victim access. By design, TSX transactions abort when addresses loaded during the transaction get evicted from the cache. As this happens when the victim accesses data that collides with the eviction set, the attacker gets notified immediately. The abort also comes with an error code, so the attacker can assess whether the abort happened due to unrelated noise (e.g., timer interrupts) or victim activity. While TSX is supported by many Intel processors, it was involved in several noteworthy side-channel attacks, and thus disabled for most CPU families via a microcode update [107]. Another notable timer-less cache attack is *S²C* [262], which uses hardware synchronization instructions on Apple M1 processors to directly measure cache evictions.

A main drawback of *Prime+Probe*-style attacks is their impact on the cache state during the priming/probing phases. If the attacker is not well-synchronized, the victim may do their access just when the attacker is refreshing the eviction set. It would be preferable if the probing would not interfere with the cache state at all, and the eviction set would only need to be reloaded after a victim access was detected. *Prime+Scope* [181] solves this problem by exploiting a property of inclusive caches: When a cache line is evicted from the LLC, it is also evicted from all other cache levels. The attacker primes the LLC in a way that they *know* which cache line from the eviction set is the next eviction candidate,

i.e., will be pushed out by the victim. They then continuously probe this cache line on a core separate from the victim. These accesses will be served by the core's L1 cache, without any effect on the LLC. When the victim does its access, the eviction candidate is removed from the LLC and thus the L1 cache as well, notifying the attacker. This attack also works with non-inclusive caches; in this case, the cache directory instead of the LLC is targeted.

TLB Attacks. Similar attacks can also be mounted against the TLB, which caches the virtual to physical address mappings. In the *TLBleed* attack [83] the attacker uses a *Prime+Probe*-like method to monitor the victim's page access patterns. Though the attack has a much lower granularity than one targeting the cache, the authors still manage to break EdDSA and RSA implementations.

Sub-Cache Line Granularity. A few attacks even achieved a spatial resolution below the 64 byte granularity offered by cache attacks. The *CacheBleed* attack [260] exploits cache bank conflicts. Cache banks are used by some processors to serve requests to different L1 cache line offsets in parallel. If the attacker runs on the same physical core as the victim, they can issue many accesses targeting a particular cache bank, slowing down the victim's accesses that hit the same cache bank. The authors show that their method can break scatter-gather implementations, which were originally recommended for mitigating cache attacks.

The *MemJam* attack [157] exploits *4K aliasing*, which is a false dependency between different addresses sharing the lower 12 bits. L1 caches often are virtually addressed, as physical and virtual addresses share the lower 12 bits, which are sufficient for indexing 64 cache sets. This means that accesses to virtual addresses that differ in the lower 12 bits can be immediately discerned, without waiting for the virtual-to-physical address translation. However, if the lower 12 bits are identical, the two virtual addresses may theoretically refer to the same physical address – a possible conflict that can only be resolved by waiting for the address translation to complete. We used *MemJam* to break the DES, AES and SM4 implementations of the Intel IPP library, which was specifically hardened against cache attacks. Later the attack was improved and applied against SGX enclaves [209].

3.1.2 Page Fault Controlled Channel

A much more powerful variant of the TLB attack is the *page fault controlled channel*, which can leak memory access patterns with page granularity and high temporal resolution. The original version [257] assumes an untrusted OS: Instead of indirectly evicting the victim's pages, the OS can clear the *present* bit of every page, indicating that they are not

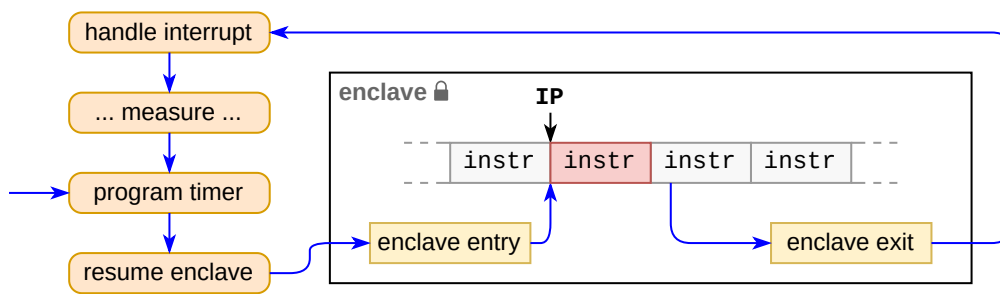


Figure 3.2: Single-stepping of an enclave. The blue arrows denote the execution flow. The attacker programs a timer to interrupt the enclave precisely after the current instruction (highlighted in red) is executed, but before the enclave can fully execute another one. After the enclave has exited, the timer interrupt handler is invoked, where the attacker can measure the instruction latency or conduct a fine-grained cache attack. Finally, they re-program the timer and resume the enclave, to execute the next instruction.

currently mapped in physical memory. When the victim tries to execute an instruction, this inevitably triggers a page fault, as the CPU cannot fetch the instruction or its data operands from memory. The OS handles the page fault and resumes execution. This stepping of the victim grants the OS an execution trace containing the page frame numbers of most executed instructions and all accessed data.

While the untrusted OS model by itself is somewhat academic (the OS could just access the victim’s data directly), that attacker model becomes highly relevant in the context of TEEs like Intel SGX or AMD SEV, which try to isolate the untrusted host (hypervisor) from the user processes and VMs through hardware-enforced access control and memory encryption. Indeed, the (nested) page fault controlled channel was used to enable and assist many attacks on TEEs [94, 136, 137, 159, 160, 250, 251, 252]. For example, we used page fault pattern-based fingerprints to identify the AES functions in the Linux kernel when attacking LUKS2 disk encryption with *SEV-Step* [251]. The *CopyCat* attack by Moghimi et al. [159] relies on the page table entry *accessed* bit to accurately count instructions while single-stepping on SGX.

3.1.3 Single-Stepping

While the page fault controlled channel allows to follow execution with page granularity (often better, if many instructions have data operands), it is possible to increase this attack to the maximum possible resolution, which is *single-stepping* the code running in the TEE. Single-stepping is a powerful primitive which allows to count instructions [159], measure their precise execution time [44] or greatly increase the precision of cache attacks [156, 208, 251].

Instead of using page faults, the attacker programs a *timer interrupt* to fire after a certain amount of cycles. Single-stepping is achieved by carefully tuning the timer threshold to interrupt the enclave as soon as the first instruction has started execution (Figure 3.2). There are several challenges: The enclave entry does not take a fixed time, as it has to restore the enclaves' current execution state from memory, which may lead to the timer firing too early, resulting in zero-stepping. This can be addressed by additionally clearing and then checking the *accessed* bit of the associated code page – if the bit is set, an instruction was fetched. Another challenge for precise measurements is the noise introduced by the enclave exit and the interrupt handler. The *Nemesis* attack tries to mitigate the latter through a user-space handler, to avoid a noisy user/kernel context switch.

An early user of timer interrupts was *CacheZoom* [156], which stepped several instructions at a time. Full SGX single-stepping was first achieved through the *SGX-Step* framework [45], which became a standard tool used in many subsequent attack papers [43, 44, 117, 158, 159, 178, 208, 209]. We later designed and published an equivalent framework for AMD SEV, called *SEV-Step* [251], and demonstrated an attack which manages to steal an AES disk encryption key within a single execution of the targeted implementation.

3.1.4 Transient Execution

Most conventional side-channel attacks are based on *observation*, i.e., they measure and exploit the effects a program's execution has on the system's internal and visible states. In case of memory access pattern leakage, the program follows its architectural control flow while leaving a footprint in the cache and other components. *Transient execution attacks* take a different route, as they *transiently* invoke behavior that is not architectural, i.e., that should not occur at all per the system's specification. There are two major categories of such attacks: Speculative execution of invalid program paths, and transient out-of-order execution of instructions that should not be executed due to a prior fault. Both attack types allow extracting data that should not be accessible architecturally, and they frequently cross privilege boundaries.

For referring to the different attack variants we use the naming scheme introduced by Canella et al. [47].

Speculative Execution Attacks. Speculative execution attacks exploit a row of CPU optimizations that *speculatively* execute code paths when one or more preconditions are still unknown (e.g., due to a non-cached operand). This way, if the predicted code path is later deemed correct, the processor has already completed a number of tasks which else would have had to wait until the stalled precondition was resolved. As shown first in the


```
int value;
if(index >= 0 && index < array->len) {
    value = array->data[index];
}
else {
    throw_outofbounds();
}
// ...
leak(value);
```

Figure 3.3: A simple out-of-bounds check forming a *Spectre-PHT* gadget, as it may be emitted by a JIT compiler. Before `array->data` is accessed, the `index` variable is checked against the array length. Architecturally, the array access is only reachable if `index` is within bounds. However, the attacker may remove `array->len` from the cache, slowing down the bounds check, which leads the CPU to speculate on the branch outcome. If the branch predictor was mistrained to assume that the check passes, the access is carried out speculatively with an invalid index. The extracted value is then leaked, e.g., through another array access where it is used as an offset.

Spectre attack [125], the attacker can specifically train the branch predictors to execute an invalid path and thus do unintended memory accesses. This would not be bad by itself, if the speculative execution would not leave a microarchitectural footprint even after being rolled back. For example, data that is accessed during speculative execution is loaded into the L1 cache, and stays there – allowing to extract information via cache-based covert channels. There are many variants of *Spectre* attacks, which differ in the exploited predictor and the covert channel that is used to leak the extracted data.

The most common example for a *Spectre* attack is *Spectre-PHT* (for *pattern history table*), which is based on an incorrect *branch taken/not taken* prediction. A typical vulnerable gadget is a bounds check before accessing an array (Figure 3.3). While *Spectre-PHT* attacks usually stay close to the architectural execution path, it is also possible to redirect execution to arbitrary locations. The *branch target buffer* (BTB) tracks recent targets of indirect branches, and the *return stack buffer* (RSB) stores the return address when a call instruction is encountered, so the processor does not need to wait for the stack read when handling a function return. Accordingly, *Spectre-BTB* and *Spectre-RSB* manipulate these buffers in a way that they point to an attacker-specified address. These variants allow building gadget chains similar to return-oriented programming attacks, and have been used for leaking data from other processes and the kernel [128, 148, 249].

After speculatively accessing secret data, the attacker needs a way to leak it outside the transient domain. Most attacks rely on a cache covert channel based on, for example, the *Prime+Probe* method. The attacker first prepares eviction sets for different cache sets, and primes the cache accordingly. Depending on the leaked value, the *Spectre* gadget

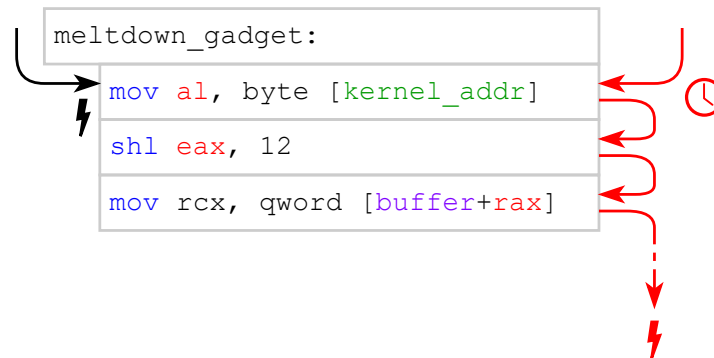


Figure 3.4: *Meltdown* attack. The black arrow on the left represents architectural execution, the red arrows on the right correspond to transient out-of-order execution. Upon accessing the privileged address `kernel_addr`, architectural execution is immediately aborted and an exception is raised. However, that is not what happens *microarchitecturally*: The exception is delayed enough that out-of-order execution can read the (cached) kernel address, rescale the result conveniently, and then leak it via an access to a `buffer`. Out-of-order execution is aborted eventually, but the accessed part of `buffer` remains in the cache and can be detected through the *Flush+Reload* method.

then accesses one of the aforementioned cache sets. Finally, the attacker probes the cache sets to learn the leaked value. Depending on the number of primed cache sets, multiple bits can be leaked in a single operation [125]. In the *NetSpectre* attack [202], the authors leak data by selectively powering on the AVX2 unit, which comes with a high and thus measurable performance penalty. Another possible covert channel is port contention, if the attacker resides on the sibling logical core [31].

Out-of-order Execution Attacks. *Meltdown*-style attacks [47, 144] do not rely on misprediction, but exploit microarchitectural race conditions. The root cause of this attack class is out-of-order execution, i.e., that instructions are (partially) executed before their predecessors are committed. Similar to speculative execution, this optimization allows the CPU to fully utilize its execution units even if a particular instruction is stalled (e.g., due to a slow load or a page table walk). Out-of-order execution is possible whenever an instruction does not depend on a prior one.

In the original *Meltdown* attack [144], the authors showed that out-of-order executed instructions can access privileged memory and leak its contents through the cache, before the processor completes the access check and raises a page fault exception, rolling back transient execution (Figure 3.4). As many operating systems map kernel addresses in the higher half of the virtual address space and only prevent user-space accesses through the *user/supervisor access* bit, the attack is able to read arbitrary kernel memory that currently happens to reside in the L1 cache. Accordingly, the attack is also referred to as *Meltdown-US-L1*. A variation of this attack even works from virtual machines: The *Foreshadow*

attack [42] (also known as *Meltdown-P-L1*) exploits short-circuiting of the nested page translation in Intel processors, which leads to the guest physical address (instead of the host physical address) being transiently used for accessing the L1 cache.

Since then, many more *Meltdown* variants have been published [43, 48, 144, 158, 188, 189, 197, 201, 216], targeting different CPU components. Some attacks do not even need to trigger exceptions, allowing them to circumvent hardware protections implemented by Intel [48, 197]. *Load Value Injection* (LVI) [43] extracts secrets from an SGX enclave by injecting malicious data that is used transiently when a load within the enclave fails. The *Downfall* attack [158] reads stale data from AVX registers through gather instructions. These registers do not only hold privileged data, but are also used by the SGX runtime for attestation. This way, the author was able to extract the SGX sealing key, breaking the entire trust model.

3.1.5 Value-Based Leakages

Cache attacks, page faults and single-stepping extract information indirectly by observing a program's memory access patterns. However, there are also channels where the CPU leaks a value itself, without the program doing a secret-dependent access, and without involving transient execution.

A widely known issue are data operand-dependent execution latencies of instructions. For example, multiplication and division instructions may have fast paths for certain simple cases, and it was shown that performance penalties for subnormal floating point numbers can be used to breach browser security boundaries and attack differentially private databases [12]. A similar issue are *silent stores*, where the CPU quietly eliminates a store because it does not change the data that is already present in the target memory location [73].

In the recent *GoFetch* attack [53], the authors exploited *data memory-dependent prefetchers* (DMP). Prefetchers are commonly used to load data into the cache that is likely to be used soon. Usually, these prefetchers rely on access pattern-based heuristics; DMP looks for pointers in the recently used memory and prefetches the target addresses, leaking the pointer to the attacker. This can also be used to extract non-pointer data, if the attacker manages to manipulate the secret in a way that it *looks* like a pointer to the prefetcher.

A TEE-specific attack class are *ciphertext side-channels* [136, 138]. AMD SEV uses deterministic tweaked memory encryption, i.e., the ciphertext depends on the plaintext and the physical address. This means that writing the same plaintext to the same address leads to the same ciphertext. In conjunction with Li et al., we showed that an attacker can break constant-time cryptographic implementations solely by observing the ciphertext during

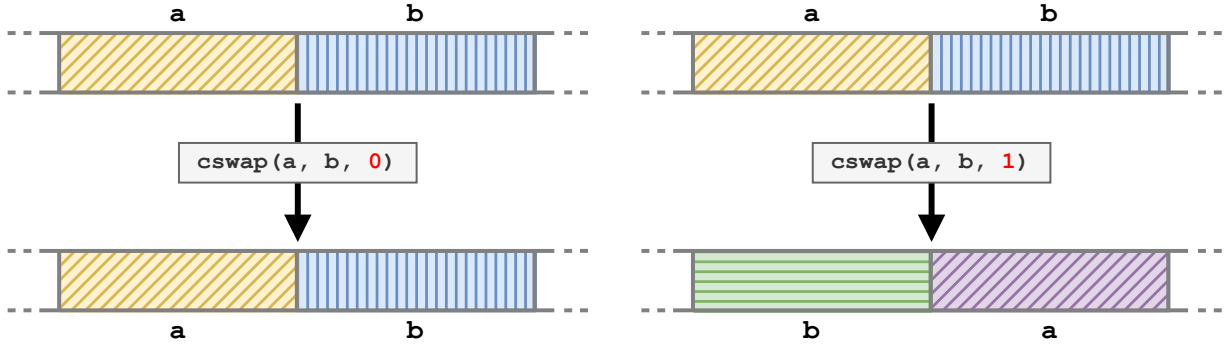


Figure 3.5: A constant-time swap operation susceptible to a ciphertext side-channel attack. The function `cswap` swaps the values of variables `a` and `b` if the secret bit is 1. The colors and patterns resemble a particular ciphertext. The secret bit is directly leaked through the ciphertext change.

encryption. The first attack primitive is the *dictionary attack*, where a variable can only hold a small range of values, so the attacker can build a mapping of observed ciphertexts to (suspected) plaintexts. The second and more generic primitive is the *collision attack*, where the attacker extracts secrets by checking whether ciphertexts repeat. Collisions happen frequently with constant-time swaps, an important primitive for elliptic curve implementations. By observing whether a memory location is changed or not, the attacker immediately learns the secret bit (Figure 3.5).

3.1.6 Other Attacks and Leakages

Execution Unit Contention. When a CPU implements SMT, logical cores share the same execution units. By observing the latency of certain operations, the attacker can leak the current workload of the neighboring core. Port contention attacks slightly delay operations of the same type as issued by the attacker [4]. Scheduler queue contention on AMD CPUs allows an attacker to temporarily block certain instructions from execution, e.g., multiplication, allowing to measure whether an application makes increased use of them [76].

Amplification. Often attacks require high-precision timers, noise reduction and repetition to extract the leaked data. *Amplification* aims to increase the signal-to-noise ratio through different means. For example, one can abuse eviction strategies and non-temporal prefetches to distinguish whether an item is cached with millisecond granularity [179, 194]. Katzman et al. [120] build logical gadgets within transient execution, allowing them to do arbitrary computations in the transient domain and generate high timing differences.

A similar approach is slowing down the victim. Such *performance degradation* attacks generate contention by, for example, repeatedly evicting cache lines [6] or executing atomic instructions which lock the memory bus [99]. This way, the attacker can more accurately target the vulnerable code sections.

Power and Frequency Side-Channel Attacks. Instead of exploiting architectural and microarchitectural behavior, side-channel attacks can also measure and manipulate physical properties of the hardware itself. This can be used both for extracting secrets like register values and for injecting faults into computations.

With *Running Average Power Limit* (RAPL) [105, Vol. 3, Sec. 15.10], Intel provides an interface for reading the system's current power consumption, that was used for several attacks. A similar interface is available on AMD CPUs. For example, in *PLATYPUS* [143], the authors were able to recover keys from AES-NI code running in an SGX enclave. Lipp et al. [140] combine AMD's version of RAPL with a prefetch side-channel to break KASLR.

The *Hertzbleed* attacks [236, 237] exploit varying CPU power consumption and frequency due to the currently processed data. They break various crypto systems by triggering specific bit sequences, which cause the CPU to increase its frequency, leading to a measurable timing difference.

Fault Attacks. Fault attacks change the architectural state in an invalid way by modifying data stored in memory or influencing the result of computations through physical means. This allows breaking protections like user/supervisor access bits in page tables or extracting cryptographic keys.

Rowhammer attacks [87, 88, 122, 127, 145, 225] exploit the increasing density of DRAM chips. While this reduces power consumption and increases capacity, close memory cells are prone to physical interactions. By repeatedly *hammering* (i.e., accessing) physically close memory locations, there is a certain chance that a neighboring memory cell flips its value. This attack strategy can also be used to *read* data from other memory cells which belong to an otherwise inaccessible address space [131].

Another fault injection vector is *undervolting* through software, which abuses a feature originally intended for under-/overclocking and energy saving. By faulting certain types of arithmetic instructions (e.g., multiplication), cryptographic keys can be extracted from SGX [121, 162, 184] and TrustZone [183] enclaves without hardware modifications.

3.2 Software Leakage Analysis

Over the years, numerous approaches for automated software leakage analysis were proposed. Most of them target timing and cache side-channel leakages, i.e., vulnerabilities stemming from secret-dependent memory accesses and control flow, which can be exploited by monitoring a program's memory access pattern. By removing these secret-dependent operations, memory access pattern-based attacks can be averted. In the following, we give an overview of the various approaches on finding such secret-dependent operations in software, for which there are *static* and *dynamic* methods. Afterward, we compare the different approaches, discuss their strengths and weaknesses, and point out directions for future work. Finally, we briefly describe related work that finds other leakage types such as transient execution vulnerabilities and ciphertext side-channels.

Note that research in this area is inherently tool-driven, i.e., a new approach is usually assigned a name and complemented with a proof-of-concept implementation and a suitable evaluation. We thus often use the tool name to refer to a specific approach.

Sound(i)ness and Completeness. We call a leakage detection tool *sound* if it never states that an insecure program is secure, or phrased differently, it never misses a real leakage (*no false negatives*). Analogously, we call a leakage detection tool *complete*, if it never falsely states that a program is insecure, i.e., it only rejects programs that are actually insecure (*no false positives*). Consequently, if the tool is both sound and complete, it always correctly determines the security of a program.

In practice, strict soundness is unrealistic. A weaker, but more realistic variation of soundness is *soundness* [147]. A tool can be called *soundy*, if it is sound within reasonable practical limitations, including some simplifications. For example, an analysis may only unroll unbounded loops until a certain depth, lose precision during conversion of a program into a formal framework, or use dynamic information to speed up an otherwise sound technique. In fact, the authors of [147] claim that every program analysis tool must make some compromises when applied to a real programming language. For this reason, when we call a tool *sound*, we actually consider it *soundy*, though the formal methods themselves may be strictly sound in an ideal setting.

Similarly, no existing tool does actually have zero false positives – apart from overapproximation, techniques like *blinding* and the internal generation of random secrets (e.g., ephemeral keys) often trip trace comparison-based methods and are not incorporated in formal models. Some dynamic approaches try to support non-determinism by applying statistical tests, but those are still susceptible to insufficient sample sizes and imperfect

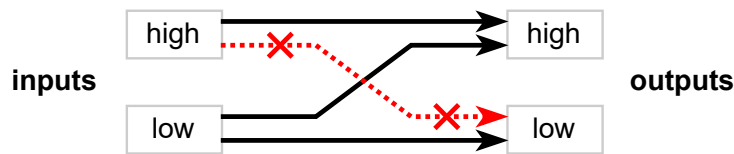


Figure 3.6: Illustration of noninterference. Low-confidentiality inputs may flow into both low- and high-confidentiality outputs. However, high-confidentiality inputs may never end up in low-confidentiality outputs.

leakage models, so not complete either. Thus, we consider completeness only for *deterministic* programs, and evaluate the ability to analyze non-deterministic programs separately.

3.2.1 Static Approaches

Static analysis aims to make general statements about the behavior of the given program, without actually executing it. As these statements typically apply to all possible inputs, static analysis can *prove* the security of the program.

A concept tightly coupled with all static analysis approaches is *noninterference*, which was coined by Goguen and Meseguer [81] to formalize secure information flow. Noninterference requires that two users on a given system with different security levels do not *interfere* with each other, i.e., confidential data does not affect data visible to other users. In the context of language security, noninterference means that given *low* and *high* confidentiality inputs, the resulting low outputs do not depend on the high inputs (Figure 3.6). As this requirement is too strong for most applications (e.g., a ciphertext clearly depends on high confidentiality inputs, but can be safely released), weaker notions were proposed, like delimited release [195], which allows certain variables to be declassified explicitly.

Noninterference can be adapted to verifying constant-time properties [7, 8]: We assume side-channel *observations* (e.g., address traces) as a public output, and then require that any pair of executions which differ only in their secret inputs produce the same observations. To prove this property, various approaches have been explored, which we summarize in the following.

Composition and Reduction to Assertion Safety. In *ct-verif* [7], Almeida et al. construct a product of the verified program with itself, getting two interleaved executions with separate sets of variables (cross product [24]). They assume that all public inputs are identical, and then assert at each position in the execution that the side-channel observations are equal. By reducing the constant-time property to assertion safety, they

can subsequently employ an SMT solver that checks whether the program is safe for all possible secret inputs. In a limited fashion, this approach also supports varying control flow that depends on public outputs, with manual annotations indicating that a given leakage is benign. However, the loss of synchronicity between the two simulated executions may lead to path explosion in the worst case.

A similar path is taken by Athanasiou et al. in SideTrail [19], who use self-composition [25] instead of a cross product, i.e., they do not interleave the two executions. However, they assume a much simpler leakage model, where they only verify whether all possible program paths have the same execution time.

Blazer [14] by Antonopoulos et al. addresses the issue of path explosion, by decomposing the program into *partitions*, where each partition holds the body of a single secret-independent conditional. By proving that all partitions have secret-independent execution time, the same is also proven for the whole program. Note that this is a weaker notion than constant-time, as there may still be memory access trace leakage through balanced branches or secret-dependent array indexes. Finally, Blazer runs on source code, as it is more difficult to identify partitions in binary code, and compiler optimizations may affect the actually observed execution time and thus introduce more leakages.

Type Systems. Type systems have long been prominent approaches for showing noninterference: A variable is assigned both a data type and a *security label*. This label is propagated through the program, similar to static taint tracking, with the compiler verifying that only permitted information flow occurs.

Conceptually, adapting an existing type system for constant-time checking is straightforward. For example, Volpano et al. [229] slightly modify their system from [228] to prevent branching based on secret data *at all*. Agat [2] relaxes this strict requirement by instead enforcing that an observer cannot distinguish which branch was taken, and provide an automated tool that pads branches accordingly.

Barthe et al. [23] take an approach that is more suited to prevent modern side-channel attacks, requiring that secret *high* data is not used in conditionals and for addressing memory. To make the system practically applicable, it is necessary to know which information is accessed in a certain part of a program. They solve this through a conventional points-to analysis, which overapproximates the set of addresses accessed through a given pointer.

A similar approach is outlined by Rodrigues et al. [193]: They exploit a property of the Static Single Assignment (SSA) form, which is often utilized by compilers, allowing them to efficiently capture implicit and explicit information flows and thus determine

secret-dependent accesses in LLVM IR. However, it is unclear how they deal with heap memory and detect which information is referenced by a given pointer.

Abstract Interpretation. Instead of building an accurate formal representation of all instructions a program contains, one can *abstract* those computations to the degree that is needed for side-channel analysis, and then *interpret* the program using those abstractions [58].

In *CacheAudit* [67], Doychev et al. approximate the number of side-channel observations through a cache leakage model. Their abstraction focuses on capturing memory accesses, to build traces of hits and misses as they would occur in practice. By counting the number of such traces, they compute an upper bound of the resulting leakage. If the approximated leakage is zero, this corresponds to a proof that the program is secure. In a follow-up paper [68], they refine their abstraction to also support dynamic memory allocations, tracking the least-significant bits of addresses (which are helpful for distinguishing whether an unknown address points to the same cache line as another address) and detecting intra-cache line leakage.

Blazy et al. [32] assume a classical leakage model, which incorporates secret-dependent branches and memory accesses of any granularity. Similarly to logic-based approaches, they reduce the security of the program to its safety, by letting the program get stuck when encountering a secret-dependent operation. They then track secrets through static tainting based on an abstract interpreter, integrated into the *CompCert* [54] verified compiler platform.

Static tainting based on abstract interpretation is also used by Schaub [198]. Their tool *STAnalyzer* traverses the AST of every function to build a dependency graph, encoding which initial (potentially secret) argument values a given variable depends on. When interpreting an instruction, the algorithm checks whether that instruction could possibly leak a secret value, and marks it accordingly.

In *CacheS* [234], Wang et al. observe that a majority of information flows in a typical cryptographic implementation relates to public values only, and only a small minority of operations depend on secrets. Thus, compared to classic static taint analysis, they reduce the accuracy of tracking public values in exchange for a highly accurate tracking of secret values. After interpreting the program, they feed all identified secret-dependent positions into a constraint solver, which generates example inputs producing the leakages (witnesses). In a rather unusual approach, *CacheS* relies on a reverse engineering tool to lift binary code into an intermediate representation. While this allows the analysis to reason about the security of the compiled code (not the source code like other approaches), it leads to loss of soundness due to inaccuracies in the conversion.

To address the general lack of completeness of abstract interpretation, Chattopadhyay et al. [51] propose to combine it with model checking or symbolic execution, which can verify leakage candidates and thus reduce false positives.

Symbolic Execution. Leakage analysis through symbolic execution [123] is tightly related to formal reduction-based approaches. It reasons about the exact effects of instructions, without making simplifications like type systems or abstract interpretation. In symbolic execution, the program is converted into a number of constraints, which are parameterized by *symbolic inputs*, i.e., variables representing the arguments given to the program. These constraints are then evaluated for satisfiability on-the-fly using a suitable solver. Most symbolic execution-based approaches make use of noninterference-related techniques like self-composition, where the program is simulated for two different secret inputs in order to prove that it behaves the same for all secrets.

In an early work on using symbolic execution for side-channel analysis, Pasareanu et al. [173] take a cost-based approach, that includes execution time, number of heap objects and I/O communication in the leakage model. They estimate the total leakage of a program by counting all possible symbolic execution paths that exhibit different cost, and then computing the Shannon entropy.

Brennan et al. [36] use a similar partitioning technique as Antonopoulos et al. [14]: With their tool *CoCo-Channel*, they decompose the program into branch and loop components, and assign each instruction a certain cost. However, instead of looking at all possible paths, they use taint analysis to find secret-dependent branches, and check whether those expose execution time differences.

CANAL [218] from Sung et al. simulates a cache by instrumenting an LLVM IR program with special store/load calls on each memory access. Additionally, it allows to query auxiliary variables containing the number of cache hits and misses. It then evaluates two executions of the same program with different symbolic inputs (self-composition) and checks whether the computed execution times and hit counts differ. Chattopadhyay et al. [52] follow a similar approach with *CacheFix*, and accompany the vulnerability detection with monitoring that synthesizes patches which introduce dummy cache hits/misses/invalidations at runtime to avert detected leakages. Cache models are refined by Brotzman et al. [39], who introduce optimistic and pessimistic models which abstract away implementation details, reducing analysis complexity at the cost of more false positives.

Disselkoeen et al. [65] verify whether a program is constant-time by propagating secret taint during symbolic execution, and checking whether a tainted value appears in a branch condition or is used as a memory address. They also support verification on protocol-level, catching attacks like *Lucky Thirteen* [5], which are not associated with

CPU-level side-channels. Tainting is also used by Yavuz et al. [261]. As a performance optimization, they ask the user to specify the information flow behavior of subroutines, so those can be omitted during symbolic execution [190].

A generic approach is taken by Daniel et al. [60, 61]. Their tool Binsec/Rel supports both constant-time verification as well as checking for proper secret erasure. While their approach also relies on self-composition for comparing the execution paths of two different secret inputs, it has one key difference: By using *relational symbolic execution* [72], they can merge redundant public paths in both programs (reducing the size of the symbolic formula), and avoid leakage tests for expressions that only involve public variables. This way, they address the performance issues that typically render symbolic execution-based techniques impractical. The good performance also allows avoiding over-approximation, making Binsec/Rel soundy and complete.

3.2.2 Dynamic Approaches

Dynamic analysis tries to reason about the behavior of the program by looking at its execution. This is usually approached by instrumenting the program and collecting one or more *execution traces* for a set of concrete inputs (contrary to static analysis, which reasons about all possible inputs).

Some approaches only use the collected runtime information to improve the performance of static analysis techniques, while others compare execution traces to find differences caused by information leaks (Figure 3.7). Many approaches share a simple idea: If changing a secret input leads to a different execution path, there must be secret-dependent code and thus leakage. In the following, we summarize the dynamic analysis techniques proposed in the literature.

Taint Tracking. `ct-grind` by Langley [133] is a small patch for the popular Valgrind debugging framework. It directly builds upon the `memcheck` tool, which checks whether uninitialized data is used for branch decisions or memory addresses. This can be trivially adapted to leakage analysis by marking all secrets as uninitialized. `memcheck` then proceeds propagating this information during execution, and reports an error as soon as a secret-dependent branch or memory access is detected. Neikes [164] shows the scalability of the method by applying it to the SUPERCOP [223] benchmarking suite, uncovering a number of vulnerabilities.

Irazoqui et al. [114] use taint tracking to find branches and memory accesses that are somehow affected by secret data. As this does not yet necessarily mean that there is leakage, they subsequently collect cache traces for those code locations, by flushing the affected variable from the cache and then measuring every access to it. Finally, they

Trace A		Trace B
read b64d[0x51]	=	read b64d[0x51]
read b64d[0x31]	≠	read b64d[0x32]
read b64d[0x6b]	≠	read b64d[0x6c]
read b64d[0x2f]	≠	read b64d[0x77]
read b64d[0x61]	=	read b64d[0x61]
read b64d[0x47]	=	read b64d[0x47]
read b64d[0x56]	=	read b64d[0x56]
read b64d[0x79]	=	read b64d[0x79]

Figure 3.7: Execution traces for the inputs Q1k/aGVy and Q21waGVy for the leaking Base64 decoding example from Figure 3.1. Different Base64 characters lead to different accessed memory addresses. If a dynamic tool observes such differences between execution traces after changing *only* the secret input, the cause was likely a secret-dependent memory access.

find dependencies between the cache traces and the secret through mutual information analysis.

Statistical Tests. In *dudect* [192], Reparaz et al. use statistical tests to compare the program’s execution time (e.g., measured by counting CPU cycles) for different secret inputs. If the tool identifies timing differences between executions, those indicate the presence of secret-dependent branches. However, the approach allows neither localizing the origin of leakages nor finding smaller leakages involving memory access patterns variations.

Zankl et al. [264] present a leakage test specifically tailored to modular exponentiation. They observe that typical optimizations skip calculations when there are 0-bits in the exponent, which in turn implies that there are more calculations for an exponent with a high amount of 1-bits. This can be turned into a leakage test by correlating the Hamming weight of the exponent with the number of times a given instruction was executed. Both positive and negative correlation indicates leakage.

Trace Comparison. STACCO [255] by Xiao et al. focuses on TLS implementations and Bleichenbacher attacks [33]. They generate pairs of random TLS packets and trace their processing in the respective libraries. The resulting control flow traces are then compared using a standard diff tool. While each small difference points to a potential vulnerability, their vulnerability analyzer allows to reduce granularity of the recorded addresses to cache line or page-level, reflecting the real attacker capabilities and thus removing leakages that are not exploitable.

DATA [240] by Weiser et al. takes a more generic approach. Leakage detection is done in three phases: First, for two sets of fixed and random secret inputs, the tool collects execution traces and then uses a custom diffing algorithm to find a pairwise alignment. In the second phase, detailed execution traces are collected for all instructions involved in trace differences, and transferred into histograms that are subsequently tested for secret-dependent differences using Kuiper's test. Finally, a specific leakage test analyzes how the leakage is correlated with the secret, e.g., through specific bytes or the Hamming weight. Internally generated secret nonces are also supported, by skipping the second phase, extracting them from the ciphertext/signature and then applying the specific leakage test [239]. A drawback of this solution is that the leakage model must be specifically tailored to the analyzed primitive, which involves significant manual intervention.

With Microwalk [243], in contrast to other leakage analysis tools, we went beyond the traditional research prototype and built a robust and extensible code base, aiming for practical usability in development workflows. At its core, Microwalk also relies on comparing execution traces: The first version collected traces for a number of random secret inputs through DBI and used mutual information analysis to find dependencies between the inputs and the observed memory access patterns. We later refined it with linear-time call tree diffing and added further metrics like minimal conditional guessing entropy, allowing the user to accurately localize and quantify the severity of deterministic leakages [247]. Besides x86 binaries, the generic leakage analysis pipeline also supports JavaScript and RISC-V programs [245]. Extension to further languages and platforms only requires a suitable instrumentation engine to generate the necessary traces.

Another tool solely relying on comparing execution traces (and optionally cache traces) is CacheQL [263] by Yuan et al.. Similar to Microwalk, they try to estimate leakage by computing mutual information between the traces. However, they estimate this value through neural networks. To assign the leakage to concrete program points, they use methods from game theory. While the authors state that applying the tool to a set of traces is fast, the neural networks it relies on need to be specifically trained for every implementation, so CacheQL cannot be straightforwardly applied to another library or primitive without very costly retraining.

Fuzzing. `ct-fuzz` [96] by He et al. utilizes an LLVM plugin to materialize a self-composition of the given program, that allows to compare two executions with different secret inputs. Suitable instrumentation tracks the leakage behavior of both instances. They then use a fuzzer to produce pairs of secret inputs until a variation is discovered. A very similar approach is followed by `DiffFuzz` [167] from Nilizadeh et al., though they target Java bytecode. What both fuzzer-based approaches have in common is that they often find a vulnerability within the first few seconds. If no vulnerability is found in that time, a longer analysis often does not lead to more results.

Dynamic Symbolic Execution. Contrary to classic symbolic execution, *dynamic symbolic execution* [80] (also called *concolic execution*) applies the symbolic formula to a concrete execution trace. This drastically reduces the search space and allows quickly finding solutions which lead to yet unexplored program paths.

Wang et al. [235] build such a symbolic expression along a concrete execution path, capturing the program state and path conditions. For any given memory access, they then ask the solver to find two secret inputs that access a different address. To reduce expensive queries to the solver, they use taint tracking to preselect potentially vulnerable instructions and apply domain knowledge.

Bao et al. [21] improve upon this idea by also including control flow variation in their expressions. Additionally, instead of querying a solver, they find that random sampling is sufficient to find a pair of inputs which produce a leakage. Based on this observation, their tool Abacus uses a variant of Monte Carlo simulation to estimate the number of inputs that can be distinguished using the given leakage. That number can then immediately be used to calculate the amount of leaked input bits.

Refinement Types. *Refinement types* [74] are an extension of type systems, where types can carry an additional *predicate* that specifies properties of the typed value. Jiang et al. [119] use this for bit-accurate tracking of secret values, allowing their type system to avoid false positives such as secret-dependent operations that involve correctly masked or randomized values. Thus, in contrast to regular type systems, their tool CaType also supports blinding. The remainder of their approach is similar to dynamic symbolic execution: They first ask the user to mark secrets and random values, collect bit-level taint for a concrete execution, and then verify adherence to their type system for the execution trace.

3.2.3 Comparison

To learn the capabilities of the many proposed static and dynamic approaches, we compare them across various categories. In the following, we briefly describe each category and discuss observations. The results are summarized in Table 3.1.



Analysis Level. The robustness of the analysis result depends on the representation of the analyzed code. If only the source code is considered, compiler optimizations can remove secret-dependent operations or even introduce them [61, 210]. Similarly, IR-based analysis may miss target-dependent low-level optimizations. For this reason, many contemporary tools target binary code, as it is executed natively on the machine and thus accurately reflects the leakage behavior. Another noteworthy approach are




certifying compilers, which validate a security property during compilation, but those suffer from similar issues as source code/IR analysis if the verification is not done on binary level. Finally, some tools rely on lifted IR, which is IR generated from a binary. During the lifting process some information gets lost, so it also suffers from inaccuracies, though it is arguably closer to the optimized binary than the compiler IR.

Security Guarantee. When evaluating the guarantees of each tool, one must consider their underlying leakage model and the resulting security level. For example, tools that only care for secret-independent execution time (*time-invar.*) or constant control flow (*const. ctrl.*) usually leave the code vulnerable to memory access trace-based attacks (e.g., cache attacks). This is averted by approaches that employ a formal cache model (*cache-invar.*), but those models do not fully match reality and miss fine-grained leakages. Most recent approaches thus adopt a generic constant-time policy (*constant-time*), which can protect against all kinds of trace-based side-channel leakages.

Sound(i)ness and Completeness. Generally, static approaches that consider some variant of noninterference allow to prove the absence of leakage, i.e., they are formally sound. This is due to considering all possible inputs, without making simplifications that omit secret-dependent information flows. In practice, peculiarities of the programming languages and proof engines still impose some restrictions, so the actual implementations of those approaches can be rather called *soundy* [147]. Naturally, dynamic tools are unsound, as they only consider a subset of all possible program paths and thus may miss leakages.

Achieving completeness (no false positives) statically is much more difficult, as the formal abstraction must not allow any over-approximation that turns a public-only information flow into one that looks like it may contain secrets. While the reduction-based tools promise completeness, this is paid with a high analysis time, making those tools hardly scalable for applications exceeding symmetric cryptography. The only other tool that provides some notion of completeness is Binsec/Rel, which is possible due to their very efficient approach at symbolic execution. On the other hand, dynamic approaches can be complete, if they do not make overapproximations that introduce false positives. For example, in deterministic programs, differences between two execution traces always originate from secret-dependent operations, so any difference is a true positive.

Scalability. For publications which contained a performance evaluation, we assigned one of four classes to illustrate their expected real-world scalability (for sensible input parameters). The lowest class, signaled by , indicates that the tool needs a long analysis time even for simple targets (or does not terminate at all in some cases), and thus is hardly usable in practice. Tools marked with  are capable of analyzing most implementations, but still take a significant analysis time or terminate without conclusion eventually.

Practically relevant are  and , which support even large asymmetric targets, though  may still have few cases with long or non-terminating analyses.

Dynamic tools generally perform better than static ones, as they often rely on native execution and only consider a concrete subset of possible inputs, avoiding costly SMT solver calls. However, FlowTracker, CacheS and Binsec/Rel demonstrate that high performance is also achievable with static approaches, by excluding irrelevant inputs from the analysis and carefully minimizing the number and size of constraints.

Capabilities. To date, `ct-verif` is the only tool that allows ignoring operations that depend on public outputs. For example, once encryption is done, any side-channel observations on the ciphertext do not increase the attacker’s knowledge, as they learn the ciphertext anyway. But it is quite difficult to automatically detect at which point exactly an encryption ended. Even `ct-verif` can only achieve this through a manual annotation by the user, after which the tool ignores all dependent operations. Being able to avoid false positives from insensitive operations (like encoding a ciphertext) is a useful property, which deserves more attention in the future.

Dealing with non-input-randomness like ephemeral keys is another difficult area. Information flow-based approaches need to be modified to also incorporate these internally generated secrets, and trace-based approaches need to rely on statistical tests to detect non-determinism. At the time of writing, only DATA was used to systematically analyze internal secrets [239], and even there they relied on a customized leakage model and lots of manual analysis. Thus, automating this kind of case remains an unsolved problem.

Blinding (i.e., transient randomization during computation) is better supported. Several dynamic tools are able to eliminate independent randomness through statistical tests, at the cost of sacrificing completeness. Deterministic information flow and trace comparison-based approaches mistake randomized for secret-dependent operations, i.e., report false positives. It thus remains an open question whether a tool can efficiently analyze blinded implementations without relying on statistical methods.

Output. In order to address a reported vulnerability, the developer needs to be given context on its cause and its severity [118, 247]. Many tools are able to tell the exact code location of a leakage and often provide a witness (i.e., secret input) triggering it. Far fewer tools offer a metric that quantifies the severity, which can be useful for triaging. The available metrics are widely different in their accuracy and validity. The metric with the arguably closest estimation of leaked bits is the one used by Abacus, which computes the number of inputs triggering secret-dependent behavior.

Table 3.1: Comparison of software side-channel leakage analysis tools. We list the analysis level, security guarantee and whether the tool is sound (*Sound*) and complete (*Compl*). We also denote support for public outputs (*PubOut*), internally generated secrets (*IntSec*) and blinded implementations (*Blind*). Finally, we checked whether the tools can localize (*Loc*) and quantify (*Quan*) leakage, and provide a suitable witness (*Witn*).

Tool	Approach	Level	Security	Properties		Scal	Capabilities			Output		
				Sound	Compl		PubOut	IntSec	Blind	Loc	Quan	Witn
[7] ct-verif	reduction ; product program	IR	constant-time									
[19] SideTrail	reduction ; self-composition	IR	time-invar.									
[14] Blazer	reduction ; decomposition	source	time-invar.									
[229] Volpano et al.	type system	-	const. ctrl.									
[2] Agat	type system	-	time-invar.									
[23] VirtualCert	type system	cert	constant-time									
[193] FlowTracker	type system	IR	constant-time									
[67] CacheAudit	abstract interpretation ; cache model	binary	cache-invar.									
[68] CacheAudit2	abstract interpretation ; cache model	binary	constant-time									
[32] Blazy et al.	abstract interpretation ; static tainting	cert	constant-time									
[198] STAnalyzer	abstract interpretation ; static tainting	source	constant-time									
[234] CacheS	abstract interpretation ; static tainting	lifted IR	constant-time									
[218] CANAL	symbolic execution ; self-comp., cache model	IR	time-invar.									
[52] CacheFix	symbolic execution ; self-comp., cache model	source	cache-invar.									
[36] CoCo-Channel	symbolic execution ; decomposition	source	time-invar.									
[39] CaSym	symbolic execution ; self-comp., cache model	IR	cache-invar.									
[65] PitchFork	symbolic execution ; static tainting	IR	constant-time									
[60] Binsec/Rel	symbolic execution ; self-comp., relational SE	lifted IR	constant-time									
[261] ENCIDER	symbolic execution ; self-comp., static tainting	IR	constant-time									
[173] Pasareanu et al.	symbolic execution ; model counting	IR	time-invar.									
[133] ct-grind	taint tracking ; tainted data-dependent accesses	binary	constant-time									
[114] Irazoqui et al.	taint tracking ; real cache attack traces	IR	cache-invar.									
[264] Zankl et al.	statistical test ; modexp bit correlation	binary	time-invar.									
[192] dudect	statistical test ; resource usage measurement	binary	time-invar.									
[255] STACCO	trace diffing ; pairwise, TLS-only, basic block traces	binary	const. ctrl.									
[240] DATA	trace diffing ; pairwise, stat. test, leakage model	binary	constant-time									
[243] Microwalk	trace diffing ; N-way, linear time	binary	constant-time									
[263] CacheQL	trace diffing ; neural networks, game theory	binary	constant-time									
[96] ct-fuzz	fuzzing ; self-composition	binary	constant-time									
[167] DifFuzz	fuzzing ; self-composition	IR	constant-time									
[235] CacheD	dyn. symbolic execution cache variance	binary	cache-invar.									
[21] Abacus	dyn. symbolic execution cache variance, sampling	binary	cache-invar.									
[119] CaType	refinement types ; cache variance	binary	cache-invar.									

3.2.4 Discussion

Application in Practice. Jancar et al. [118] conducted a study among cryptographic library developers, where they asked them which kinds of tools they are using for verification and what properties a tool should have. First of all, they noted that most available tool implementations were unmaintained research prototypes with often insufficient language support, which are hardly usable in everyday programming. Other points of criticism were high resource consumption and usability issues like difficult setup and the need for manual code annotations (which need maintenance as well). For the latter reason, the developers mostly preferred tools based on dynamic instrumentation, and perceived formal analysis tools as requiring too much effort. Finally, purely statistical tools like *dudect* were disliked due to their susceptibility to external factors like noisy shared hosts used for CI, and the fact that their are neither sound nor complete.

With our work *Microwalk-CI* [247], we took up these results and showed how the *Microwalk* framework can be adapted to use in a CI workflow. Additionally, we addressed several usability issues with a new low-maintenance analysis template and a fast analysis algorithm that works in a constrained CI environment. We continue to maintain the tool and, at the time of writing, work on integrating it into an actual library development workflow.

Test Case Generation. One issue of dynamic approaches is the necessity of a set of secret inputs (test cases) that yield good coverage of the analyzed code. Most tools rely on purely random test cases, which tend to work well with cryptographic implementations due to their high degree of diffusion (i.e., changing a single bit is likely to trigger varying behavior if it is present). It was thus found that a small set of random inputs already yields good coverage [240, 247]. However, this method may still miss code paths that trigger in very rare situations.

One approach to address that is fuzzing (as employed by *ct-fuzz* and *DifFuzz*). Basu et al. [27] propose a test case generation method based on simulated annealing, aiming to maximize the number of observed cache misses, and show that it performs better than the AFL [135] and Radamsa [97] fuzzers.

3.2.5 Other Vulnerability Types

The aforementioned methods can also be adapted for finding other vulnerability types.

Transient Execution. Transient execution attacks like *Spectre* [125] exploit invalid code paths that can be triggered by speculative execution, accessing data that should not be accessible architecturally.

To detect potentially vulnerable code gadgets, Microsoft quickly integrated a number of heuristics into the Visual C++ compiler [153].

Guarnieri et al. [91] introduce *speculative noninterference* (SNI), an adaption of the non-interference property that also considers *mispredicted* execution paths until a certain depth. Their Spectector algorithm then uses symbolic execution to prove the property. Fabian et al. [70] extend Spectector with more generic semantics which can also catch other types of *Spectre* vulnerabilities. An alternative notion introduced by Brotzman et al. [40] is *speculation-aware noninterference* (SANI), which is more generic than SNI, as it does not assume that the program is side-channel free. They aim to find more cases by also considering leakage from the occurrence of speculation *itself*, instead of only focusing on memory index-based leakage.

Other work uses symbolic execution as well: One can model speculative behavior by extending conventional cache leakage models with a simulation of branch misprediction (i.e., visiting both code paths for a limited number of instructions). SpecuSym [93] by Guo et al. and KLEESpectre [232] by Wang et al. take this approach.

As a fully dynamic method, oo7 by Wang et al. [233] uses binary taint analysis to identify branch instructions which depend on attacker inputs, and finds memory access instructions reachable from them. Finally, the tool checks whether there are further memory access instructions depending on the values of the former, which constitutes typical *Spectre-PHT* gadgets. Their work additionally hardens the code by inserting fences in the identified gadgets.

Spectre vulnerabilities can also be found through fuzzing: SpecFuzz by Oleksenko et al. [169] simulates misprediction at branches and executes otherwise invalid code paths, rolling back execution state afterward. By applying AddressSanitizer [1] to the executed code, they can find transient out-of-bounds memory accesses.

Tol et al. [221] train a classifier to detect many different types of *Spectre* gadgets in code.

Memory Content-Based Side-Channels. Contrary to cache and transient execution attacks, which target address-based access patterns, ciphertext side-channel attacks [136, 138] and observation of silent stores [227] or data memory-dependent prefetching (DMP)[226] target the secret values themselves.

CipherH [63] by Deng et al. uses dynamic taint analysis and static symbolic execution to detect whether consecutive writes to the same address lead to ciphertext collisions.

While they aim to find ciphertext side-channels, their method only considers consecutive writes without any other accesses to the same address in between. So while the tool is not able to detect all cases of ciphertext side-channels, it is suitable to find silent store leakages.

In our work *Cipherfix* [244], we take a less precise approach at finding ciphertext side-channel vulnerabilities: We use taint analysis to track secrets through the program and identify which variables and instructions are affected by them. We successfully use this analysis for automated and reliable code hardening, though it is a clear overapproximation which would be unsuitable for a pure detection tool.

3.3 Side-Channel Countermeasures

When dealing with side-channel vulnerabilities, two approaches at countermeasures are possible: Modifying the hardware to remove the cause of the side-channel, or adapting the software to detect or mitigate the leakages caused by the side-channel. In the following, we discuss the state of the art for both hardware and software solutions with respect to various side-channel leakages.

3.3.1 Cache Attacks

A large class of attacks is based on caching, i.e., the fact that observable access behavior differs between seldom used data and data that was accessed recently. While cache attacks mostly refer to the “classic” L1 to LLC memory caches, there are more caching layers in modern processors, like the translation look-aside buffer (TLB) for page table mappings. However, the memory caches have received lots of scrutiny recent in attack research, as they feature high temporal and spatial accuracy, and attacks against LLC even work cross-core. Thus, in the following, we focus on countermeasures for memory cache attacks, though the main ideas also apply to other microarchitectural components.

The countermeasures against cache attacks fall into three main classes: Secure cache designs, software hardening through code linearization and randomization, and dynamic attack detection.

Secure Cache Architectures. Caches come with significant advantages, making them an essential part of modern microarchitectures. Most notably, they are usually fully transparent, i.e., no software support is needed to benefit from them. This transparency also means that CPU vendors can freely adjust various parameters like set associativity and replacement policies, improving performance without having to adapt existing

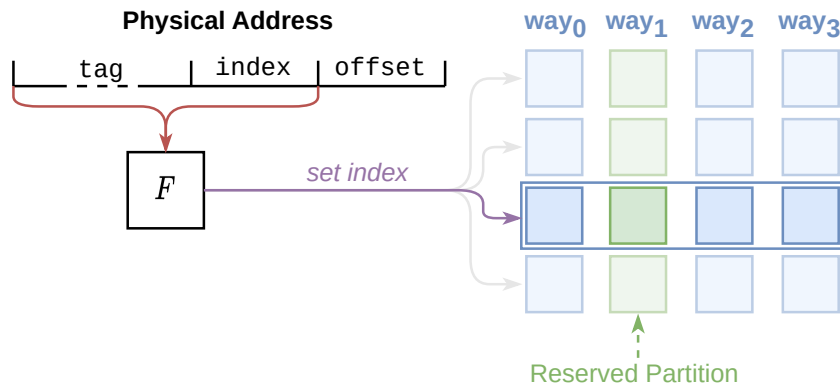


Figure 3.8: A basic way-partitioned set-associative cache with 4 ways, where way₁ is fully reserved for a sensitive process. Cache lines of other processes cannot be stored there, so their available cache capacity is reduced.

software. Simply disabling caches due to security concerns is not viable, as this would lead to a major loss of performance. Similarly, secure cache designs should not introduce a notable overhead, as this would run counter to the purpose of the cache.

As for other microarchitectural components, the main security issue of caches is that they are shared between processes and behave predictably. For example, an attacker process can manipulate the (shared) cache state and then observe how that state is affected by a victim process.

A straightforward solution to this issue is *cache partitioning*, where processes are prevented from evicting each other's cache lines [172, 187, 196, 238, 256]. One simple method is way partitioning, where parts of the cache are reserved exclusively for a certain code section (Figure 3.8). Similarly, in a method called *page coloring* [220], the operating system picks physical pages which map to a known cache set, allowing to “reserve” a given cache set for the private data of a process. However, this assumes a known mapping from address to set indexes, which is not always available for modern processors. The Sanctum TEE [57] uses page coloring-based partitioning. Such static methods are generally considered inflexible and wasteful, and there have been many proposals on dynamic approaches which make better use of the limited resources.

Wang et al. [238] propose PLcache, which comes with new instructions for *locking* selected memory areas in the cache. A process cannot evict a locked cache line from another process, and locked cache lines can only evict each other if they are from the same process. The approach relies on the operating system to ensure that a single process does not lock too many resources. The cache partitioning system Vantage from Sanchez et al. [196] takes a different route: They divide the cache into unmanaged and managed regions, where the latter takes the majority of ways and is divided into a fixed amount of partitions.

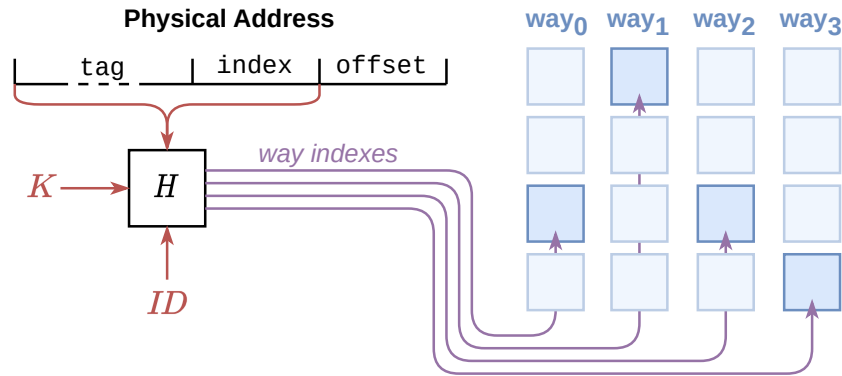


Figure 3.9: A ScatterCache [241] with 4 ways. The address-to-set mapping function F is replaced by a hash (or symmetric encryption) function H , which takes the physical address, a hardware-generated key and an operating system-determined ID. Instead of producing a single index for a fixed cache set, the function yields separate indexes for each cache way, greatly increasing the number of possible cache sets.

Partitions may briefly overflow into the unmanaged region. Their management strategy keeps track of insertion and eviction rates of each partition, and balances them in order to keep them to their desired sizes. However, this means that there is still interference between the partitions of different processes.

Contrary to cache partitioning, *set randomization* tries to thwart the attacker’s ability to build eviction sets and infer secrets from cache state, by breaking the formerly fixed mapping of physical addresses to cache sets [35, 185, 238]. RPcache by Wang et al. [238] adds a key to the set index computation function to generate a unique permuted cache set mapping for the current process. However, this requires frequent rekeying (which amounts to a cache flush), as an attacker may prepare fixed eviction sets for every cache set and then only needs to figure out which eviction set is required to evict the target address.

Pre-computed eviction sets can be avoided through *full randomization* with skewed caches [186, 241]: ScatterCache by Werner et al. [241] does not only permute the mapping of addresses to set indexes, but also the sets *themselves* are comprised of a random selection of ways, leading to an amount of possible cache sets that is exponential in the number and size of ways (Figure 3.9). The permutation function of ScatterCache is keyed both with a random hardware-managed key and a software-controlled ID. This way, the attacker can no longer construct reliable eviction sets, as the allocation of ways to form a cache set is fully random. The attacker’s context has a different ID than the victim, so they also get an entirely different cache layout.

Nevertheless, it was shown that such designs can be attacked. For example, in a proba-

```
const unsigned char b64d[256] = { ... };

int decode_base64_char(char c) {
    return b64d[c];
}
```

```
const unsigned char b64d[256] = { ... };

int decode_base64_char_ct(char c) {
    int bits = 0;
    for(int i = 0; i < 256; i++) {
        int mask = ~((i == c) - 1);
        bits |= b64d[i] & mask;
    }

    return bits;
}
```

Figure 3.10: Linearization example. The left box shows the leaking Base64 decoder from Figure 3.1, the right box a simple (and inefficient) linearized version of the same code. Instead of accessing only the desired table index (and thus leaking the value of *c*), the code accesses *every* table index, but masks out all values except for the requested one. The linearized code exhibits no secret-dependent branches or memory accesses. However, it is possible that the compiler’s optimization passes detect the masking pattern and nevertheless emit the same machine code as for the original leaking lookup.

bilistic version of *Prime+Probe*, called *Prime+Prune+Probe* [180], the attacker gradually builds an eviction set that does not have any self evictions (prune). The eviction set size is increased until it manages to evict the target address. A follow-up proposal to ScatterCache, SassCache [79], addresses this problem by introducing a second mapping function that reduces the set of cache lines a given process can reach. This way, the attacker only has a partial overlap with the victim, drastically reducing their ability to build a full eviction set, as some victim cache lines are not visible to them at all.

Yet, while skewed caches seem promising due to their security properties and generally low overheads in their evaluations, Song et al. [214] note that skewed-cache designs may perform poorly in highly parallelized conditions, and recommend using randomized set-associative caches with automatic attack detection instead.

In summary, there are many different proposals for hardware-based secure caches, where each comes with its own advantages and drawbacks. Despite the long line of research, the major CPU vendors have yet to adopt one of the concepts in their processors. Randomized designs were shown to be vulnerable to probabilistic attacks, so even if they become widely available eventually, the user may need to harden software to mitigate the remaining risk. Thus, for the foreseeable future, software-level solutions to cache attacks remain necessary.

Code Linearization and Randomization. The primary software defense against cache attacks is *linearization*, i.e., removal of input-dependent runtime behavior (Figure 3.10). The resulting *constant-time code* exhibits the same sequence of executed instructions and accessed memory addresses for every secret input. An alternative approach is

randomization through means like *blinding*, where a random value is introduced into the computation at the beginning and removed from the result afterward, to decouple the observed execution trace and the secret inputs. However, randomization only works with selected applications. Approaches and tools for checking the side-channel security of linearized and randomized code are discussed in-depth in Section 3.2.

Hardening software against cache attacks is a mostly manual task, which requires experience, reduces code readability, and often involves fighting with the compiler to keep it from re-introducing leakages through optimizations [61, 210]. To address this, several methods for aiding and even automating code linearization were proposed.

The first category are domain-specific programming languages (DSLs), which are specifically designed to enable constant-time programming and ease static analysis. FaCT by Cauligi et al. [50] is a C-like language with a built-in type system that allows marking data as public and secret. The programmer can write readable code without constant-time idioms, as the compiler takes care of the subsequent constant-time transformation and formal verification of the security properties. FaCT was later extended to also support speculative load hardening [205].

Other proposals focus on rewriting existing C code instead of translating all cryptographic code into a dedicated programming language, which avoids the compatibility, community support and maintenance issues that come with a DSL. SC-Eliminator by Wu et al. [254] *standardizes* control flow by replacing secret-dependent *break* and *continue* statements by *if* conditions and rewriting memory accesses to conditional ones. Lookup tables are prefetched into the cache to ensure equal execution time. However, an attacker may work around this. Additionally, the method fully unrolls loops, so their bounds must be known at compile time. Soares et al. [212, 213] use a similar approach called *partial control-flow linearization*, which was originally designed for enabling program vectorization. Their tool Lif also supports unbounded loops, though loops whose iteration count is secret-dependent are turned into infinite loops, reducing practicality of the method. To solve the loop bound problem, Constantine by Borrello et al. [34] takes a hybrid approach: Control flow is linearized using masked decoy paths, and secret-dependent memory accesses are replaced by *oblivious* operations that always touch every memory location that may be accessed by that instruction. Secret-dependent instructions are identified through dynamic taint analysis, so the security of the resulting program depends on the achieved coverage. Finally, to also support secret-dependent loops, Constantine has a runtime component that adjusts loop iteration counts dynamically.

Finally, side-channel security can be achieved through *oblivious RAM* (ORAM). The most simple ORAM is a linear array that is fully scanned on each access. More advanced schemes like Path ORAM [217] use trees to reduce the amount of accessed data. However,

they are mostly tailored towards a client/server setting with large databases and have a lot of overhead when implemented to work with small amounts of data in a constant-time fashion [191, 246]. *Raccoon* by Rane et al. [191] uses static taint analysis to find secret-dependent branches, and subsequently forces control flow to execute both paths. Each path is run as a transaction, that is either committed or discarded after execution. Memory accesses in the real and decoy paths go to an ORAM, so the attacker cannot distinguish whether the program just executed a real or a dummy access. Due to the cost of ORAM, *Raccoon* has a much higher overhead than tools like *Constantine*. However, for threat models beyond the classic cache side-channel attacker who wants to extract secret data, ORAM-based approaches yield good results. For example, in TEEs, the owner may not only want to hide their secret data, but also the code itself, which is not accomplished through traditional linearization. *Obfuscuro* by Ahmad et al. [3] protects code by splitting it into blocks, which are stored in a dedicated ORAM. To execute a code block, it is copied into a *scratchpad*, so the attacker always observes the same memory addresses. The same applies to the accessed data blocks. Our work *Obelix* [246] further highlights the potential of ORAM by extending *Obfuscuro* with support for strong TEE-specific attacker models like single-stepping and ciphertext side-channels.

Attack Detection. Another line of research is dynamic detection and mitigation of attacks. Instead of hardening software or hardware to prevent cache attacks *passively*, the system actively monitors the cache state to identify malicious processes and contain attacks. This leads to an arms race between new detection methods and more stealthy attack techniques evading them [37].

Most detection methods rely on hardware performance counters (HPCs) [129]. HPCs allow measuring microarchitectural behavior like execution unit usage and counts of memory accesses and hits/misses in the various cache layers. HPC-based detection assumes that cache side-channel attacks cause abnormal cache usage patterns which are then reflected in the measurements. The various approaches differ in their choice of HPCs to monitor, privilege level of the detection, the sampling rates, and the classification method for the sampled performance values. Common issues and trade-offs of HPC-based side-channel detection are accuracy (i.e., false positive rate), performance overhead (directly tied to the sampling rate), delay between attack and detection, and an incomplete threat model [129].

Another related approach for detecting cache attacks is analyzing the execution time of the victim process, where noticeable differences may be caused by an attacker [38].

3.3.2 Single-Stepping

The most effective software-side countermeasure against single-stepping attacks is constant-time code, which is free from data-dependent branches and memory accesses. Weaker notions which, for example, permit intra-cache line differences, are insecure, as single-stepping allows very precise counting and measuring of instructions [44, 159, 178, 208]. Another approach is oblivious execution, which can also protect the code itself from leaking to the attacker, though at a high performance cost. The code blocks need to be sufficiently uniform in order to be indistinguishable by a single-stepping attacker [246].

Alternatively, the enclave can monitor itself to detect attacks and deploy appropriate countermeasures at runtime. Cui et al. [59] propose a compiler modification *QuanShield* that generates Intel SGX enclaves which self-destruct after receiving an interrupt. For that, they store crucial runtime information (e.g., base pointers) in a second stack, which resides in the state save area (SSA). After an interrupt, the current enclave state is written into the SSA, overwriting the pointers in the second stack with invalid data. When the enclave resumes execution, it soon crashes due to an illegal access. This approach is quite effective, but requires the operating system to put the respective CPU core into a tickless mode (i.e., no scheduler interrupts).

A less intrusive approach is monitoring the number of asynchronous exits and deploying mitigations when there is suspicious activity. For example, *Varys* by Oleksenko et al. [168] starts a second enclave thread on the sibling logical core, which watches the SSA for changes. They additionally prevent cache attacks by filling the cache with dummy data on enclave exits.

Other tools use transactional memory to suppress single-stepping and enforce execution of instruction sequences [132, 204]. However, Intel TSX is known to come with its own vulnerabilities, and was therefore disabled in many processors via microcode updates [107].

AEX-Notify from Constable et al. [55] takes a combined hardware/software approach which introduces a new control bit, allowing the enclave to return to a dedicated entrypoint after an interrupt. That entrypoint atomically prefetches data operands of the next application instruction, so it is executed quickly after the enclave resumes. This way, the attacker is unable to reliably place an interrupt between the entrypoint and the resumed instruction. *AEX-Notify* was deployed as a microcode and runtime update by Intel, making it also available on many legacy systems.

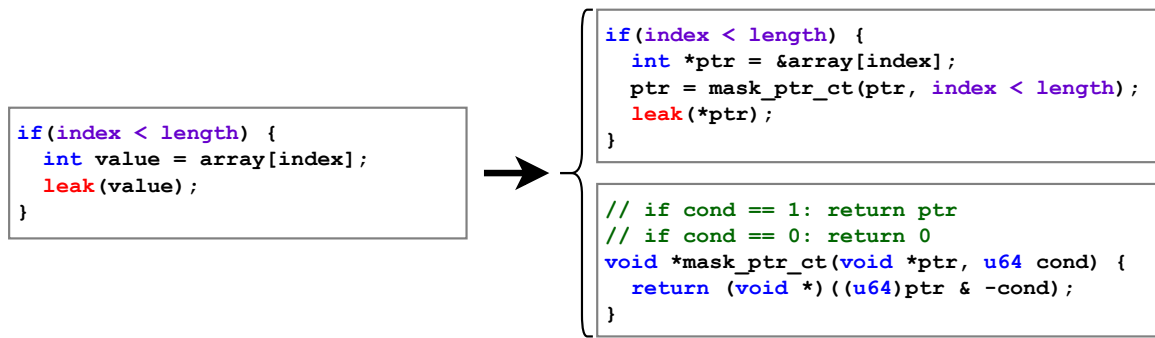


Figure 3.11: Example for speculative load hardening [49]. The left side shows a simple bounds check that is vulnerable to *Spectre-PHT*. The condition may be mispredicted during speculative execution, leading to an access of an invalid array index. The read value is leaked subsequently. With SLH, the pointer to the targeted memory address is masked first, where the mask depends on the condition. This introduces a data dependency on the condition. If the branch is predicted correctly, the access goes through; if the branch is mispredicted, transient execution will use a null pointer.

3.3.3 Transient Execution

Most transient execution attacks fall into two major categories: Leakages caused by variants of speculative execution (*Spectre* [125]) and vulnerabilities from out-of-order execution and delayed exception handling (*Meltdown* [144]). While often mentioned together, the attack classes are very different and thus need to be mitigated separately.

Speculative Execution. *Spectre*-style attacks target a problem that is inherent to any implementation of speculative execution: The fact that code is executed transiently which is not *meant* to be executed, accessing data which is otherwise protected. In CPUs, this leads to side-effects like transiently accessed data being loaded into the cache, which can be observed by the attacker. There are different countermeasures suggested for each *Spectre* variant.

As a first measure, several browser vendors reduced the resolution of available timers in order to prevent attackers from using the cache covert channel [95, 231]. However, this was shown to be insufficient [248]. Chrome uses site isolation by default, which stores sensitive data in an own virtual address space that should not be accessible by attacker-induced speculative execution [230].

For generic software, a straightforward solution to mitigating several *Spectre* variants is serialization, i.e., inserting serializing instructions like `lfence` before/after potentially vulnerable branches [103, 125]. This way, speculation is fully prevented, but at a high performance cost. To allow more targeted application of serialization, static analysis approaches for finding vulnerable gadgets were proposed (see Section 3.2.5).

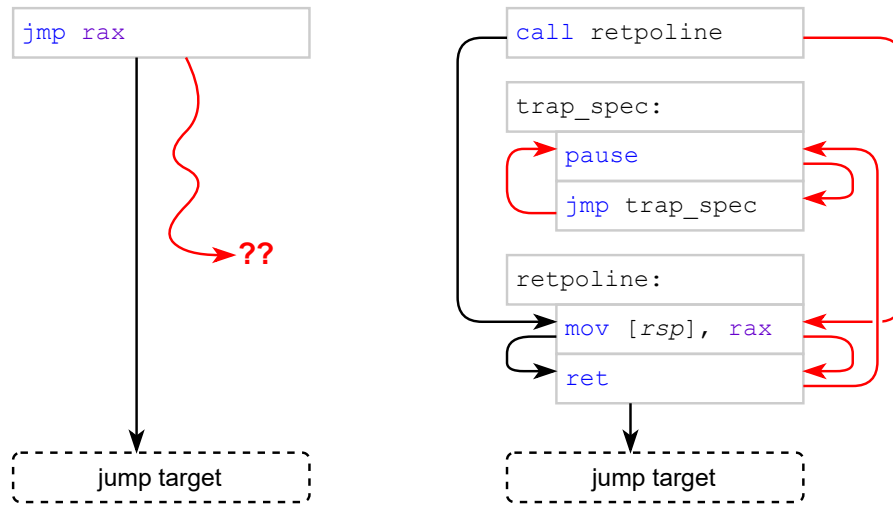


Figure 3.12: Retpoline illustration. The original program on the left side consists of an indirect jump to the address stored in `rax`. Architectural execution (black arrows) jumps to the desired target. However, speculative execution (red arrows) relies on the BTB to predict the address, ending up in an attacker-determined location. The retpoline on the right side leads both architectural and speculative execution into a simple dummy function which overwrites the return address on the stack with the value in `rax` and then returns to that address. However, while architectural execution continues with the jump target, speculative execution relies on the RSB for determining the return address, which is not updated by the stack overwrite. Thus, speculative execution returns to the old return address and gets stuck in the `trap_spec` loop.

Another more efficient software defense against *Spectre-PHT* is *speculative load hardening* (SLH) [49, 265] (Figure 3.11). The key observation underlying SLH is that in most cases data is exfiltrated through a data-dependent memory access, which results in an observable modification of the cache state. SLH introduces a branchless predicate that checks the original branch condition and generates a mask, which is applied to the pointer. If the condition is mispredicted, the pointer becomes invalid, and the access is prevented. Another variant of SLH protects the loaded *value* instead, preventing the attacker from leaking it through another covert channel. SLH is supported both by the LLVM and the GCC projects. SLH in its original form does only protect against *Spectre-PHT* with a cache covert channel. Proposed extensions of SLH with higher guarantees are *strong SLH* [174] and *ultimate SLH* [265].

Spectre-BTB was initially prevented with so-called *retpolines* [110]. A retpoline is a special code gadget that initiates a function call and then replaces the return address on the stack by the branch target. Speculative execution reads the stale return address from the RSB, leaving it stuck in a loop until the misprediction is detected (Figure 3.12). However, in *Retbleed*, Wikner et al. [249] showed that the RSB may underflow into the BTB in some cases. To mitigate that attack, AMD proposed to replace all returns in the kernel by jumps

to a single fully protected return thunk [249].

There are also hardware/microcode mitigations available for the aforementioned attacks. Initially, Intel proposed and released *Indirect Branch Restricted Speculation* (IBRS) [112]. The kernel can prevent trained branch targets from user processes from being applied in privileged mode by enabling IBRS during the user/kernel context switch. However, IBRS is not effective if it is still enabled from an earlier context switch, so it must be disabled when leaving privileged mode. As writing to the corresponding MSR on every user/kernel context switch is quite expensive, Intel later introduced enhanced IBRS (eIBRS). eIBRS only needs to be activated once and effectively isolates branch target prediction of user and kernel mode. IBRS is complemented by *Single Thread Indirect Branch Predictors* (STIBP) and *Indirect Branch Predictor Barrier* (IBPB), which isolate hardware threads and processes running on the same logical core, respectively.

Besides the available countermeasures, researchers have proposed many more methods on software and hardware level. For example, Blade by Vassena et al. [224] uses a type system to detect speculative data flows from secrets to the cache. These data flows are then suppressed using serializing instructions. SpecShield by Barber et al. [22] is a pure hardware solution that prevents usage of data for leaking operations like loads by delaying them until speculative execution was resolved. SpectreGuard by Fustos et al. [75] takes a similar approach, but allows the programmer to annotate sensitive memory regions. ConTEXT by Schwarz et al. [200] does hardware-level taint tracking by adding a bit to each page table entry, register and status flag, indicating that the element holds secret data. This bit is checked during transient execution, and the hardware ensures that the secret never leaks into the microarchitectural state. Software then needs to correctly classify memory locations as secret and not secret. With ProSpeCT, Daniel et al. [62] propose a formal processor model to prove the security of such approaches. They show that the protections in ConTEXT can be weakened to allow some secret-dependent operations speculatively without compromising on security.

Out-of-order Execution and Exception Handling. Contrary to speculative execution leakages, which trick a process (or the kernel) into leaking its own data and which are limited to the current execution context, *Meltdown*-style attacks have generally no such restrictions. Instead, attacks like *Meltdown* [144], *Foreshadow* [42] and *ZombieLoad* [201] sample data from various CPU buffers independent from the attacker's privilege level. As these attacks are often tightly linked to hardware peculiarities which cannot be controlled by software, they can only be properly fixed by the CPU vendors. In the following, we discuss a few software-level mitigations and proposals to harden systems.

The original *Meltdown* attack exploits the fact that modern operating systems use the upper half of the virtual address space for kernel memory and keep it mapped even in

user mode. User space processes are kept from accessing such kernel pages through a supervisor bit in the respective page table entries. Out-of-order execution may then use these existing mappings to read kernel data and leak it via the cache, before the access violation is detected and an exception is raised. Until Intel introduced suitable hardware patches [102], operating system vendors adopted a countermeasure called *kernel page table isolation* (KPTI, originally proposed as KAISER by Gruss et al. [86]). KPTI removes the kernel mapping during user mode, except for a few necessary components like the interrupt handlers.

Load value injection [43] (LVI) allows to inject data into various CPU buffers which is then used transiently by load instructions, even in secure contexts like SGX. To defend against LVI until hardware fixes become available, Intel and the authors of the attack paper recommended inserting `lfence` instructions after each vulnerable load. This leads to a considerable slowdown, and we found that it greatly improves accuracy of other attacks [208].

While not a countermeasure, *Unique Program Execution Checking* (UPEC) by Fadiheh et al. [71] is a formal verification technique for hardware designs that allows to prove the absence of transient execution side-channels. The method is able to detect *Meltdown*-like vulnerabilities without assuming prior knowledge thereof, and the authors recommend to integrate such formal checks into the release process of new CPUs.

Finally, many (but not all) transient attacks can be averted by disabling SMT, as they are caused by microarchitectural resources shared between victim and attacker thread. For example, Intel recommended disabling SMT for SGX enclaves to mitigate the Foreshadow attack [109]. However, this also comes with a performance penalty for the entire system.

3.3.4 Value-Based Leakages

Constant-time code and serialization do little to prevent side-channel attacks that exploit memory content and register value-based behavior. As some attacks are quite new for general-purpose CPUs, there are only few approaches for mitigating them automatically.

Computation simplifications which lead to an operand-dependent instruction latency can be addressed by moving to fixed-latency instructions. Recently, Intel introduced *data operand independent timing* (DOIT) [104], an upcoming ISA extension that ensures constant-time execution for a majority of arithmetic instructions. It also prevents data-dependent prefetching.

Silent stores occur when the CPU detects that the same data is written to a memory address and thus skips the store. To ensure that the processor always executes a store,

`cio` by Flanders et al. [73] compute an auxiliary value that is guaranteed to be different than the old and new value, and stores it in between. This way, the stored value always changes, and silent stores are suppressed. Besides eliminating silent stores, `cio` is also able to mask operands in a way that certain computation simplifications are prevented.

In general, masking is an effective method for preventing value-based side-channels. Our ciphertext side-channel hardening tool `Cipherfix` [244] uses dynamic taint tracking to find instructions and memory areas which touch secret data. We then rewrite the program and dependency binaries to add a random mask before each store to sensitive memory locations. The mask is stored alongside the masked data and subtracted again when reading it. This way, the observed ciphertext becomes independent from the stored secret, and changes even when the stored value does not change. Thus, `Cipherfix` would also prevent silent stores, though it arguably assumes a stronger threat model and thus overapproximates.

In our ciphertext side-channel attack paper [136] we also explored two other mitigation approaches: Interleaving and address rotation. Both are specific to ciphertext side-channels, as they break the otherwise deterministic mapping between physical address and plaintext to ciphertext. Interleaving breaks up each memory block into two parts, where the former holds the original data and the latter holds a counter that is incremented with each store. The size of the aforementioned memory block matches the block size of the memory encryption scheme. If the block size is 16 byte (AES), this results in an 8-byte counter, so a ciphertext can only repeat after 2^{64} stores to the same address. While very secure and efficient, interleaving has the drawback that it breaks common patterns like copying of contiguous memory (`memcpy`) and unaligned accesses, making its practical implementation very complicated and unstable.

Address rotation takes an alternative route, by not changing the plaintext itself, but the address where it is stored. That approach has a high memory cost and requires considerable bookkeeping, but it is suitable to protect small well-known workloads like the stack on user/kernel space context switches. Our generic TEE hardening framework `Obelix` [246] uses interleaving to protect data stored in the data ORAM, and address rotation of the code and data scratchpads to avoid repeating code/data block ciphertexts.

Conclusion

Defending against side-channel attacks remains a challenge given the variety of attack methods. While there are many proposals for countermeasures, only few of them are actually deployed in practice. At the moment, the most relevant defense is manually created constant-time code, which is resistant against memory access pattern attacks. However, writing constant-time code is not trivial and may fail subtly, either due to the developer missing small leakages or the compiler re-introducing input-dependent behavior. We have surveyed many tools that aim to detect such remaining vulnerabilities, and found that there are trade-offs between accuracy and scalability: While static tools allow proving the absence of vulnerabilities, dynamic tools are generally much more efficient and easier to use.

In an effort to combine high scalability with accurate results, we designed the Microwalk framework, which relies on finding differences between execution traces for concrete inputs. By design, Microwalk is complete for deterministic programs, and we are working on further reducing the false negative rate by incorporating coverage into the analysis. In contrast to usual research prototypes, we took extra steps to ensure practical suitability in line with the results from Jancar et al. [118], which includes simple analysis templates and integration into CI workflows. Microwalk is subject to ongoing development, with the goal to eventually offer all analysis capabilities that we discussed in the state-of-the-art. DATA is currently the only tool that was shown to be able to analyze implementations with internal secrets, and no other tool than `ct-verif` can declassify public outputs. By developing methods to integrate these capabilities into a trace comparison-based approach, we plan to offer a comprehensive solution suitable for practical development.

Automated leakage analysis assists developers of cryptographic software, who are generally aware of memory access pattern side-channels and know constant-time programming techniques. However, users of TEEs may not be as knowledgeable to side-channels, but still wish to fully protect their workloads. We have discussed various proposals for automated hardening against different attacks. Hardware-based approaches promise efficient and robust solutions, but depend on processor vendors adopting them. On the contrary, software solutions can be applied by anyone, though potentially at an increased performance cost. We explored automated hardening schemes against attacks that exist

in modern processors and which do not have hardware mitigations available. First, with `Cipherfix`, we built a binary rewriting framework which added masking to any memory access to combat ciphertext side-channels. `Cipherfix` requires the input program to be constant-time. We went several steps further with `Obelix`, where we took a holistic approach at side-channels in TEEs, creating a framework that automatically hardens any program against memory access pattern attacks, single-stepping and ciphertext side-channels, with the option to include measures to prevent speculative execution and fault injection attacks as well. While providing yet unmatched security guarantees, `Obelix` comes with a significant performance overhead for non-trivial workloads. We hope that our first exploration in the area of *generic* countermeasures inspires further research that leads to a reduction of the performance impact in the future, while maintaining the security guarantees.

Finally, future work may look into combining leakage analysis and code rewriting to precisely harden programs without any manual intervention and at a minimal performance overhead. This way, developers can focus on writing clean code, leaving all side-channel hardening to automated tools. Such tools can also take into account processor-specific leakages and can deploy application-specific countermeasures, e.g., yielding highly hardened (but slower) code for TEEs and leaky (but very fast) code for local applications.

References

- [1] *AddressSanitizer*. <https://github.com/google/sanitizers/wiki/AddressSanitizer>. (Visited on 2024-05-21).
- [2] Johan Agat. “Transforming Out Timing Leaks”. In: *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2000. DOI: 10.1145/325694.325702.
- [3] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyong Lee. “OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX”. In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. 2019. URL: <https://www.ndss-symposium.org/ndss-paper/obfuscuro-a-commodity-obfuscation-engine-on-intel-sgx/>.
- [4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. “Port Contention for Fun and Profit”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00066.
- [5] Nadhem J. AlFardan and Kenneth G. Paterson. “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols”. In: *2013 IEEE Symposium on Security and Privacy (SP)*. 2013. DOI: 10.1109/SP.2013.42.
- [6] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. “Amplifying side channels through performance degradation”. In: *32nd Annual Computer Security Applications Conference (ACSAC)*. 2016. URL: <http://dl.acm.org/citation.cfm?id=2991084>.
- [7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>.
- [8] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. “Formal verification of side-channel countermeasures using self-composition”. In: *Sci. Comput. Program.* 78.7 (2013), pp. 796–812. DOI: 10.1016/j.scico.2011.10.008.

- [9] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>. 2020. (Visited on 2024-05-21).
- [10] AMD. *AMD64 Architecture Programmer's Manual*. 2024.
- [11] Dennis Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. No Starch Press, 2018.
- [12] Marc Andryscio, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "On Subnormal Floating Point and Abnormal Timing". In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.44.
- [13] *angr*. <https://angr.io/>. (Visited on 2024-05-21).
- [14] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terachi, and Shiyi Wei. "Decomposition instead of self-composition for proving the absence of timing channels". In: *38th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2017. DOI: 10.1145/3062341.3062378.
- [15] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a Minute! A fast, Cross-VM Attack on AES". In: *17th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2014. DOI: 10.1007/978-3-319-11379-1_15.
- [16] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. "LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage". In: *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020. DOI: 10.1145/3372297.3417268.
- [17] ARM. *Learn the architecture - Introducing Arm Confidential Compute Architecture*. <https://developer.arm.com/documentation/den0125/latest>. (Visited on 2024-05-21).
- [18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, 2023-11.
- [19] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. "SideTrail: Verifying Time-Balancing of Cryptosystems". In: *10th International Conference on Verified Software Theories, Tools, and Experiments (VSTTE)*. 2018. DOI: 10.1007/978-3-030-03592-1_12.
- [20] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018), 50:1–50:39. DOI: 10.1145/3182657.

-
- [21] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. “Abacus: Precise Side-Channel Analysis”. In: *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2021. DOI: 10.1109/ICSE43902.2021.00078.
 - [22] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. “Spec-Shield: Shielding Speculative Data from Microarchitectural Covert Channels”. In: *28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2019. DOI: 10.1109/PACT.2019.00020.
 - [23] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. “System-level Non-interference for Constant-time Cryptography”. In: *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014. DOI: 10.1145/2660267.2660283.
 - [24] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Relational Verification Using Product Programs”. In: *17th International Symposium on Formal Methods (FM)*. 2011. DOI: 10.1007/978-3-642-21437-0_17.
 - [25] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. “Secure Information Flow by Self-Composition”. In: *17th IEEE Computer Security Foundations Workshop (CSFW)*. 2004. DOI: 10.1109/CSFW.2004.17.
 - [26] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. “ARMore: Pushing Love Back Into Binaries”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/di-bartolomeo>.
 - [27] Tiyyash Basu and Sudipta Chattopadhyay. “Testing Cache Side-Channel Leakage”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICST Workshops)*. 2017. DOI: 10.1109/ICSTW.2017.16.
 - [28] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *2005 USENIX Annual Technical Conference (USENIX ATC)*. 2005. URL: <http://www.usenix.org/events/usenix05/tech/freenix/bellard.html>.
 - [29] Sebastian Berndt, Jan Wichelmann, Claudius Pott, Tim-Henrik Traving, and Thomas Eisenbarth. “ASAP: Algorithm Substitution Attacks on Cryptographic Protocols”. In: *2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2022. DOI: 10.1145/3488932.3517387.
 - [30] Daniel J Bernstein. *Cache-timing attacks on AES*. 2005.
 - [31] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. “SMoTherSpec: Exploiting Speculative Execution through Port Contention”. In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019. DOI: 10.1145/3319535.3363194.

- [32] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying Constant-Time Implementations by Abstract Interpretation”. In: *22nd European Symposium on Research in Computer Security (ESORICS)*. 2017. DOI: 10.1007/978-3-319-66402-6_16.
- [33] Daniel Bleichenbacher. “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1”. In: *18th Annual International Cryptology Conference (CRYPTO)*. 1998. DOI: 10.1007/BFb0055716.
- [34] Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. “Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484583.
- [35] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel S. Emer, and Mengjia Yan. “CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches”. In: *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020. DOI: 10.1109/MICRO50266.2020.00092.
- [36] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. “Symbolic path cost analysis for side-channel detection”. In: *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2018. DOI: 10.1145/3213846.3213867.
- [37] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. “RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/briongos>.
- [38] Samira Briongos, Pedro Malagón, José L. Risco-Martín, and José Manuel Moya. “Modeling side-channel cache attacks on AES”. In: *Summer Computer Simulation Conference (SummerSim)*. 2016. URL: <http://dl.acm.org/citation.cfm?id=3015611>.
- [39] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00022.
- [40] Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. “SpecSafe: detecting cache side channels in a speculative world”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–28. DOI: 10.1145/3485506.
- [41] Robert Buhren. “Insecure Until Proven Updated: Analyzing AMD SEV’s Remote Attestation”. In: *31. Krypto-Tag, Berlin*. 2019. DOI: 10.18420/CDM-2019-31-25.

-
- [42] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/bulck>.
 - [43] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00089.
 - [44] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018. DOI: 10.1145/3243734.3243822.
 - [45] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *2nd Workshop on System Software for Trusted Execution (SysTEX@SOSP)*. 2017. DOI: 10.1145/3152701.3152706.
 - [46] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.
 - [47] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/canella>.
 - [48] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019. DOI: 10.1145/3319535.3363219.
 - [49] Chandler Carruth. *Speculative Load Hardening*. <https://l1vm.org/docs/SpeculativeLoadHardening.html>.

- [50] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. “FaCT: A Flexible, Constant-Time Programming Language”. In: *IEEE Cybersecurity Development (SecDev)s*. 2017. DOI: 10.1109/SecDev.2017.24.
- [51] Sudipta Chattopadhyay and Abhik Roychoudhury. “Scalable and precise refinement of cache timing analysis via path-sensitive verification”. In: *Real Time Syst.* 49.4 (2013), pp. 517–562. DOI: 10.1007/S11241-013-9178-0.
- [52] Sudipta Chattopadhyay and Abhik Roychoudhury. “Symbolic Verification of Cache Side-Channel Freedom”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37.11 (2018), pp. 2812–2823. DOI: 10.1109/TCAD.2018.2858402.
- [53] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. “GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers”. In: *33rd USENIX Security Symposium*. 2024.
- [54] CompCert Project. *The CompCert formally-verified C compiler*. <https://github.com/AbsInt/CompCert>. (Visited on 2024-05-21).
- [55] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. “AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/constable>.
- [56] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: <http://eprint.iacr.org/2016/086>.
- [57] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan>.
- [58] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *4th ACM Symposium on Principles of Programming Languages (POPL)*. 1977. DOI: 10.1145/512950.512973.
- [59] Shujie Cui, Haohua Li, Yuanhong Li, Zhi Zhang, Lluís Vilanova, and Peter R. Pietzuch. “QuanShield: Protecting against Side-Channels Attacks using Self-Destructing Enclaves”. In: *CoRR abs/2312.11796* (2023). DOI: 10.48550/ARXIV.2312.11796. arXiv: 2312.11796.

-
- [60] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00074.
 - [61] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure”. In: *ACM Trans. Priv. Secur.* 26.2 (2023), 11:1–11:42. DOI: 10.1145/3563037.
 - [62] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. “ProSpeCT: Provably Secure Speculation for the Constant-Time Policy”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/daniel>.
 - [63] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. “CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/deng-sen>.
 - [64] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00009.
 - [65] Craig Disselkoen, Sunjay Cauligi, Dean Tullsen, and Deian Stefan. “Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork”. In: *TECHCON*. 2020.
 - [66] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen>.
 - [67] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *22nd USENIX Security Symposium*. 2013. URL: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev>.
 - [68] Goran Doychev and Boris Köpf. “Rigorous analysis of software countermeasures against cache attacks”. In: *38th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2017. DOI: 10.1145/3062341.3062388.
 - [69] *DynamoRIO: Dynamic Instrumentation Tool Platform*. <https://dynamorio.org/>.

- [70] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. “Automatic Detection of Speculative Execution Combinations”. In: *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022. DOI: 10.1145/3548606.3560555.
- [71] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. “An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors”. In: *IEEE Trans. Computers* 72.1 (2023), pp. 222–235. DOI: 10.1109/TC.2022.3152666.
- [72] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. “Relational Symbolic Execution”. In: *21st International Symposium on Principles and Practice of Programming Languages (PPDP)*. 2019. DOI: 10.1145/3354166.3354175.
- [73] Michael Flanders, Reshabh Sharma, Alexandra Michael, Dan Grossman, and David Kohlbrenner. “Avoiding Instruction-Centric Microarchitectural Timing Channels Via Binary-Code Transformations”. In: *2024 Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2024.
- [74] Timothy S. Freeman and Frank Pfenning. “Refinement Types for ML”. In: *1991 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 1991. DOI: 10.1145/113445.113468.
- [75] Jacob Fustos, Farzad Farshchi, and Heechul Yun. “SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks”. In: *56th Annual Design Automation Conference (DAC)*. 2019. DOI: 10.1145/3316781.3317914.
- [76] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. “SQUIP: Exploiting the Scheduler Queue Contention Side Channel”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023. DOI: 10.1109/SP46215.2023.10179368.
- [77] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *J. Cryptogr. Eng.* 8.1 (2018), pp. 1–27. DOI: 10.1007/s13389-016-0141-6.
- [78] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. “A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries”. In: *2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023. DOI: 10.1145/3576915.3623112.

-
- [79] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. "Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023. DOI: 10.1109/SP46215.2023.10179440.
- [80] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *2005 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2005. DOI: 10.1145/1065010.1065036.
- [81] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy (SP)*. 1982. DOI: 10.1109/SP.1982.10014.
- [82] Cosmin Gorgovan, Amanieu D'Antras, and Mikel Luján. "MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM". In: *ACM Trans. Archit. Code Optim.* 13.1 (2016), 14:1–14:26. DOI: 10.1145/2896451.
- [83] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>.
- [84] Marc Green, Leandro Rodrigues Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think". In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/green>.
- [85] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed Javascript". In: *20th European Symposium on Research in Computer Security (ESORICS)*. 2015. DOI: 10.1007/978-3-319-24174-6_6.
- [86] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR". In: *9th International Engineering Secure Software and Systems Symposium (ESSoS)*. 2017. DOI: 10.1007/978-3-319-62105-0_11.
- [87] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoecl, and Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018. DOI: 10.1109/SP.2018.00031.
- [88] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *13th Detection of Intrusions and Malware, and Vulnerability Assessment Conference (DIMVA)*. 2016. DOI: 10.1007/978-3-319-40667-1_15.

- [89] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *13th Detection of Intrusions and Malware, and Vulnerability Assessment Conference (DIMVA)*. 2016. DOI: 10.1007/978-3-319-40667-1_14.
- [90] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *24th USENIX Security Symposium*. 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>.
- [91] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. “Spectector: Principled Detection of Speculative Information Flows”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00011.
- [92] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games - Bringing Access-Based Cache Attacks on AES to Practice”. In: *2011 IEEE Symposium on Security and Privacy (SP)*. 2011. DOI: 10.1109/SP.2011.22.
- [93] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. “SpecuSym: speculative symbolic execution for cache timing leak detection”. In: *42nd International Conference on Software Engineering (ICSE)*. 2020. DOI: 10.1145/3377811.3380428.
- [94] Marcus Hähnel, Weidong Cui, and Marcus Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: *2017 USENIX Annual Technical Conference (USENIX ATC)*. 2017. URL: <https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel>.
- [95] John Hazen. *Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer*. <https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/>. 2018-01-03. (Visited on 2024-05-21).
- [96] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. “ct-fuzz: Fuzzing for Timing Leaks”. In: *13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 2020. DOI: 10.1109/ICST46399.2020.00063.
- [97] Aki Helin. *radamsa*. <https://gitlab.com/akihe/radamsa>. (Visited on 2024-05-21).
- [98] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.
- [99] Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Hit by the Bus: QoS Degradation Attack on Android”. In: *2020 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2017. DOI: 10.1145/3052973.3053028.

-
- [100] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. “Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud”. In: *IACR Cryptol. ePrint Arch.* (2015), p. 898. URL: <http://eprint.iacr.org/2015/898>.
- [101] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *18th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2016. DOI: 10.1007/978-3-662-53140-2_18.
- [102] Intel. *Affected Processors: Guidance for Security Issues on Intel Processors*. <https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html>.
- [103] Intel. *Analysis of Speculative Execution Side Channels*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analysis-speculative-execution-side-channels.html>.
- [104] Intel. *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html>. 2023.
- [105] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals*. 2023.
- [106] Intel. *Intel Software Guard Extensions*. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf. 2016. (Visited on 2024-05-21).
- [107] Intel. *Intel Transactional Synchronization Extension (Intel TSX) Disable Update for Selected Processors*. <https://cdrdv2.intel.com/v1/dl/getContent/643557>. 2023.
- [108] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*. <https://cdrdv2-public.intel.com/733575/intel-tdx-module-1.5-base-spec-348549002.pdf>. 2023. (Visited on 2024-05-21).
- [109] Intel. *L1 Terminal Fault*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html>.
- [110] Intel. *Retpoline: A Branch Target Injection Mitigation*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html>.

- [111] Intel. *Runtime Encryption of Memory with Intel® Total Memory Encryption Multi-Key (Intel® TME-MK)*. <https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html>. (Visited on 2024-05-21).
- [112] Intel. *Speculative Execution Side Channel Mitigations*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html>.
- [113] Intel. *Supporting Intel SGX on Multi-Socket Platforms*. <https://web.archive.org/web/20230302235023/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-multi-socket-platforms.pdf>. Archived version. (Visited on 2024-05-21).
- [114] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. "Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries". In: *CoRR* abs/1709.01552 (2017). arXiv: 1709.01552. URL: <http://arxiv.org/abs/1709.01552>.
- [115] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES". In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.42.
- [116] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors". In: *2015 Euromicro Conference on Digital System Design (DSD)*. 2015. DOI: 10.1109/DSD.2015.56.
- [117] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/islam>.
- [118] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ""They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833713.
- [119] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. "Cache Refinement Type for Side-Channel Detection of Cryptographic Software". In: *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022. DOI: 10.1145/3548606.3560672.

-
- [120] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. “The Gates of Time: Improving Cache Attacks with Transient Execution”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/katzman>.
 - [121] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. “VOLTpwn: Attacking x86 Processor Integrity from Software”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar>.
 - [122] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014. DOI: 10.1109/ISCA.2014.6853210.
 - [123] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.
 - [124] *KLEE Symbolic Virtual Machine*. <https://github.com/klee/klee>. (Visited on 2024-05-21).
 - [125] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00002.
 - [126] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *16th Annual International Cryptology Conference (CRYPTO)*. 1996. DOI: 10.1007/3-540-68697-5_9.
 - [127] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. “Half-Double: Hammering From the Next Row Over”. In: *31st USENIX Security Symposium*. 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double>.
 - [128] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. “Spectre Returns! Speculation Attacks using the Return Stack Buffer”. In: *12th USENIX Workshop on Offensive Technologies (WOOT)*. 2018. URL: <https://www.usenix.org/conference/woot18/presentation/koruyeh>.

- [129] William Kosasih, Yusi Feng, Chitchanok Chuensatiansup, Yuval Yarom, and Ziyuan Zhu. “SoK: Can We Really Detect Cache Side-Channel Attacks by Monitoring Performance Counters?” In: *2024 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2024.
- [130] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “NetCAT: Practical Cache Attacks from the Network”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00082.
- [131] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. “RAMBleed: Reading Bits in Memory Without Accessing Them”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00020.
- [132] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. “MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape”. In: *2022 Annual Computer Security Applications Conference (ACSAC)*. 2022. DOI: 10.1145/3564625.3568002.
- [133] A Langley. *ctgrind: Checking that functions are constant time with Valgrind*. 2010.
- [134] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snaveley. “PEBIL: Efficient static binary instrumentation for Linux”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452024.
- [135] lcamtuf. *american fuzzy lop*. <https://lcamtuf.coredump.cx/afl/>. (Visited on 2024-05-21).
- [136] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. “A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833768.
- [137] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. “CrossLine: Breaking “Security-by-Crash” based Memory Isolation in AMD SEV”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3485253.
- [138] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. “CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel”. In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan>.

-
- [139] Jens Lindemann and Mathias Fischer. “A memory-deduplication side-channel attack to detect applications in co-resident virtual machines”. In: *33rd Annual ACM Symposium on Applied Computing (SAC)*. 2018. DOI: 10.1145/3167132.3167151.
 - [140] Moritz Lipp, Daniel Gruss, and Michael Schwarz. “AMD Prefetch Attacks through Power and Time”. In: *31st USENIX Security Symposium*. 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/lipp>.
 - [141] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. “Practical Keystroke Timing Attacks in Sandboxed JavaScript”. In: *22nd European Symposium on Research in Computer Security (ESORICS)*. 2017. DOI: 10.1007/978-3-319-66399-9_11.
 - [142] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp>.
 - [143] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. “PLATYPUS: Software-based Power Side-Channel Attacks on x86”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021. DOI: 10.1109/SP40001.2021.00063.
 - [144] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>.
 - [145] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. “Nethammer: Inducing Rowhammer Faults through Network Requests”. In: *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&P Workshops)*. 2020. DOI: 10.1109/EUROSPW51379.2020.00102.
 - [146] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.43.
 - [147] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. “In defense of soundness: a manifesto”. In: *Commun. ACM* 58.2 (2015), pp. 44–46. DOI: 10.1145/2644805.

- [148] Giorgi Maisuradze and Christian Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers”. In: *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018. DOI: 10.1145/3243734.3243761.
- [149] MAMBO: A Low-Overhead Dynamic Binary Modification Tool for RISC architectures. <https://github.com/beehive-lab/mambo>. (Visited on 2024-05-21).
- [150] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.
- [151] Heiko Mantel, Alexandra Weber, and Boris Köpf. “A Systematic Study of Cache Side Channels Across AES Implementations”. In: *9th International Engineering Secure Software and Systems Symposium (ESSoS)*. 2017. DOI: 10.1007/978-3-319-62105-0_14.
- [152] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2015. DOI: 10.1007/978-3-319-26362-5_3.
- [153] Microsoft C++ Team. *Spectre mitigations in MSVC*. <https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/>. (Visited on 2024-05-21).
- [154] Microwalk Project. *Microwalk Source Code and Templates*. <https://github.com/microwalk-project>.
- [155] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: 10.1145/96267.96279.
- [156] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *19th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2017. DOI: 10.1007/978-3-319-66787-4_4.
- [157] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *Int. J. Parallel Program.* 47.4 (2019), pp. 538–570. DOI: 10.1007/s10766-018-0611-9.
- [158] Daniel Moghimi. “Downfall: Exploiting Speculative Data Gathering”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi>.

-
- [159] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat>.
 - [160] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *11th European Workshop on Systems Security (EUROSEC)*. 2018. DOI: 10.1145/3193111.3193112.
 - [161] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. "SEVerity: Code Injection Attacks against Encrypted Virtual Machines". In: *2021 IEEE Security and Privacy Workshops (SP Workshops)*. 2021. DOI: 10.1109/SPW53761.2021.00063.
 - [162] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00057.
 - [163] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. "BIRD: Binary Interpretation using Runtime Disassembly". In: *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2006. DOI: 10.1109/CGO.2006.6.
 - [164] Moritz Neikes. *TIMECOP: Automated Dynamic Analysis for Timing Side-Channels*. 2020. URL: <https://www.post-apocalyptic-crypto.org/timecop/>.
 - [165] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *2007 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2007. DOI: 10.1145/1250734.1250746.
 - [166] Michael Neve and Kris Tiri. "On the complexity of side-channel attacks on AES-256 - methodology and quantitative results on cache attacks". In: *IACR Cryptol. ePrint Arch.* (2007), p. 318. URL: <http://eprint.iacr.org/2007/318>.
 - [167] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. "DiffFuzz: Differential Fuzzing for Side-Channel Analysis". In: *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik, Innsbruck*. 2020. DOI: 10.18420/SE2020_37.
 - [168] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference (USENIX ATC)*. 2018. URL: <https://www.usenix.org/conference/atc18/presentation/oleksenko>.

- [169] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. “SpecFuzz: Bringing Spectre-type vulnerabilities to the surface”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko>.
- [170] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015. DOI: 10.1145/2810103.2813708.
- [171] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *The Cryptographers’ Track at the RSA Conference 2006 (CT-RSA)*. 2006. DOI: 10.1007/11605805_1.
- [172] Dan Page. “Partitioned Cache Architecture as a Side-Channel Defence Mechanism”. In: *IACR Cryptol. ePrint Arch.* (2005), p. 280. URL: <http://eprint.iacr.org/2005/280>.
- [173] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. “Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT”. In: *29th IEEE Computer Security Foundations Symposium (CSF)*. 2016. DOI: 10.1109/CSF.2016.34.
- [174] Marco Patrignani and Marco Guarnieri. “Exorcising Spectres with Secure Compilers”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484534.
- [175] Colin Percival. *Cache missing for fun and profit*. 2005.
- [176] *Pin - A Dynamic Binary Instrumentation Tool*. <https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html>. (Visited on 2024-05-21).
- [177] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SymCC: Don’t interpret, compile!” In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau>.
- [178] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. “Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend”. In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/puddu>.
- [179] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. “Show-Time: Amplifying Arbitrary CPU Timing Side Channels”. In: *2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2023. DOI: 10.1145/3579856.3590332.

-
- [180] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. "Systematic Analysis of Randomization-based Protected Cache Architectures". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021. DOI: 10.1109/SP40001.2021.00011.
 - [181] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks". In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484816.
 - [182] QEMU: A generic and open source machine emulator and virtualizer. <https://www.qemu.org/>. (Visited on 2024-05-21).
 - [183] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Abusing the Processor Voltage to Break Arm TrustZone". In: *GetMobile Mob. Comput. Commun.* 24.2 (2020), pp. 30–33. DOI: 10.1145/3427384.3427394.
 - [184] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. "VoltJockey: A New Dynamic Voltage Scaling-Based Fault Injection Attack on Intel SGX". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 40.6 (2021), pp. 1130–1143. DOI: 10.1109/TCAD.2020.3024853.
 - [185] Moinuddin K. Qureshi. "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping". In: *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018. DOI: 10.1109/MICRO.2018.00068.
 - [186] Moinuddin K. Qureshi. "New attacks and defense for encrypted-address cache". In: *46th International Symposium on Computer Architecture (ISCA)*. 2019. DOI: 10.1145/3307650.3322246.
 - [187] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2006. DOI: 10.1109/MICRO.2006.49.
 - [188] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks". In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/ragab>.
 - [189] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021. DOI: 10.1109/SP40001.2021.00020.

- [190] David A. Ramos and Dawson R. Engler. “Under-Constrained Symbolic Execution: Correctness Checking for Real Code”. In: *24th USENIX Security Symposium*. 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos>.
- [191] Ashay Rane, Calvin Lin, and Mohit Tiwari. “Raccoon: Closing Digital Side-Channels through Obfuscated Execution”. In: *24th USENIX Security Symposium*. 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane>.
- [192] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time?” In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017. DOI: 10.23919/DATE.2017.7927267.
- [193] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. “Sparse representation of implicit flows with applications to side-channel detection”. In: *25th International Conference on Compiler Construction (CC)*. 2016. DOI: 10.1145/2892208.2892230.
- [194] Stephen Röttger and Artur Janc. *A Spectre proof-of-concept for a Spectre-proof web*. <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>.
- [195] Andrei Sabelfeld and Andrew C. Myers. “A Model for Delimited Information Release”. In: *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium (ISSS)*. 2003. DOI: 10.1007/978-3-540-37621-7_9.
- [196] Daniel Sánchez and Christos Kozyrakis. “Vantage: scalable and efficient fine-grain cache partitioning”. In: *38th International Symposium on Computer Architecture (ISCA)*. 2011. DOI: 10.1145/2000064.2000073.
- [197] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “RIDL: Rogue In-Flight Data Load”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00087.
- [198] Alexander Schaub. “Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations. (Méthodes formelles pour l’analyse de fuites cache-timing et la génération de clés dans les implémentations cryptographiques)”. PhD thesis. Polytechnic Institute of Paris, France, 2020. URL: <https://tel.archives-ouvertes.fr/tel-03205242>.
- [199] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *2010 IEEE Symposium on Security and Privacy (SP)*. 2010. DOI: 10.1109/SP.2010.26.

-
- [200] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre". In: *27th Annual Network and Distributed System Security Symposium (NDSS)*. 2020. URL: <https://www.ndss-symposium.org/ndss-paper/context-a-generic-approach-for-mitigating-spectre/>.
- [201] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019. DOI: 10.1145/3319535.3354252.
- [202] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network". In: *24th European Symposium on Research in Computer Security (ESORICS)*. 2019. DOI: 10.1007/978-3-030-29959-0_14.
- [203] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. "Remote Memory-Deduplication Attacks". In: *29th Annual Network and Distributed System Security Symposium (NDSS)*. 2022. URL: <https://www.ndss-symposium.org/ndss-paper/auto-draft-235/>.
- [204] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017. URL: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/>.
- [205] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. "Spectre Declassified: Reading from the Right Place at the Wrong Time". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023. DOI: 10.1109/SP46215.2023.10179355.
- [206] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016. DOI: 10.1109/SP.2016.17.
- [207] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. "Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses". In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman>.

- [208] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. “Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484783.
- [209] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. “TeeJam: Sub-Cache-Line Leakages Strike Back”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 457–500. DOI: 10.46586/TCHES.V2024.I1.457-500.
- [210] Laurent Simon, David Chisnall, and Ross J. Anderson. “What You Get is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018. DOI: 10.1109/EuroSP.2018.00009.
- [211] *Skylake (client) - Microarchitectures - Intel*. [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)). (Visited on 2024-05-21).
- [212] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. “Side-channel Elimination via Partial Control-flow Linearization”. In: *ACM Trans. Program. Lang. Syst.* 45.2 (2023), 13:1–13:43. DOI: 10.1145/3594736.
- [213] Luigi Soares and Fernando Magno Quintão Pereira. “Memory-Safe Elimination of Side Channels”. In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021. DOI: 10.1109/CGO51591.2021.9370305.
- [214] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. “Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021. DOI: 10.1109/SP40001.2021.00050.
- [215] Raphael Spreitzer and Thomas Plos. “Cache-Access Pattern Attack on Disaligned AES T-Tables”. In: *4th International Constructive Side-Channel Analysis and Secure Design Workshop (COSADE)*. 2013. DOI: 10.1007/978-3-642-40026-1_13.
- [216] Julian Stecklina and Thomas Prescher. “LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels”. In: *CoRR abs/1806.07480* (2018). arXiv: 1806.07480. URL: <http://arxiv.org/abs/1806.07480>.
- [217] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *J. ACM* 65.4 (2018), 18:1–18:26. DOI: 10.1145/3177872.
- [218] Chunga Sung, Brandon Paulsen, and Chao Wang. “CANAL: a cache timing analysis framework via LLVM transformation”. In: *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 2018. DOI: 10.1145/3238147.3240485.

-
- [219] *SymCC: efficient compiler-based symbolic execution*. <https://github.com/eurecom-s3/symcc>. (Visited on 2024-05-21).
- [220] George Taylor, Peter Davies, and Michael Farmwald. “The TLB slice—a low-cost high-speed address translation mechanism”. In: *ACM SIGARCH Computer Architecture News* 18.2SI (1990-05), pp. 355–363. ISSN: 0163-5964. DOI: 10.1145/325096.325161.
- [221] M. Caner Tol, Berk Gülmezoglu, Koray Yurtseven, and Berk Sunar. “FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings”. In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021. DOI: 10.1109/EUROSP51992.2021.00047.
- [222] *Valgrind*. <https://valgrind.org/>. (Visited on 2024-05-21).
- [223] VAMPIRE lab. *SUPERCOP*. 2020. URL: <https://bench.cr.yp.to/supercop.html>.
- [224] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. “Automatically eliminating speculative leaks from cryptographic code with blade”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434330.
- [225] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016. DOI: 10.1145/2976749.2978406.
- [226] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. “Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833570.
- [227] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. “Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data”. In: *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 2021. DOI: 10.1109/ISCA52012.2021.00035.
- [228] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. “A Sound Type System for Secure Flow Analysis”. In: *J. Comput. Secur.* 4.2/3 (1996), pp. 167–188. DOI: 10.3233/JCS-1996-42-304.

- [229] Dennis M. Volpano and Geoffrey Smith. “Eliminating Covert Flows with Minimum Typings”. In: *10th IEEE Computer Security Foundations Workshop (CSFW)*. 1997. DOI: 10.1109/CSFW.1997.596807.
- [230] Luke Wagner. *Mitigating Side-Channel Attacks*. <https://www.chromium.org/Home/chromium-security/ssca/>. (Visited on 2024-05-21).
- [231] Luke Wagner. *Mitigations landing for new class of timing attack*. <https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/>. 2018-01-03. (Visited on 2024-05-21).
- [232] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. “KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution”. In: *ACM Trans. Softw. Eng. Methodol.* 29.3 (2020), 14:1–14:31. DOI: 10.1145/3385897.
- [233] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. “oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis”. In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2504–2519. DOI: 10.1109/TSE.2019.2953709.
- [234] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. “Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation”. In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai>.
- [235] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. “Cached: Identifying Cache-Based Timing Channels in Production Software”. In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>.
- [236] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. “Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86”. In: *31st USENIX Security Symposium*. 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen>.
- [237] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. “DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023. DOI: 10.1109/SP46215.2023.10179326.
- [238] Zhenghong Wang and Ruby B. Lee. “New cache designs for thwarting software cache-based side channel attacks”. In: *34th International Symposium on Computer Architecture (ISCA)*. 2007. DOI: 10.1145/1250662.1250723.

-
- [239] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. “Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/weiser>.
- [240] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. “DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [241] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. “ScatterCache: Thwarting Cache Attacks via Cache Set Randomization”. In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/werner>.
- [242] Jan Wichelmann, Sebastian Berndt, Claudius Pott, and Thomas Eisenbarth. “Help, My Signal has Bad Device! - Breaking the Signal Messenger’s Post-Compromise Security Through a Malicious Device”. In: *18th Detection of Intrusions and Malware, and Vulnerability Assessment Conference (DIMVA)*. 2021. DOI: 10.1007/978-3-030-80825-9_5.
- [243] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *34th Annual Computer Security Applications Conference (ACSAC)*. 2018. DOI: 10.1145/3274694.3274741.
- [244] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. “Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/wichelmann>.
- [245] Jan Wichelmann, Christopher Peredy, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. “MAMBO-V: Dynamic Side-Channel Leakage Analysis on RISC-V”. In: *20th Detection of Intrusions and Malware, and Vulnerability Assessment Conference (DIMVA)*. 2023. DOI: 10.1007/978-3-031-35504-2_1.
- [246] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. “Obelix: Mitigating Side-Channels through Dynamic Obfuscation”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024. DOI: 10.1109/SP54263.2024.00182.
- [247] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. “Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications”. In: *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022. DOI: 10.1145/3548606.3560654.

- [248] Johannes Wikner, Christiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks”. In: *2022 Workshop on Offensive Technologies (WOOT)*. 2022.
- [249] Johannes Wikner and Kaveh Razavi. “RETBLEED: Arbitrary Speculative Code Execution with Return Instructions”. In: *31st USENIX Security Symposium*. 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/wikner>.
- [250] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. “SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00080.
- [251] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. “SEV-Step A Single-Stepping Framework for AMD-SEV”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 180–206. DOI: 10.46586/TCHES.V2024.I1.180-206.
- [252] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. “undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation”. In: *IEEE Security and Privacy Workshops (SP Workshops)*. 2021. DOI: 10.1109/SPW53761.2021.00064.
- [253] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. “Egalito: Layout-Agnostic Binary Recompile”. In: *2020 Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020. DOI: 10.1145/3373376.3378470.
- [254] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. “Eliminating timing side-channel leaks using program repair”. In: *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2018. DOI: 10.1145/3213846.3213851.
- [255] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. “STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves”. In: *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134016.
- [256] Yuejian Xie and Gabriel H. Loh. “PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches”. In: *36th International Symposium on Computer Architecture (ISCA)*. 2009. DOI: 10.1145/1555754.1555778.
- [257] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.45.

-
- [258] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00004.
- [259] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium*. 2014. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.
- [260] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *18th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2016. DOI: 10.1007/978-3-662-53140-2_17.
- [261] Tuba Yavuz, Farhaan Fowze, Grant Hernandez, Ken Yihang Bai, Kevin R. B. Butler, and Dave Jing Tian. "ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs". In: *IEEE Trans. Dependable Secur. Comput.* 20.2 (2023), pp. 1577–1595. DOI: 10.1109/TDSC.2022.3160346.
- [262] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. "Synchronization Storage Channels (S2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions". In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong>.
- [263] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. "CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software". In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/yuan-yuanyuan-cacheql>.
- [264] Andreas Zankl, Johann Heyszl, and Georg Sigl. "Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software". In: *15th International Conference on Smart Card Research and Advanced Applications (CARDIS)*. 2016. DOI: 10.1007/978-3-319-54669-8_14.
- [265] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. "Ultimate SLH: Taking Speculative Load Hardening to the Next Level". In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh>.

Part II

Publications

MicroWalk: A Framework for Finding Side Channels in Binaries

Publication

Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. *MicroWalk: A Framework for Finding Side Channels in Binaries*. In *34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

Contribution

Main author.

Outline

1	Introduction	102
2	Background	104
3	<i>MicroWalk</i> Analysis Technique	108
4	<i>MicroWalk</i> Framework	113
5	Case Study I: Intel IPP	116
6	Case Study II: Microsoft CNG	121
7	Related Work	123
8	Conclusion	125
	References	127

MicroWalk: A Framework for Finding Side Channels in Binaries

Jan Wichelmann¹, Ahmad Moghimi², Thomas Eisenbarth^{1,2}, and Berk Sunar²

¹Universität zu Lübeck

²Worcester Polytechnic Institute

Microarchitectural side channels expose unprotected software to information leakage attacks where a software adversary is able to track runtime behavior of a benign process and steal secrets such as cryptographic keys. As suggested by incremental software patches for the RSA algorithm against variants of side-channel attacks within different versions of cryptographic libraries, protecting security-critical algorithms against side channels is an intricate task. Software protections avoid leakages by operating in constant time with a uniform resource usage pattern independent of the processed secret. In this respect, automated testing and verification of software binaries for leakage-free behavior is of importance, particularly when the source code is not available. In this work, we propose a novel technique based on Dynamic Binary Instrumentation and Mutual Information Analysis to efficiently locate and quantify memory based and control-flow based microarchitectural leakages. We develop a software framework named *MicroWalk* for side-channel analysis of binaries which can be extended to support new classes of leakage. For the first time, by utilizing *MicroWalk*, we perform rigorous leakage analysis of two widely-used closed-source cryptographic libraries: *Intel IPP* and *Microsoft CNG*. We analyze 15 different cryptographic implementations consisting of 112 million instructions in about 105 minutes of CPU time. By locating previously unknown leakages in hardened implementations, our results suggest that *MicroWalk* can efficiently find microarchitectural leakages in software binaries.

1 Introduction

Side-channel attacks exploit information leakage through physical behavior of computing devices. The physical behavior depends on the processed data. The resulting data-dependent patterns in physical signals such as power consumption, electromagnetic emanations or timing behavior can be analyzed to extract secrets such as cryptographic keys [19, 33, 50, 59]. Despite the physical proximity requirement for most physical attacks, there exist remotely exploitable side channels such as microarchitectural attacks [32].

Microarchitectural attacks exploit shared hardware features such as cache [13, 65, 67], branch prediction unit (BPU) [2], memory order buffer (MOB) [62] and speculative execution engine [49] to extract secrets from a process executed *on the same system*. These attacks can be mounted remotely or locally on systems where untrusted entities can execute code on a shared hardware, either because the system is shared or untrusted code is executed. Scenarios include but are not limited to cross-VM attacks in the cloud environment [44, 57], drive-by JavaScript trojans inside the browser sandbox [54], attacks originating from untrusted mobile applications [55] and system-adversarial attacks against Intel Software Guard eXtensions (SGX) [20, 61]. Microarchitectural leakage can be used to break software implementations of cryptographic schemes where the adversaries recover the secret key by combining the leaked partial information from key-dependent activities [12, 31, 83]. These side channels can be further exploited to violate user's privacy through activity profiling [38], or to steal user's keystrokes [36]. Memory protections such as Address Space Layout Randomization (ASLR) can be bypassed by exploiting microarchitectural side-channel leakages [30].

Defense against microarchitectural side channels have been proposed based on new hardware design [24, 47], systematic mitigation [56] and activity monitoring [18, 87]. However, the most widely-used protection against microarchitectural leakage is software hardening using constant-time programming techniques [17, 40]. In this context, constant-time programming implies using microarchitectural resources in a secret-independent fashion. Therefore, timing, or trace-based leakages [26] in the hardware would not reveal any information about the secret. These techniques depend on the underlying microarchitecture and side-channel knowledge, i.e. software implementations are hardened to follow a constant-time behavior based on published attacks on the target microarchitecture. Consequently, a novel microarchitectural attack demands new changes to these software protections. While true constant-time code avoids such problems, manual verification of the software implementation for constant-time behavior is an error-prone task, and it requires extensive, and ever growing knowledge of side channels. Besides, what we observe in the source code is not always what is executed on the processor [72], and there are leakages in the program binary that remain unobserved in the source code [46]. The state of art tools and techniques for automated finding of side-channel leakages in software binaries fall short in practice, particularly when the source code is not available. As a result, commercial cryptographic products such as *Microsoft Cryptography API Next Generation (CNG)*, which is used everyday by millions of users, have never been externally audited for side-channel security.

1.1 Our Contribution

We propose a leakage detection technique, and develop a framework named *MicroWalk* to locate leakages within software binaries. We apply *MicroWalk* to analyze two commercial closed-source cryptographic libraries hardened toward constant-time protections and report previously unknown vulnerabilities, in summary:

- We propose a technique based on Dynamic Binary Instrumentation (DBI) and Mutual Information (MI) Analysis to locate memory based and control-flow based microarchitectural leakages in software binaries.
- We develop the *MicroWalk* framework to perform automated leakage testing and quantification based on our technique. Our framework can be extended to locate other and new types of microarchitectural leakages.
- We demonstrate the ease-of-use of *MicroWalk* by showing how it significantly eases the analysis of binary code even in cases where source code is not accessible to the analyst.
- We apply *MicroWalk* to cryptographic schemes implemented in *Microsoft CNG* and *Intel IPP*, which are both widely used, yet closed source crypto libraries. Our results include previously unknown leakages in these libraries.
- We perform analysis and quantification of the critical leakages, and discuss the security impact of these leakages on the relevant cryptographic schemes.

1.2 Analysis Setup and Targeted Software

Our machine for analysis is a Dell XPS 8920 machine with Intel(R) Core i7-7700 processor, 16 GB of RAM and a traditional hard disk drive running *Microsoft Windows 10*. The *MicroWalk* Framework uses *Pin* v3.6 as the DBI backend, and *IDA Pro* v6.95 for binary visualization and leakage analysis. The tested cryptographic modules are *Microsoft bcryptprimitives.dll* v10.0.17134.1 as part of *Microsoft CNG*, and *Intel IPP* v2018.2.185.

2 Background

2.1 Dynamic Binary Instrumentation

Dynamic program analysis is more accurate compared to static analysis due to availability of real system states and data [63]. Dynamic analysis requires instrumentation of the program binary, and it analyzes the program when it executes. The instrumentation code

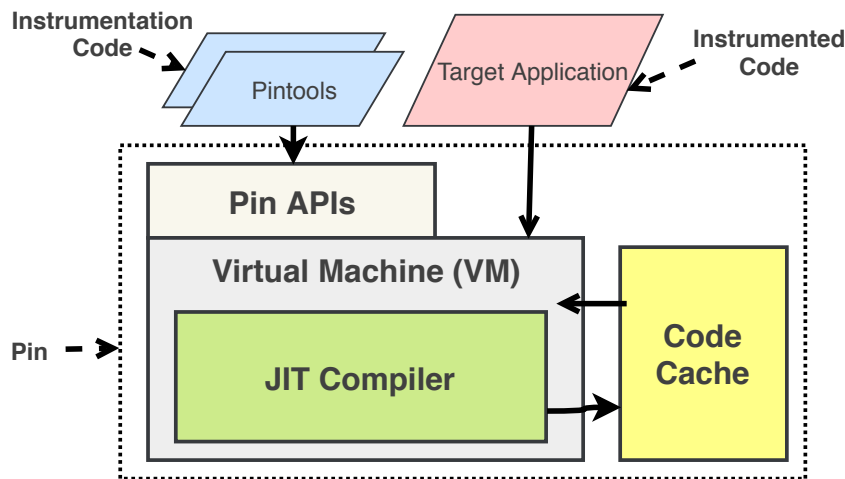


Figure 1: Pin: The JIT compiler combines application and instrumentation codes, and it stores the transformed binary in code cache. The virtual machine maintains and tracks program states, while it executes from the code cache.

is added to the program binary without changing the normal logic and execution flow of the program under analysis, and it contains minimal instructions and subroutines for collecting metadata and measurements. The instrumentation code and the instrumented code execute at the same time following each other. Indeed, adding instrumentation is easier during the compilation phase and when the source code is available [53], but source code is not always available, and the analysis would not be as accurate due to compiler transformations. Thanks to Dynamic Binary Instrumentation (DBI) frameworks such as Pin [58], it is possible to instrument program binaries without source code.

Pin is a DBI framework based on just-in-time (JIT) compilation. In general, JIT compilers transform a source language to executable binary instructions at runtime. Figure 1 shows how an embedded JIT engine is part of *Pin* to recompile the binary instructions at runtime and combine the program's instruction with instrumentation codes, named *Pintools*. To avoid the performance pitfall of JIT compilation, *Pin* uses a code cache that stores the combined code, and re-execution of the same basic blocks occur from the code cache. Binary instrumentation using *Pintools* gives us an easy to use interface to collect runtime metadata about program states such as the accessed memory addresses, targets of indirect branches and memory allocations. *Pin* makes sure the instrumentation is transparent, i.e., it preserves the original application behavior [58]. These events can be measures as accurate as they occur on the OS and the processor and as it would be an uninstrumented execution. In terms of microarchitectural analysis, we can observe the program behavior and resource usage as they appear on the hardware, and this gives us the ability to model a known microarchitectural leakage based on the observation of states from a real system.

2.2 Microarchitectural Leakage

Modern microarchitectures feature various shared resources, and these resources are distributed among malicious and benign processes with different permissions. A malicious process, sharing the same hardware, can cause resource contention with a victim and measure the timing of either the victim or herself to learn about the victim's runtime. In a cache attack, the adversary accesses the same cache set that the victim's security-critical memory accesses are mapped to, and she measures the memory accesses' timing. A slow memory access reveals some information about the address bits of the victim's memory access. As motivated by cache attacks on AES [13, 65], knowledge of secret-dependent memory accesses such as S-Box operations leaks information about the internal runtime state, and this information can be used for cryptanalysis and secret key recovery. In cache attacks, the size of each cache block is 64 B which stops adversaries from gaining information about the $\log_2(64) = 6$ least significant address bits. While some constant-time software countermeasures assume that the adversary cannot leak these bits, there are microarchitectural attacks on cache banks and MOB that leak beyond this assumption [62, 84]. In this work, we consider all secret-dependent memory accesses and treat them as memory-based leakages disregarding their spatial resolution.

Memory operations are not the only source of leakage. A conditional statement, or a processing loop that depends on a secret to choose an execution path can leak information about the secret. Each unique execution path operates on a different set of instructions, and it consumes the shared resources uniquely. Shared resources such as instruction cache and BPU leak information about the state of branches [1, 3]. Figure 2 resembles a classical side-channel leakage in RSA Montgomery modular exponentiation. This algorithm processes a secret exponent one bit at a time, and it performs an additional arithmetic operation when the secret bit is one. An adversary who is able to track the execution of the left branch is able to determine the secret value that affected the conditional jump decision. We treat all the attacks that are triggered due to secret-dependent branches as control-flow based attacks.

2.3 Mutual Information Analysis

Mutual information (MI) measures the mutual dependence of two random variables, and it can be used to quantify the average amount of obtainable information about one variable through observation of the second variable [37]. Mutual information using Shannon entropy is defined as

$$I(X, Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2 \left(\frac{p(x, y)}{p(x)p(y)} \right),$$

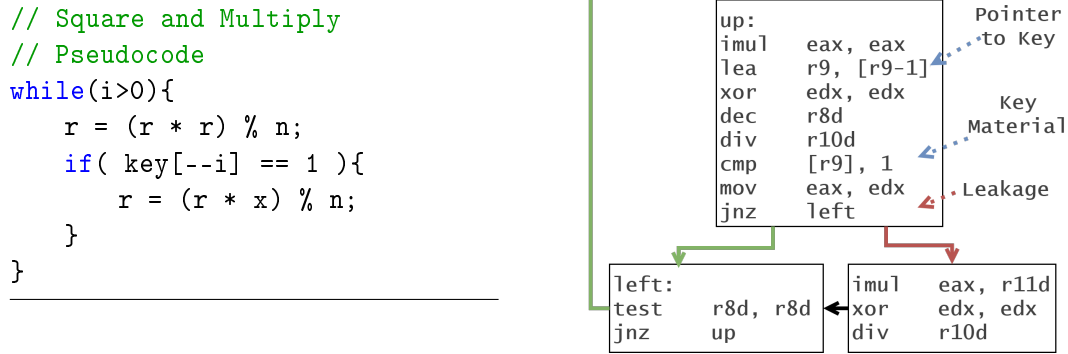


Figure 2: Montgomery Square and Multiply operations can leak information about the secret exponent. While `r9` points to the exponent in memory, comparison of a value from the exponent determines if the left jump should occur which leaves a key-dependent microarchitectural footprint.

where $p(x)$ and $p(y)$ are the probability distributions of random variables X and Y respectively. $p(x, y)$ is the joint probability of X and Y , and $I(X, Y)$ tells us the average amount of dependent information in bits¹ between the variables X and Y . MI has been utilized to quantify side-channel security [10, 43, 75, 86], or to mount side-channel attacks [34]. Redefining MI in the side-channel context, we can define variable X as the secret and variable Y as an internal physical state of a system leaked through a side channel. $I(X, Y)$ will measure the average amount of leakage from secret X , through observing the side-channel information Y .

2.4 Signing Algorithms

2.4.1 DSA

Digital Signing Algorithm (DSA) [68] is a signature scheme based on the discrete logarithm problem (DLP) [48]. Choosing a prime p , another prime q divisor of $p - 1$, the group generator g , a secret key x , the public key $y = g^x \bmod p$, and the hash of the message to be signed z , the DSA signing operation is defined as

$$k \leftarrow \text{RANDOM} \mid 1 < k < q$$

$$r = (g^k \bmod p) \bmod q, \quad s = k^{-1}(z + r \cdot x) \bmod q$$

where (r, s) are the output signature pairs.

¹ \log_2 measures the MI in *bit* unit.

2.4.2 ECDSA

Elliptic-Curve DSA (ECDSA), as an analogue of *DSA*, is a signature scheme based on elliptic curves [45], in which the subgroup of a prime p is replaced by the group of points on an elliptic curve over a finite field. Choosing an elliptic curve, a point on the curve G , the integer order n of G , a secret key d_A , the public key $Q_A = d_A \times G$, and the hash of message to be signed z , the ECDSA signing operation is defined as

$$\begin{aligned} k &\leftarrow \text{RANDOM} \mid 1 < k < n - 1 \\ (x_1, y_1) &= k \times G \\ r &= x_1 \bmod n, \quad s = k^{-1}(z + r \cdot d_A) \bmod n \end{aligned}$$

Both *DSA* and *ECDSA* use an ephemeral secret k that needs to be chosen randomly for each operation.

2.4.3 Modified Elliptic Curve Signature

Elliptic-Curve Nyberg-Rueppel (ECNR) [64] and *SM2* [8], a standard signature scheme, are modified schemes based on *ECDSA* that allow signatures with message recovery. *ECNR* and *SM2* are widely used, and they are both supported by *Intel IPP*. Public parameters, the private/public key pair and the ephemeral secrets are chosen similar to *ECDSA*. The pair (x_1, y_1) is also calculated similarly, but the signature generation for *ECNR* is defined as

$$r = x_1 + z, \quad s = k - r \cdot d_A,$$

and the signature generation for *SM2* is defined as

$$r = x_1 + z, \quad s = (1 + d_A)^{-1}(k - r \cdot d_A)$$

3 MicroWalk Analysis Technique

MicroWalk aims to find microarchitectural leakages in software binaries. A binary implementation is vulnerable to microarchitectural side-channel attacks when there is a dependency between a secret and internal computation states observable through the side channels. We expose such relationships and quantify the amount of observable leakage in these implementations. This helps security analysts **1)** to reveal whether an implementation has leakages, **2)** to locate the exact location of each leakage in the binary, and **3)** to measure the dependency between the secret and the internal state, i.e., it can

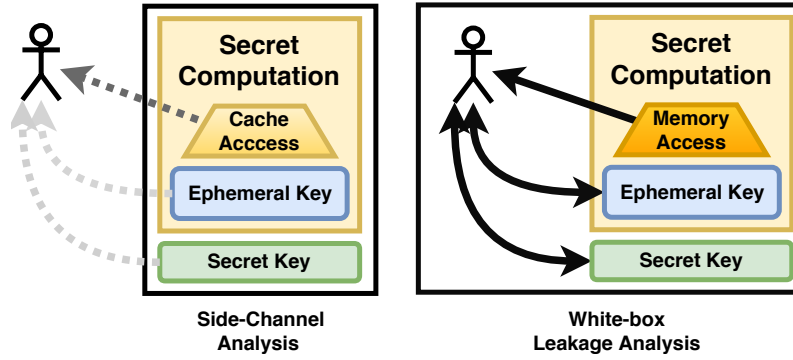


Figure 3: Left: side-channel analyst finds relationship between a real leakage such as cache access pattern and secrets such as cryptographic keys. Right: *MicroWalk* follows a white-box model where the security analyst has full access to runtime states such as memory accesses, and she can find dependencies between arbitrary secrets and internal states.

give some confidence value on the severity of the leakage. In contrast to side-channel analysis model, we are able to perform this analysis in a white-box model.

3.1 Leakage Analysis Model

We assume a strong adversary with full access to runtime events such as memory accesses, execution path and even register values. Further, the adversary can choose and modify any secret input of the system. This strong adversary can define any internal computation state such as addresses of memory accesses and register values as a potential leakage vector, based on her knowledge of a category of side-channel attacks, e.g., memory based attacks (Section 2.2). The adversary executes the system under her full control, and feeds the system with arbitrary secrets while collecting runtime traces for the defined leakage vector. Figure 3 compares our leakage analysis model with the side-channel analysis model. As an example, if we try to analyze a binary implementation of AES, we need to define certain operations as our leakage vectors. Based on cache attacks, an adversary defines memory accesses as a leakage vector, and she collects all memory accesses during the execution of AES using arbitrary secret keys. If there is a dependency between different secret keys and the variation of memory accesses, the adversary can locate which instructions relate to any secret-dependent memory accesses, and identify potential leakages.

3.2 Capturing Internal States

We choose two common sources of leakage as our leakage vectors: 1) execution path and 2) memory accesses. A true constant-time implementation follows a linear execution path

for any given secret input; each time a software performs a secret-dependent conditional branch, it leaks some amount of information about the secret. Defining execution path as a leakage vector helps us to check whether for any secret input the same operations are performed. The second common source of leakage are memory accesses. A constant-time implementation should follow a secret-independent memory access pattern. If, for example, an implementation of a cryptographic algorithm does key-dependent table look ups which can be exploited by measuring cache timings, an attacker will be able to extract parts of the secret key; we ensure that memory accesses are either invariant, or at least uncorrelated to the input (e.g. blinding in RSA [50]).

To be able to detect these two types of leakages, we need to collect the internal state for all memory accesses and branch operations. First of all, we generate a set of arbitrary inputs for a chosen secret. These inputs can be either random (e.g. plain texts for encryption) or have a special structure with some random components (e.g. private keys or ephemeral secrets). We then execute the target binary on each input and log the following events:

- memory allocations
- branches, calls and returns
- memory reads and writes
- stack operations.

Absolute memory addresses may vary even for constant-time programs, e.g., due to ASLR and dynamic heap allocation. We use the trace of memory allocations and stack operations to compute relative memory addresses; our meta data then consists of a list of relative addresses for memory accesses and the branches to, from and within the code we are analyzing. Note that one can define other leakage sources based on the underlying microarchitecture and collect the state of relevant instructions for analysis, e.g., the multiplication on some *ARM* platforms leaks information [7].

3.3 Preparing State Variables

We do not make any assumptions on the leakage granularity; compared to similar techniques, that stop at cache line level, we keep this parameter freely configurable. This has the advantage that the analysis can be restricted to leakage sizes that are actually relevant to the analyst: For example, as of writing this paper, on Intel processors the finest known attack has a leakage granularity of 4 bytes [62]. Applying our technique in 1-byte mode will give all positions where a leakage might occur, but if one only expects 4-byte leakages to be exploitable, this may yield some false positives. Instead, the security analyst can choose the leakage granularity that fits to the desired spatial resolution. After

applying the chosen leakage granularity of $g \in \mathbb{N}$ bytes by discarding the lower $\log_2 g$ bits of each address, we can acquire an efficient representation of a specific execution state by computing a hash value of all or a subset of the trace entries; a truly constant-time program should have identical hashes of the full trace for every secret input. If we are only interested in analyzing individual instructions, e.g., memory access leakages of a specific subroutine, we can as well just compute the hash for the subset of traces for a single instruction.

3.4 Leakage Analysis

Our approach identifies any variations resulting from unique inputs and captured internal states per input. A naive approach is to compare the collected traces and divide them into classes. Observing more than one class informs us about secret-dependent operations. One can also compare raw traces sequentially which outlines all positions where the program behaves input-dependent and thereby allows to isolate the problematic sections. In addition to these simple approaches, we use MI to detect/locate these leakages, and to quantify the observable information.

To simplify MI analysis, we assume that X is a set of unique uniformly distributed input test cases, which trigger deterministic behavior of the investigated program. If the program makes use of randomization (e.g. blinding in RSA [50]), the test cases $x \in X$ should contain the corresponding sources of randomness too.

Let Y be a set of possible internal states (e.g. hashes of execution traces). We then define the execution state $T_i \subset X \times Y$ of the analyzed program at time point i as

$$(x, y) \in T_i \wedge (x, y') \in T_i \Rightarrow y = y',$$

i.e. each test case $x \in X$ appears at most once in T_i . The probability of one observed state $y \in Y$ is

$$p_i(y) = \frac{|\{(x', y') \in T_i \mid y = y'\}|}{|T_i|}.$$

For the probability of pairs $(x, y) \in X \times Y$ we get

$$p_i(x, y) = \begin{cases} \frac{1}{|X|} & \text{if } (x, y) \in T_i, \\ 0 & \text{else,} \end{cases}$$

since each input and therefore each input/state tuple occur exactly once: $|T_i| = |X|$.

With this knowledge we can finally compute the mutual information between test cases

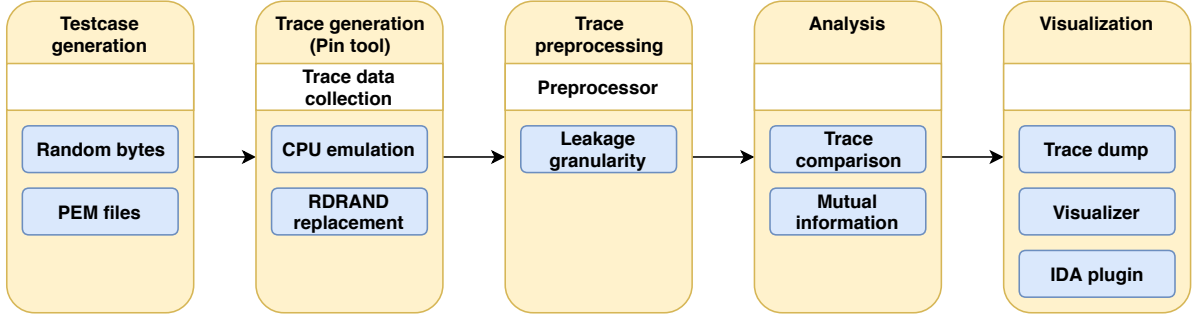


Figure 4: The *MicroWalk* pipeline: Given the software binary under test, the framework generates test cases using the selected source, that are then used to produce execution traces. These traces need to be preprocessed to extract important information. The resulting trace files can then be analyzed for leakages, which are shown to the user in the visualization stage. Each stage can be easily modified to add further functionality, that is used either interchangeably or in addition to existing features.

X and the set of all occurring states $Y_i := \{y \mid (x, y) \in T_i\}$:

$$\begin{aligned}
 I_i(X, Y_i) &= \sum_{(x,y) \in T_i} \frac{1}{|X|} \log_2 \left(\frac{\frac{1}{|X|}}{\frac{1}{|X|} \cdot \frac{|\{(x', y') \in T_i \mid y = y'\}|}{|T_i|}} \right) \\
 &= \sum_{(x,y) \in T_i} \frac{1}{|X|} \log_2 \left(\frac{|T_i|}{|\{(x', y') \in T_i \mid y = y'\}|} \right).
 \end{aligned}$$

3.5 Interpretation of MI Score

As mentioned before, we can compute the MI for the entire trace, or a single instruction. A non-zero score for whole-trace MI tells us that an implementation has leakages, but it cannot locate the leakage point, and an implementation that has multiple leakage points over the execution period will have an aggregated MI value. The MI for single instructions is more precise, in which we can locate the instructions with positive score. The MI score $I_i(X_i, Y_i)$ is bounded by the amount of input bits $\log_2 |X|$, and (for instruction MI) by the operand size: For example, an instruction that once accesses memory depending on 8 bits of the input will generate MI $\min\{8, \log_2 |X|\}$. If we only execute $|X| = 128$ test cases, we get MI score 7; for 256 or more test cases we get MI score 8. The analyzed MI score is an estimate of the average leakage over the given test cases. MI is the appropriate metric in cases where the analyzed inputs are not under the attackers control and commonly used in leakage quantification. Alternatively, the *worst case* leakage for any attacker-chosen input is given by the *min entropy*, which only considers the most likely guess. The use of min entropy instead of MI in *MicroWalk* is recommended if the adversary has full control over the inputs and specific high-leakage inputs exist [74].

4 *MicroWalk* Framework

The *MicroWalk* framework is built as a pipeline with separate stages for test case generation, tracing and analysis (Figure 4). This modular design reduce the complexity, leading to easier extensibility: If one wants to implement additional analysis techniques apart from the ones that we already provide, she can directly add a new analysis stage, without needing to touch other parts like the trace generation. We will continue explaining these stages in more detail.

4.1 Investigated Binary

Although we are only interested in analyzing a specific function within a binary, we have to instrument and collect traces for the entire setup stage of the application before reaching to the analysis point, including the target library or executable itself, and parts of dependencies like system components. This process leads to an enormous decrease in analysis speed. A more efficient approach is to load and instrument the setup code only once, and then process the incoming test cases in a controlled loop. For libraries, we create a wrapper executable that executes an interface in a loop with new test cases. For executable applications, we can adopt in-memory fuzzing techniques where we inject hooks at the beginning and end of the target function and control the execution of the function to reset to the beginning with new test cases [11]. To separate traces of the different test cases and avoid that the loop code causes false positives, we place calls to two instrumented dummy functions `PinNotifyTestcaseStart` and `PinNotifyTestcaseEnd`, which mark the start and the end of the analyzed section. A similar approach is taken by some fuzzers like WinAFL [35], which use a built-in functionality of DynamoRIO [29] to exchange the argument list of `main` or a similar function.

4.2 Input Generation

The *MicroWalk* framework utilizes cryptographically secure pseudorandom number generators to create random test cases of any specified length. This performs well when analyzing cryptographic code, e.g. decrypting random ciphertexts. If a special input format is required, the test case generation code can be easily extended to produce such inputs, e.g. cryptographic keys in PEM format; this way, parts of the input can be kept constant while other parts are randomized, allowing to isolate the parameters which cause non-constant time behavior. Further, the framework supports passing a directory containing already generated inputs of any format.

4.3 Trace Generation

To trace the execution of individual test cases, we create a custom so-called *Pintool*, which is a client library making use of Intel Pin's dynamic instrumentation capabilities. In summary, our *Pintool* logs the following events in a custom binary format on disk:

- module loads and the respective start and end addresses;
- calls to dummy functions in the instrumented executable, to identify start and end of a test case execution;
- sizes and addresses of allocated memory blocks through heap allocation functions such as `malloc` and `free` (Platform dependent), for resolving relative memory addresses;
- Stack pointer modifications, for resolving relative addresses;
- branches, calls and returns to and from all involved modules;
- memory reads and writes in investigated modules.

4.3.1 Instruction Emulation

Several cryptographic libraries use the `CPUID` instruction to detect the supported instructions for the respective processor and select a fitting implementation (that e.g. makes use of AES-NI). We enabled the *Pintool* to change the output of this instruction. This allows to test arbitrary subsets of the instruction set that is available on the computer running *Pin*.

As mentioned in Section 3, cryptographic implementations might use randomization techniques like blinding to hide correlations between secret inputs and execution, or use ephemeral secrets. Some of these rely on the `RDRAND` instruction, which provides random numbers seeded with hardware entropy [70]. We provide an option to override the output of this instruction with arbitrary fixed values to control the randomization of the program under investigation.

4.4 Trace Preprocessing

The resulting raw trace files now need some preprocessing: First we add the common trace prefix that is generated before running the first test case, and which contains allocation data from the setup phase. In a second step, we calculate relative offsets of memory addresses. This involves associating branch targets with instruction offsets in the respective libraries, and identify offsets of memory accesses, such that traces

generated using the same test case but during different runs of the *Pintool* still match, regardless of the usage of randomized virtual addresses, e.g., *ASLR*. For accesses to heap memory, we need to maintain a list of all currently allocated blocks: We use a stack to match the allocation size with their respective returned memory addresses, since in some implementations the heap allocator tends to call itself to reserve memory for internal bookkeeping. Finally the resulting preprocessed trace file is much smaller than the raw one (which can be discarded after this step), saving disk space and speeding up the following analysis stage.

4.4.1 Applying Leakage Granularity

We apply the leakage granularity immediately before the analysis starts; this way the preprocessed trace files are not modified, so the analysis can be performed on the same traces with different parameters. An analysis granularity of $g = 2^b$ bytes ($b \in \mathbb{N}$) is introduced by discarding the b least significant bits of each relative address.

4.5 Leakage Analysis

We implemented three different analysis methods in our framework:

4.5.1 Analysis 1: Trace Comparison

The first analysis method implements the trace comparison technique; given two preprocessed traces, we compare them entry by entry to check whether they differ at all. This performs well for leakage detection of particularly small algorithms such as symmetric ciphers. Optionally, the user can use trace diffs to manually inspect varying sections.

4.5.2 Analysis 2: Whole-trace MI

For leakage detection of an entire logic and calculation of the average amount of input bits that might leak over arbitrary parts of the execution (assuming that the attacker has full access to the trace), we provide an option to estimate the MI between input data and resulting trace. Given a set X of unique test cases, we need to determine matching outputs for each trace prefix. Since we can compute the final MI only after waiting for completion of all test cases, it would be inefficient to store the entire trace; instead we reduce the trace data by encoding information like relative memory accesses and branch targets into 64-bit integers, and then compress them into one 64-bit integer $y \in \{0, \dots, 2^{64} - 1\}$ using a hash function. We store the resulting tuples of inputs and

hashes in sets $T_i \subset X \times \{0, \dots, 2^{64} - 1\}$ for each prefix length i . We then apply the methods from Section 3 to measure the trace leakage.

4.5.3 Analysis 3: Single-instruction MI

The average amount of bits leaked by a single memory instruction is calculated analogously to the trace prefixes: Here, for a specific instruction i , $T_i \subset X \times \{0, \dots, 2^{64} - 1\}$ contains hashes of the accessed memory addresses for each input x . These hashes change when the accessed addresses, their amount or their order vary, thus we get the maximum amount of information that is leaked by the respective instruction.

4.6 Manual Inspection and Visualization

To be able to manually inspect the preprocessed traces, the program has an option to convert binary traces into a readable text representation. If MAP files with function names are available (exported by some compilers or disassemblers), these can be used to symbolize memory addresses. We also created an IDA python plugin to import our single-instruction MI results as disassembly annotations. This helps further analysis on which parts of functions and loops leak.

Further we developed an experimental visualization tool, that renders function names and then draws an execution path. It also provides an option to render two traces simultaneously and highlight all sections where they have differences. This gives a quick overview of potential leakages and their structure.

5 Case Study I: Intel IPP

Intel's *Integrated Performance Primitives (IPP)* cryptographic library aims to provide high performance cryptographic primitives that are compatible with various generations of Intel's processor [41]. *Intel IPP* supports symmetric operations such as AES, as well as asymmetric signature and encryption schemes such as ECDSA. Intel IPP is used as the cryptographic backend for many of Intel's security products such as Intel SGX. Each of the implemented schemes in this library comes in variants optimized for different processors [42]. The dynamic library checks the supported instruction set at runtime and chooses the most optimized implementation. However, developers can statically link toward a specific implementation by choosing the proper architecture code, e.g., `n8_ippsAESInit` rather than `ippsAESInit`. In this case study, we test implementations

for the variant optimized for processors supporting *Intel® Advanced Vector Extensions 2* (*Intel® AVX2*) with architecture code 19.

5.1 Applying *MicroWalk* MI Analysis to IPP

To be able to test *Intel IPP* cryptographic implementations, we prepared wrappers that perform encryption and signing operations. For each tested implementation, we configured the wrappers for testing multiple test case scenarios: **1)** randomized plaintexts/-ciphertexts to be encrypted/decrypted, or the message to be signed, **2)** randomized symmetric keys or private asymmetric keys, and **3)** random ephemeral secrets, when it is applicable, e.g., *DSA* and *ECDSA* as the input to MI Analysis. As suggested by chosen plaintext/ciphertext attacks, attacks on the cipher key and lattice attacks on ephemeral secrets [12], using these scenarios, we are able to detect leakages that are dependent on various types of secrets.

Table 1 shows the single-instruction MI analysis results, where symmetric ciphers: (*Triple*) *DES*, *AES* and *SM4* and asymmetric ciphers: *DSA*, *RSA*, *ECDSA*, *ECNR* and *SM2* have been tested. On our analysis setup, the total computational time to analyze 10 different implementations with about 92 million total instructions is 73 minutes of CPU time, highlighting the efficiency of our method. Note that we performed analysis with input size $2^7 = 128$ (7-bit MI) and input size $2^{10} = 1024$ (10-bit MI), for analysis of symmetric and asymmetric operations respectively. Although analysis with more iterations is possible, state-of-the-art side-channel attacks on these implementations suggest that the random secret should show leakage behavior after this number of iterations. *Intel IPP* uses two separate interfaces for the key schedule, and ephemeral secret generation for most implementations (Table 1).

(*Triple*) *DES*, *AES* and *SM4* are block ciphers that use table-based S-Box operations. The results suggest that these implementations are heavily protected against memory-based leakages. Our target architecture code uses the AES-NI instruction set for *AES* and *SM4* operations. AES-NI is inherently secure against known attacks. However, testing the *CTR mode* reveals some leakages. All asymmetric ciphers suffer from at least one leakage. For schemes that are based on elliptic curves such as *ECDSA*, *ECNR* and *SM2*, *Intel IPP* supports various standard curves. As some developers optimize curve arithmetic differently for various standard elliptic curves, we tested the *ECDSA* signing operation with three different curves: *SECP256R1*, *BN256* and *SM2*. However, the MI analysis results are exactly the same for different choices of elliptic curves. We found a total of 13 leakages in *Intel IPP*, while some of these leakages are triggered through calling the same subroutine, e.g., both *ECDSA* and *SM2* use the leaky subroutine for scalar multiplication. We will discuss these subroutines in more detail.

Table 1: Single-instruction MI Analysis of Intel IPP cryptographic implementations v2018.2.185. All implementations are chosen from the *I9* architecture code.

Scheme	Interfaces	Executed / Unique Instructions	Analysis Time (ms)	Leakage Found
3DES/ECB	ippsDESInit ippsTDESDecryptECB	4074613 / 70205	11921	0
SM4/ECB	ippsSMS4Init ippsSMS4EncryptECB	4085517 / 68221	10004	0
AES/CTR	ippsAESInit ippsAESEncryptCTR	2138799 / 49181	27289	2
DSA (512)	ippsDLPGenKeyPair ippsDLPSignDSA	12245281 / 57423	1735153	2
RSA (512)	ippsRSA_Decrypt	43987943 / 55167	275090	1
ECDSA (SECP256R1)	ippsECCPGenKeyPair ippsECCPSignDSA	4085155 / 63785	358373	3
ECDSA (BN256)	ippsECCPGenKeyPair ippsECCPSignDSA	5383210 / 63699	750188	*
ECDSA (SM2)	ippsECCPGenKeyPair ippsECCPSignDSA	5158607 / 63741	353435	*
ECNR (SECP256R1)	ippsECCPGenKeyPair ippsECCPSignNR	4028592 / 62447	281937	2
SM2	ippsECCPSignSM2	6021005 / 64273	554035	3
Total		91208722 / 618142	73 minutes	13

* Different curves did not change the results for ECDSA.

5.2 Discovered leakages in Intel IPP

We have found 7 different subroutines that have leakages, i.e., perform data-dependent memory accesses or branch decisions (Table 2). We performed an initial analysis of these leakages using our visualization tool and *IDA Pro*. The subroutine `gfec_MulBasePoint` performs scalar multiplication of a scalar and point on the elliptic curve, as a common operation in all curve-based signature schemes: *ECDSA*, *ECNR*, *SM2*. As defined by the signing algorithms (Section 2.4), `gfec_MulBasePoint` leaks information about the ephemeral secret. This leakage occurs due to the dependability of the number of times the window-based multiplier loop processes the ephemeral secret. Further leakages exist

Table 2: Discovered leakage subroutines within Intel IPP cryptographic implementations v2018.2.185. Some of the subroutines expose critical and potentially exploitable leakages.

Subroutine	Affected	MI	Leakage Source
gfec_MulBasePoint	ECDSA, ECNR, SM2	0.86 / 10	Conditional Loop
cpMontExpBin	DSA	3.73 / 10	Conditional Loop
cpModInv	DSA, SM2, ECDSA	3.88 / 10	Conditional Loop
ExpandRijndaelKey	AES/CTR	7.00 / 7	Memory Lookup
ippsAESEncryptCTR	AES/CTR	0.13 / 7	Conditional Loop
gsMontExpWin	RSA	1.12 / 10	Conditional Loop
		3.11 / 10	Memory Lookup
alm_mont_inv	ECDSA, ECNR, SM2	5.33 / 10	Conditional Loop
		9.98 / 10	Memory Lookup

in the curve operations after the scalar multiplication: The subroutine *alm_mont_inv* leaks information during the mapping of x coordinate of computed public point. As (x_1, y_1) are not secrets in the signing operation, this leakage is not critical, and we refrain from further root cause analysis. Similarly, the subroutine *cpModInv* has leakages with a relatively high MI score that is due to the secret-dependent loop count. *cpModInv* performs a modular inversion operation using Extended Euclidean Algorithm (EEA). In ECDSA, k^{-1} leaks information about the secret ephemeral, and in SM2, $(1 + d_A)^{-1}$ leaks information about the secret signing key. ECNR does not perform any modular inversion and is safe from leakages due to this subroutine. The existing leakage in *cpModInv* subroutine also applies to DSA where a modular inversion on ephemeral secret, k^{-1} can leak.

Intel IPP supports two distinct functions for performing Montgomery exponentiation. Exponentiation of big numbers is a common operation in schemes such as RSA and DSA. The RSA algorithm uses the *gsMonthExpWin* subroutine which is a window-based implementation of the Montgomery exponentiation. This function has leakages based on both memory lookup and conditional loop. The second Montgomery exponentiation subroutine *cpMontExpBin* is a protected binary implementation that has leakage due to the conditional loop count. DSA uses the latter, which leaks information about the ephemeral secret during computation of $(g^k \bmod p)$.

The only leakage exposed during testing of symmetric ciphers are due to AES key generation subroutine *ExpandRijndaelKey*, and calculation of the nonce length in CTR mode. *ExpandRijndaelKey* is called every time the *ippsAESInit* is used. As the high MI score shows, AES key schedule used during the CTR mode has full leakage. This leakage can be considered critical in scenarios such as the SGX environment where an

Algorithm 1 Bitmasked Montgomery Exponentiation

```

1: procedure BINEXP(base  $g$ , exponent  $k$ )
2:    $A \leftarrow R \bmod p$ 
3:    $\tilde{g} = \text{MontMul}(g, R^2 \bmod p)$ 
4:    $m \leftarrow 0$ 
5:    $i = 1$ 
6:   while  $i < (\text{BitLength}(k) \bmod 64)$  do
7:      $t \leftarrow A \& \sim m \mid \tilde{g} \& m$ 
8:      $A \leftarrow \text{MontMul}(A, t)$ 
9:      $m = \sim m \& k_i$ 
10:     $i = i + 1 - m$ 
11:  end while
12:  for  $j \leftarrow 1$  to  $\text{BitLength}(k)/64$  do
13:    perform the same operations as above.
14:  end for
15:  return  $A$ 
16: end procedure

```

adversary has a high resolution side channel [61, 78]. When the symmetric key is passed to the AES key schedule, a high resolution adversary can steal the secret key before any encryption/decryption. While *AES/CTR* encryption uses AES-NI, there is a loop within this implementation where calculating the length of nonce leaks about the leading zero bits.

5.2.1 Leakage of Scalar Multiplication

Scalar multiplication in *Intel IPP* uses a fixed-window algorithm with a window size of 5: for a 256-bit ephemeral secret, as defined by SECP256R1, the algorithm performs 51 iterations of the window operation. However, our dynamic analysis of the algorithm with various random ephemeral secrets shows that `gfec_MulBasePoint` skips the leading zero bits and applies fewer windows if there are leading zero bits in the beginning, as the multiple of the window size. In this case, the main loop performs 50 times for 2, 49 for 7 and 48 times for 12, etc, leading zero bits. *CacheQuote* [25] exploits a similar vulnerability used by *Intel EPID* signature scheme, but *EPID* uses a different function of *Intel IPP* for scalar multiplication `cpEcGFpMulPoint`. As our discovery suggests, this was a common issue in *Intel IPP* that was existed among other curve implementations. Although this implementation has countermeasure based on Scatter-Gather technique [17], this vulnerability can easily be exploited in high resolution settings using a lattice attack [25].

5.2.2 Bitmasked Montgomery Exponentiation

The Montgomery exponentiation in *Intel IPP* follows a bit-by-bit operation based on the Montgomery Reduction technique [39]. However, the implementation is protected by obfuscating the conditional statements as bit-masked operations. Therefore, the subroutine always executes the same Montgomery multiplication (`MontMul`) subroutine disregarding the value of the exponent bits. However, the exponent bits are used as a mask to choose the operand of the `MontMul` and to execute the `MontMul` two times with two different operands when the exponent bit is one. Although this implementation looks secure at first sight, the exponent bits are used **1**) to calculate the exponent bit length, i.e., leading zero-bit leakage, and **2**) to decide the number of iterations of the loop. Based on Algorithm 1, the main loop executes two times if an exponent bit is one and once if the exponent bit is zero. This leaks the Hamming weight of the ephemeral secret to a microarchitectural adversary.

Further, the algorithm performs a similar operation with separate instructions for different parts of the key. For example, for a 160-bit DSA exponent, the algorithm first processes the first 32 bits, and then another code section processes the remaining 128 bits of exponent. This gives an adversary a local Hamming weight leakage of the first 32-bit of the secret exponent.

6 Case Study II: Microsoft CNG

The *Cryptography API: Next Generation* (CNG) is the cryptography platform supplied with every Windows system beginning with Windows Vista, and replaces the older *CryptoAPI* as the default cryptographic stack. It includes many common algorithms, including *RSA*, *AES*, *ECDSA*. While the public API for *Microsoft CNG* resides in the `BCrypt.dll` system file, its cryptographic implementations themselves are located in another library file, `BCryptPrimitives.dll`. Microsoft does provide neither source code nor documentation for the internal functionality, but one can download PDB symbol files from Microsoft's symbol server, which contain most of the internal function names, helping to reduce the reverse engineering effort.

6.1 Applying *MicroWalk* MI Analysis to CNG

As we did with IPP, we again created wrapper executables to call the respective library functions of *RSA*, *DSA*, *ECDSA* and *AES/ECB*. For *AES*, the library uses the `CPUID` instruction to choose between two different implementations, one that uses *AES-NI* vector instructions, and a plain T-table based implementation. We tested both implementations

Table 3: Singe-instruction MI Analysis of some of the bcryptprimitives.dll v10.0.17134.1 cryptographic implementations.

Scheme	Interfaces	Executed / Unique Instructions	Analysis Time (ms)	Leakage Found
AES/ECB	SymCryptAesEcbEncrypt	2384298 / 55451	17546	0
AES/ECB	SymCryptAesEcbEncryptAsm	2324391 / 63179	26211	2
DSA (512)	MSCryptDsaSignHash	3586162 / 63748	223356	1
RSA (1024)	MSCryptRsaDecrypt	8073605 / 66454	760450	0
ECDSA (SECP256R1)	MSCryptEcDsaSignHash	4764783 / 64732	831136	1
Total		21133239 / 313564	31 minutes	4

Table 4: Discovered leakage subroutines within bcryptprimitives.dll v10.0.17134.1 cryptographic implementations.

Subroutine	Affected	MI	Leakage Source
SymCryptFdefModInvGeneric	DSA, ECDSA	10.00 / 10	Conditional Loop
SymCryptAesEncryptAsmInternal	AES	7.96 / 10	Memory Lookup

by emulating the CPUID instruction, as explained in Section 4.3.1. The results are shown in Table 3. We analyzed a total of 21 million instructions in 31 minutes of CPU time, finding four different leakage points. For *RSA*, we discovered that Microsoft’s implementation behaves truly constant-time. *ECDSA* and *DSA* implementations both suffer from leakage due to calling the same subroutine for modular inversion.

6.2 Discovered leakages in Microsoft CNG

Analyzing the aforementioned algorithms yielded two leakage candidates (see Table 4); the first one resides within the modular inversion function of *DSA* and *ECDSA* and is used for all processors. The MI returns full leakage for the modular inversion leakage, implying that the implementation is heavily unprotected. The second one is in the encryption function of *AES* and only used by processors not supporting AES-NI. As it is a table-based implementation, the leakage is expected.

6.2.1 Leakage of Modular Inversion

The modular inversion function that is used for *DSA* and *ECDSA* gives full *MI* on 1024 signing operations for random ephemeral secrets with fixed key and plaintext. This subroutine does not have any constant-time protection. However, while this is a non-constant time behavior and suggests that the ephemeral leaks, we considered this as not exploitable; Microsoft protects this implementation through a masking countermeasure. The masking countermeasure for modular inversion works as follow:

1. A mask value m is generated randomly.
2. The ephemeral secret k is multiplied by m before the modular inversion: $s = (k \cdot m)^{-1}(z + x) \bmod q$
3. Then the signature s is multiplied again with m to produce the correct signature:

$$s = sm = (k \cdot m)^{-1}(z + x) \cdot m = k^{-1}(z + x)$$

Thus, the implementation leaks $k \cdot m$, where m is a random per-signature generated mask, effectively preventing extraction of useful information. Leakage of ephemeral keys is exploitable [12], the randomized product of ephemeral key and a random value is not.

6.2.2 Leakage of AES T-table Lookup

The non-vector version of *AES* uses a common lookup table implementation, where four so-called *T-tables* combine the steps *SubBytes*, *ShiftRows* and *MixColumns*. Each round consists of four of such lookups per table, leading to $16 \cdot r$ memory accesses per encryption, where $r \in \{10, 12, 14\}$ is the number of rounds. The 8-bit indices used for the table accesses depend on the plaintext and the key; since the *MI* is 7.96 for 1024 measurements, these indices can be considered fully leaking. Each table entry has 4 bytes size, thus each T-table has 1024 bytes, and therefore takes 16 cache lines on an Intel processor; such implementations have already been shown to be exploitable with cache attacks [13].

7 Related Work

Programming languages can support constant-time code generation and verification [16, 21, 73]. The general approach is to support annotation of security-critical variables and to generate instructions that operate obliviously on annotated secrets. Annotated secrets

can be verified for constant-time behavior using SMT-based techniques [16]. Constant-time behavior can be enforced for some operations by using primitives such as oblivious RAM (ORAM) [76] and obfuscated execution [69]. Language-based approaches are not widely used, and annotation is an error-prone task.

Black-box testing approaches use statistical methods to quantify leakages of physical channels [23]. In particular, *Dudect* [71] performs black-box timing analysis, in which the timing of a target system with different inputs will be analyzed using the *t-test* [80], but these black-box techniques do not scale to microarchitectural attacks with a gray-box model. With an abstract model of the leakage channel, methods based on **Static Program Analysis** are proposed to analyze program code and to quantify leakages [4, 5, 9, 14, 51]. Similar to language-based approaches, these techniques are limited to correct annotation of the source code. While some of these approaches are limited to the source code and cannot find leakages that are potentially introduced by the compiler [5], others perform the analysis on the lower level LLVM bitcode [4] or the annotated machine code [9, 14]. However, they rely on the availability of the source code. *CacheAudit* [27, 28] is based on Static Binary Analysis (SBA). SBA approaches need to initially reconstruct the original basic blocks and control flow graph. Precise reconstruction of the program semantic and control flow graph is infeasible without the runtime information, by just using static disassembly [6]. As a result, while they give formal guarantees on the absence of leakages, they do not scale to accurately analyze large program binaries, e.g., *CacheAudit* approach has only been tested on rather simple algorithms such as sorting and symmetric encryption. Other proposals based on **Symbolic execution** quantify side-channel leakage by determining symbolic secret inputs that affect the runtime behavior [22, 66]. However symbolic execution is an expensive approach, and the proposed methods require access to the source code.

In this work, we leverage **Dynamic Program Analysis** techniques to accurately locate microarchitectural leakage in software binaries, as they execute on the processor. *ct-grind* [52] based on LLVM memcheck can check all branches and memory accesses to make sure that they do not have dependency on secret data. Irazoqui et al. instrument the source code to obtain and analyze cache traces using MI [43]. Sensitive code sections are identified by taint analysis. On binary-only approaches, *CacheD* [77] analyzes binaries based on symbolic execution and constraint solving. They initially use DBI to get execution traces for a set of input values; then, given the information which input values are considered secret, a taint analysis extracts all instructions that work with secrets, either directly or indirectly. These instructions are then analyzed using symbolic execution to detect whether cache leakages exist. In comparison, our method aims at maximum performance without too much loss of accuracy by only storing necessary information and using hash compression to get small execution states. The symbolic execution approach introduces a large bottleneck, as their analysis time suggest. This

saving of computation time allows us to detect also other types of leakages like differing loop counts or byte-level memory access differences. Also, since *MicroWalk* is designed as a modular open source framework, one can implement arbitrary analysis stages for other types of leakages. Zankl et al. [85] use DBI to collect traces for instruction based leakage detection. They use *t-tests* for leakage analysis and only test for execution flow leakages. *STACCO* [81] is focused on differential trace analysis for Bleichenbacher attacks [15]. Independently, *DATA* [79] follows a similar approach based on DBI. They use trace differentiation and *t-tests* for leakage analysis. As of our knowledge, our work is the first that has been tested on actual closed-source binaries.

8 Conclusion

The lack of efficient and practical tools for leakage analysis of binaries leave the reliability of these untested deployed implementations a mystery. To be able to analyze the compiler outputs and closed-source libraries, we have created an extensible framework that supports various types of microarchitectural leakages based on instruction and data cache, M0B, BPU, etc. *MicroWalk* can be extended to analyze other and future side channels. Our framework leverages *DBI* to collect the internal state of a program under test, and it applies multiple analysis techniques based on trace comparison and *MI*. *MicroWalk* is open source and is publicly accessible: <https://github.com/UzL-ITS/Microwalk>. We used this framework to thoroughly analyze two widely used closed-source libraries, *Intel IPP* and *Microsoft CNG*. The tested implementations are optimized for the current generation of Intel processors. Our report shows that side-channel countermeasures for these implementations are still not fully leakage-free, e.g., all the curve-based signature schemes in *Intel IPP* suffer from at least one vulnerability. We have identified several leakages in symmetric and asymmetric ciphers, and reported them to the respective vendors. Our analysis shows that despite the existing efforts on protecting these implementations, some of them still suffer from security-critical leakages.

8.1 Future Work

8.1.1 Coverage-based Fuzzing

We use random test cases to get a uniform random distribution of potential memory accesses and execution paths; while this works well with cryptographic implementation, it would not scale to targets such as protocols or data structures. Coverage-based *Fuzzing* [60] is a technique to generate test cases with the aim of achieving maximum code coverage; while it was originally developed to find software bugs, e.g., memory

corruption, the same approach can be applied for finding side-channel leakages, e.g., leakage in the *JPEG* library [82]. We have already implemented an experimental support for using WinAFL [35] as a test case generator; in that setting AFL helps to generate samples with higher coverage, while at the same time the test cases are sent to our framework for further processing. It is desirable to enhance this experimental feature and apply it to non-cryptographic implementations that are critical in terms of side-channel security.

8.1.2 Distinguishing leakages in call graph

We observed that in some cases control flow leakages in the higher level algorithm residing at the top of the call chain hide leakages in the subroutines invoked in deeper levels. Also, if separate functions use a common subroutine, a positive MI result in this subroutine can not easily be assigned to its root cause. We therefore propose to add an option to *MicroWalk* to take the call graph into account when computing mutual information.

Responsible Disclosure

We have informed the *Intel Product Security Incident Response Team (PSIRT)* and *Microsoft Security Response Center (MSRC)* of our findings. MSRC has not responded. After the initial report, we noticed that Intel have already patched `gfec_MulBasePoint` in Intel IPP v2018.3.240. Intel have acknowledged the receipt for the remaining vulnerabilities. Here is the time line for the responsible disclosure:

- **06/22/2018:** We informed our findings to the Intel Product Security Incident Response Team (Intel PSIRT) and the Microsoft Security Response Center.
- **06/25/2018:** Intel PSIRT acknowledged the receipt.
- **07/31/2018:** Intel PSIRT confirmed a work-in-progress patch for IPP 2018 update 4 (CVE-2018-12155, CVE-2018-12156).

Acknowledgements

This work is supported by the National Science Foundation, under grant CNS-1618837.

References

- [1] Onur Aciicmez, Billy Bob Brumley, and Philipp Grabher. “New Results on Instruction Cache Attacks”. In: *12th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2010. DOI: 10.1007/978-3-642-15031-9_8.
- [2] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *2nd ACM Symposium on Information, Computer and Communications Security*. Singapore, 2007. ISBN: 1595935746. DOI: 10.1145/1229285.1266999.
- [3] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “Predicting Secret Keys Via Branch Prediction”. In: *The Cryptographers’ Track at the RSA Conference 2007 (CT-RSA)*. 2007. DOI: 10.1007/11967668_15.
- [4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>.
- [5] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. “Formal verification of side-channel countermeasures using self-composition”. In: *Sci. Comput. Program.* 78.7 (2013), pp. 796–812. DOI: 10.1016/j.scico.2011.10.008.
- [6] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. “An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse>.
- [7] ARM. *Cortex-M3 Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf. (Visited on 2024-05-21).
- [8] L. Bai, Y. Zhang, and G. Yang. “SM2 cryptographic algorithm based on discrete logarithm problem and prospect”. In: *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*. 2012-04.
- [9] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. “System-level Non-interference for Constant-time Cryptography”. In: *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014. DOI: 10.1145/2660267.2660283.
- [10] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. “Automatic Application of Power Analysis Countermeasures”. In: *IEEE Trans. Computers* 64.2 (2015), pp. 329–341. DOI: 10.1109/TC.2013.219.

- [11] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. "Finding Software Vulnerabilities by Smart Fuzzing". In: *Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2011. DOI: 10.1109/ICST.2011.48.
- [12] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "'Ooh Aah... Just a Little Bit' : A Small Amount of Side Channel Can Go a Long Way". In: *16th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2014. DOI: 10.1007/978-3-662-44709-3_5.
- [13] Daniel J Bernstein. *Cache-timing attacks on AES*. 2005.
- [14] Sandrine Blazy, David Pichardie, and Alix Trieu. "Verifying Constant-Time Implementations by Abstract Interpretation". In: *22nd European Symposium on Research in Computer Security (ESORICS)*. 2017. DOI: 10.1007/978-3-319-66402-6_16.
- [15] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *18th Annual International Cryptology Conference (CRYPTO)*. 1998. DOI: 10.1007/BFb0055716.
- [16] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. "Vale: Verifying High-Performance Cryptographic Assembly Code". In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond>.
- [17] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. "Mitigating cache/ timing based side-channels in AES and RSA software implementations". In: *RSA Conference 2006 session DEV-203*. 2006.
- [18] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. "CacheShield: Detecting Cache Attacks through Self-Observation". In: *Eighth ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2018. DOI: 10.1145/3176258.3176320.
- [19] David Brumley and Dan Boneh. "Remote timing attacks are practical". In: *Comput. Networks* 48.5 (2005), pp. 701–716. DOI: 10.1016/j.comnet.2005.01.010.
- [20] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck>.

- [21] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. “FaCT: A Flexible, Constant-Time Programming Language”. In: *IEEE Cybersecurity Development (SecDev)s*. 2017. DOI: 10.1109/SecDev.2017.24.
- [22] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. “Quantifying the information leak in cache attacks via symbolic execution”. In: *15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2017. DOI: 10.1145/3127041.3127044.
- [23] Jean-Sébastien Coron, Paul C. Kocher, and David Naccache. “Statistics and Secret Leakage”. In: *4th International Financial Cryptography Conference (FC)*. 2000. DOI: 10.1007/3-540-45472-1_12.
- [24] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan>.
- [25] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. “CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 171–191. DOI: 10.13154/tches.v2018.i2.171-191.
- [26] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen>.
- [27] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *22nd USENIX Security Symposium*. 2013. URL: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev>.
- [28] Goran Doychev and Boris Köpf. “Rigorous analysis of software countermeasures against cache attacks”. In: *38th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2017. DOI: 10.1145/3062341.3062388.
- [29] *DynamoRIO: Dynamic Instrumentation Tool Platform*. <https://dynamorio.org/>.
- [30] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016. DOI: 10.1109/MICRO.2016.7783743.

- [31] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. ““Make Sure DSA Signing Exponentiations Really are Constant-Time””. In: *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016. DOI: 10.1145/2976749.2978420.
- [32] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *J. Cryptogr. Eng.* 8.1 (2018), pp. 1–27. DOI: 10.1007/s13389-016-0141-6.
- [33] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. “Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation”. In: *17th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2015. DOI: 10.1007/978-3-662-48324-4_11.
- [34] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. “Mutual Information Analysis”. In: *10th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2008. DOI: 10.1007/978-3-540-85053-3_27.
- [35] Ivan Fratric. *WinAFL*. <https://github.com/ivanfratric/win afl>. (Visited on 2024-05-21).
- [36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *24th USENIX Security Symposium*. 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>.
- [37] Silviu Guiaşu. *Information theory with new applications*. McGraw-Hill Companies, 1977.
- [38] Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. “PerfWeb: How to Violate Web Privacy with Hardware Performance Events”. In: *22nd European Symposium on Research in Computer Security (ESORICS)*. 2017. DOI: 10.1007/978-3-319-66399-9_5.
- [39] JaeCheol Ha and Sang-Jae Moon. “A Common-Multiplicand Method to the Montgomery Algorithm for Speeding up Exponentiation”. In: *Inf. Process. Lett.* 66.2 (1998), pp. 105–107. DOI: 10.1016/S0020-0190(98)00031-3.
- [40] Mike Hamburg. “Accelerating AES with Vector Permute Instructions”. In: *11th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2009. DOI: 10.1007/978-3-642-04138-9_2.
- [41] Intel. *Symmetric Cryptography Primitive Functions*. <https://intel.ly/2xwNvCM>. (Visited on 2018-02-27).
- [42] Intel. *Understanding CPU Dispatching in the Intel® IPP Libraries*. <https://intel.ly/2QAcQo6>. (Visited on 2018-02-27).

- [43] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. "Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries". In: *CoRR abs/1709.01552* (2017). arXiv: 1709.01552. URL: <http://arxiv.org/abs/1709.01552>.
- [44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES". In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.42.
- [45] Don Johnson, Alfred Menezes, and Scott A. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Int. J. Inf. Sec.* 1.1 (2001), pp. 36–63. DOI: 10.1007/s102070100002.
- [46] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. "When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015". In: *15th International Cryptology and Network Security Conference (CANS)*. 2016. DOI: 10.1007/978-3-319-48965-0_36.
- [47] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael B. Abu-Ghazaleh, Dmitry V. Ponomarev, and Aamer Jaleel. "RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks". In: *54th Annual Design Automation Conference (DAC)*. 2017. DOI: 10.1145/3061639.3062313.
- [48] S McCURLEY Kevin. "The discrete logarithm problem". In: *Cryptology and computational number theory* 42 (1990), p. 49.
- [49] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00002.
- [50] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *16th Annual International Cryptology Conference (CRYPTO)*. 1996. DOI: 10.1007/3-540-68697-5_9.
- [51] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. "Automatic Quantification of Cache Side-Channels". In: *24th Computer Aided Verification Conference (CAV)*. 2012. DOI: 10.1007/978-3-642-31424-7_40.
- [52] A Langley. *ctgrind: Checking that functions are constant time with Valgrind*. 2010.

- [53] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Life-long Program Analysis & Transformation”. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*. 2004. DOI: 10.1109/CGO.2004.1281665.
- [54] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. “Practical Keystroke Timing Attacks in Sandboxed JavaScript”. In: *22nd European Symposium on Research in Computer Security (ESORICS)*. 2017. DOI: 10.1007/978-3-319-66399-9_11.
- [55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp>.
- [56] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. “CATalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016. DOI: 10.1109/HPCA.2016.7446082.
- [57] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.43.
- [58] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *2005 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2005. DOI: 10.1145/1065010.1065034.
- [59] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9.
- [60] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. *Fuzzing: The State of the Art*. <https://apps.dtic.mil/sti/citations/ADA558209>. 2012. (Visited on 2024-05-21).
- [61] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *19th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2017. DOI: 10.1007/978-3-319-66787-4_4.
- [62] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *Int. J. Parallel Program.* 47.4 (2019), pp. 538–570. DOI: 10.1007/s10766-018-0611-9.

- [63] Nicholas Nethercote. “Dynamic binary analysis and instrumentation: or building tools is easy”. PhD thesis. University of Cambridge, UK, 2004. URL: <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.616254> (visited on 2024-05-21).
- [64] Kaisa Nyberg and Rainer A. Rueppel. “A New Signature Scheme Based on the DSA Giving Message Recovery”. In: *1993 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1993. DOI: 10.1145/168588.168595.
- [65] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *The Cryptographers’ Track at the RSA Conference 2006 (CT-RSA)*. 2006. DOI: 10.1007/11605805_1.
- [66] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. “Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT”. In: *29th IEEE Computer Security Foundations Symposium (CSF)*. 2016. DOI: 10.1109/CSF.2016.34.
- [67] Colin Percival. *Cache missing for fun and profit*. 2005.
- [68] NIST FIPS PUB. *Digital signature standard*. 1993.
- [69] Ashay Rane, Calvin Lin, and Mohit Tiwari. “Raccoon: Closing Digital Side-Channels through Obfuscated Execution”. In: *24th USENIX Security Symposium*. 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane>.
- [70] Intel. *Intel Digital Random Number Generator (DRNG) Software Implementation Guide*. <https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html>. (Visited on 2024-05-21).
- [71] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time?”. In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017. DOI: 10.23919/DATE.2017.7927267.
- [72] Laurent Simon, David Chisnall, and Ross J. Anderson. “What You Get is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018. DOI: 10.1109/EuroSP.2018.00009.
- [73] Rohit Sinha, Sriram K. Rajamani, and Sanjit A. Seshia. “A compiler and verifier for page access oblivious computation”. In: *2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2017. DOI: 10.1145/3106237.3106248.
- [74] Geoffrey Smith. “On the Foundations of Quantitative Information Flow”. In: *12th International Foundations of Software Science and Computational Structures Conference (FOSSACS)*. 2009. DOI: 10.1007/978-3-642-00596-1_21.

- [75] François-Xavier Standaert, Tal Malkin, and Moti Yung. “A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks”. In: *28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 2009. DOI: 10.1007/978-3-642-01001-9_26.
- [76] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *J. ACM* 65.4 (2018), 18:1–18:26. DOI: 10.1145/3177872.
- [77] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. “CachedD: Identifying Cache-Based Timing Channels in Production Software”. In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>.
- [78] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134038.
- [79] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. “DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [80] Bernard L Welch. “The generalization of student’s’ problem when several different population variances are involved”. In: *Biometrika* 34.1/2 (1947), pp. 28–35.
- [81] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. “STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves”. In: *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134016.
- [82] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.45.
- [83] Yuval Yarom and Naomi Benger. “Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack”. In: *IACR Cryptol. ePrint Arch.* (2014), p. 140. URL: <http://eprint.iacr.org/2014/140>.

- [84] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *18th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2016. DOI: 10.1007/978-3-662-53140-2_17.
- [85] Andreas Zankl, Johann Heyszl, and Georg Sigl. "Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software". In: *15th International Conference on Smart Card Research and Advanced Applications (CARDIS)*. 2016. DOI: 10.1007/978-3-319-54669-8_14.
- [86] Tianwei Zhang and Ruby B. Lee. "New models of cache architectures characterizing information leakage from cache side channels". In: *30th Annual Computer Security Applications Conference (ACSAC)*. 2014. DOI: 10.1145/2664243.2664273.
- [87] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds". In: *19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2016. DOI: 10.1007/978-3-319-45719-2_6.

Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications

Publication

Jan Wichelmann, Florian Sieck, Anna Pätchke, and Thomas Eisenbarth. *Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications*. In *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

Contribution

Main author.

Outline

1	Introduction	138
2	Background	141
3	A Fast Leakage Analysis Algorithm	144
4	JavaScript Leakage Analysis	154
5	Integration into Development Workflow	157
6	Evaluation and Discussion	160
7	Related Work	168
8	Conclusion	169
	References	170

Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications

Jan Wichelmann, Florian Sieck, Anna Päsche, and Thomas Eisenbarth

Universität zu Lübeck

Secret-dependent timing behavior in cryptographic implementations has resulted in exploitable vulnerabilities, undermining their security. Over the years, numerous tools to automatically detect timing leakage or even to prove their absence have been proposed. However, a recent study at IEEE S&P 2022 showed that, while many developers are aware of one or more analysis tools, they have major difficulties integrating these into their workflow, as existing tools are tedious to use and mapping discovered leakages to their originating code segments requires expert knowledge. In addition, existing tools focus on compiled languages like C, or analyze binaries, while the industry and open-source community moved to interpreted languages, most notably JavaScript.

In this work, we introduce Microwalk-CI, a novel side-channel analysis framework for easy integration into a JavaScript development workflow. First, we extend existing dynamic approaches with a new analysis algorithm, that allows efficient localization and quantification of leakages, making it suitable for use in practical development. We then present a technique for generating execution traces from JavaScript applications, which can be further analyzed with our and other algorithms originally designed for binary analysis. Finally, we discuss how Microwalk-CI can be integrated into a continuous integration (CI) pipeline for efficient and ongoing monitoring. We evaluate our analysis framework by conducting a thorough evaluation of several popular JavaScript cryptographic libraries, and uncover a number of critical leakages.

1 Introduction

Collection of sensitive data is common in today's cloud and Internet of Things (IoT) environments, and affects everyone. Protecting this private and sensitive data is of utmost importance, therefore requiring secure cryptography routines and secrets. However, especially the cloud allows attackers to observe the execution of victim code using side-channels in co-located environments [26]. These attacks range from Last Level Cache (LLC) [35] and de-duplication attacks [34] to the observation of memory access patterns [64] or main memory access contention [44]. The spatial resolution depends on

the granularity of the attacked buffer and the temporal resolution on the capabilities of the attacker, meaning either the ability to achieve a sufficiently high measurement frequency [65] or to interrupt and pause the victim code [14, 64].

To avoid side-channel vulnerabilities, programmers should write constant-time code, i.e., software which does not contain input or secret-dependent control flow or memory accesses. Depending on the problem at hand, this can be achieved by different means: For example, a secret-dependent data access may be replaced by accessing every element of the target array and then choosing the correct one with a mask. Conditionals can be adjusted by always executing both branches and then selecting the result.

However, for complex projects like large cryptographic libraries, finding such vulnerabilities is a difficult and time-intensive task. Thus, the research community has developed a number of analysis strategies [5, 13, 16, 17, 33, 46, 60, 62, 63], that aim at automating the detection of side-channel leakages in a given code base. However, a recent study [29] that conducted a survey between crypto library developers found that while most developers were aware of and welcome those tools, they had major difficulties using them due to bad usability, lack of availability and maintenance or high resource consumption. The authors worked out a number of recommendations for creators of analysis tools: The tools should be well-documented and easily usable, such that adoption requires low effort from the developer. Another focus is on compatibility: The analysis shouldn't require use of special languages or language subsets. Finally, the tools should aid *efficient* development, i.e., quickly yield results with less focus on completeness, making them suitable for inclusion in a continuous integration (CI) workflow.

In this work, we study how these challenges can be addressed, and adapt the existing Microwalk [63] framework to fit the given objectives. Microwalk was originally designed for finding leakages in binary software, for which it generates a number of execution traces for a set of random inputs and then compares them. The dynamic analysis approach of Microwalk is quite fast, as it does only run the target program several times, and then compares the resulting execution traces with a simple linear algorithm. However, due to the simplistic leakage quantification, the resulting analysis reports contain a lot of potential vulnerabilities, with little or even misleading information about their cause. This makes it difficult to assess their severity and address them efficiently, especially for complex libraries. Finally, the initial setup can be time-consuming, as the different components need to be compiled from source.

We mitigate these issues by designing Microwalk-CI, which features a new leakage analysis algorithm that combines the performance benefits of dynamic analysis with an accurate leakage localization and quantification, easing the assessment and investigation of the reported leakages. In addition, we add support for running Microwalk-CI in an automated environment like a CI pipeline, and create a Docker image that contains

Microwalk-CI and its dependencies for easy use. Finally, we create simple templates that allow quick adoption of Microwalk-CI's analysis capabilities into a cryptographic library's CI workflow with little effort by the developer.

During the research on leakage detection tools, as part of their evaluation, many vulnerabilities in popular cryptographic libraries have been uncovered and fixed. However, the developer community is moving away from compiled languages like C or C++ and instead embraces interpreted scripting languages like JavaScript or Python. In fact, the 2021 Stack Overflow developer survey and the January 2021 Redmonk programming language ranking found that those two languages are the most popular, both for private and professional contexts [45, 56]. JavaScript was originally designed as a client-side language for web browsers, but, with the arrival of Node.js [40], it has seen growing adoption for server-side software as well. Consequently, the community has come up with a number of cryptographic libraries written in pure JavaScript. However, due to the lack of appropriate tooling and attention of the research community, these libraries have never been vetted for their robustness against side-channel attacks, which is worrying given the fact that the servers using them may be hosted in IaaS cloud environments.

To address this, Microwalk-CI offers a novel method for applying Microwalk's original binary analysis algorithms to JavaScript libraries by using the Jalangi2 [48, 55] source code instrumentation library to generate compatible traces. The new tracing backend comes with a simple code template and supports full automation, such that the analysis can be easily added to the CI workflows of respective libraries. We evaluate several popular JavaScript cryptographic libraries, uncovering a number of high-severity leakages.

By supporting the analysis of JavaScript, we strive to improve the security of software and raise awareness for the importance of constant-time cryptographic code in the community of web and cloud developers. The underlying concepts of our source-based trace generator can be used for building analysis support for other programming languages as well, making side-channel leakage analysis available for all common platforms and at a low barrier.

1.1 Our Contribution

In summary, we make the following contributions:

- We introduce a novel call tree-based analysis method, which allows efficient and accurate localization and quantification of leakages.
- We show the first dynamic leakage analysis tool for JavaScript libraries.

- We propose a new approach for integrating a fully automated timing leakage analysis into the crypto library development workflow, which requires low effort from the developer and immediately reports newly introduced vulnerabilities.
- We evaluate the new analysis framework with several widely-used JavaScript libraries and uncover a significant number of previously unaddressed leakages.

The source code of Microwalk-CI is available at <https://github.com/microwalk-project/Microwalk>.

1.2 Disclosure

We contacted the authors of the affected libraries, informed them about our findings and offered to aid in fixing the vulnerabilities.

The author of `elliptic` acknowledged the discovered vulnerabilities, but noted that the package is no longer maintained and that fixing the vulnerabilities would require major changes, as side-channel resistance wasn't part of the underlying design considerations. There was no response from the other library authors.

2 Background

2.1 Microarchitectural Timing Attacks

Implementations of cryptographic algorithms are often run on hardware resources that are shared between different processes. If the code exhibits secret-dependent behavior, malicious processes can use the resulting information leakage to extract secrets like private keys through side-channel analysis. Cache attacks are a prominent example for exploiting resource contention with a victim process: By measuring the time it takes for repeatedly clearing and accessing a specific cache entry, the attacker can see whether the victim accessed a similar cache entry in the meantime [1, 10, 42, 53, 65]. Other attack vectors include the translation lookaside buffer (TLB) [22] and the branch prediction unit [2].

The most widely used software countermeasure against these attacks is writing constant-time code that does not contain secret-dependent memory accesses or branches, and that uses instructions that do not come along with operand-dependent runtime [12]. There exists a variety of tools [5, 13, 16, 17, 33, 46, 60, 62, 63], that feature different analysis approaches. Some of these tools are open-source, with varying performance and usability [29].

2.2 Microwalk

Microwalk [63] is a framework for checking the constant-time properties of software binaries in an automated fashion. It follows a dynamic analysis approach, i.e., it executes the target program with a number of random inputs and collects execution traces, which contain branch targets, memory allocations and memory accesses. This is done through a three-stage pipeline, where traces are generated, preprocessed, and analyzed. Each stage has various *modules*, which are chosen by the user depending on their application. Furthermore, Microwalk has a plugin architecture, that allows easy extension by loading custom modules.

Currently, Microwalk only has one trace generation module, which is based on Intel Pin [27] and produces traces for binary software. Correspondingly, there is a preprocessor module that converts the raw traces generated by Pin into Microwalk's own format. Finally, these preprocessed traces can be fed into a number of analysis modules, e.g., for computing the mutual information between memory access patterns and inputs, or for dumping the preprocessed traces in a human-readable format.

2.3 Mutual Information and Guessing Entropy

Mutual information (MI) quantifies the interdependence of two random variables, i.e., it models how much information an attacker can learn about one variable on average by observing the other one [23]. It has been widely used for quantifying side-channel leakages [9, 28, 63, 67].

The mutual information of the random variables $X: K \rightarrow \mathcal{X}$ and $Y: L \rightarrow \mathcal{Y}$ is defined as

$$I(X, Y) = \sum_{\substack{x \in \mathcal{X} \\ y \in \mathcal{Y}}} \Pr[X = x, Y = y] \cdot \log_2 \left(\frac{\Pr[X = x, Y = y]}{\Pr[X = x] \cdot \Pr[Y = y]} \right).$$

The information is measured in bits. In our setting, the random variable X represents a secret and Y the information that can be gathered by observing the system state through a side-channel.

The guessing entropy (GE) of a random variable $X: K \rightarrow \mathcal{X}$ quantifies the average number of guesses that have to be made in order to guess the value of X correctly [32]. If \mathcal{X} is indexed such that $\Pr[X = x_i] \geq \Pr[X = x_j]$ for $x_i, x_j \in \mathcal{X}$ and $i \leq j$, the guessing

entropy is defined as

$$G(X) = \sum_{1 \leq i \leq |\mathcal{X}|} i \cdot \Pr[X = x_i].$$

The *conditional guessing entropy* (conditional GE) $G(X | Y)$ for random variables X and Y is defined as

$$G(X | Y) = \sum_{y \in \mathcal{Y}} \Pr[Y = y] \cdot G(X | Y = y).$$

$G(X | Y)$ measures the expected number of guesses that are needed to determine the value of X for a known value of Y .

A variant of the conditional GE, the *minimal conditional guessing entropy* (minimal GE), determines the lower bound of expected guesses. It is defined as

$$\hat{G}(X | Y) = \min_{y \in \mathcal{Y}} G(X | Y = y),$$

i.e., it outputs the minimal number of guesses that are needed to find out one of the possible values of X .

2.4 JavaScript Instrumentation

JavaScript code can be instrumented in different ways, each coming with their own benefits and drawbacks.

FoxHound [49] modifies Firefox's JavaScript engine. While this allows many optimizations, it comes with the downside of being constrained to one specific JavaScript engine and requiring constant maintenance to keep up with the upstream project. OpenTelemetry [41] and Google's tracing framework [21] create program traces to monitor and profile software, but require the developer to insert instrumentation calls into their source code manually. While being very specific and thus only introducing the necessary overhead, they are not generally applicable without a lot of manual effort.

Lastly, the JavaScript code can be dynamically instrumented in a source-to-source fashion. Jalangi2 [48, 52, 55] wraps the loading process of JavaScript files and injects instrumentation code into the source code. The user of the instrumentation framework can write and register custom callback routines, which are supplied with the current execution state. This approach comes with a certain overhead, but it is flexible and works with arbitrary JavaScript code without manual adjustments.

3 A Fast Leakage Analysis Algorithm

We propose a new leakage analysis algorithm that is optimized for quickly delivering detailed leakage information, aiding developers in efficiently locating and fixing issues. Before we dive into the algorithm, we define the leakage model and discuss the objectives a thorough leakage analysis must meet. Then, we describe how the traces are processed to build a call tree, which in a final step is broken down to compute leakage metrics for specific instructions.

3.1 Leakage Model

To ensure that we detect all leakages which may be exploited by current and future attack methods, we choose a strong leakage model: An attacker tries to extract secret inputs from an implementation through a side-channel attack, which allows them to get a trace of all executed instructions and all accessed memory addresses. They also have access to all public inputs and outputs.

Under certain conditions, a hypervisor/OS-level adversary can single step instructions [4, 38, 54], or have below cache-line resolution [37, 66]. However, for more relaxed adversarial scenarios like cross-VM attacks, granularities of 32 or 64 bytes and hundreds of instructions may be more appropriate. Adjusting the processing of the leakage accordingly allows an analysis under such a leakage model as well, but, we believe that the most conservative approach should be applied, i.e., assuming a maximum resolution attacker. Attacks exploiting speculative execution are considered off-scope, as we focus on leakages caused by actual secret-dependent control flow or memory accesses, i.e., code paths that are reached architecturally.

This leakage model and the following analysis approach are consistent with the models used by Microwalk [63] and DATA [62].

3.1.1 Analysis approach

The leakage model can be turned into a dynamic analysis approach by making the following observation: Since the attacker tries to infer a secret solely by looking at an execution trace and public inputs/outputs, they can only succeed if the trace depends on the secret. I.e., if changing the secret does never influence the observed trace, the implementation does not leak the secret and is *constant-time*.

We model this by giving the attacker a number of secret inputs and corresponding execution traces, and asking them to map the inputs to the respective traces. If they

```

1  int func(int secret) {                // func+0
2      lookup(secret);                  // func+1
3
4      int result = 0;
5      for(int i = 0; i < secret; ++i) { // func+4
6          result += lookup(1);          // func+5
7      }                                // func+6
8      return result;                   // func+7
9  }
10 int table[] { ... };
11 int lookup(int index) {               // lookup+0
12     return table[index];              // lookup+1
13 }

```

Figure 1: A sample program illustrating different kinds of leakages: The `lookup` function is not constant-time, since it does an input-based array lookup, so the memory access to `table[index]` would be marked as leaking if `index` is secret. Another cause of leakage is in `func`, which calls `lookup` a varying number of times depending on a secret value.

perform better than guessing, we consider the implementation as leaking. If all traces are identical, the implementation is considered constant-time.

3.2 Objectives

For an efficient and useful dynamic leakage analysis, we identified three major objectives: Accurate *localization* of leakages, a *quantification* of leakage severity, and *performance*.

Localization While varying address traces for a memory read instruction are a clear sign that there *is* leakage, which can be extracted by monitoring that particular instruction [63], they do not indicate where the leakage is actually *caused*. E.g., a non-constant-time function may be called two times, once with a secret-dependent parameter, and once a varying number of times in a loop, but with a constant parameter (Figure 1). A correct analysis should distinguish the two invocations of `lookup` and mark the table access in line 12 as leaking for the first invocation (line 2); for the second invocation (line 6), the secret-dependent branch in line 5 should be reported, as the table access in line 12 itself does not add any leakage.

Quantification In addition to an accurate localization, there is a need for a rough quantification of the severity of leakages. For example, a chain of nested if statements may only leak a few bits of the secret each, but the leakage aggregates up to a point which allows an attacker to easily distinguish different secrets just by looking at the resulting sequence of branch instructions. At the same time, a lone if statement which merely handles a special case during key file parsing (e.g., whether a parameter has some additional byte) does not necessarily pose an urgent problem. The analysis should assign each leakage with a score allowing the developer to prioritize between findings.

Performance Finally, for integrating the leakage analysis into a development workflow, performance is important: When checking whether a proposed change impacts security, or whether a given patch fixes a previously discovered leakage, the developer should not need to wait several ten minutes or hours until analysis results are available. The analysis should be efficient enough to run it both on a standard developer machine and in a hosted CI environment.

3.3 Algorithm Idea

In order to find leakages, we need to compare the generated traces, and find sections where they diverge and, later, merge again. However, due to the performance requirements and the immense size of traces, especially for asymmetric primitives, we cannot afford running a traditional diff or trace alignment algorithm, which usually have quadratic complexity. At the same time, we do not want to lose information, as we want to accurately pinpoint the detected leakages. Thus, we opt for a data structure that preserves all necessary information in an efficient way, and which allows to conduct a thorough leakage analysis which can discover and quantify trace divergences in linear time.

For that, we merge the traces into a call tree, where each function call and a few other trace entries form the nodes, and where subsequent function calls *and trace divergences* generate branches. Each node holds the IDs of the traces which reach that node. The tree can be built on-the-fly while the traces are processed, so it can be integrated into a leakage analysis pipeline like the one offered by Microwalk. After the traces have been processed, a final step traverses the tree and evaluates for each instruction in each call stack, whether it caused a divergence and how severe that divergence is. In the following sections, we elaborate on the respective steps.

3.4 Step 1: Building the Call Tree

We merge the traces into a tree in a greedy way, i.e., we simultaneously iterate over a trace and the current tree entries, and add the trace entries to the tree. In order to save memory and get a readable representation of the traces with little tree depth, we can exploit the fact that traces of constant-time implementations tend to have long shared sequences without any differences, and thus use a radix trie instead of a plain tree, such that each node holds an as long as possible sequence of consecutive trace entries.

3.4.1 Types of trace entries

In order to address the leakage model, the execution traces used by Microwalk contain information about branches, memory allocations and memory accesses.

Branches cover call, return and jump instructions. A *branch* trace entry has a source address, a target address and a *taken* bit that denotes whether the branch was taken or skipped (e.g., due to a failed comparison). The source and target addresses consist each of an image ID (i.e., the binary which contains the corresponding instruction) and an offset.

Memory allocations are used to keep track of memory blocks on the heap and stack. Each time the analyzed program calls *malloc* or a similar function, a new allocation block is registered with a unique ID and the allocation block size.

Memory accesses contain the image ID and offset of the corresponding instruction and the allocation block ID and offset of the accessed address. This relative addressing allows to compare traces even when they each operate on their own allocated memory regions, which have different absolute addresses.

3.4.2 Tree layout

As mentioned above, we chose a radix trie-like representation of the merged trace entries, as this reduces tree depth, speeds up analysis and enhances readability of tree dumps.

A tree node consists of two parts: The consecutive trace entries which are present for all traces hitting this node, and a list of (edges to) *split nodes* (Figure 2), which represent divergences between the different traces. The trace entry list may contain *call nodes* (Figure 3) which open their own sub tree, but always return back into the current node and may be followed by other trace entries.

Edges start from within the trace entry list (for calls) or from the split node list. If an edge leads to a split node, it is annotated with the trace IDs taking this specific edge.

3.4.3 Inserting trace entries into the tree

The handling of equal and conflicting trace entries depends on the respective type. Split nodes are only created when a function call or a jump targets a different instruction than the already existing trace entry, as the resulting sub tree may be fairly different. Other differences like varying memory access offsets are only recorded in the respective trace entry, as they don't affect control flow and the current entry is thus likely followed by other, non-conflicting entries.

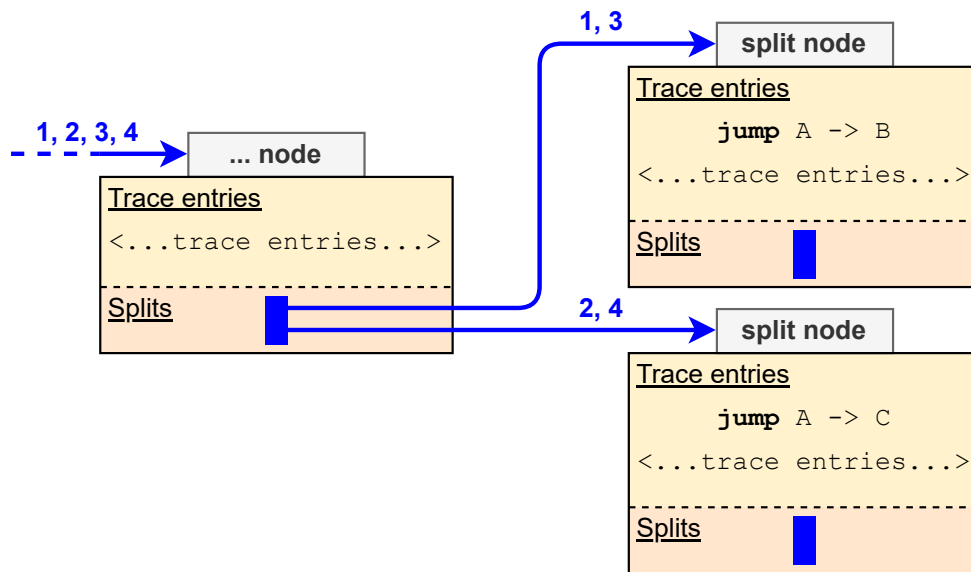


Figure 2: A generic trace divergence with two split nodes. While traces 1 to 4 share the entries in the left node, they differ at the jump statement at location A: Traces 1 and 3 jump to location B, while traces 2 and 4 jump to C. Here, each case gets its own split node, and processing of trace entries is resumed there.

A function call is handled by creating a new tree node at the current position in the list of consecutive trace entries of the current tree node. Afterwards, the current node is pushed onto a stack and the new call node is set as the current node, such that subsequent trace entries are stored in the new node. When encountering a return statement, the last node is popped from the stack, and insertion of trace entries is resumed after the earlier created call node. If the target address of the current call entry does not match the target address recorded in an existing call node, a split is triggered.

If a conflict between an existing and a new trace entry is detected, the algorithm generates two new split nodes: One node receives the original conflicting trace entry, the remaining consecutive trace entries and the split node list of the current node; the other node is initialized with the new conflicting trace entry and an empty split node list. The branches to both nodes are annotated with the corresponding trace IDs. The current node is then set to the new split node, such that the new trace entries end up in the new node. The call node stack is *not* updated, i.e., the next return statement ends the divergence and restores the state before the last call node. This way, we can recover from a trace divergence and discover additional leakages in other function calls.

Cases where there are more than two possible targets for an instruction (e.g., an indirect jump) are handled appropriately, by generating further split nodes at the same level.

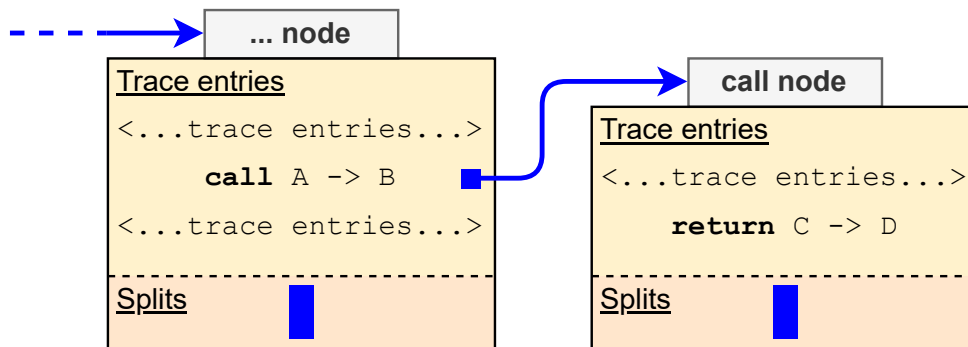


Figure 3: A generic function call with a call node. When a function call entry is encountered, a new call node is created, that subsequently receives the trace entries for the given function. Once the function ends (return statement), the trace entry list of the prior call node is continued. Note that the return statement may also end up in the split node tree of the call node, if there are trace divergences within the function.

3.5 Step 2: Leakage Analysis

After trace processing has concluded, we have a call tree that encodes the similarities and differences of all traces. We now perform a final step that collects this information and computes leakage measures, such that we can assign leakage information to each instruction, meeting our localization and quantification objectives.

3.5.1 Building call stacks with trace ID trees per instruction

First, we consolidate the call tree into a number of call stacks, and store the trace split information for each instruction in the corresponding call stack. The split information consists of *trace ID trees*, which encode how multiple executions of the given instruction for a certain function invocation led to trace divergence. This greatly simplifies the computation of leakage measures for individual instructions, and allows to display expressive information about the leakage behavior of a given instruction to the developer. If a function is called multiple times (i.e., the same call stack occurs repeatedly), additional trace ID trees are created (no merging).

When a split is encountered, new child nodes for each edge of the split are added to the trace ID tree for the responsible instruction. Figure 4 illustrates the resulting trace ID tree for a simple program counting bits in a secret variable: When the jump instruction in question is encountered first, all traces are identical (tree level 0). At that point, execution diverges for traces with an even versus an odd secret. After the second iteration, traces are again split depending on the second bit of the secret. In the end, there are four different possible traces for the given function call.

```

void f1(int secret) { f2(secret); } // f1+0
void f2(int secret) { f3(secret); } // f2+0
void f3(int secret) {           // f3+0
    int tmp = 0;
    for(int i = 0; i < 2; ++i) { // f3+2
        if(secret & (1 << i)) { // f3+3
            ++tmp;
        }
    } // f3+6
}

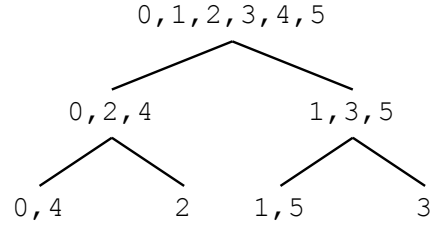
```

(a) Program

Call stack:
main+X -> f1+0
f1+0 -> f2+0
f2+0 -> f3+0

Instructions:
jump at f3+2
jump at f3+3
jump at f3+6

(b) Call stack and instruction info



(c) Trace ID tree for jump at f3+3

Figure 4: Example for call stack and trace ID generation. The program in (a) counts the number of 1s in the two least-significant bits of a secret variable by repeatedly executing a secret-dependent if statement. When calling f1 from main with secret values from 0 (trace ID 0) to 5 (trace ID 5), we get the call stack as shown in (b), with three detected jump instructions. The secret-dependent jump at f3+3 leads to divergence of traces, as is visible in the resulting trace ID tree in (c). Traces sharing a tree node at tree level $h \geq 0$ are identical for at least h consecutive executions of the instruction.

3.5.2 Computing leakage measures

After recording the divergence behavior of instructions per call stack, we can compute various measures to quantify the corresponding leakage. We feature three efficiently computable metrics that give the developer an indication of the severity of each detected leakage: Mutual information, conditional guessing entropy and minimal conditional guessing entropy. If the function containing the analyzed instruction is invoked multiple times for the same call stack and thus produces multiple trace ID trees, the algorithm computes the metrics for each tree separately and outputs the mean, minimum, maximum, and standard deviation for each metric.

All metrics depend on the size of leaves in the trace ID tree. For n traces, let $T = \{0, 1, \dots, n-1\}$ be the set of trace IDs. The set of leaves L for a given trace ID tree is then defined as $L = \{L_i \mid L_i \subseteq T \wedge L_i \neq \emptyset\}$ with $L_1 \cup L_2 \cup \dots \cup L_\ell = T$ and $L_i \cap L_j = \emptyset$ for $L_i, L_j \in T$ and $i \neq j$. This can be read as the tree having ℓ leaves L_i ($i = 1, \dots, \ell$), where each L_i holds the trace IDs ending up in this particular leaf. Those traces are considered identical.

Let $X: T \rightarrow \mathbb{N}$ be a random variable for picking a trace ID. The trace IDs are uniformly distributed, hence $\Pr[X = k] = \frac{1}{n}$ for each $k = 1, \dots, n$. Let $Y: L \rightarrow \mathbb{N}$ be a random variable for observing a particular trace, with $\Pr[Y = i] = \frac{|L_i|}{|T|} = \frac{|L_i|}{n}$ for $i = 1, \dots, \ell$.

Mutual information measures the *average amount of information* an attacker learns when observing a trace.

The MI of the trace ID X and the observed trace Y is defined as

$$I(X, Y) = \sum_{k=1}^{|T|} \sum_{i=1}^{|L|} \Pr[X = k, Y = i] \cdot \log_2 \left(\frac{\Pr[X = k, Y = i]}{\Pr[X = k] \cdot \Pr[Y = i]} \right).$$

With

$$\Pr[X = k, Y = i] = \begin{cases} 0 & \text{if } k \notin L_i \\ \frac{1}{n} & \text{if } k \in L_i \end{cases}$$

we get

$$I(X, Y) = \sum_{i=1}^{|L|} \frac{|L_i|}{n} \cdot \log_2 \left(\frac{\frac{1}{n}}{\frac{1}{n} \cdot \frac{|L_i|}{n}} \right) = \frac{1}{n} \sum_{i=1}^{\ell} |L_i| \cdot \log_2 \left(\frac{n}{|L_i|} \right).$$

The value of $I(X, Y)$ can be interpreted as bits: In the best case, there is only one leaf containing all trace IDs, such that the attacker learns nothing (0 bits). In the worst case, with one leaf for each trace ID, the attacker learns $\log_2(n)$ bits. The MI of the example in Figure 4 is $\frac{1}{6} (2 \cdot 2 \cdot \log_2(3) + 2 \cdot 1 \cdot \log_2(6)) \approx 1.33$ bits.

This metric has a few drawbacks: Due to its logarithmic nature, with an increasing number of traces it only grows slowly. Another shortcoming is the averaging, i.e., a high leakage in a few cases may get suppressed by the smaller leakage of all other cases. Finally, it may be mistakenly interpreted as additive due to its “bits” unit (i.e., 10 instructions leaking 3 bits each does not mean that there is a leakage of 30 bits). However, it does perform well for small and balanced leakages, e.g., when an instruction constantly divides the traces into two groups of similar size.

Conditional guessing entropy measures the *expected number of guesses* an attacker needs for associating a given trace with a secret input. The conditional GE $G(X | Y)$ for determining a trace ID, modeled as random variable X , for a known value of an observed

trace (random variable Y) is calculated as

$$\begin{aligned} G(X | Y) &= \sum_{i=1}^{|L|} \Pr[Y = i] \cdot G(X | Y = i) \\ &= \sum_{i=1}^{|L|} \Pr[Y = i] \cdot \sum_{k=1}^{|T|} k \cdot \Pr[X = k | Y = i]. \end{aligned} \quad (6.1)$$

Since

$$\Pr[X = k | Y = i] = \begin{cases} 0 & \text{if } k \notin L_i \\ \frac{1}{|L_i|} & \text{if } k \in L_i, \end{cases}$$

we can simplify (6.1) to

$$G(X | Y) = \sum_{i=1}^{\ell} \Pr[Y = i] \cdot \frac{1}{|L_i|} \sum_{k=1}^{|L_i|} k = \frac{1}{2n} \sum_{i=1}^{\ell} |L_i| \cdot (|L_i| + 1).$$

Note that the value of $G(X | Y)$ is upper-bounded by $\frac{n+1}{2}$, which is the best case where there is only one leaf which contains all trace IDs, i.e., all traces are identical. For the example in Figure 4, we get $G(X | Y) = \frac{1}{2 \cdot 6} (6 + 2 + 6 + 2) \approx 1.33$ guesses.

Small values for the conditional GE convey that an instruction sequence leads to almost unique traces, implying that there is widespread leakage affecting most to all traces. On the other side, a high value means that most traces are similar and do not leak much information. However, this being an average measure just like MI, there may well be special cases where there is a very high leakage. Those risk being obscured by this metric, thus we add an additional worst-case metric designed for catching these cases.

Minimal conditional guessing entropy measures the *minimal number of guesses* an attacker needs for associating a given trace with a secret input. It is calculated similarly to the conditional GE, but takes the minimum of all individual outcomes instead of weighting them:

$$\hat{G}(X | Y) = \min_{i=1, \dots, |L|} G(X | Y = i) = \min_{i=1, \dots, \ell} \frac{|L_i| + 1}{2}.$$

For the example in Figure 4, we get $\hat{G}(X | Y) = \min\{1.5, 1, 1.5, 1\} = 1$ guesses, i.e., there is at least one trace that is unique.

Minimal GE is the most definite leakage measure; it gives the number of guesses needed for the trace which leaks most. A high value for the minimal GE affirms that there is no outlier with high leakage. We thus recommend using this metric when evaluating the severity of a detected leakage.

3.5.3 Leakage severity and score

While the full analysis report provides detailed information about each leakage, we also seek to condense this information into a single, uniform score, such that the developer can quickly prioritize. That score should require little context: The developer should not need to be familiar with entropy, nor know analysis details like the particular number of test cases, which determines the upper bounds for the various metrics. Additionally, providing a single score allows easy integration of the leakage report into the user interface of modern development platforms like GitLab. The platform can then use that score for sorting and assigning a severity to the leakage.

We chose minimal GE for computing the leakage score, as it represents the worst-case leakage. Instead of reporting the minimal GE value directly, we map it onto a linear scale of 0 to 100, where 0 corresponds to a minimal GE of $\frac{n+1}{2}$ (i.e., no leakage), and 100 corresponds to a minimal GE of 1 (i.e., maximum leakage). If there are multiple trace ID trees for a given instruction (see Section 3.5.1), we show the mean and the standard deviation over the individual minimal GE values.

3.6 Implementation

We implemented the described algorithm as a new analysis module in Microwalk-CI's source tree. It integrates directly into the leakage analysis pipeline, i.e., it receives and handles preprocessed traces from the previous pipeline stage, the trace preprocessor. The tree is implemented as a recursive data structure, where each node holds a list of successor and split nodes. We do not store the consecutive non-diverging trace entries as a plain `ITraceEntry` list (as is suggested in the algorithm description), but as full-featured tree nodes as well. Apart from making the code more readable, this simplifies adding new divergences and storing temporary data for the final leakage analysis step, at the cost of additional memory overhead (we discuss this trade-off in Section 6.2).

Our implementation offers functionality for generating leakage reports and other detailed analysis result files optimized for readability, including an optional full call tree dump for debugging purposes. All features can be controlled via the Microwalk-CI configuration file infrastructure, allowing easy adoption of the new analysis module. In total, the module has 1,363 lines of C# code.

4 JavaScript Leakage Analysis

We now show how we can apply Microwalk-CI's generic analysis methods to JavaScript libraries, despite them being originally designed for binary analysis. First, we present a simple trace generator relying on the Jalangi2 instrumentation framework. Then, we show how these traces can be preprocessed such that they use the generic trace format from Microwalk-CI.

4.1 Instrumenting JavaScript code

Microwalk-CI expects multiple execution traces with varying secret input for the analyzed target function. These execution traces are then fed into various analysis modules for finding non-constant-time behavior, i.e. control flow or data flow-dependencies from secret input. A trace needs to contain the following information:

- Address and size of all loaded program modules (e.g., binaries or source files, called “images” internally);
- the control-flow of the analyzed program, encoded as a sequence of branch source and target addresses;
- address and size of all heap memory objects; and
- the instructions and target addresses of all memory accesses.

We translate this to JavaScript by collecting a trace of all executed code lines, and recording access offsets to any object or array. For instrumentation, we use Jalangi2 [48]. Jalangi2 instruments the code at load time by inserting callbacks before and after certain source tokens, e.g., conditionals, expressions or return statements.

First, we register the provided *SMemory* analysis module, which assigns a shadow object to each object, that contains a unique ID and the object value, allowing us to map accesses to known objects. We then create an own analysis front-end, called *tracer*, which registers some callbacks to record the necessary information and write it to a file for further processing. The tracer has 252 lines of code, and is chained after the *SMemory* analysis, which supplies the means for memory access tracking.

4.2 Trace File Structure

Figure 5 illustrates the structure of the trace files for a simple toy example. The example has an input-dependent branch in line 10 and a secret-dependent memory access in line 11, which should be detected by our analysis toolchain.

Each trace is structured as follows: The first element defines the type of the trace event, e.g. *Call* or *Expr* (for Expression). This is followed by the exact source location of the event, meaning the script file name with start/end line and column number. For a *Call*, the first location describing the source of the call is followed by a second location describing the target, which in turn is followed by the name of the called function. *Expr* entries log the locations of all executed expressions; this information is only needed for reconstructing control flow edges. Similarly, *Ret1* records the occurrence of a return-statement, which must be tracked due to not being covered by an expression. *Ret2* is generated *after* a function has returned, and records the entire ranges of the function call and the executed function; however, the associated callback does not know where the control flow originated from, thus the necessity of tracking expressions and return statements. The same is true for *Cond* entries, which mark the execution of a conditional and thus the begin of a control-flow edge.

To illustrate this, Figure 5b shows the case of a taken *else*-branch with the assignment `ret = 0` in line 14 of the trace. If we compare this trace to the Figures 5c and 5d, which both show a taken *if*-branch, it becomes apparent that the control-flow deviation only shows up due to the differences in lines 14 and 15; everything else is identical. Thus, only tracking all expressions and read/write operations allows us to reconstruct the entire control flow.

Comparing line 14 of Figures 5c and 5d demonstrates how the traces enable us to discover secret-dependent memory accesses. The last two elements of the *Get* entry represent the ID of the shadow object, and the accessed property or offset. Both elements differ between the traces: The object IDs are assigned by *Jalangi2* and thus vary for subsequent invocations of *processTestcase*, and the accessed offset depends on the input. The analysis conducted by *Microwalk-CI* will later match the object IDs belonging to identical objects between traces, such that it can compare the offsets.

This example shows a very short excerpt of a trace for a toy program. Analyzing real world code may result in traces with millions of events, resulting in huge files. To reduce the storage overhead, we compress the trace by shortening strings and encoding repeating lines. For most targets, these measures are sufficient to keep the trace files within a few ten megabytes. Additional compression could be achieved e.g. by using LZMA, which due to the high rate of repetitions and hence low entropy usually manages to bring down the trace file size to a few hundred kilobytes.

```

1 /**
2  * Simplified demo test case
3  */
4  function processTestcase(buffer)
5  {
6      var val = parseInt(buffer);
7      var array = [0, 1, ..., 15];
8      var ret = -1;
9
10     if(val % 2 === 0) {
11         ret = array[val] + 1;
12     } else {
13         ret = 0;
14     }
15
16     return ret;
17 }
18
19

```

(a) Source

```

1 ...
2 Call;/index.js:28:5:28:43;
3 target.js:4:1:17:2;;
4 processTestcase
5 Call;target.js:6:15:6:39;
6 [extern]:parseInt::parseInt
7 Ret2;[extern]:parseInt;;
8 target.js:6:15:6:39
9 Expr;target.js:6:15:6:39
10 Expr;target.js:7:17:7:71
11 Expr;target.js:8:15:8:17
12 Cond;target.js:10:8:10:21
13 Expr;target.js:10:5:14:6
14 Expr;target.js:13:9:13:17
15 Ret1;target.js:16:12:16:15
16 Expr;target.js:16:5:16:16
17 Ret2;target.js:4:1:17:2;;
18 /index.js:28:5:28:43
19 ...

```

(b) Trace for buffer = 1.

```

1 ...
2 Call;/index.js:28:5:28:43;
3 target.js:4:1:17:2;;
4 processTestcase
5 Call;target.js:6:15:6:39;
6 [extern]:parseInt::parseInt
7 Ret2;[extern]:parseInt;;
8 target.js:6:15:6:39
9 Expr;target.js:6:15:6:39
10 Expr;target.js:7:17:7:71
11 Expr;target.js:8:15:8:17
12 Cond;target.js:10:8:10:21
13 Expr;target.js:10:5:14:6
14 Get;target.js:11:15:11:25;17;0
15 Expr;target.js:11:9:11:30
16 Ret1;target.js:16:12:16:15
17 Expr;target.js:16:5:16:16
18 Ret2;target.js:4:1:17:2;;
19 /index.js:28:5:28:43
20 ...

```

(c) Trace for buffer = 0.

```

1 ...
2 Call;/index.js:28:5:28:43;
3 target.js:4:1:17:2;;
4 processTestcase
5 Call;target.js:6:15:6:39;
6 [extern]:parseInt::parseInt
7 Ret2;[extern]:parseInt;;
8 target.js:6:15:6:39
9 Expr;target.js:6:15:6:39
10 Expr;target.js:7:17:7:71
11 Expr;target.js:8:15:8:17
12 Cond;target.js:10:8:10:21
13 Expr;target.js:10:5:14:6
14 Get;target.js:11:15:11:25;19;2
15 Expr;target.js:11:9:11:30
16 Ret1;target.js:16:12:16:15
17 Expr;target.js:16:5:16:16
18 Ret2;target.js:4:1:17:2;;
19 /index.js:28:5:28:43
20 ...

```

(d) Trace for buffer = 2.

Figure 5: Traces created by Microwalk-CI for a JavaScript toy example. Indented lines are wrapped for readability and are formatted in a single line in the original trace file.

4.3 Trace Preprocessing

These raw traces are not yet suitable for use by Microwalk-CI; we need to translate the sequence of executed lines to branch entries, generate allocation information for the objects showing up in the traces, and finally produce compatible binary traces, which can be fed into analysis modules like the one described in Section 3. For this, we implemented a new preprocessor module, which has 702 lines of code and resides in a plugin. The module iterates through each entry of the raw trace, generating a preprocessed trace on-the-fly. It recognizes branches by waiting for the next code location that is outside the corresponding conditional; if an access to a previously unknown object is detected, an

allocation is created.

Note that our analysis module is designed for binary analysis, i.e., it works with actual memory addresses and offsets. In fact, this proves valuable for later analysis, as this simplifies encoding trace entries and gives clear identifiers for referring to certain instructions. Thus, we chose to generate a mapping of observed source locations to dummy addresses, by encoding the line and column numbers onto a base address belonging to the respective source file. This mapping is stored in a special map file, such that it can be mapped back to a human-readable source line after analysis.

In summary, we now have a tool chain that instruments JavaScript programs, generates raw execution traces and converts them into the Microwalk-CI binary trace format, allowing us to analyze arbitrary JavaScript software with the existing and new generic analysis algorithms, without having to create a dedicated analysis tool.

5 Integration into Development Workflow

In this section, we show how one can simplify usage of Microwalk-CI to a degree that it only needs a one-time effort by the developer to set it up and register the functions that need to be analyzed. From that point, the tool is part of the CI pipeline of the respective library, and runs each time a new commit is submitted. The developer is then able to easily verify whether a code change introduces new leakages, without requiring any manual intervention.

5.1 Dockerizing the Analysis Framework

In order to use the analysis framework in an automated environment, we must ensure that all its dependencies are present and the environment is configured correctly. For this task, common CI systems allow the use of Docker containers. When a job starts, a new container is started from a predefined Docker image. The CI system checks out the current source code and then executes a user-defined script within the container. This has the advantage of being independent of the host system: The analysis job may run on the developer's private server, but also on cloud infrastructure administrated by external providers. We thus create a pre-configured Docker image containing the components needed for our JavaScript analysis: The Jalangi2 runtime, the analysis script and the Microwalk-CI binaries. The image is uploaded to a Docker registry.

```

/                                # Project source tree
index.js                        # Analysis entrypoint
package.json
...
microwalk/                      # Analysis-specific files
  analyze.sh                    # Script executed by CI
  config-preprocess.yml         # Microwalk config. for preprocessing
  config-analyze.yml           # Microwalk config. for analysis
  target-aes.js                 # AES target
  target-rsa.js                 # RSA target
  ...
  testcases/                    # Input files for trace generation
    target-aes/
      0.testcase
      1.testcase
      ...
    target-rsa/
      0.testcase
      1.testcase
      ...
    ...
  ...

```

Figure 6: Generic source tree of a JavaScript project containing our analysis template.

5.2 Analysis Template

Having solved the installation and configuration problem, we now need to setup the necessary infrastructure to actually run the analysis for the specific library. Instead of requiring the developer to dive into the proper usage of the analysis toolchain, we designed a template that is simple and generic enough to work with most libraries, and which only needs minimal understanding and adjustment. The resulting file structure is depicted in Figure 6.

The template features a script file `index.js`, which serves as analysis entry point and is responsible for loading test cases and executing the target implementations. A *target* is any independently testable code unit, e.g., a single primitive in a cryptographic library. The individual `microwalk/target-*.js` script files consist of a single function, that receives the current test case data buffer and is expected to call the associated library code. Each target also needs a number of test cases, which may have a custom format and thus need to be generated once by the developer. The test cases are stored in the `microwalk/testcases/` subdirectory. Finally, the `microwalk` folder has a bash script `analyze.sh`, that is called by the CI. The analysis script iterates through the target files, and runs the Microwalk-CI pipeline. The Microwalk-CI configuration is located in two generic YAML files, which can be adjusted by the developer if they wish to use other analysis modules or options than the preconfigured ones.

The abstractions offered by our template allows the developer to focus on supplying simple wrappers for their library interface and generating a number of random test

◆	Critical - (target-toy-example) Found vulnerable memory access instruction, leakage score 100.00% +/- 0%. Check analysis result in artifacts for details. in target.js:11
▼	Major - (target-toy-example) Found vulnerable jump instruction, leakage score 53.33% +/- 0%. Check analysis result in artifacts for details. in target.js:10

Figure 7: GitLab report for the toy example from Figure 5a. The leakage score is a relative representation of the minimal GE as explained in Section 3.5.3.

cases; everything else is taken care of by the existing scripts. We implemented a similar template for compiled software, so the approach is the same for C libraries.

5.3 Reports

After the CI job has completed, it yields a couple of analysis result files. As of our analysis objectives in Section 3, these files are designed to be human-readable and offer as much insight into a leakage as possible. However, if there are a lot of leakage candidates, going through this list may be tedious, especially if the result files are stored separately and need to be inspected manually for each commit. We thus looked into ways for integrating these results into the usual development workflow.

For GitLab, there is a *Code Quality Reports* [20] feature, which shows up in the merge request UI. It allows to assign a severity, a description and a source code file and line to each entry, which makes it suitable to display the results from our leakage analysis. Microwalk-CI consolidates the analysis result into a report that can be parsed by GitLab. For this, the leakages must be mapped to their originating locations in the source code. This is straightforward for JavaScript, as this information already shows up in the analysis result file; for binary programs, we resort to parsing the DWARF debug information in order to map offsets to file names and lines. The code quality report also shows a severity of a given problem, which can be one of *info*, *minor*, *major*, *critical* and *blocker* (a continuous scale is not supported). Assigning these levels to specific leakages is somewhat arbitrary and depends on the preferences of the individual developer; we settled for *minor* if the minimal GE is higher than 80% of its upper bound, *critical* if the minimal GE is lower than 20% of its upper bound, and *major* for everything in between. This ensures that instances with high leakage are displayed prominently. Figure 7 shows an example report.

6 Evaluation and Discussion

To evaluate Microwalk-CI, we applied it to several popular JavaScript crypto libraries. In the following, we describe our experimental setup and discuss the performance and discovered vulnerabilities.

6.1 Experimental Setup

As targets we pulled eight popular JavaScript libraries for cryptography and utility functions from NPM, and set up a local GitLab repository for each. Using the version from NPM instead of the version from GitHub allows us to analyze the code deployed to millions of users. We then applied our template and created `target-*.js` files for selected cryptographic primitives and utility functions that deal with secret data. For each target we generated 16 random test cases, which were subsequently checked in into the source tree.

The GitLab instance takes care of managing the CI jobs and visualizing the resulting code quality reports. The analysis jobs themselves are executed through a Docker-based GitLab Runner on a separate machine (build server), which has an AMD EPYC 7763 processor with 128 GB DDR4 RAM. We configured the Microwalk-CI trace preprocessor step to use up to 4 CPU cores. After all CI jobs had completed, we collected the performance statistics generated by GitLab and the CI jobs, and went through the leakage reports. The results are visualized in Table 1.

6.2 Performance

6.2.1 Computation time

The CPU time spent for trace generation, preprocessing and analysis mostly depends on two factors. First, it correlates with the complexity of the analyzed targets: For the investigated libraries, symmetric algorithms and utility functions performed very well, while asymmetric primitives took significantly longer, which is expected. Second, the CPU time scales linearly with the number of test cases. We discuss the corresponding trade-off between accuracy and performance in Section 6.4.

The computational cost for the **trace generation step** mainly stems from the instrumentation itself, as our tracer script is already quite minimal. Significant optimizations would thus need to target the Jalangi2 implementation.

The CPU time spent for the **preprocessing step** correlates with the size of the raw traces. The implementation is parallelized, so each trace can be processed independently.

Table 1: Targets analyzed with JavaScript Microwalk-CI, performance metrics and the number of detected leakages (total and unique code lines). “Tr. CPU” shows the CPU time for generating the raw traces, “Prep. CPU” for trace preprocessing, and “An. CPU” for the analysis step. “Duration” denotes the wall clock time spent for the entire CI job (including setup and cleanup). Finally, “Prep. RAM” and “An. RAM” show the peak memory usage for the preprocessing and analysis steps, respectively.

Target	Type	Tr. CPU	Prep. CPU	An. CPU	Duration	Prep. RAM	An. RAM	# Lkgs.	# Uniq.
aes-js [3] 3.1.2, \approx 800k weekly downloads									
AES-ECB	cipher	1 sec	< 1 sec	< 1 sec	7 sec	294 MB	180 MB	16	16
base64-js [7] 1.5.1, \approx 28M weekly downloads									
base64-encode	utility	< 1 sec	< 1 sec	< 1 sec	6 sec	283 MB	173 MB	7	7
base64-decode	utility	< 1 sec	< 1 sec	< 1 sec	6 sec	291 MB	189 MB	7	7
crypto-js [15] 4.1.1, \approx 4M weekly downloads									
AES-ECB	cipher	2 sec	< 1 sec	< 1 sec	8 sec	289 MB	191 MB	44	44
Rabbit	cipher	2 sec	< 1 sec	< 1 sec	8 sec	304 MB	182 MB	0	0
base64-encode	utility	4 sec	1 sec	1 sec	10 sec	349 MB	250 MB	0	0
base64-decode	utility	3 sec	< 1 sec	< 1 sec	9 sec	339 MB	225 MB	2	2
pbkdf2	utility	5 sec	1 sec	1 sec	11 sec	384 MB	221 MB	0	0
elliptic [18] 6.5.4, \approx 13M weekly downloads									
secp256k1	signature	110 sec	26 sec	21 sec	139 sec	853 MB	3,123 MB	58	45
p192	signature	237 sec	35 sec	12 sec	261 sec	2,112 MB	1,835 MB	98	57
p224	signature	303 sec	45 sec	14 sec	334 sec	1,700 MB	2,357 MB	76	58
p256	signature	469 sec	84 sec	45 sec	545 sec	3,347 MB	6,674 MB	78	50
p384	signature	977 sec	145 sec	45 sec	1,063 sec	3,383 MB	7,522 MB	391	53
ed25519	signature	175 sec	46 sec	32 sec	222 sec	2,884 MB	4,607 MB	111	40
js-base64 [8] 3.7.2, \approx 6M weekly downloads									
base64-encode	utility	< 1 sec	< 1 sec	< 1 sec	6 sec	290 MB	187 MB	0	0
base64-decode	utility	< 1 sec	< 1 sec	< 1 sec	6 sec	290 MB	155 MB	0	0
node-forge [19] 1.2.1, \approx 17M weekly downloads									
AES-ECB	cipher	5 sec	< 1 sec	< 1 sec	11 sec	298 MB	193 MB	36	36
AES-GCM	cipher	9 sec	2 sec	2 sec	16 sec	387 MB	349 MB	126	52
base64-encode	utility	5 sec	< 1 sec	< 1 sec	11 sec	287 MB	192 MB	0	0
base64-decode	utility	5 sec	< 1 sec	< 1 sec	11 sec	296 MB	198 MB	4	4
rsa	signature	62 sec	18 sec	13 sec	82 sec	364 MB	1,926 MB	223	111
ed25519	signature	124 sec	33 sec	9 sec	144 sec	1,145 MB	509 MB	0	0
pbkdf2 [43] 3.1.2, \approx 13M weekly downloads									
pbkdf2	utility	< 1 sec	< 1 sec	< 1 sec	6 sec	298 MB	179 MB	0	0
tweetnacl [58] 1.0.3, \approx 21M weekly downloads									
secretbox	cipher	2 sec	< 1 sec	< 1 sec	8 sec	288 MB	189 MB	0	0
box	asymmetric	75 sec	20 sec	7 sec	91 sec	335 MB	487 MB	0	0
ed25519	signature	117 sec	33 sec	9 sec	138 sec	1,137 MB	509 MB	0	0

Profiling shows a slight bottleneck in the string parsing code, so switching to a binary trace format may further improve preprocessing performance, at the cost of higher code complexity in the trace generation.

The **analysis step** took less than one CPU minute for every investigated target; this underlines the efficiency of the presented analysis algorithm, and that it is fast enough to be used in a productive setting. The time spent for the analysis mostly depends on the trace size, as when building the call tree, each trace entry is converted into a tree node or embedded into an existing one. Another factor is the number of leakages, as is apparent when comparing the analysis times of the various ed25519 implementations.

The measured **overall duration** heavily depends on where most CPU time is spent: While the trace generation and the analysis are mostly sequential, the trace preprocessing is heavily parallelized. Thus, a high CPU time for preprocessing does contribute less to the overall duration. Apart from one outlier, `elliptic's p384`, the measured times stayed well within a few minutes, which can be considered acceptable for productive use in a CI pipeline.

6.2.2 Memory usage

The inherently different pipeline steps also reflect in different memory requirements.

The **trace generation step** has a negligible memory footprint, which mostly depends on the size of the array that is used for buffering trace entries before writing them to the output file.

The memory consumption of the **preprocessing step** is mainly caused by loading chunks of the trace file into memory and decompressing them. Parallelization of the preprocessing step means that several trace files are being held in memory simultaneously. The memory usage of the preprocessing can be reduced by decreasing the number of parallel threads (4 in our experiment).

In the call tree **analysis step**, the memory demand is driven by the size of the preprocessed traces and, most notably, their level of divergence. If the target is constant-time and thus all traces are identical, the tree does not have any split nodes, so all traces end up in the same nodes. Adding a trace ID to an existing node does not involve any significant memory cost, as the trace IDs assigned to a call tree node are stored as a bitfield.

However, if the traces heavily diverge, the analysis produces many split nodes with partially redundant subtrees. This distinction becomes apparent by the implementations

of `ed25519` in `elliptic` and in `tweetnacl`: While using comparable tracing and preprocessing time, the constant-time implementation in `tweetnacl` requires much less memory than the implementation in `elliptic`, which relies on the leaking `bn.js` and `hash.js` libraries. Through continuously applying Microwalk-CI and mitigating non-constant-time behavior such that only small leakages pop up during analysis, the peak memory usage of the analysis step can be kept within the bounds of a typical CI environment.

Overall, the **peak memory usage** of Microwalk-CI is on an acceptable level. The highest memory consumption was observed when analyzing `elliptic`'s `p384`. This is certainly a worst case example, as large parts of its code are non-constant time, while Microwalk-CI is optimized for finding mid-level leakages in an otherwise fairly constant-time software. However, most of `p384`'s code is shared with the other curve implementations, which contain the same leakages, but can be analyzed more efficiently. Also, a significant part of the identified leakages reside in the the SHA-512 implementation of `hash.js`, which should be analyzed separately.

As expected, more complex algorithms like asymmetric cryptography require more memory in the analysis. But, even those only require an amount of memory which, today, is commonly available.

6.3 Vulnerabilities

Our leakage analysis identified many leakages in the given libraries. We evaluated whether those are in fact actual vulnerabilities, and discuss a few examples in the following. In general, the leakages were correctly assigned to the respective leaking code lines, and we did not encounter any false positives (i.e., code lines that don't leak by themselves). In addition to the report shown in the user interface (Figure 7), a detailed leakage report is generated, which provides the full calling context for each leakage and shows how the different test cases contributed to tree divergences.

6.3.1 Leakages in AES

All investigated implementations of AES use table lookups into S-boxes or precomputed T-tables, making those highly susceptible to timing attacks. The exploitability of such lookups was previously shown in other work [10]. All leakages found in `aes.js` by Microwalk-CI have a maximum leakage score.

Additionally, Microwalk-CI discovers input-dependent behavior in the AES-GCM encryption of `node-forge`. Manual inspection shows that these leakages in the `tableMultiply` function in the file `cipherModes.js` occur during the computation of the GHASH which

is used for the final computation of the authentication tag. The `tableMultiply` function uses a table precomputed from the hash key and multiplies by accessing this table with an index which is an intermediate value computed from the current ciphertext block and the previous hash value. Learning this intermediate value potentially allows to gain information about the GHASH key, compromising the authentication property. The implementation in `node-forge` uses 4-bit tables. Whether this implementation and leakage is exploitable, is left to future work. We recommend not having any secret-dependent non-constant-time code.

6.3.2 Elliptic curve implementations

`node-forge` and `tweetnacl` feature custom constant-time big number arithmetic that is specifically designed for the supported curves. The `elliptic` library, however, relies entirely on arithmetic from the general-purpose `bn.js` [11] library, which features a lot of input-dependent control flow and memory accesses. Thus, we see very high leakage over all supported primitives. The leakages detected in the big number and elliptic code itself are mostly assigned scores between 80 and 100.

In addition, for computing the signature, `elliptic`'s ECDSA implementation uses the `hash.js` [24] library, which offers pure-JavaScript implementations for SHA-1 and SHA-2. For ECDSA and EdDSA signatures with the curves `p384` and `ed25519`, respectively, the leakage report points to a significant amount of leakage in `lib/hash/sha/512.js` for a variety of call stacks. Here, the implementation works around a limitation of JavaScript, which represents all numbers in IEEE-754 double precision floating point, and temporarily converts them to 32-bit signed integers for bitwise arithmetic. If the most-significant bit ends up being 1, JavaScript sign-extends it such that the result is negative. The implementation checks for this in an `if` statement and adds `0x100000000` to get a positive number. This leakage may pose a security issue, as ECDSA and EdDSA use the hash function for generating a nonce from the private key. Microwalk-CI assigns leakage scores between 60 and 70 for most of the leakages in `lib/hash/sha/512.js`. Future work could investigate whether the leakage of the most-significant bit can be used to learn parts of the private key. The libraries `elliptic`, `bn.js` and `hash.js` are from the same author.

6.3.3 Base64 encoding

We also found leakages in some of the various Base64 implementations. All of them were caused by the use of lookup tables, where 6-bit chunks are mapped to ASCII characters

and vice versa. The only known attack against Base64 encoding relies on a precise controlled channel that is not available for common JavaScript deployments [54]. However, depending on the memory layout of the respective lookup tables, partial information may be accessible via a cache attack. `js-base64` does also feature a vulnerable Base64 implementation; however, it first checks whether the `Buffer` class with native Base64 support is present, which is the case for our Node.js build.

6.4 Number of Test Cases

As mentioned in the performance analysis, computation time and, to a lesser degree, memory consumption, scale with the number of test cases. A higher number of test cases increases the chance of triggering uncommon code paths and thus finding more leakages. In the following, we analyze this trade-off and point out approaches for striking a good balance between accuracy and performance.

In our performance analysis, we ran 16 test cases for each library. This number is within the same order of magnitude as the one used for the evaluation in [62], where the authors recommend running 10 test cases. To check whether the small number of test cases had impact on the number of detected leakages, we repeated our analysis with 48 additional test cases (64 total) for each target and compared the results with those of the first analysis.

6.4.1 Performance

Increasing the number of test cases does not affect every pipeline step in the same way. Doubling the number of test cases roughly doubles the CPU time needed for trace generation, but that does not apply to the analysis step: There, the first test case takes much longer than subsequent ones, as it needs to build the tree from scratch, which involves spending a lot of time in the memory allocator. Later non-diverging test cases only need to iterate the existing tree, which takes considerably less resources. We observed that the duration increased by factor 3 to 3.5, although we ran 4 times as many test cases.

6.4.2 Leakages

Except for targets in the libraries `elliptic` and `node-forge`, Microwalk-CI found the same amount of leakages with 64 test cases as with 16. For `elliptic`, all targets show a small single digit increase in the number of overall and unique leakages. For all new leakages, we determined that these were initially missed due to a saturation effect (see

Section 6.6) and not by lack of coverage, and would have been found by re-running the analysis after fixing the preceding leakages.

For `node-forge`'s RSA implementation, the difference is a bit larger. While Microwalk-CI finds 223 overall and 111 unique leakages with 16 test cases, it was able to discover 255 overall and 125 unique leakages with 64 test cases. Manual investigation shows again that most leakages were missed due to a saturation effect. However, a small number was missed due to insufficient coverage of the initial 16 test cases.

6.4.3 Recommendations

We recommend the developer to choose an overall duration that is acceptable during ongoing development and determine an according test case number. In addition, the coverage of the generated test cases could be checked with a separate tool to ensure that all relevant code gets executed. Finally, the developer could add another larger collection of test cases that runs as a final check before releasing the next version, where a longer analysis time is acceptable.

6.5 Comparison with Microwalk's original Analysis Module

Microwalk originally features two analysis modules that implement the *memory access trace (MAT)* analysis method for finding leakages. The method was first presented in [63]. For each memory accessing instruction, the modules generate a hash over all accessed offsets. By comparing the hashes between traces, the amount of leakage for each memory accessing instruction is computed. Due to the focus on memory accesses, control flow leakages are only discovered indirectly or may even be missed entirely.

The first module, that was originally published with [63], generates only one leakage report per instruction. The later added second module (referred to by us as *CMAT module*) is an extension of the first module that additionally distinguishes between call stacks to achieve a higher accuracy. To compare the existing analysis method with our new approach, we ran a selection of the targets with the CMAT module, using the same 16 test cases as for the initial analysis. The results are shown in Table 2.

Since the CMAT module only stores a single mapping of call stacks and instructions to hashes, it generally takes less resources than our new tree-based approach, both in computation time and memory consumption. However, the preceding trace generation and preprocessing, which take most of the time, are identical, so the actual difference in overall duration is limited.

Table 2: Results of the analysis step of selected targets with the original Microwalk CMAT module, and its resource usage. Time and memory consumption of the trace generation and preprocessing steps are identical to those shown in Table 1.

Target	CPU	Duration	RAM	# Lkgs.	# Unique
aes-js					
AES-ECB	< 1 sec	8 sec	168 MB	16	16
elliptic					
p192	4 sec	253 sec	289 MB	4,003	811
tweetnacl					
ed25519	5 sec	126 sec	286 MB	0	0

For `aes-js`' AES-ECB implementation, the CMAT module reports a number of secret-dependent table accesses with full leakage, which are identical to the leakages reported by our new analysis module. This is the kind of leakage that the MAT analysis was designed for: Through hashing the sequence of memory addresses that a given instruction accesses, secret-dependent variations are discovered. Our new analysis detects these leakages through the address lists stored in the individual memory access trace entries, which ultimately yields the same result, but takes more memory.

The result from the CMAT module for `elliptic`'s `p192` is very imprecise and contains many false positives: It reports 811 leaking lines in total, which includes lines like `"this.pendingTotal = 0;"`. As a fixed offset is accessed, this line is a clear false positive. The leakage in question was in fact caused by a control flow variation higher up in the call chain, leading to a varying number of executions of the given instruction, which in turn produced a different memory access offset hash. The other false positives follow a similar pattern. Our new tree-based approach handles control flow and memory access leakages separately, which reduces false positives and allows accurately attributing a leakage to a specific code line.

6.6 Limitations of the Analysis Algorithm

As other dynamic analysis approaches, Microwalk-CI needs a good coverage of the program in order to give an accurate leakage detection result. If a particular path is never executed, it does not appear in the traces and thus never reaches the analysis modules. However, for cryptographic code, randomly generated test cases tend to work very well [62, 63]. For other targets, it may be worth exploring other methods for generating coverage, e.g., fuzzing.

Finally, in our analysis algorithm, some leakages may be obscured by other leakages at a higher tree level. If leakages on higher levels cause splits that result in a unique

sub tree for each trace, the lower leakages can not cause any more divergences and thus are overlooked. This “saturation” is an inherent property of the analysis approach, and the price paid for having a linear-time algorithm. We do not believe that this impacts practical usage: After having a library reach a certain state of “constant-time-ness”, we only expect few new leakages being reported, as certain functions are touched. And even if a leakage is not reported in a first pass, it will show up after committing the fixes for the previously reported leakages. It is unlikely that a number of unfixed low-severity leakages obscure a subsequent severe leakage. This would imply a fully split up tree, which, in itself, signals a high-severity leakage.

Other work tries to find all trace leakages in a single pass, but uses significantly more resources with every CI run and thus is not suitable for integration into an everyday-development workflow.

7 Related Work

Constant-time program analysis has a long tradition as there are different classes of vulnerabilities that can be found through various analysis techniques [36]. Some tools for checking constant-time behavior depend on the availability of source code. Irazoqui et al. [28] introduce secret-dependent cache trace analysis, *ct-fuzz* [25] specializes fuzzing for timing leakages, *ct-verif* [5] describes constant-time through safety properties and *CaSym* [13] uses symbolic execution to model the execution behavior of a program. Microwalk-CI does not require access to the source code for compiled languages.

Unlike Microwalk-CI which uses dynamic program analysis and compares real execution traces, **static binary analysis** tries to simulate the execution of every possible program path. *BINSEC/REL* [16] uses relational symbolic execution of two execution traces to efficiently analyze binary code, however is limited by the high performance impact of static analysis. *CacheS* [59], based on *CacheD* [60], combines taint tracking and symbolic execution to find cache line granular leakage and secret-dependent branches. Moreover, *CacheAudit* [17] tracks relational information about memory blocks to compute upper bounds for leakages. In contrast with these works, Microwalk-CI finds any leakage with byte granularity.

DATA [62] and its (EC)DSA-specific extension [61] find microarchitectural and timing side-channels in binaries via **dynamic binary analysis**. The trace alignment approach of *DATA* is based on computing pairwise differences between traces, leading to a computation time that is quadratic both in the number of traces and in the trace length. While it yields more leakage candidates after a single pass, it needs more computational resources and thus is not suited for use in a CI environment. *Abacus* [6] identifies secret-dependent

memory access instructions using symbolic execution. Then, the authors use Monte Carlo sampling to estimate the amount of leaked information. A shortcoming of the approach is that *Abacus* only uses one trace and therefore suffers from low coverage. *dudect* [46] measures timing behavior in a statistical way without any model of the underlying hardware, which is fast, but also yields imprecise results. *ctgrind* [33] and *TIMECOP* [39] search the code for secret-dependent jump or memory accesses like table-lookups and variable-time CPU instructions, but are rather manual.

Analysis of JavaScript code recently received more focus in the research community as it is widely used in browsers including many security-critical workloads. Basic properties of JavaScript regarding security of code have been widely analyzed [30, 31, 52, 57]. Just as in other programming languages, various attacks on secret-dependent behavior have been conducted [51, 53]. A common prerequisite for exploiting timing-dependent properties of code is having precise timers [47], though this can be bypassed [53]. Apart from countermeasures like disabling timers or blocking certain functionality [50], little work has gone into finding non-constant-time JavaScript code.

8 Conclusion

With Microwalk-CI we have shown how one can design a side-channel analysis framework that is suitable for integration into a day-to-day development workflow. We have presented a new trace processing algorithm that merges the recorded traces into a call tree, allowing us to precisely localize and quantify leakages in a short time frame. Moreover, by “dockerizing” the analysis, we have provided the means for easy and fast usage without the necessity of understanding the details of the framework.

With the design and implementation of a tracer for JavaScript and the integration with Microwalk-CI, we have built the first comprehensive constant-time verifier for JavaScript code and demonstrated how analysis techniques originally developed for binary analysis can be used for interpreted or just-in-time compiled languages. Microwalk-CI is constructed in a modular fashion and allows to add tracing backends for other languages with limited effort.

Overall, Microwalk-CI carries the potential to increase the side-channel security for many popular libraries written in potentially any programming language, and raises awareness for the risks of non-constant-time code in new communities.

Acknowledgements

The authors thank Julia Tönnies for her help in evaluating the suitability of the leakage metrics, and the anonymous reviewers for their helpful comments and suggestions. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under grants 427774779 and 439797619, and by Bundesministerium für Bildung und Forschung (BMBF) through projects ENCOPIA and PeT-HMR.

References

- [1] Onur Aciıçmez, Billy Bob Brumley, and Philipp Grabher. “New Results on Instruction Cache Attacks”. In: *12th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2010. DOI: 10.1007/978-3-642-15031-9_8.
- [2] Onur Aciıçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “Predicting Secret Keys Via Branch Prediction”. In: *The Cryptographers’ Track at the RSA Conference 2007 (CT-RSA)*. 2007. DOI: 10.1007/11967668_15.
- [3] AES-JS. <https://github.com/ricmoo/aes-js>. (Visited on 2024-05-21).
- [4] Alejandro Cabrera Aldaya and Billy Bob Brumley. “When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 196–221. DOI: 10.13154/tches.v2020.i2.196-221.
- [5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>.
- [6] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. “Abacus: Precise Side-Channel Analysis”. In: *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2021. DOI: 10.1109/ICSE43902.2021.00078.
- [7] base64-js. <https://github.com/beatgammit/base64-js>. (Visited on 2024-05-21).
- [8] base64.js. <https://github.com/dankogai/js-base64>. (Visited on 2024-05-21).
- [9] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. “Automatic Application of Power Analysis Countermeasures”. In: *IEEE Trans. Computers* 64.2 (2015), pp. 329–341. DOI: 10.1109/TC.2013.219.
- [10] Daniel J Bernstein. *Cache-timing attacks on AES*. 2005.
- [11] bn.js. <https://github.com/indutny/bn.js>. (Visited on 2024-05-21).

- [12] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. “Software mitigations to hedge AES against cache-based software side channel vulnerabilities”. In: *IACR Cryptol. ePrint Arch.* (2006), p. 52. URL: <http://eprint.iacr.org/2006/052>.
- [13] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00022.
- [14] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX@SOSP)*. 2017. DOI: 10.1145/3152701.3152706.
- [15] crypto-js. <https://github.com/brix/crypto-js>. (Visited on 2024-05-21).
- [16] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00074.
- [17] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *22nd USENIX Security Symposium*. 2013. URL: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev>.
- [18] Elliptic. <https://github.com/indutny/elliptic>. (Visited on 2024-05-21).
- [19] Forge. <https://github.com/digitalbazaar/forge>. (Visited on 2024-05-21).
- [20] GitLab. *Code Quality*. https://docs.gitlab.com/ee/ci/testing/code_quality.html. (Visited on 2024-05-21).
- [21] Google. *Tracing Framework*. <https://github.com/google/tracing-framework>. (Visited on 2024-05-21).
- [22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>.
- [23] Silviu Guiaşu. *Information Theory with Applications*. McGraw-Hill Companies, 1977.
- [24] hash.js. <https://github.com/indutny/hash.js>. (Visited on 2024-05-21).
- [25] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. “ct-fuzz: Fuzzing for Timing Leaks”. In: *13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 2020. DOI: 10.1109/ICST46399.2020.00063.

- [26] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *18th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2016. DOI: 10.1007/978-3-662-53140-2_18.
- [27] Intel. *Pin 3.22 User Guide*. [https://software.intel.com/sites/landingpage/putool/docs/98547/Pin/html/](https://software.intel.com/sites/landingpage/puttool/docs/98547/Pin/html/). (Visited on 2024-05-21).
- [28] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. “Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries”. In: *CoRR abs/1709.01552* (2017). arXiv: 1709.01552. URL: <http://arxiv.org/abs/1709.01552>.
- [29] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ““They’re not that hard to mitigate”: What Cryptographic Library Developers Think About Timing Attacks”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833713.
- [30] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “Type Analysis for JavaScript”. In: *16th International Symposium on Static Analysis (SAS)*. 2009. DOI: 10.1007/978-3-642-03237-0_17.
- [31] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. “JSAI: a static analysis platform for JavaScript”. In: *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 2014. DOI: 10.1145/2635868.2635904.
- [32] Boris Köpf and David A. Basin. “An information-theoretic model for adaptive side-channel attacks”. In: *2007 ACM Conference on Computer and Communications Security (CCS)*. 2007. DOI: 10.1145/1315245.1315282.
- [33] A Langley. *ctgrind: Checking that functions are constant time with Valgrind*. 2010.
- [34] Jens Lindemann and Mathias Fischer. “A memory-deduplication side-channel attack to detect applications in co-resident virtual machines”. In: *33rd Annual ACM Symposium on Applied Computing (SAC)*. 2018. DOI: 10.1145/3167132.3167151.
- [35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.43.
- [36] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. “A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography”. In: *ACM Comput. Surv.* 54.6 (2022), 122:1–122:37. DOI: 10.1145/3456629.

- [37] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *Int. J. Parallel Program.* 47.4 (2019), pp. 538–570. DOI: 10.1007/s10766-018-0611-9.
- [38] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. “CopyCat: Controlled Instruction-Level Attacks on Enclaves”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat>.
- [39] Moritz Neikes. *TIMECOP: Automated Dynamic Analysis for Timing Side-Channels*. 2020. URL: <https://www.post-apocalyptic-crypto.org/timecop/>.
- [40] OpenJS Foundation. *Node.js - JavaScript Runtime*. <https://nodejs.org>. (Visited on 2024-05-21).
- [41] OpenTelemetry. *OpenTelemetry JavaScript*. <https://github.com/open-telemetry/opentelemetry-js>. (Visited on 2024-05-21).
- [42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *The Cryptographers’ Track at the RSA Conference 2006 (CT-RSA)*. 2006. DOI: 10.1007/11605805_1.
- [43] pbkdf2. <https://github.com/crypto-browserify/pbkdf2>. (Visited on 2024-05-21).
- [44] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl>.
- [45] Red Monk. *The RedMonk Programming Language Rankings: January 2022*. <https://redmonk.com/sograzy/2022/03/28/language-rankings-1-22/>. (Visited on 2024-05-21).
- [46] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time?” In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017. DOI: 10.23919/DATE.2017.7927267.
- [47] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. “SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers”. In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021. DOI: 10.1109/EuroSP51992.2021.00039.
- [48] Samsung. *Jalangi2 Source*. <https://github.com/Samsung/jalangi2>. (Visited on 2024-05-21).
- [49] SAP. *Project Foxhound*. <https://github.com/SAP/project-foxhound>. (Visited on 2024-05-21).

- [50] Michael Schwarz, Moritz Lipp, and Daniel Gruss. “JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks”. In: *25th Annual Network and Distributed System Security Symposium (NDSS)*. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-3_Schwarz_paper.pdf.
- [51] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *21st International Financial Cryptography and Data Security Conference (FC)*. 2017. DOI: 10.1007/978-3-319-70972-7_13.
- [52] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. “Jalangi: a selective record-replay and dynamic analysis framework for JavaScript”. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2013. DOI: 10.1145/2491411.2491447.
- [53] Anatoly Shusterman, Ayush Agarwal, Sioli O’Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. “Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses”. In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman>.
- [54] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. “Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484783.
- [55] Manu Sridharan, Koushik Sen, and Liang Gong. *Jalangi2 Presentation*. <https://manu.sridharan.net/files/JalangiTutorial.pdf>. (Visited on 2024-05-21).
- [56] Stack Overflow. *2021 Developer Survey - Programming, scripting, and markup languages*. <https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-programming-scripting-and-markup-languages>. (Visited on 2024-05-21).
- [57] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. “Automated Analysis of Security-Critical JavaScript APIs”. In: *2011 IEEE Symposium on Security and Privacy (SP)*. 2011. DOI: 10.1109/SP.2011.39.
- [58] TweetNaCl.js. <https://tweetnacl.js.org>. (Visited on 2024-05-21).
- [59] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. “Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation”. In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai>.

- [60] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. "CacheD: Identifying Cache-Based Timing Channels in Production Software". In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>.
- [61] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations". In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/weiser>.
- [62] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [63] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries". In: *34th Annual Computer Security Applications Conference (ACSAC)*. 2018. DOI: 10.1145/3274694.3274741.
- [64] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.45.
- [65] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium*. 2014. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.
- [66] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *18th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2016. DOI: 10.1007/978-3-662-53140-2_17.
- [67] Tianwei Zhang and Ruby B. Lee. "New models of cache architectures characterizing information leakage from cache side channels". In: *30th Annual Computer Security Applications Conference (ACSAC)*. 2014. DOI: 10.1145/2664243.2664273.

MAMBO-V: Dynamic Side-Channel Leakage Analysis on RISC-V

Publication

Jan Wichelmann, Christopher Peredy, Florian Sieck, Anna Pättschke, and Thomas Eisenbarth. *MAMBO-V: Dynamic Side-Channel Leakage Analysis on RISC-V*. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) - 20th International Conference*, 2023.

Contribution

Main author.

Outline

1	Introduction	178
2	Background	181
3	Overview	183
4	MAMBO-V Implementation	185
5	Side-Channel Leakage Analysis	188
6	Evaluation	191
7	Discussion and Future Work	195
8	Related Work	196
9	Conclusion	198
	References	198

MAMBO-V: Dynamic Side-Channel Leakage Analysis on RISC-V

Jan Wichelmann, Christopher Peredy, Florian Sieck, Anna Pättschke,
and Thomas Eisenbarth

Universität zu Lübeck

RISC-V is an emerging technology, with applications ranging from embedded devices to high-performance servers. Therefore, more and more security-critical workloads will be conducted with code that is compiled for RISC-V. Well-known microarchitectural side-channel attacks against established platforms like x86 apply to RISC-V CPUs as well. As RISC-V does not mandate any hardware-based side-channel countermeasures, a piece of code compiled for a generic RISC-V CPU in a cloud server cannot make safe assumptions about the microarchitecture on which it is running. Existing tools for aiding software-level precautions by checking side-channel vulnerabilities on source code or x86 binaries are not compatible with RISC-V machine code.

In this work, we study the requirements and goals of architecture-specific leakage analysis for RISC-V and illustrate how to achieve these goals with the help of fast and precise dynamic binary analysis. We implement all necessary building blocks for finding side-channel leakages on RISC-V, while relying on existing mature solutions when possible. Our leakage analysis builds upon the modular side-channel analysis framework Microwalk, that examines execution traces for leakage through secret-dependent memory accesses or branches. To provide suitable traces, we port the ARM dynamic binary instrumentation tool MAMBO to RISC-V. Our port named MAMBO-V can instrument arbitrary binaries which use the 64-bit general purpose instruction set. We evaluate our toolchain on several cryptographic libraries with RISC-V support and identify multiple leakages.

1 Introduction

Executing workloads in cloud environments with shared hardware resources is becoming more and more important, promising great flexibility and scalability. From a security viewpoint, however, this trend comes with a number of challenges, as shown by manifold examples of attacks that exploit microarchitectural side-channels in cloud systems [21, 22, 53].

While most of these cloud systems and the corresponding attacks are based on the conventional x86 architecture, a new architecture called RISC-V is gaining traction in both embedded applications and general-purpose hardware. The royalty-free license [4] of RISC-V enables affordable hardware through lower development costs, and helps innovation: For example, there now are several open-source CPU designs which can be analyzed and extended by anyone [26, 33, 45], promising the development of new hardware features like secure trusted execution environments (TEEs) which avoid the issues of existing commercial solutions. The software support for the RISC-V platform is growing as well, with major compiler vendors adding backends for emitting RISC-V machine code, which in turn allows porting operating systems like Linux.

The growing importance of RISC-V in general-purpose and cloud computing, coupled with a wide spectrum of CPU designs from various vendors, still necessitates caution to prevent repeating the mistakes that caused a lot of security issues on the established platforms. One particular example is *microarchitectural timing leakage* in cryptographic libraries, where subtle differences in how the microarchitecture processes certain operations lead to exploitable leakages, allowing a co-located attacker running on the same hardware as the victim code to extract cryptographic secrets. By microarchitectural timing leakage, we refer to architectural traces only, excluding transient execution attacks. As most of the existing RISC-V hardware finds usage in the IoT or the automotive domain, there has been more focus on physical attacks like power side-channels, and little work on analyzing the co-location scenario so far. However, it is likely that many attack vectors from x86 and ARM will apply to RISC-V systems as well. While there are several proposals for hardware countermeasures that would address this issue (e.g., resistant cache designs [11, 43, 49]), it is unlikely that all CPU vendors will include one of those mitigations in their processors. Thus, absent a proven hardware-based countermeasure, software-level mitigations are needed.

By now, most established libraries address timing leakages by employing so-called *constant-time code*, i.e., code that exhibits the same control flow and memory access pattern independent of its secret inputs. However, the new compiler backends and different instruction set of RISC-V may re-introduce leakage previously fixed at source level [3, 10]. In addition, there is ongoing work on assembly-level implementations of cryptographic primitives, which are carefully optimized to fully utilize the underlying hardware to achieve best performance [44], but may have subtle leakages. While there are lots of approaches for finding leakages on source-level or via generic languages, those cannot detect leakage introduced by the compiler. Finally, most of the corresponding proof-of-concept implementations lack usability [23] or do not apply to RISC-V.

In this work, we discuss the requirements of analyzing RISC-V software for side-channel leakages, and show how an established side-channel analysis framework can be adapted

to also support RISC-V binaries. For that, we build upon the *Microwalk* framework [51], that analyzes execution traces in order to identify vulnerabilities, and then yields a detailed leakage report. While Microwalk generates its execution traces through dynamic binary instrumentation (DBI), no such tool is yet available for RISC-V. Thus, we develop the first DBI tool for RISC-V, called MAMBO-V, which sets up on the MAMBO toolkit [18] for ARM, and show how we can use this tool to generate Microwalk-compatible traces. We evaluate our leakage analysis toolchain on several cryptographic libraries with support for RISC-V, and uncover multiple vulnerabilities.

1.1 Our Contribution

In summary, our contributions are:

- We analyze the similarities and differences between RISC-V and established architectures in terms of side-channel vulnerabilities, and extract requirements for building side-channel-resistant software on RISC-V.
- We implement MAMBO-V, a RISC-V port of the ARM-based DBI tool MAMBO, enabling us to natively instrument RISC-V binaries.
- We include MAMBO-V in the Microwalk framework for finding timing side-channels in software binaries, building the first toolchain for automatically analyzing RISC-V programs.
- We analyze several RISC-V builds of cryptographic libraries and detect various leakages.

The source code is available at <https://github.com/UzL-ITS/MAMBO-V>.

1.1.1 Responsible Disclosure

We disclosed the potentially exploitable AES vulnerabilities to the developers of the respective libraries, who all acknowledged our findings. They were mostly aware of the issues of the relevant implementations, and WolfSSL and OpenSSL have (undocumented) compiler flags which partially fix the leakages (see Section 6.3). At the time of submission, there is ongoing work on patches that ensure that the default implementations are secure, or on appropriate documentation changes.

2 Background

2.1 RISC-V

RISC-V is a reduced instruction set computer (RISC) load-store architecture, with a focus on broad availability through permissive licensing and high modularity to support all applications from small low-power IoT devices over personal mobile devices to large-scale general purpose computers. Its open-source character allows easy extensibility through a so-called base-plus-extension instruction set architecture (ISA). As a RISC architecture, only designated instructions operate on memory, whereas the arithmetic merely happens in registers. The most important standardized extensions for RISC-V are I, M, A, C, F, D, Zicsr and Zifencei, which are often grouped together as *RV64GC*. Also, more specialized extensions are drafted and partially ratified, such as the vector extension and scalar cryptographic extension [42]. Instruction encodings are designed to simplify hardware implementations to increase performance and efficiency [47].

2.2 Dynamic Binary Instrumentation

Binary instrumentation allows inserting code into an existing binary in order to monitor or modify the program's behavior. The insertion points are determined through user-supplied rules or callback functions.

Static binary instrumentation (SBI), also called binary rewriting, permanently inserts instrumentation code into the binary in an offline phase [12]. While this approach promises a small runtime overhead, it is error-prone due to relying on a correct disassembly of the program. In addition, SBI cannot handle special cases like just-in-time compilation or self-modifying code.

In *dynamic binary instrumentation* (DBI), the instrumentation code is added with the help of an instrumentation framework at runtime. The DBI framework combines application and instrumentation code and executes the resulting code directly on the target platform. DBI engines introduce a slightly higher overhead than SBI due to the code translation at runtime, but most prevalent instrumentation frameworks feature optimizations like caching, so each code block needs to be instrumented only once. Popular DBI engines include Intel Pin [30], DynamoRIO [9], QBDI [39] and the heavyweight analysis framework Valgrind [35], which were initially built for x86 and then, in some cases, extended to also support other architectures like ARM's AArch32 and AArch64.

However, as ARM is a RISC architecture and thus quite different to x86, x86-specific optimizations in a DBI engine may have little or even negative effects. MAMBO [18] is a DBI tool specifically designed and developed for ARM, making it suitable for efficiently

handling RISC architectures. In addition to some ARM-specific optimizations, MAMBO has general DBI features like a cache for storing already instrumented code and scanning new code in basic block units. Moreover, it supports behavioral transparency, which means that the execution of all ABI-compliant binaries is guaranteed to be correct. The application binary interface (ABI) defines the calling convention, which includes register allocation for parameters and stack pointer behavior.

2.3 Microarchitectural Side-Channels

In a cloud setting, usually, many processes from different customers share the same underlying hardware. These processes may work with sensitive data, which should not be leaked to an attacker. While there are many architectural safeguards in place to prevent data from flowing from one process to another directly, there are more subtle *side-channels* that use properties of the underlying microarchitecture to extract some information from the running code. One prominent example are so-called *cache attacks* [1, 7, 37, 53], where the attacker brings the (shared) CPU cache into a known state, and then monitors changes to this state in order to learn whether the victim has accessed data within a certain address range. This way, the attacker can infer the code line the victim is currently executing, or determine the index of a table lookup. Besides the cache, there are many more shared resources that the attacker can monitor and exploit, like the translation look-aside buffer [19] and the branch prediction unit [2]. Note that we only consider attacks that target architectural traces, so transient execution attacks like Spectre [25] are out-of-scope.

A commonly used software-based countermeasure against side-channel attacks is constant-time code without any secret-dependent memory accesses or branches [3]. This code exhibits the same control flow and data flow independent of the processed secret, so a side-channel attacker cannot learn anything by looking at an execution trace as provided by a cache attack. As cryptographic implementations are a primary target for side-channel attacks, most current cryptographic libraries feature constant-time code.

2.3.1 Leakage Detection Tools

To ease checking implementations for side-channel vulnerabilities, numerous tools and approaches have been proposed. Tools that analyze source code include *ct-fuzz* [20] that uses a specialized form of fuzzing, *ct-verif* [3] based on formal verification methods and *CaSym* [8] that symbolically executes the source code. Moreover, there are various tools that analyze binaries through static techniques, like *BINSEC/REL* [10] using symbolic execution, *CacheS* [46] combining symbolic execution with taint analysis, or

CacheAudit [14] which uses formal methods to find leakages on all paths of a program. Finally, dynamic binary approaches comprise statistical timing measurements like in *dudect* [40], constraint modeling in *Abacus* [5], as well as trace alignment in *DATA* [48] or trace merging in *Microwalk* [51].

3 Overview

We first describe requirements and our approach for analyzing the side-channel security of RISC-V implementations running in a co-located setting.

3.1 Analysis Approach

As described in Section 2.3, there are numerous tools and approaches for finding side-channel leakages in software. Any useful tool should unify the following properties [23, 51]: First, it should accurately localize the respective leakages, so the developer can directly understand the cause of a leakage and start building a patch. Then, the analysis should be fast enough, so there is immediate feedback whenever there is a code change. Finally, to aid adoption in the developer community, the tool should not be too hard to set up and use.

To check whether RISC-V code is leakage-free, focusing on the source code alone is insufficient. For example, there have been cases where a misguided compiler pass “optimized” constant-time code, producing binaries with leakages that are not present in the source code [3, 24]. Daniel et al. [10] further provide an extensive evaluation of different compiler versions, optimization levels and target architectures, showing that constant-time properties always need to be validated on the binary level. Compiling the code for x86 and using existing analysis tools is not sufficient either, as x86 compilers may use different optimization passes than RISC-V compilers. In addition, x86 has special extensions like AES-NI or the `pclmulqdq` instruction for carry-less multiplication (used in Galois counter mode), which may substitute otherwise leaking code paths.

The necessity to work with RISC-V specific assembly leaves the option to use either static or dynamic binary analysis. While static binary approaches offer some guarantees that purely dynamic tools cannot give, they often suffer from poor performance and require lots of manual interaction. On the other hand, dynamic analysis is heavily dependent on the achieved coverage, i.e., leakage can only be found in code that is actually executed. However, for cryptographic implementations, it was found that a small number of random test cases is sufficient to cover the relevant code [48, 51]. In addition, dynamic analysis is easy to use, as the user only has to call the respective primitives.

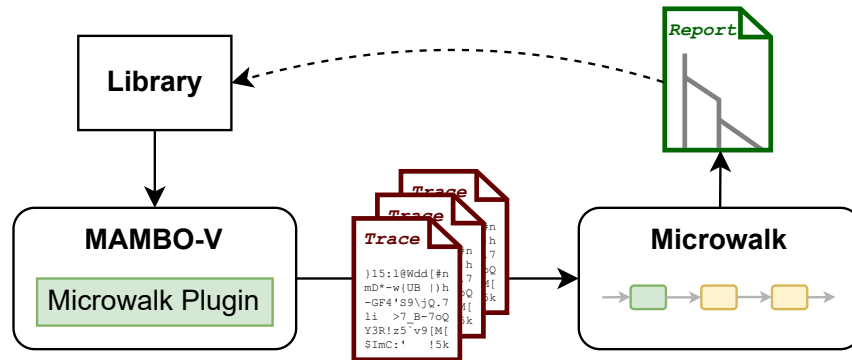


Figure 1: RISC-V side-channel analysis overview. MAMBO-V instruments a RISC-V library and generates execution traces, which are subsequently analyzed using Microwalk. The resulting analysis report then helps the developer to find and fix the identified leakages.

3.2 Toolchain

With the aforementioned requirements in mind, we picked the Microwalk framework [50, 51] as a basis for our RISC-V leakage analysis. Microwalk uses DBI to generate execution traces from user-supplied programs, and offers several analysis modules that compare these traces in order to find leakage. While the authors originally designed Microwalk for x86 binaries, its modular structure and generic trace format encourage addition of trace generators for other architectures.

This leaves the problem of generating Microwalk-compatible execution traces for RISC-V. At the time of writing, there is no generic DBI framework for RISC-V available, that offers the necessary flexibility for generating the information Microwalk needs. Another requirement is transparency, such that the execution traces are not influenced by the DBI engine itself, which would otherwise distort the analysis result. Instead of building a new DBI framework, we decided to port an existing framework for another RISC architecture, that is MAMBO [18] for ARM. The similarities between ARM and RISC-V allow us to reuse most of the general-purpose logic from MAMBO, like plugin handling or memory management. Our port, named MAMBO-V, implements the most significant performance optimizations from MAMBO, which are inline hash table lookups and direct branch linking. Additionally, we add support for atomic sequences, which need special handling on RISC-V hardware. We are working with the maintainers of MAMBO to contribute our RISC-V patches to the main project.

The resulting toolchain is illustrated in Figure 1.

4 MAMBO-V Implementation

We now describe our RISC-V port of the MAMBO DBI framework, named MAMBO-V. We give an overview over its generic features and discuss notable performance optimizations as well as RISC-V specifics to be considered.

4.1 Instrumentation Approach

4.1.1 Target Platform

MAMBO-V targets RV64GC platforms, i.e., processors with support for the RV64I base instruction set and its most common extensions. Like MAMBO, MAMBO-V aims for *behavioral transparency*: Binaries that are compliant to the standard RISC-V ABI are executed correctly. This does not affect the correctness of our side-channel analysis, as we can expect that compilers emit standard-compliant code and that the analyzed programs are not malicious.

4.1.2 Execution Model

Just as the ARM implementation of MAMBO, MAMBO-V unifies the instrumentation framework and the target application in a single process. On startup, a custom ELF loader reads the RISC-V ELF file and potential dependencies of the target application into the memory of the MAMBO-V process, such that the engine can access the target's full code. After initialization is done, MAMBO-V's dispatcher proceeds loading and translating chunks of the target's code on-the-fly, while inserting instrumentation at the points specified by the user. Each chunk consists of a single basic block, i.e., a sequence of instructions with a single entry point at the beginning and a single exit point at the end. This way, the dispatcher can safely hand over control to the translated chunk, and reclaim it after the chunk has fully executed.

4.1.3 Plugin API

In order to facilitate the usage of MAMBO-V for application developers who want to analyze their applications, we also ported the plugin API from MAMBO. A plugin contains user-supplied functions, which are called at certain events, e.g., when translating a basic block. With these functions, the user can then insert instrumentation code during translation. Other supported events are function entry/exit, threads and system calls. In our analysis, we primarily utilize the instrumentation to insert trace writing code.

4.1.4 Optimizations

To speed up analysis, we have ported a number of performance optimizations from MAMBO. Most of the overhead that arises during DBI comes from the code translation and context switches between the dispatcher and the target application. The most notable optimization is the code cache, which is a common feature of DBI frameworks: It is located outside the target application's address space and stores a limited amount of translated basic blocks. This avoids re-translation of frequently executed code, improving overall performance significantly. Other optimizations are hash tables for faster resolution of translated blocks and direct branch linking to speed up jumping between different blocks in the code cache without invoking a costly context switch to the dispatcher.

4.2 New Features for RISC-V

4.2.1 Atomic Sequences

A challenge we encountered on RISC-V cores are tightly constrained *atomic sequences*, which ensure exclusive memory operations for multiprocessor systems and process synchronization. Software locks for resources that should only be accessed by a single thread or process at a time are often translated to *atomic loops* by the compiler. An atomic loop contains an atomic sequence, which begins with a load-reserved (LR) instruction and ends with a store-conditional (SC) instruction. The atomic loop loops over the atomic sequence until the SC eventually succeeds. The result of the SC instruction depends on whether the reserved value was accessed during the atomic sequence and on the environmental constraints defined by the ISA. Among others, the ISA defines a maximum of 16 consecutive instructions between LR and SC, and allows only the base (I) instruction set, disallowing loads, stores, backward jumps or calls.

While the compiler enforces the constraints within an atomic sequence, the instrumentation done by MAMBO-V can insert arbitrary instructions that break one of the above constraints. Figure 2 shows an example of how a direct port of MAMBO would add unconstrained instructions to an atomic sequence: First, the original loop in Figure 2a is split into two blocks because of the conditional branch in line 3. Then, the resulting code cache blocks undergo optimization and are instrumented as shown in Figure 2b, leading to the insertion of unconstrained instructions (line 3-5). The result is a non-sequential sequence that includes loads, stores, calls, and potential backward jumps, and is therefore not guaranteed to succeed on RISC-V. However, requiring all instrumentation to adhere to the constraints would cause some instrumentation features to be lost in the process.

On ARM, where atomic sequences are available as well, MAMBO allows users to freely insert instrumentation, which when breaking a constraint causes undefined behavior,

```

-----
# a0: value to store
# a1: lock status
# s3: memory address
-----
1:loop:
2:  LR.D a1, (s3)
3:  BNE a1, zero, loop
4:  SC.D a1, a0, (s3)
5:  BNE a1, zero, loop
-----

```

(a) Original lock-acquire-loop.

```

-----
1:block1:
2:  LR.D a1, (s3)
3:  <branch condition evaluation>
4:  <call trace_conditional_branch>
5:  <cond. branch to block1 or block2>
6:block2:
7:  SC.D a1, a0, (s3)
8:  BNE a1, zero, loop
-----

```

(b) Instrumented lock-acquire-loop.

Figure 2: Exemplary instrumentation of a lock-acquire-loop: The instrumentation may insert unconstrained instructions (marked in **blue**) into the atomic sequence, e.g., add a function call with parameters to trace a conditional branch instruction. In order to set the argument registers, the original register contents have to be written to the stack using an unconstrained store instruction.

but does not affect stability on ARM Cortex processors. However, on our SiFive U54 core, violating a constraint can block the SC instruction from succeeding entirely, leaving the process stuck in a deadlock. We encountered such a deadlock when instrumenting the dynamic linker.

Thus, for reliable instrumentation on RISC-V cores, we designed a lightweight and behaviorally transparent solution for handling atomic sequences: We use hardware-assisted software emulation to relax the hardware constraints by replacing the LR and the SC instructions. The LR is replaced by an equivalent normal load instruction, which marks the beginning of the software-emulated atomic sequence. To emulate the reserve, we also back up the original value for later comparison. The subsequent code is not bound by constraints anymore and safe for arbitrary instrumentation. Finally, we replace the SC instruction with a semantically equivalent atomic sequence that conditionally stores the new value if the value at the destination is equal to the previously created backup. Since we include a native atomic sequence to check for changes at the destination, our emulation remains thread-safe. The observable behavior of the emulated atomic sequence is nearly identical to the original, with the only difference being that the emulation cannot detect stores on the reserved value that do not modify it. To the best of our knowledge, this difference does not effectively change the semantics of the emulated sequence, and therefore the traces remain identical.

4.2.2 Global Pointer and Thread Pointer Register

In contrast to ARM, the RISC-V standard calling convention defines a global pointer register `gp` and a thread pointer register `tp`. Applications use these registers to access structures such as the global offset table and global/thread-local variables. MAMBO-V does not share these structures with its client, so `gp` and `tp` must be updated on each of the

context switch between MAMBO-V and the client. Originally, on ARM, a unidirectional context switch was sufficient, as the dispatcher does not make assumptions on register contents on entry. Thus, only the context of the client is fully saved when entering the MAMBO-V context and restored when leaving again. To support the distinct gp/tp contexts on RISC-V, we implemented a full context switch for these two registers, while keeping the unidirectional context for all other registers to minimize the overhead.

4.2.3 Shorter Jump Encoding

RISC-V and ARM do not have direct branch instructions that take an absolute immediate address. Due to different instruction encodings, the maximum range of ARM branch instructions is ± 128 MiB, while on RISC-V it is only ± 1 MiB. The code cache in MAMBO-V can be much larger than 1 MiB. Hence, for MAMBO-V, we decided to use indirect jumps to transfer control flow back to the dispatcher. Loading the address and performing the jump takes 14 additional bytes in the code cache, but due to the long lifetime of translated code and runtime overhead of the client-dispatcher context switch the effect on the overall performance and memory consumption is negligible.

5 Side-Channel Leakage Analysis

In the following, we describe our approach for finding architecture-specific leakage in code compiled for RISC-V with the help of MAMBO-V. We focus on implementations of cryptographic algorithms, as their impact on the security of systems and communication is high. However, the concepts do apply to any scenario where secret information should not be exposed to an attacker recording execution traces. As discussed in Section 3, source-level analysis is often not sufficient, and binaries may contain leakages even though the original source code is constant-time. Therefore, we opted for a binary approach based on RISC-V-specific DBI for execution trace generation and Microwalk for leakage analysis.

5.1 Leakage Model

We adopt the leakage model as specified for Microwalk [51]: We supply the attacker with an implementation, a number of secret inputs and corresponding *execution traces*. An execution trace consists of a sequence of all executed instructions and accessed memory addresses, but does not contain actual processed data. The attacker also gets access to all public inputs and outputs. We consider the implementation constant-time if all traces are identical, i.e., when the attacker does not learn anything about the secret input by

looking at a trace. In other words, in a constant-time program, the observed control flow and memory accesses are independent of the secret inputs.

This leakage model assumes a rather strong attacker, as the known side-channel attacks can only retrieve a fraction of the information expressed in a full execution trace. For example, cache attacks are limited to granularities of 32 or 64 bytes on most systems, and control flow tracking techniques like single-stepping only work in very specific scenarios. Due to the lack of suitable hardware, there has not yet been much work on side-channels for RISC-V. Thus, while we expect similar vulnerabilities on upcoming RISC-V processors as are already known for other architectures, sticking to a strong leakage model is the safest way forward. We only consider secret-dependent control flow and memory accesses that are architecturally reachable, so transient execution attacks are out-of-scope.

5.1.1 Implementation in Microwalk

Microwalk implements the above leakage model through a simple dynamic analysis pipeline, which generates secret inputs (called *test cases*), collects and preprocesses corresponding execution traces, and finally compares those traces with each other. If Microwalk finds a difference between two or more traces at a given code position, this difference is reported as *leakage*, as an attacker may exploit this difference to tell apart two or more secret inputs. If all traces are identical, the attacker does not learn anything about the underlying secret inputs, and the implementation is reported as non-leaking.

5.2 Required Information

Microwalk uses a common generic execution trace format to run its analysis modules on, so we build a toolchain that collects RISC-V execution traces and converts them into Microwalk's format. Microwalk already offers two raw trace preprocessors, one for converting source-based execution traces from languages like JavaScript, and another one for binary traces from compiled code. While the binary trace preprocessor was originally written for x86, we found that its raw trace format is generic enough to also be used on other architectures. We thus only need to create a trace generator for RISC-V, that emits raw execution traces in the same format as the existing Intel Pin module (Figure 3).

A raw binary execution trace from Microwalk's Intel Pin module combines the following information:

- taken/non-taken branches, with source and (if applicable) target address;
- memory accesses, with instruction address and accessed memory address;

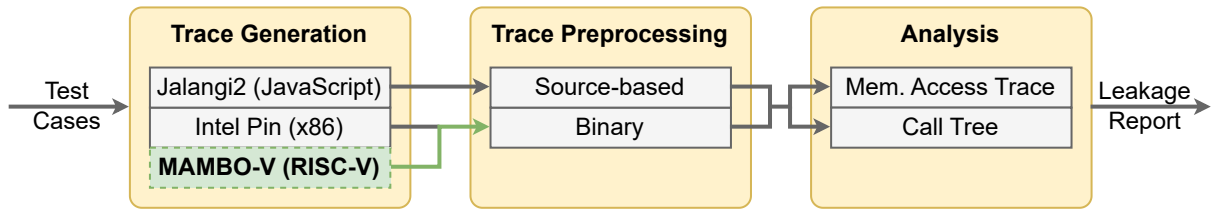


Figure 3: Microwalk pipeline with a new trace generation module based on MAMBO-V. Each trace generation module may emit either source-based or binary execution traces, which are then preprocessed into a common trace format that can be parsed by all analysis modules.

- heap/stack allocation blocks, with start and end address;
- start and end addresses of the memory-mapped executable binaries.

We collect this data using a plugin for the MAMBO-V DBI framework.

5.3 MAMBO-V Trace Plugin

5.3.1 Interaction with the Target Program

In order to analyze a cryptographic primitive, the primitive has to be made available to the DBI framework. We follow Microwalk’s approach by asking the user to supply a small function that receives a test case file with secret inputs and then calls the cryptographic primitive. Our MAMBO-V plugin registers a *function call* event callback for detecting execution of that function, so it can detect when test case execution starts and ends. This method has the advantage that we do not need to re-instrument the binary for each test case, but can reuse the existing instrumentation, which speeds up trace generation significantly. Before the first test case begins, we record a *trace prefix*, that contains initializations of all global objects that may be referenced during test case execution.

5.3.2 Recording Control Flow and Memory Accesses

When a test case begins, which is signaled by the respective event callback, our plugin opens a new binary trace file. We also register an instrumentation callback, which is called whenever a new basic block is instrumented. In this callback, we check each instruction for control flow and memory accesses, and add instrumentation to that instruction if necessary. The resulting instrumented code then writes to the trace file whenever the respective instruction is executed. To avoid tracing information outside our target functions, the plugin receives a list of binaries that should be traced.

5.3.3 Tracking Memory Allocations

Microwalk needs both a list of allocated heap memory blocks and the regions of the memory-mapped executables. To collect heap blocks, we register *function call* and *function return* event callbacks for the `malloc`, `calloc`, `realloc` and `free` functions, and log their parameters and return addresses. For the static memory regions, we hook into the *VM operation* event handler and extract the required information from `VM_MAP` events, which are triggered whenever a new ELF file is loaded.

6 Evaluation

To evaluate the performance of our toolchain and assess the current state of side-channel security on RISC-V, we analyze a number of frequently used cipher and signature functions for several popular libraries. We describe the experimental setup, analyze the performance of trace creation and analysis, and discuss and evaluate the discovered leakages. The results are summarized in Table 1.

6.1 Experimental Setup

As described in Section 3, we combine MAMBO-V with Microwalk to natively analyze the leakage of binaries on RISC-V. We record the traces with MAMBO-V on a Microchip PolarFire SoC FPGA Icicle Kit with four SiFive U54 cores featuring RV64GC. The trace analysis with Microwalk is executed on an AMD Ryzen 9 7950X with 16 cores.

6.1.1 Libraries

Due to its modular structure, the RISC-V architecture allows for a broad range of target applications, from small embedded devices to server CPUs. To reflect this, we chose to analyze WolfSSL [52] and Mbed TLS [31] as examples for libraries that support many architectures and that are optimized for the embedded market. OpenSSL [36] and GNU Nettle [16], on the other hand, are general purpose cryptography libraries that are used across different architectures and chip sizes. In addition, as an example of a library specifically written for RISC-V, we investigated SCL (SiFive Cryptographic Library) [44]. Finally, as a reference for constant-time implementations, we included libsodium [29].

Table 1: Result of leakage analysis of several cryptographic libraries on RISC-V. “Tr. CPU” shows the CPU time for generating the raw traces and “An. CPU” the CPU time for trace preprocessing and analysis. The columns “# Lkgs.” and “# Uniq.” show the total and unique number of detected leaking code lines.

Target	Type	Tr. CPU	An. CPU	# Lkgs.	# Uniq.
WolfSSL [52] 5.5.4					
AES-ECB	cipher	1 sec	< 1 sec	157	157
AES-GCM	aead-cipher	2 sec	< 1 sec	493	184
ChaCha20-Poly1305	aead-cipher	< 1 sec	< 1 sec	0	0
Ed25519	signature	36 sec	< 1 sec	0	0
ECDSA (secp192r1)	signature	880 sec	7 sec	105	10
Mbed TLS [31] 3.3.0					
AES-ECB	cipher	2 sec	< 1 sec	68	68
AES-GCM	aead-cipher	4 sec	< 1 sec	216	76
ChaCha20-Poly1305	aead-cipher	7 sec	< 1 sec	0	0
OpenSSL [36] 3.0.0					
AES-ECB	cipher	115 sec	< 1 sec	52	52
AES-GCM	aead-cipher	117 sec	< 1 sec	166	60
ChaCha20-Poly1305	aead-cipher	117 sec	< 1 sec	0	0
Ed25519	signature	556 sec	4 sec	0	0
ECDSA (secp192r1)	signature	3128 sec	30 sec	1647	284
GNU Nettle [16] 3.8.1 with GMP [15] 6.2.1					
AES-ECB	cipher	2 sec	< 1 sec	32	32
AES-GCM	aead-cipher	3 sec	< 1 sec	108	40
ChaCha20-Poly1305	aead-cipher	2 sec	< 1 sec	0	0
Ed25519	signature	104 sec	4 sec	0	0
SCL - SiFive Cryptographic Library [44] 20.08.00					
ECDSA (secp256r1)	signature	102 sec	< 1 sec	5	2
libsodium [29] 1.0.18					
ChaCha20-Poly1305	aead-cipher	2 sec	< 1 sec	0	0
Ed25519	signature	12 sec	< 1 sec	0	0

6.1.2 Analyzed Primitives

We wrote analysis wrappers for AES-ECB, the authenticated encryption schemes AES-GCM and ChaCha20-Poly1305, and the signature algorithms Ed25519 and ECDSA (curve secp192r1; secp256r1 for SCL). The wrappers initialize the necessary environment and call the target functions, if supported by the respective library. We skipped the ECDSA implementations in GNU Nettle and Mbed TLS, as those are comparably slow and thus lead to traces which exceed the limited resources of our evaluation platform.

All libraries and target wrappers were cross-compiled with the RISC-V GNU Compiler

Toolchain 12.2.0 [41] for RV64GC and ISA specification 2.2. We built all libraries with default options and appropriate additional security flags as stated in their documentation. All libraries except OpenSSL are built with optimization level -O2. OpenSSL was built with optimization level -O3.

6.1.3 Test cases

We generated 16 test cases for each primitive by creating 16 random keys, and supplied these test cases to the target function. Since Microwalk measures differences in the execution traces, any other input outside the test cases must be kept constant to avoid false positives. Therefore, inputs such as initialization vectors were set to fixed values. Random values like the ephemeral key in ECDSA were generated by custom test case-dependent RNGs. We opted for using smaller key sizes, as the cryptographic procedures are invariant of the key size, and larger key sizes increase the resource consumption of the leakage analysis without uncovering further vulnerabilities [51].

6.2 Performance Results

The performance of the side-channel analysis on RISC-V depends on the time required for tracing the target function and analyzing the traces. The runtime for all targets is summarized in Table 1.

6.2.1 Tracing

The duration of tracing 16 executions for each target is inherently constrained by the limited performance of the SiFive U54 core. For the symmetric ciphers and Ed25519, the tracing took at most a few minutes, which suggests that our toolchain is suitable for everyday use on a developer's computer. With newer and more performant RISC-V cores, the tracing time should further decrease.

One outlier is OpenSSL, where a majority of the tracing time was spent in the library initialization, which is mostly irrelevant for the leakage analysis. To reduce this overhead, the developer could disable most features when compiling the library for vulnerability evaluation and target low-level functions.

6.2.2 Analysis

With one exception, the trace preprocessing and analysis of nearly all targets took less than 5 seconds. The fast analysis allows for frequent execution of any test. The outlier, ECDSA for OpenSSL, was slowed down by preprocessing the huge traces, so optimizing the tracing time should fix this as well.

6.3 Vulnerabilities

The leakage analysis for the chosen popular libraries shows many vulnerabilities across the board, except for libsodium which only implements a limited number of ciphers and signature algorithms that allow for an implementation with better resistance against timing attacks by design. Indeed, all analyzed implementations of ChaCha20-Poly1305 and Ed25519 are constant-time. We summarize the results in Table 1 in the columns “# Lkgs.” (total leakages) and “# Uniq.” (unique leakages). An instruction or function can be called or reached from multiple contexts, thus potentially leaking different secrets with varying leakage severity. Therefore, we also count unique occurrences of leaking instructions.

In-depth analysis of the libraries showed that most provide specific assembly implementations for x86 and other architectures that use constant-time primitives. For RISC-V though, due to lack of specifically optimized implementations, the libraries fell back to default ones, which often turned out to be non-constant-time, even when using the hardening flags specified in the documentation.

6.3.1 Symmetric Ciphers

All analyzed AES-ECB implementations leak secret information through their timing behavior. The examined libraries do not provide RISC-V-specific code, but fall back to their default C/C++ implementations, which use either T-table or S-box lookups for AES encryption and round key generation. Previous work has shown that table lookups are exploitable by timing measurements [7]. The number of unique leakages varies between the different libraries depending on whether the encryption rounds are unrolled and how the final step is scheduled. After informing the OpenSSL developers that we found several leakages in the default AES-ECB implementation, we were pointed to an undocumented compiler flag that enables an alternative AES implementation, which we verified to be constant-time. However, they also stated that the flag leads to a 95% performance loss, which is why it is not enabled by default.

The authenticated encryption algorithm AES-GCM builds upon the same primitives as AES-ECB and thus also shows the same table lookup leakage for the encryption step. In addition, the GCM mode adds authentication through computation of a GHASH, which involves encryption of a 128-bit string of zeros and the IV. The result of the latter encryption is used for the final computation of the authentication data. The multiplication used for the GHASH is implemented with a hash lookup table, where the accessed index depends on the current ciphertext and the hash value of the previous block.

We compared the leakage result of AES-GCM on RISC-V for the libraries OpenSSL and Mbed TLS against the analysis on x86. While the RISC-V binaries contain many leakages as explained above, we observed no leakages for x86 binaries. The x86 implementations use the AES-NI hardware extension for encryption and the `clmul` extension for computation of the GHASH. Until such extensions are available for RISC-V, cryptographic libraries must feature constant-time software implementations. For WolfSSL, we learned during disclosure that there is a `GCM_SMALL` flag, which enables a non-table-based GHASH implementation. While designed (and documented) primarily for small code size, we found that it is constant-time and thus a secure alternative for the default implementation.

6.3.2 Asymmetric Signature Algorithms

None of the analyzed implementations of Ed25519 shows any non-constant-time behavior, emphasizing its inherent resistance against timing attacks, even though there are no specific assembly implementations for RISC-V. However, we found leakage for all analyzed implementations of ECDSA, especially in the implementation from OpenSSL. Even the specially crafted RISC-V implementation from SCL reveals non-constant-time behavior, though the library is not yet deemed production-ready. Despite the high number of potential vulnerabilities, we found that all analyzed ECDSA implementations use blinding, rendering the discovered leakages likely unexploitable.

7 Discussion and Future Work

7.0.1 Limitations of Microwalk

As we base our analysis on Microwalk, we inherit some of its limitations. Currently, Microwalk only supports deterministic implementations. Thus, all entropy must come from the secret inputs. While this scenario works well with symmetric and constant-time asymmetric cryptographic primitives, it has some issues with blinded implementations which obscure the computation by randomizing the input parameters. Disabling the

randomness is not sufficient either, as this would just expose leakages which are normally obscured by blinding. As a solution, Microwalk should be extended to support randomized implementations. Another limitation of Microwalk’s analysis algorithm is the possibility of several small leakages higher up in the call chain hiding leakages further down, though we did not observe this during our evaluation. Finally, Microwalk’s dynamic approach heavily depends on the coverage. While it was found that few random test cases usually suffice [48, 51], the user should check that all relevant code locations have been reached.

7.0.2 Other Applications of MAMBO-V

While we used MAMBO-V for generating execution traces, the tool is far more versatile. The plugin API supports a variety of different callbacks, making it on par with other widely-used frameworks like Intel Pin. For example, new plugins can aid with control-flow checks or help in bug detection. The broad similarities to ARM allow reusing analysis code originally written for MAMBO with little adjustments.

7.0.3 Leakage Analysis on ARM

The proximity of RISC-V and ARM suggests that the MAMBO-V trace generator plugin can be ported to the original MAMBO implementation with little adjustments. With that plugin, one could generate execution traces from ARM binaries, and analyze these traces for side-channel vulnerabilities using Microwalk, yielding a dynamic leakage analysis toolchain for ARM. Thus, our toolchain comprising a tracer plugin and Microwalk provides a solid basis for fast and accurate side-channel leakage analysis on various systems.

8 Related Work

8.0.1 Analysis of Code on Intermediate Representations

Instead of instrumenting code natively, the machine code can be lifted to a generic intermediate representation. This approach is taken by the ongoing RISC-V port [38] of the heavyweight instrumentation framework Valgrind [35] and the full-system emulator QEMU [6], which do an emulated analysis of RISC-V instructions on the intermediate representations of the respective framework. Thereby, it is possible to re-use existing analysis tools like memory leaks detection or call graphs. Apart from that, the whole system reverse engineering tool PANDA [13] provides a way to capture an execution trace, replay

it afterwards and combine it with extensive analysis through different plugins. However, emulated analysis meets a different objective than analyzing architecture-specific leakage, as the leakage may be hidden during lifting to the intermediate representation. Furthermore, the emulators impose a very high overhead and are too resource-consuming to use them in restricted environments or for an efficient analysis with Microwalk.

8.0.2 Side-Channel Analysis

Side-channel attacks on RISC-V are receiving growing attention by security research. Apart from the timing side-channels we analyze in this work, there have been efforts to secure RISC-V implementations against leakage through power side-channels [32]. Further, electromagnetic leakage builds the basis for a successful fault attack in [34], showing that manifold leakage channels need to be addressed. As some RISC-V systems also support out-of-order execution, they are susceptible to Spectre [25] attacks [17, 27]. Recently, it was shown that data can be leaked from speculative execution through cache attacks [28]. The vulnerability to Spectre-style attacks further motivates the development of a framework to automatically detect timing side-channels in software, because apart from direct exploitation, the timing differences can also be used as a way to leak speculatively accessed secrets.

8.0.3 Hardware-Based Countermeasures

A RISC-V working group developed a number of extensions intended for secure cryptography, which were ratified in 2022 [42]. This includes hardware-acceleration for symmetric encryption and hash functions, but also the *Zkt* extension, which specifies constant-time properties for certain instructions. If a vendor implements the *Zkt* extension, certain arithmetic instructions are guaranteed to have data-independent execution time. However, solely instruction-based approaches are insufficient, as most vulnerabilities are caused by higher-level data-dependent behavior. Yu et al. propose support for oblivious memory accesses, which would block most timing side-channels [54] and thus go far beyond simply avoiding data-dependent instruction latency like in the *Zkt* extension. With hardware-integrated fully automated Boolean masking [45], hardly any software-level precautions need to be taken against power side-channels. To protect against data leakages in ALU, memory and memory interfaces, INVITED [32] uses state-of-the-art masking techniques.

However, these hardware mechanisms are always applied, not only for secret inputs, making the solutions potentially inefficient for workloads where only a small fraction of all executed instructions is truly security-critical. Moreover, in a cloud scenario, the

clients have limited control about the hardware actually used, making secure software implementations indispensable.

9 Conclusion

In this paper, we have presented the first comprehensive side-channel analysis for implementations of cryptographic primitives on RISC-V. We have shown that some of the most popular open-source cryptographic libraries lack proper side-channel resistance on RISC-V. For our work, we have studied the requirements for leakage detection on RISC-V and designed a thorough approach to incorporate all requirements into a mature side-channel analysis framework that we have extended with all necessary building blocks. We have based our analysis toolchain on Microwalk and augmented the framework with the necessary RISC-V specific tracing capabilities by implementing the DBI tool MAMBO-V. Our evaluation pinpoints several potentially exploitable leakages that should be fixed by the developers and emphasizes the need for complete and precise side-channel analysis capabilities on RISC-V to pave the way for secure computations on shared RISC-V hardware in the cloud.

Acknowledgements

We thank the library maintainers for the smooth disclosure process, and the reviewers and our shepherd for their helpful comments and suggestions. This work has been supported by DFG under grants 427774779 and 439797619, and by BMBF through projects ENCOPIA and PeT-HMR.

References

- [1] Onur Aci mez, Billy Bob Brumley, and Philipp Grabher. “New Results on Instruction Cache Attacks”. In: *12th International Cryptographic Hardware and Embedded Systems Workshop (CHES)*. 2010. DOI: 10.1007/978-3-642-15031-9_8.
- [2] Onur Aci mez,  etin Kaya Ko , and Jean-Pierre Seifert. “Predicting Secret Keys Via Branch Prediction”. In: *The Cryptographers’ Track at the RSA Conference 2007 (CT-RSA)*. 2007. DOI: 10.1007/11967668_15.

- [3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>.
- [4] Krste Asanović and David A Patterson. “Instruction Sets Should be Free: The Case for RISC-V”. In: *EECS Dpt., Univ. of CF, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).
- [5] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. “Abacus: Precise Side-Channel Analysis”. In: *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2021. DOI: 10.1109/ICSE43902.2021.00078.
- [6] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *2005 USENIX Annual Technical Conference (USENIX ATC)*. 2005. URL: <http://www.usenix.org/events/usenix05/tech/freenix/bellard.html>.
- [7] Daniel J Bernstein. *Cache-timing attacks on AES*. 2005.
- [8] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00022.
- [9] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. “An Infrastructure for Adaptive Dynamic Optimization”. In: *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*. 2003. DOI: 10.1109/CGO.2003.1191551.
- [10] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure”. In: *ACM Trans. Priv. Secur.* 26.2 (2023), 11:1–11:42. DOI: 10.1145/3563037.
- [11] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. “HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky>.
- [12] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00009.

- [13] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. “Repeatable Reverse Engineering with PANDA”. In: *5th Program Protection and Reverse Engineering Workshop (PPREW@ACSAC)*. 2015. DOI: 10.1145/2843859.2843867.
- [14] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *22nd USENIX Security Symposium*. 2013. URL: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev>.
- [15] GMP. <https://ftp.gnu.org/gnu/gmp/>. (Visited on 2024-05-21).
- [16] GNU Nettle. <https://git.lysator.liu.se/nettle/nettle>. (Visited on 2024-05-21).
- [17] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanovic. “Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture”. In: *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2019.
- [18] Cosmin Gorgovan, Amanieu D’Antras, and Mikel Luján. “MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM”. In: *ACM Trans. Archit. Code Optim.* 13.1 (2016), 14:1–14:26. DOI: 10.1145/2896451.
- [19] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>.
- [20] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. “ct-fuzz: Fuzzing for Timing Leaks”. In: *13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 2020. DOI: 10.1109/ICST46399.2020.00063.
- [21] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *18th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2016. DOI: 10.1007/978-3-662-53140-2_18.
- [22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.42.

- [23] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ““They’re not that hard to mitigate”: What Cryptographic Library Developers Think About Timing Attacks”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833713.
- [24] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. “When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015”. In: *15th International Cryptology and Network Security Conference (CANS)*. 2016. DOI: 10.1007/978-3-319-48965-0_36.
- [25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00002.
- [26] Vinay B. Y. Kumar, Suman Deb, Naina Gupta, Shivam Bhasin, Jawad Haj-Yahya, Anupam Chattopadhyay, and Avi Mendelson. “Towards Designing a Secure RISC-V System-on-Chip: ITUS”. In: *J. Hardw. Syst. Secur.* 4.4 (2020), pp. 329–342. DOI: 10.1007/s41635-020-00108-8.
- [27] Anh-Tien Le, Ba-Anh Dao, Kuniyasu Suzuki, and Cong-Kha Pham. “Experiment on Replication of Side Channel Attack via Cache of RISC-V Berkeley Out-of-Order Machine (BOOM) Implemented on FPGA”. In: *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2020.
- [28] Anh-Tien Le, Trong-Thuc Hoang, Ba-Anh Dao, Akira Tsukamoto, Kuniyasu Suzuki, and Cong-Kha Pham. “A cross-process Spectre attack via cache on RISC-V processor with trusted execution environment”. In: *Comput. Electr. Eng.* 105 (2023), p. 108546. DOI: 10.1016/j.compeleceng.2022.108546.
- [29] libsodium. <https://github.com/jedisct1/libsodium>. (Visited on 2024-05-21).
- [30] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *2005 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2005. DOI: 10.1145/1065010.1065034.
- [31] Mbed-TLS. <https://github.com/Mbed-TLS/mbedtls>. (Visited on 2024-05-21).
- [32] Elke De Mulder, Samatha Gummalla, and Michael Hutter. “Protecting RISC-V against Side-Channel Attacks”. In: *56th Annual Design Automation Conference (DAC)*. 2019. DOI: 10.1145/3316781.3323485.

- [33] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. “HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment”. In: *2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2021. DOI: 10.1145/3433210.3453112.
- [34] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. “Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 28–68. DOI: 10.46586/tches.v2022.i1.28-68.
- [35] Nicholas Nethercote and Julian Seward. “Valgrind: A Program Supervision Framework”. In: *Third Workshop on Run-time Verification (RV@CAV)*. 2. 2003. DOI: 10.1016/S1571-0661(04)81042-9.
- [36] OpenSSL. <https://github.com/openssl/openssl>. (Visited on 2024-05-21).
- [37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *The Cryptographers’ Track at the RSA Conference 2006 (CT-RSA)*. 2006. DOI: 10.1007/11605805_1.
- [38] Petr Pavlu. *Valgrind RISC-V Port*. <https://github.com/petrpavlu/valgrind-riscv64>. Free and Open Source Software Developers’ European Meeting (FOSDEM) 2022. (Visited on 2024-05-21).
- [39] QBDI. *Quarkslab Dynamic binary Instrumentation*. <https://qbdi.quarkslab.com/>. (Visited on 2024-05-21).
- [40] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time?” In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017. DOI: 10.23919/DATE.2017.7927267.
- [41] RIn. *RISC-V GNU Compiler Toolchain*. <https://github.com/riscv-collab/riscv-gnu-toolchain>. (Visited on 2024-05-21).
- [42] RISC-V. *RISC-V Cryptography Extensions Volume I*. <https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar>. (Visited on 2024-05-21).
- [43] Gururaj Saileshwar and Moinuddin K. Qureshi. “MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design”. In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar>.
- [44] SiFive Cryptographic Library (SCL). <https://github.com/sifive/scl-metal>. (Visited on 2024-05-21).
- [45] Kleber Stangherlin and Manoj Sachdev. “Design and Implementation of a Secure RISC-V Microprocessor”. In: *IEEE Trans. Very Large Scale Integr. Syst.* 30.11 (2022), pp. 1705–1715. DOI: 10.1109/TVLSI.2022.3203307.

- [46] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. "Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation". In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai>.
- [47] Andrew Waterman. "Design of the RISC-V Instruction Set Architecture". PhD thesis. University of California, Berkeley, USA, 2016. URL: <https://www.escholarship.org/uc/item/7zj0b3m7>.
- [48] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [49] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization". In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/werner>.
- [50] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries". In: *34th Annual Computer Security Applications Conference (ACSAC)*. 2018. DOI: 10.1145/3274694.3274741.
- [51] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. "Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications". In: *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022. DOI: 10.1145/3548606.3560654.
- [52] WolfSSL. <https://github.com/wolfSSL/wolfssl>. (Visited on 2024-05-21).
- [53] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium*. 2014. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.
- [54] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing". In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. 2019. URL: <https://www.ndss-symposium.org/ndss-paper/data-oblivious-isa-extensions-for-side-channel-resistant-and-high-performance-computing/>.

Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software

Publication

Jan Wichelmann*, Anna Pättschke*, Luca Wilke, and Thomas Eisenbarth. *Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software*. In *32nd USENIX Security Symposium*, 2023.

Contribution

Co-main author. Anna Pättschke contributed the information flow analysis, while I designed and implemented the actual code hardening.

Personal contributions:

- development of a binary rewriting framework that hardens all identified memory accesses by inserting masking logic, without requiring recompilation;
- design of various runtime secrecy tracking and mask generation strategies;
- evaluation of the performance and security of the different Cipherfix variants on several cryptographic libraries.

Outline

1	Introduction	207
2	Background	209
3	CIPHERFIX Design	212
4	Leakage Localization and Preprocessing	217
5	Static Mitigation	221
6	Evaluation	227
7	Discussion	234
8	Related Work	237
9	Conclusion	238
	References	239
A	Static Instrumentation Example	245
B	Evaluation Results	246

Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software

Jan Wichelmann*, Anna Päsche*, Luca Wilke, and Thomas Eisenbarth

Universität zu Lübeck

Trusted execution environments (TEEs) provide an environment for running workloads in the cloud without having to trust cloud service providers, by offering additional hardware-assisted security guarantees. However, main memory encryption as a key mechanism to protect against system-level attackers trying to read the TEE's content and physical, off-chip attackers, is insufficient. The recent Cipherleaks attacks infer secret data from TEE-protected implementations by analyzing ciphertext patterns exhibited due to deterministic memory encryption. The underlying vulnerability, dubbed the ciphertext side-channel, is neither protected by state-of-the-art countermeasures like constant-time code nor by hardware fixes.

Thus, in this paper, we present a software-based, drop-in solution that can harden existing binaries such that they can be safely executed under TEEs vulnerable to ciphertext side-channels, without requiring recompilation. We combine taint tracking with both static and dynamic binary instrumentation to find sensitive memory locations, and mitigate the leakage by masking secret data before it gets written to memory. This way, although the memory encryption remains deterministic, we destroy any secret-dependent patterns in encrypted memory. We show that our proof-of-concept implementation protects various constant-time implementations against ciphertext side-channels with reasonable overhead.

1 Introduction

The current trend for data processing and provisioning of infrastructure heads towards cloud computing, with many co-located clients sharing the same physical hardware instead of working in isolated self-hosted environments. To protect different clients from each other, as well as the hypervisor from the clients, virtual machines (VMs) are used to provide isolation. However, especially when processing sensitive data, users may also want isolation from the hypervisor for data privacy or regulative reasons. This kind of isolation can be provided by trusted execution environments (TEEs), which model the hypervisor as an untrusted party. To achieve this kind of isolation, TEEs use a combination of additional access rights and cryptography to prevent the hypervisor, or

more general, any privileged attacker, from reading the content of the TEE or interfering with its execution state.

Nevertheless, sharing the same hardware leads to traces in shared resources like caches which in turn provides an attack surface for timing or microarchitectural side-channels [6, 10, 28, 34, 40]. A widely used countermeasure against these side-channels is constant-time code that is data oblivious, i.e., does not access memory or decide for branch targets based on secrets [1, 52]. To support developers, there are various mostly automated constant-time analysis tools that observe different properties of software traces for finding microarchitectural or timing leakage that could lead to exploitable side-channels [1, 17, 49, 50, 51, 52]. As these tools advance the constant-time properties of code, leakages get smaller and harder to find, though recent research has shown that even very small leakages are exploitable, especially when the strong attacker model of TEEs is considered [5, 36, 47].

The recent Cipherleaks paper [33] and its follow-up [31] introduced a new attack vector on code running in TEEs, dubbed the ciphertext side-channel. The core idea is that some TEEs use deterministic memory encryption, resulting in a one-to-one mapping between plaintexts and ciphertexts for a given memory block. As a result, the attacker can correlate changes in the ciphertext to the processed data. For example, the secret decision bit of a constant-time swap operation can be leaked by observing whether the ciphertext of the corresponding memory location changes, showing that state-of-the-art constant-time code is not secure under this attacker model. Thus, this attack vector demands for new analysis methods and countermeasures.

In this work, we introduce an analysis technique to mitigate ciphertext side-channel leakages in constant-time code. A naive approach hardening every memory write access would result in a very high performance overhead. Thus, our technique uses secret-tracking to pinpoint critical memory accesses, that are then safeguarded by randomizing observable write patterns such that the resulting binary does not leak information through the ciphertext side-channel. By combining static and dynamic approaches, we design a solution that covers all program components and works without recompilation.

1.1 Our Contribution

We present the CIPHERFIX framework, the first general-purpose drop-in mitigation for ciphertext side-channel-based leakages. This includes the following contributions:

- We propose an analysis technique based on dynamic taint analysis to find all secret-containing memory locations in constant-time binaries that are potentially vulnerable to the ciphertext side-channel.
- We employ dynamic binary analysis to locate stack variables and enable context-aware tracking of heap allocations, in order to support robust static instrumentation.
- We develop a mitigation technique, based on static binary instrumentation, that hardens the software binary across library boundaries without requiring recompilation and that provides three different security levels.
- We evaluate our proof-of-concept implementation of CIPHERFIX regarding performance and security on various primitives from four widely-used cryptographic libraries and discuss the effects of different mitigation approaches.

Our source code is available at <https://github.com/UzL-ITS/Cipherfix>.

Outline. After providing background in Section 2, we give an overview over the design of CIPHERFIX in Section 3. In Section 4, we present our dynamic analysis, which we use to build the static mitigation as described in Section 5. We evaluate the performance and security of our mitigation in Section 6. Finally, in Section 7, we discuss design decisions of CIPHERFIX and point out angles for future work.

2 Background

2.1 Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (SEV) is a trusted execution environment (TEE) that is designed as a drop-in solution to protect whole virtual machines. It encrypts the RAM content of the VM with an encryption key inaccessible to the hypervisor [26]. The latest iteration, SEV Secure Nested Paging (SEV-SNP) [3], prevents the hypervisor from remapping or modifying VM memory, thwarting attacks like [12, 20, 32, 37, 53]. For the memory encryption, SEV uses AES-128 in the XOR-Encrypt-XOR (XEX) [45] mode of operation, where a tweak value is XOR-ed before and after encryption. SEV derives the tweak values from the physical address of a 16-byte memory block and a random seed generated at boot time.

2.2 Ciphertext Side-Channel

The ciphertext side-channel was first introduced in [33] and later generalized to arbitrary memory regions and implementations in [31]. Both papers extract cryptographic keys from state-of-the-art constant-time cryptographic implementations running in SEV-SNP VMs. While the attack vector in [33] has been fixed on a firmware level [2], the attacks from [31] remain unaddressed. The core idea is exploiting the deterministic encryption at a fixed memory location, to leak information by precisely observing changes in the ciphertext and correlating them with the (known) executed code.

The authors of [31] introduce two attack variants: The *collision* and the *dictionary* attack. Both attacks exploit repeated write operations to the same memory address. The collision attack extracts information from observing the same ciphertext over multiple writes. One common example is the `cswap` pattern (Figure 1): A variable is always written, but depending on a secret decision bit the old or the new value is selected. While in the former case the deterministic ciphertext remains unchanged, in the latter case a new value is written, producing a different ciphertext. Thus, by observing the ciphertext of the memory location before and after the `cswap`, the attacker can immediately infer the secret decision bit. In the dictionary attack, the attacker does not only rely on collisions, but maps ciphertexts to (partially) known plaintexts. As the dictionary attack relies on repeating ciphertexts as well, mitigating the collision attack also mitigates the dictionary attack.

While the attacks above target values explicitly written to memory by the application, they can also be used to extract register values. For this, the authors of [31] exploit that the operating system running in the SEV-protected VM stores the user space register values upon context switches on the stack. This mechanism allows an attacker to extract secrets residing in registers by forcing context switches and observing the ciphertexts. However, the authors also describe how to fix this issue, by randomizing the stack layout.

2.3 Binary Instrumentation

Binary instrumentation allows modifying compiled programs without access to the source code. This is commonly used to insert new code that gathers information.

Dynamic binary instrumentation (DBI) gives the opportunity to include the architectural state by executing the analysis routines while the program is running. There are numerous DBI frameworks, e.g., Valgrind [39], Intel Pin [35], DynamoRIO [9] or DynInst [11]. The Intel Pin framework compiles and inserts analysis instructions at runtime through an x86 just-in-time (JIT) compiler. The code is processed in units called *basic blocks*, which are defined as instruction sequences that have a single entry and exit

```

cswap(p, q, b):
  c = ~(b - 1);    // b = 0 -> c = 00...00
  t = c & (p ^ q);
  p ^= t;
  q ^= t;

```

(a) Constant-time swap of p and q , depending on bit b .

Ciphertext of p		
b	before cswap	after cswap
0	e4c80f2a	e4c80f2a
1	e4c80f2a	aa2f2a61

(b) Ciphertext of p , before and after calling cswap.

Figure 1: cswap and resulting ciphertexts for the encrypted RAM accessible by the attacker. 1a shows the procedure of a constant-time swap. Depending on the value of a secret decision bit b , the values p and q are swapped ($b = 1$), or left as-is ($b = 0$). 1b shows the effect on the resulting ciphertext: If the ciphertext did not change, the attacker can infer that $b = 0$; if the ciphertext changed, the attacker learns that $b = 1$.

point. Through a number of callbacks, a so-called *Pintool* specifies the analysis code to be inserted during JIT compilation. The original instructions and the analysis code are combined such that the instrumentation is transparent to the analyzed program.

Static binary instrumentation (SBI) results in a modified standalone binary that is obtained by the use of rewriting or redirecting techniques. The execution of an instrumented binary does not depend on an instrumentation framework, which means that the main overhead comes from the inserted analysis code [4]. However, static instrumentation struggles with analyzing indirect branches, shared libraries and dynamically generated code [30, 35]. There are different approaches for adding analysis code to the binary at specific instrumentation points and then redirecting the control flow, such that both analysis and original application code are executed in the right order. To avoid breaking references, the instrumented code can be put into a separate `.instrument` section. An instrumentation point then redirects execution to this section, either through software breakpoints via the `int3` [38] instruction and a custom signal handler, or through direct jumps via so-called *trampolines* [11, 23, 24]. It is possible to combine multiple approaches to minimize their shortcomings, e.g., by inserting 5-byte jumps where possible, and falling back to 2-byte jumps or `int3` when not enough space is available. An example of trampoline-based instrumentation is illustrated in Figure 8 in the appendix. Recent binary rewriting approaches further optimize the instrumentation through using available

metadata for lifting [54] or symbolization of references [19].

2.4 Dynamic Taint Analysis

Dynamic taint analysis (DTA) tracks the flow of selected information through a program during code execution. The data to be tracked is marked as a *taint source*, and its propagation is defined through a *taint policy*. The policy also determines the *taint sinks* that can be reached by the data. All instructions that process secret data are considered for the taint propagation. Data flow tracking can be done in various granularities, whereby byte-level tracking is the most commonly used. For each memory location and register, there is shadow memory containing the taint label information, so the performance overhead is directly connected to the granularity. If too much data is marked as tainted, this is called *overtainting*; tainting too little data is referred to as *undertainting* [4, 27, 46].

A widely-used x86 taint analysis tool providing fast taint propagation based on Intel Pin is libdft [27]. In order to also support 64-bit binaries, libdft has been extended for VUzzer64 [44] and the AngoraFuzzer [14]. The data flow-based byte-level taint propagation in libdft64 is implemented through handwritten rules for every instruction class.

3 CIPHERFIX Design

We first give an overview of the generic design of our ciphertext side-channel counter-measure.

3.1 Attacker Model

We assume an attacker that tries to extract secret information from a TEE, that is protected with a deterministic block-based memory encryption with address-dependent tweaks. The attacker knows the exact binary which is executed by the victim, but cannot access secret data that is stored within the TEE. They have root access to the machine running the TEE and are able to read the entire encrypted memory, but cannot decrypt or modify it. Furthermore, the attacker can make use of a controlled channel that allows them to track and interrupt the code running inside the victim's TEE. This means that they can reconstruct the entire control flow of the targeted application and annotate it with snapshots of the corresponding ciphertexts in memory. One instance of such a scenario is a malicious hypervisor attacking a VM that is protected with AMD SEV-SNP. Finally, we assume that potential operating systems running alongside the targeted application

inside the TEE do properly protect register values from ciphertext side-channels attacks, as discussed in Section 2.2.

3.2 Countermeasure Requirements

Our overall goal is to produce a hardened binary which does not contain leaking memory writes. The countermeasure should not only protect the targeted program itself, but all its dependencies as well, as leakage may span multiple libraries (e.g., a crypto library calls `memcpy` in `libc`), and library developers are unlikely to widely adopt ciphertext side-channel countermeasures themselves. Finally, we target application developers who build code on top of third-party libraries and who do not have the necessary insight to manually fix leakages in those libraries. Thus, a drop-in solution with little manual interaction is desirable here.

There are two major approaches to this: One could either create a compiler extension that rewrites vulnerable memory accesses at compile time, or modify existing binaries through SBI. A pure compiler-based solution needs to recompile all dependencies, which is complex and requires manual intervention. A combination of DBI and SBI can work directly with the compiled binaries and, given sufficient coverage, accurately identify and harden vulnerable memory writes. For these reasons, CIPHERFIX aims for a binary instrumentation-based solution. The trade-off between binary vs. source-based approaches is further discussed in Section 7.1.

3.3 Protecting Memory Writes

In order to protect an existing binary from being attacked through a ciphertext side-channel, the content-based patterns of write accesses to memory have to be obscured. In [31], the authors propose various approaches for randomizing observed ciphertexts: First, by limiting reuse of memory locations through using a new address for each memory write; second, by interleaving data with random nonces; and third, by applying a random mask when writing data. The first approach uses the fact that different memory addresses get different tweak values in the memory encryption, but has a high overhead when applied outside of well-defined conditions. The second approach requires extensive changes to data structures, which has many pitfalls and needs to be done by the compiler. Due to lower overhead and higher practicability, we thus opt for the last approach, i.e., we add a random mask whenever an instruction writes secret data to main memory. We further discuss the different approaches in Section 7.3.

The masking of data takes place before memory writes and after memory reads. To store the masks belonging to a particular memory chunk (e.g., a C++ object), we allocate a

mask buffer of the same size, so there is a one-to-one mapping of data bytes to mask bytes. When writing data, we generate and store a new mask, XOR it with the plaintext, and store the masked plaintext; when reading, we read the mask and then decode the masked plaintext. Note that we need to ensure that at no point non-encoded secret data is written to memory, so all decoding must be done in secure locations like registers.

3.4 Tracking Data Secrecy at Runtime

While masking all memory writes provides good protection, it comes with a high overhead. In fact, only a fraction of all memory writes relate to secret information: As we assume that the implementation is constant-time, there is no secret-dependent control flow, so, for example, return addresses pushed onto the stack by function calls can be safely written in clear text. The same is true for the data structures used by the heap memory allocator to keep track of memory chunks. Finally, there may be a point where data is no longer considered secret, e.g., when sending a signature over the network. We thus aim to find and protect those instructions that actually deal with secret data. However, this is non-trivial, as there may be instructions that access both public and secret data, depending on the context (e.g., from `memcpy`).

Thus, we need a way to detect at runtime whether a given memory address should be considered secret, i.e., whether the data at that address is masked, and whether we should apply a new mask when writing to said address. We propose two approaches for storing this *secrecy* information (Figure 2): In the first approach, which we denote CIPHERFIX-BASE, we allocate another buffer of the same size as the mask buffer, called the *secrecy buffer*. In the second approach, CIPHERFIX-FAST, we encode this information directly into the mask buffer.

3.4.1 Storing secrecy information separately

In CIPHERFIX-BASE we allocate a buffer that holds the secrecy information for each memory location. If a byte is public, the corresponding secrecy byte is `0x00`; if a byte is secret, the secrecy byte is `0xff`. The secrecy buffer is initialized on allocation, and may be updated during the lifetime of the object. This construction allows us to read and update data without branching, as we can combine the secrecy value S with the mask M via a bitwise AND (\otimes), before applying it to the data via a bitwise XOR (\oplus): When reading, we compute $P = \hat{P} \oplus (M \otimes S)$, so we only decode the stored (potentially masked) plaintext \hat{P} if the address is considered secret. For writing, we always generate and store a new mask, and then compute $\hat{P} = P \oplus (M \otimes S)$ for plaintext P . As we make no assumptions

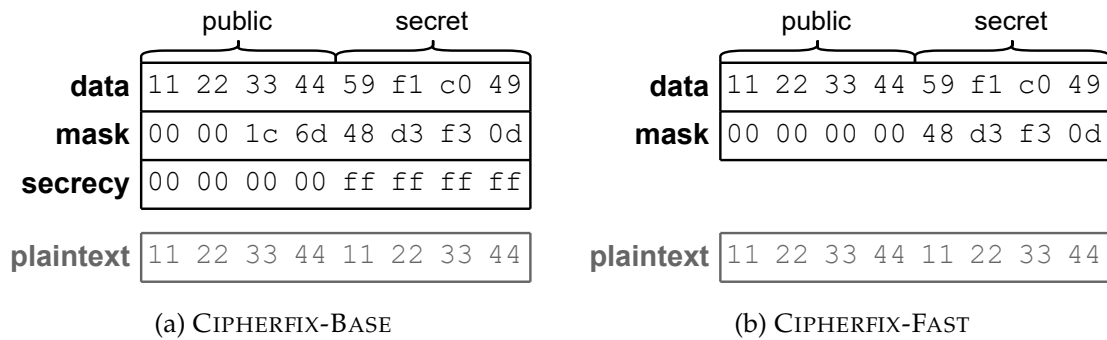


Figure 2: CIPHERFIX-BASE stores the secrecy information in a separate buffer, and uses it to decide whether a given mask byte should be applied or not. This allows to safely have non-zero mask bytes behind public data, as they are ignored if the corresponding secrecy bytes are zero. In contrast, CIPHERFIX-FAST stores this information directly in the mask buffer, i.e., a mask byte is zero iff the corresponding data is public.

about the mask, this generally functions as a one-time pad: The mask M is fully random and independent from the plaintext P , thus \hat{P} is independent from P as well.

3.4.2 Storing secrecy as zero masks

By separating mask and secrecy information, CIPHERFIX-BASE can generate uniform masks, yielding a one-time pad encoding. However, this comes at a cost: First, we get high memory overhead by allocating the mask and secrecy buffers. Second, each read is replaced by three reads, namely to the data, mask and secrecy buffers. To reduce this overhead, we make an observation: If the data is public, ANDing the mask and the secrecy value yields zero; if the data is secret, we use the mask value directly. Thus, for CIPHERFIX-FAST, we merge the secrecy information and the mask into the mask buffer, by setting the mask to zero when the data is public, and to a random non-zero value otherwise.

For writes, we check whether the old mask is zero before generating a new one, saving a memory write in some cases; for reads, we directly XOR the mask value, saving a memory read compared to CIPHERFIX-BASE. Thus, in addition to the reduced memory overhead, we get a performance improvement due to fewer memory accesses. We discuss the security implications of this in Section 6.3.

3.4.3 Reducing risk of mask collision

While CIPHERFIX can be used with secret data of any size, the width of the masks influences the robustness against attackers that observe ciphertexts over longer periods

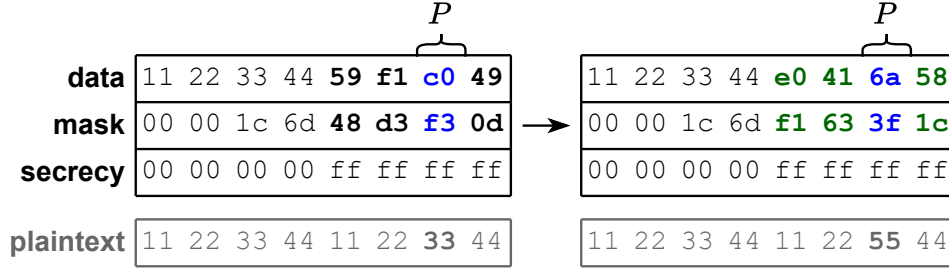


Figure 3: Extended write in CIPHERFIX-ENHANCED. Instead of updating only $w = 8$ data and mask bits at offset 6, CIPHERFIX-ENHANCED extends the write to $w' = 32$ bits, by also updating the mask of the surrounding three bytes, reducing the probability of a mask collision.

of time. For example, for a $w = 8$ bit wide mask, a mask collision can be expected in as few as $\sqrt{2^w} = 16$ writes. To address this issue, we propose CIPHERFIX-ENHANCED, which, as an extension of CIPHERFIX-BASE, converts writes with a size w below a certain threshold to a bigger size w' that is considered safe: Instead of updating w bits, we generate a new mask of size w' bits and update w' data bits at once. This is possible due to architectures like x86 supporting multiple write sizes from 1 byte to 8 bytes (and even more with vector instructions). We thus read and decode the existing masked plaintext \hat{P}' around the given address, merge it with the new plaintext P and then re-encode it. A write access protected with CIPHERFIX-ENHANCED is illustrated in Figure 3.

3.5 Toolchain

The CIPHERFIX framework is a drop-in solution that analyzes existing binaries with DTA to identify vulnerable code and then statically instruments the binaries to mitigate the detected leakages. CIPHERFIX consists of two distinct steps (Figure 4). In the analysis step, a taint analysis tool detects instructions and memory locations like stack frames and heap objects, that touch secret data. In parallel, a structure analysis tool extracts information about basic blocks and register/flag usage per instruction to aid the static mitigation. Finally, the mitigation step uses the analysis results to statically instrument the vulnerable binaries, inserting masking code for secret memory accesses and installing infrastructure for initializing newly allocated memory. In the following sections, we discuss the respective steps in more detail.

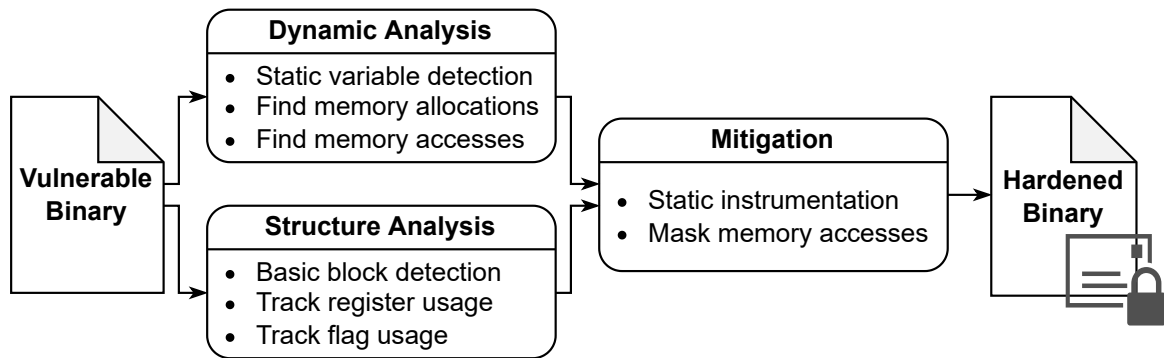


Figure 4: Structure of the CIPHERFIX framework. The vulnerable binary is dynamically analyzed and then hardened through static instrumentation.

4 Leakage Localization and Preprocessing

In order to protect read/write operations, we first need to identify all vulnerable memory locations and the instructions accessing them. Our static mitigation relies on some additional structural information, i.e., the offsets of basic blocks and liveness of registers and flags. In the following, we describe our leakage localization technique and the other analysis steps.

4.1 Dynamic Secret Tracking

With the help of DBI and DTA, we can collect information that is only available at runtime. As constant-time code does not include secret-dependent control-flow, DTA covers all paths of the implementation. For the cases of non-constant control flow in public paths, we use multiple iterations of the program with different inputs. We further discuss this in Section 7.2. If an exact analysis is not possible, we stay on the safe side and avoid undertainting so that in combination with full path coverage we reliably identify all secret accesses.

Our proof-of-concept implementation is based on libdft64 data flow tracking. When combined with a Boolean taint, we found that byte-level tainting of memory is fast enough to analyze complex cryptographic libraries while maintaining a high accuracy. While that leads to some overtainting (i.e., some memory locations get protected unnecessarily), we avoid undertainting. We also extended libdft64 by adding support for many SSE/AVX vector instructions, which are heavily used in optimized cryptographic code. All in all, we added 4355 lines of code (LoC) to libdft64 for new instruction support and 2269 LoC for our tracking logic.

4.1.1 Taint policy

We offer several venues for specifying taint sources, depending on the use case: First, if the main application itself can be easily recompiled (e.g., a custom network program linked against OpenSSL), the developer can call a special `classify` function, which takes a memory address and a size parameter. The taint analysis Pintool tracks this function and introduces taint for the corresponding memory when observing a call. In addition, we support fully automated assigning of taint sources without recompilation: Many cryptographic implementations read their private keys from the file system, so by intercepting the `open` and `read` system calls we can detect accesses to such files and taint the incoming data.

Our policy does not introduce taint sinks in the classical way; instead, those consist of all traced memory accesses and information that is needed for the static countermeasure. However, we offer a `declassify` function that explicitly marks data as no longer secret, i.e., all associated taint is deleted. In addition, functions that transmit data over insecure channels (e.g., network functions) remove taint as well. Thereby, we ensure that data that is meant to be publicly available does not get damaged by remaining secrecy features.

4.1.2 Tracking secret-related instructions

In order to protect memory accesses in our mitigation, we need to identify all instructions that read or write secret data at some point of the execution. The analysis distinguishes between three different cases: For instructions that only process public data there is no need to apply any ciphertext side-channel protection, whereas for instructions that only process private data the content written to memory always gets randomized. Finally, there are instructions that only occasionally access secrets and thus need to be able to distinguish between public and secret memory. As the latter may come with a certain performance overhead, the information about secrecy of accessed memory should be included in the taint analysis result used for the static mitigation.

4.2 Identifying Memory Locations

As the taint analysis itself tracks secrets only through “raw” memory addresses, it misses a lot of context: For example, there is no distinguishing between heap and stack memory, and which function a given accessed stack frame belongs to. However, for a static mitigation, we need certain information about each object in memory, like where it is allocated and which offsets need to be protected. There are various kinds of memory locations, i.e., static variables in the binary itself and dynamically allocated heap blocks and stack frames, so we need to distinguish between those cases.

4.2.1 Finding static variables

During the execution of cryptographic code, some instructions access data that lies within the memory region of the mapped binary, i.e., static initialized or uninitialized variables. Since we cannot access source-code level information about the program, we develop a method to locate these variables and determine their size, as we aim to only protect those that contain secret information. These fine-grained memory objects keep their secrecy status during the whole execution (i.e., if a variable contains secrets at some point, it is secret from the beginning until the end of a program run). For the static variable detection, we implemented a small Pintool with 338 LoC that collects traces of memory accesses in the data segments, matches these accesses to contiguous blocks in the binary's memory region and then produces an output file that can be parsed by the main taint tracking Pintool.

4.2.2 Heap allocations

Heap allocations are tracked through explicit (de)allocations, e.g., the `malloc`, `realloc`, `calloc` and `free` standard library functions. Similar to the static variables, the secrecy status of a heap object is kept for its entire lifetime. However, the heap layout may be different for each execution, so we cannot rely on fixed addresses to identify a heap object. Generating a flat list of heap allocations and retrieving the secrecy information using a counter variable is not useful either, as this restricts the hardened binary to a single control flow path. Instead, we use the call stacks of the heap allocations: Apart from rare cases where allocations are done in a loop, the call stack of each allocation is unique and thus suitable for identifying it both during analysis and at runtime. The call stack of an allocation is determined by keeping track of all calls and returns during analysis and emitting the current call stack whenever an allocation function is observed (Figure 5).

As for heap objects the application itself has full control over their layout and the stored data types may vary depending on context, we cannot safely make assumptions about relative offsets within a heap object. Thus, we opted for marking the entire object as secret whenever a part of it gets tainted. While this overapproximation may lead to a slightly higher overhead due to protecting more instructions than strictly necessary, it reduces complexity and makes the static mitigation more robust. We also found that in practice the impact is limited, as generally the size of a heap object correlates with the amount of (private) data stored in it (e.g., big integer objects).

Call Tree	Allocation Call Stacks
1007 : call <sign>	
11e1 : call <multiply>	
140b : call <malloc> secret	1007—11e1—140b
1211: call <multiply>	
140b : call <malloc> public	1007—1211—140b
1432 : call <malloc> secret	1007—1432

Figure 5: Call tree and resulting call stacks for three heap allocations. The call stack is accompanied with secrecy information, i.e., whether a secret block was allocated. The offsets of call instructions that lead to at least one secret allocation are marked bold and red; the offsets of instructions that only lead to public allocations are marked green. This information is directly reused in the instrumentation (see Section 5.2.2).

4.2.3 Tracking stack frames

The stack memory area is characterized by rather liberal (de)allocation and access strategies, which makes separating individual stack frames difficult and thus complicates tracking the exact offsets and lifetimes of secret variables. An easy solution would be marking the entire stack as secret and protecting all instructions that ever access stack memory, but this would introduce a lot of unnecessary overhead, since the stack is mostly used for temporarily storing registers and small local variables that often do not contain secret data. Instead, in order to avoid overtainting and the aforementioned performance penalty, we developed a generic stack frame tracking strategy that allows to keep track of secret data throughout the program execution by means of stack frame offsets. Contrary to the heap, the stack usually conforms to fixed patterns built by the compiler, so we can assume that relative offsets within a function’s stack frame are valid over multiple executions.

Our proof-of-concept implementation does not rely on source code or function symbols, but works with any standard-conforming binary. The stack allocation tracking consists of identifying function calls, mapping a call target to an actual function for which a stack frame initialization of the static instrumentation is needed and determining its respective stack frame size, and building a list of secret offsets within that stack frame.

Most function calls are detected through `call/ret`-pairs; in addition, our analysis includes a heuristic for detecting tail calls, i.e., when a function is exited via a `jmp` instruction to another function. Calls to functions in shared libraries present another challenge, as the application invokes those through a call to the `.plt` section, which may in turn jump into the dynamic runtime linker to resolve the actual function call target. In order to find the function in the shared library and not its stub code in the caller’s `.plt` section, we

need to follow the resolution process in the dynamic linker until we reach the actual call target. This is done through a state machine that keeps track of the current linking state and generates a mapping of `.plt` offsets to the corresponding functions.

After detecting a function, we proceed with determining its stack frame size. This is achieved through several means: First, there may be explicit stack frame allocations through instructions like `push/pop` and `sub/add`, which directly modify the stack pointer. In addition, the x86-64 ABI permits functions to freely use a small chunk above the stack pointer (which usually marks the end of a stack frame), the so-called red zone. We handle this by updating the stack frame size whenever we observe an access outside a known stack frame.

4.3 Binary Structure Analysis

Contrary to DBI, where the executed code is recorded and instrumented at runtime, SBI must apply all changes in an offline manner, without being able to handle unexpected states. Our proof-of-concept SBI-based mitigation needs further information besides the DTA results, namely the precise bounds of all basic blocks and, for each instruction, the usage of registers and status flags. The latter is necessary since the masking operations need scratch registers to store intermediate results, and inadvertently clobber the status flags. While this information can be collected through static liveness analysis or heuristics [19, 54], we decided to employ dynamic analysis here as well, as we already have the necessary code coverage from the DTA. This approach marks only registers and flags that are indeed used, avoiding unnecessary saves/restores and thus reducing the runtime overhead. We created a specialized Pintool with 599 LoC that collects the aforementioned information and passes it to the SBI tool.

5 Static Mitigation

With the information from the dynamic analysis we can now statically instrument the affected binaries, hardening them against ciphertext side-channel attacks. We identify consecutive basic block chains (functions), which are then copied and instrumented at a new section in the binary. The original code locations are replaced by a number of jumps to their instrumented counterparts, following an optimized trampoline-approach described in Section 2.3. We then modify all vulnerable memory accesses to apply masks, such that each of these memory writes is randomized. The resulting hardened binaries are self-contained, i.e., they can be executed without an external instrumentation framework.

```

-----
00: mov rcx, qword [rbp-0x20] ; read encoded (?) data
04: mov rax, qword [rbp-0x3ffff020] ; read mask
0b: and rax, qword [rbp-0x2ffff020] ; AND secrecy value
12: xor rcx, rax ; decode (?) data
15: shr rcx, 8 ; do actual computation
-----
19: rdrand rax ; generate random mask
1d: jnc 19 ; retry on failure
-----
23: mov qword [rbp-0x3ffff020], rax ; store new mask
2a: and rax, qword [rbp-0x2ffff020] ; AND secrecy value
31: xor rcx, rax ; encode (?) data
34: mov qword [rbp-0x20], rcx ; store encoded (?) data
-----

```

Figure 6: Assembly code generated by CIPHERFIX-BASE for the instruction `shr qword [rbp-0x20], 8`, that accesses both public and secret memory. As an in-place shift, it has to first read and decode the left operand, compute the shift, and then encode and store the result. The instrumentation tool identified `rax` and `rcx` as scratch registers, which did not need to be preserved.

5.1 Masking Memory Accesses

After copying all affected basic blocks to a separate section, we can replace the vulnerable memory accesses by hardened instruction sequences. As described in Section 3.4, we mitigate the ciphertext side-channel by adding a random mask to each memory write to a secret location. Some instructions have read and write accesses (e.g., arithmetic with a memory operand acting both as source and destination), so they may need decoding and encoding (Figure 6). String operations like `rep movsq` are replaced by an explicit loop that decodes each word of the source data and re-encodes it for the destination, as not the entire copied memory block may be secret. Our proof-of-concept implementation supports protection of common arithmetic and move instructions, and a number of vector instructions that occur in cryptographic code.

Each memory block is accompanied by a mask buffer and a secrecy buffer, which have a constant distance d_M resp. d_S to the memory block’s address. Using a constant distance saves expensive look-ups for finding the appropriate buffers, reducing the total overhead of the mitigation. For our test setup, we found that $d_M = 0x3ffff000$ and $d_S = 0x2ffff000$ work well. These provide sufficient memory space while still fitting into the signed 32-bit memory displacement immediate which is supported by x86-64, and avoid penalties like aliasing when two addresses share too many low bits.

5.1.1 Updating the masks

Apart from initializing the mask and/or secrecy buffers during setup (see Section 5.2), we need to update the mask values before every write operation. To ensure that masks do not have repeating or easily exploitable patterns, we sample them from a pseudorandom number generator (PRNG). As we want to keep the overhead low, any such PRNG should have a small code footprint and require as few registers as possible, which rules out most classic software-based PRNGs. A natural choice on x86-64 is the `rdrand` instruction, which fills a single general purpose register with random bytes. The instruction offers cryptographically secure randomness. However, its security guarantees also lead to a noticeable slowdown when the instruction is used extensively.

To work around this, we devised two additional PRNGs for mask generation. The first one, named AES, makes use of the AES-NI `vaesenc` instruction to repeatedly apply the first round of AES to an initially random 16-byte state with a random 16-byte round key. The second PRNG is XorShift128+, a widely-used and fast full-period generator [48], for which we created a vectorized implementation. In both cases, the new mask is extracted from the state. For best performance, the AES PRNG needs two vector registers and the XorShift128+ PRNG needs three. We found that usually enough such registers are available, and, if not, the overhead for the additional save/restore is still smaller than calling `rdrand`. We discuss the properties of the different PRNGs in Section 6.3.1.

5.1.2 Scratch registers and flags

For some operations, we need additional scratch register space for storing intermediate results. Since we are restricted to working with an existing binary, we cannot exclude registers from being allocated by the compiler and thus have to look for registers which hold stale values, or save those values in a secure location. We use the results from the structure analysis in Section 4.3 to identify suitable registers. To save general purpose registers, we prefer using SSE vector registers via the `vmovq` and `vpinsrq` instructions, as those are fast and immune to ciphertext side-channels. In the rare case where no vector register is available, we store the scratch register's original value in memory. To avoid the expensive masking when writing a secret value to memory, we prioritize registers that the taint tracking did identify as not holding secret data.

Similar to the registers, our instrumentation may overwrite status flags through the encoding/decoding instructions. To save and restore single flags, we use the `setcc` instruction family, while for multiple flags we rely on the `lahf` instruction, which copies the entire flag state into the `ah` register.

5.2 Managing Mask and Secrecy Buffers

The instrumented instructions assume that there is a mask buffer and a secrecy buffer with a constant distance to the accessed memory address. Thus, for each memory block that is accessed by such an instruction, we need to allocate a mask buffer at the corresponding address and initialize it with random data, if it contains secret data. This comes with a few challenges: First, there are several ways of allocating memory, namely the stack, the heap and static fixed-size arrays in the binary itself. Then, not all memory blocks in these regions are considered secret, so their masks and secrecy values need to be initialized context-aware. In the following, we discuss strategies for handling the various memory regions.

5.2.1 Stack

The stack is allocated by the operating system at application start and is used for storing return addresses, register values and small local variables. The taint analysis produces stack frame information for each function, which contains the size of the stack frame and the relative offsets where secret data is stored. Accordingly, we insert a small code gadget at the beginning of each function, that prepares its stack frame by generating a random mask or setting the secrecy value for the respective offsets. The mask and secrecy buffers for the stack are allocated on startup; the constant buffer distances work well for the stack, as it usually resides within a well-known memory range and does not grow beyond a few megabytes.

5.2.2 Heap

For most Linux applications, the heap is a contiguous memory region that is managed by the standard library's allocator. The heap starts at a random base address, and is resized via the `brk` system call. The user then typically allocates memory by calling `malloc` or `realloc`, which ensure that enough heap memory is available and return an appropriate memory range.

To guarantee that there are mask and secrecy buffers backing the entire heap region, we instrument the `brk` system call and (de)allocate corresponding memory each time the heap grows or shrinks. The buffers are initially set to zero. We also replace the `malloc` and `realloc` calls by custom code, which ensures that the corresponding mask and secrecy buffers are correctly initialized depending on whether the allocated memory should contain secret data or not. To identify the particular heap allocation, we resort to tracking its call stack, as explained in Section 4.2.2. We achieve this through an *allocation tracker*, which is an integer residing at a fixed memory address, and which is updated




Call Tree	Allocation Tracker
1007: call <sign>	0...000 1
11e1: call <multiply>	0...00 11
140b: call <malloc> secret	0...0 111 
1211: call <multiply>	0...00 10
140b: call <malloc> public	0...0 101 
1432: call <malloc> secret	0...00 11 

Figure 7: Allocation tracking for the example from Figure 5. Each time a `call` instruction is executed, the allocation tracker is shifted to the left, and 1 is added when this particular `call` is part of a call tree leading to an allocation of a secret heap object. On return, the tracker is shifted back to the right. The `malloc/realloc` handler code then checks whether the allocation tracker has the value $2^n - 1$, i.e., whether it is all ones starting with the least significant bit. In this case, the new heap object is considered secret; else, it is public.

on each `call` instruction that is part of a call stack that leads to a heap allocation. Before each `call`, we left-shift the tracker variable, and add 1 if the call is part of a call stack that leads to allocation of a secret heap memory object. With our allocation tracker, we can reliably handle heap allocations even if we encounter non-constant control flow or when a function is reused in a different context. An example is illustrated in Figure 7.

Contrary to `malloc`, the `realloc` function allows resizing or reallocating an existing heap memory object, while keeping its contents. As the new object may have a different secrecy setting than the old one, we have to ensure that the data is correctly decoded, copied and encoded. However, `realloc` itself is not aware of the masks and secrecy information, so to avoid losing information, our `realloc` handler copies the old data, mask and secrecy buffers to a separate memory location, runs `realloc`, and then restores the contents at the new location with the appropriate encoding.

If the instrumented program allocates lots of memory, the constant distance to the mask and secrecy buffers may be insufficient, as the heap could at some point overlap with its mask buffers. In this case, one could replace the affected `malloc` calls by a custom allocator, that is injected into the instrumentation and operates outside the usual heap area. Note that this still limits the maximum memory object size to the distance between a memory address and its buffers, i.e., at most two gigabytes, if the instrumentation should do without another scratch register for computing larger offsets.

5.2.3 Static arrays

Finally, the binary may have a number of static global variables, which reside in its data sections. We embed the information about static memory objects containing secret data

in the instrumented binary. On startup, an initialization routine walks through this list and allocates and initializes the respective mask and secrecy buffers.

5.3 Implementation

We created a proof-of-concept implementation of our mitigation in C#, which takes the dynamic analysis results and the target program and produces statically instrumented binaries. The instrumentation tool has 7346 LoC, which includes a specifically developed library for patching ELF64 files.

5.3.1 Instruction instrumentation

The instrumentation tool loads and parses the outputs from the taint tracking and the structure analysis tools, and decodes the target ELF files. Then, for each individual binary, the instrumentation is applied: First, we look for contiguous basic block chains and identify appropriate code locations for inserting jumps to instrumentation code. Next, we replace each memory accessing instruction marked by the DTA by a masked version. After handling all basic blocks, we obtain a list of unmodified and instrumented instructions, grouped by their respective basic blocks. In a final step, we re-assemble those instructions and write them into a newly allocated ELF section, while patching the basic blocks in the old `.text` section to jump to the instrumentation code.

5.3.2 Initialization

After the instruction-level instrumentation is done, we need to install infrastructure for handling the `int3` signals and some initialization code that allocates mask and secrecy buffers. For this, we created an *instrumentation header*, which consists of 966 lines of assembly code interleaved with some static constants which are later replaced by the instrumentation tool. The instrumentation header hooks into the *constructor* of each binary, which is executed by the dynamic linker when a binary is loaded into memory. This way, we ensure that our initialization runs before all other application code. The initializer of the main program sets up the signal handler, and determines the stack size and base address. Then, it allocates mask and secrecy buffers for the stack. The initializers of the main program and of all dynamic libraries iterate through the list of secret static variables deposited by the instrumentation tool, and allocate and initialize mask and secrecy buffers.

6 Evaluation

We now evaluate the performance and security of the different CIPHERFIX variants. We analyze whether there is remaining leakage with regard to collision attacks, and discuss trade-offs between security and performance.

6.1 Experimental Setup

We evaluate our proof-of-concept implementation of CIPHERFIX against a number of typical algorithms which are used in widespread protocols like TLS or SSH. To observe variations caused by different implementations of the same primitive, we spread our analysis over several common libraries, that are OpenSSL 3.0.2, WolfSSL 5.3.0, Mbed TLS 3.3.0 and libsodium 1.0.18. As primitives which were shown to be vulnerable to ciphertext side-channel attacks [31], we picked EdDSA (Ed25519) and ECDSA (secp256r1), and verified that these are still vulnerable in the given implementations. [33] demonstrated an attack against the RSA signature scheme, which we included as well. We also added ECDH (X25519) as a primitive that is widely used in cryptographic protocols and likely to be vulnerable as well. As additional benchmarks, we included the symmetric primitives AES-GCM and ChaCha20-Poly1305, the hash function SHA-512, and finally the Base64 decoding function as a non-cryptographic algorithm, that is nevertheless often present in cryptographic applications.

The analysis, instrumentation and all measurements were performed on an AMD EPYC 7763 CPU with Zen3 microarchitecture, which supports SEV-SNP. All libraries were compiled with GCC 9.4.0 on Ubuntu 20.04.4 LTS. MbedTLS was linked statically, while the other libraries were linked as shared libraries.

6.2 Performance

To get the information necessary for the mitigation, we ran the dynamic analysis as described in Section 4. We found that we achieve sufficient coverage by executing each target 10 times with random inputs in a loop, except for WolfSSL RSA, which required 20 due to high control flow variation introduced by blinding. In all cases, the time required for dynamic analysis was less than 5 minutes, with around 80% of the time taken by the register tracking in the structure analysis, and most of the remaining time by the taint tracking. The most expensive target, Mbed TLS ECDH, required tracking 170 532 009 executed instructions (5167 unique). While the register tracking could be scrapped in favor of a faster but potentially less precise static liveness analysis (as done by several

binary rewriting tools), note that these steps are executed offline and only need to be done once to protect a binary, so we deem an analysis time of a few minutes acceptable.

6.2.1 Runtime overhead

To measure the runtime overhead of the different CIPHERFIX variants, we executed each target with 1000 random inputs, averaged the measured execution times, and computed the relative overhead compared to the original implementation. An overview of the resulting overall slowdowns of the different CIPHERFIX variants is given in Table 2. As expected, CIPHERFIX-FAST has the lowest overhead, CIPHERFIX-ENHANCED has the highest, and CIPHERFIX-BASE lies in between. The slowdown of CIPHERFIX-BASE compared to CIPHERFIX-FAST is caused by the additional read for each protected memory access; in most cases, CIPHERFIX-ENHANCED performs quite similar to CIPHERFIX-BASE, except for the symmetric primitives and utility functions which have a vastly higher number of 1-byte writes.

Moreover, generating masks with `rand` introduces a much higher overhead than with one of the other PRNGs. This is caused by the continuous reseeding of the underlying shared hardware PRNG, in combination with `rand` not being designed for sampling random numbers at a high frequency. The smallest overhead is achieved with the AES PRNG, as it consists of a single `vaesenc` instruction and only needs two vector registers. A detailed overview over all runtime overhead measurements is given in Table 1.

6.2.2 Code properties contributing to overhead

We identified several major factors that determine the overhead when hardening a particular implementation with CIPHERFIX. First of all, code that heavily relies on memory accesses for dealing with secret information is clearly more susceptible to overhead introduced by instrumentation than code that performs most computations in registers. This becomes apparent when comparing the RSA implementations of Mbed TLS and WolfSSL: Though for WolfSSL a higher percentage of the memory accesses is instrumented (78% writes vs. 65%), Mbed TLS has an order of magnitude more memory operations than WolfSSL and thus gets a higher overhead. Similarly, some instructions are more expensive than others in terms of ciphertext side-channel hardening: For example, arithmetic directly applied to memory operands requires a full decoding and re-encoding cycle (cf. Figure 6), which is slow due to direct data dependencies between the steps. We observed this for Mbed TLS ECDSA, which gets a much higher overhead (7.1x vs. 1.6x) than the comparable implementation in WolfSSL, mostly due to expensive adds in a hot code path.

Table 1: Runtime overhead of instrumented binaries. For each CIPHERFIX variant and PRNG, we measured the execution time in milliseconds (ms) of 1000 executions of each primitive and the corresponding overhead factor to the original implementation. The target AES refers to AES-GCM, the target CC20 to ChaCha20-Poly1305. The last row shows the geometric mean of the respective overheads for each CIPHERFIX variant.

Target				CF-FAST			CF-BASE			CF-ENHANCED		
				AES	XS+	rand	AES	XS+	rand	AES	XS+	rand
libodium	EdDSA	time	29	159	166	1159	189	248	1133	214	245	1134
		factor	-	5.5x	5.7x	40.0x	6.5x	8.6x	39.1x	7.4x	8.4x	39.1x
	SHA512	time	9	14	20	196	21	22	194	22	25	194
		factor	-	1.6x	2.2x	21.8x	2.3x	2.4x	21.6x	2.4x	2.8x	21.6x
mbedtls	AES	time	104	297	377	2849	364	371	2576	1204	1213	2683
		factor	-	2.9x	3.6x	27.4x	3.5x	3.6x	24.8x	11.6x	11.7x	25.8x
	Base64	time	10	12	13	58	16	16	45	28	30	46
		factor	-	1.2x	1.3x	5.8x	1.6x	1.6x	4.5x	2.8x	3.0x	4.6x
	CC20	time	144	324	332	2945	542	570	2952	1785	1721	3059
		factor	-	2.3x	2.3x	20.5x	3.8x	4.0x	20.5x	12.4x	12.0x	21.2x
	ECDH	time	1855	3674	3778	8559	9425	9440	14419	9926	10208	14827
		factor	-	2.0x	2.0x	4.6x	5.1x	5.1x	7.8x	5.4x	5.5x	8.0x
	ECDSA	time	472	3367	3558	8920	3912	3929	8297	4265	4301	8374
		factor	-	7.1x	7.5x	18.9x	8.3x	8.3x	17.6x	9.0x	9.1x	17.7x
	RSA	time	896	3276	3777	28886	5436	5339	27148	5527	5663	27208
		factor	-	3.7x	4.2x	32.2x	6.1x	6.0x	30.3x	6.2x	6.3x	30.4x
OpenSSL	ECDH	time	172	541	550	2408	664	657	2323	708	807	2369
		factor	-	3.1x	3.2x	14.0x	3.9x	3.8x	13.5x	4.1x	4.7x	13.8x
	ECDSA	time	516	939	1121	7181	1795	1855	9980	2051	1973	10072
		factor	-	1.8x	2.2x	13.9x	3.5x	3.6x	19.3x	4.0x	3.8x	19.2x
WolfSSL	AES	time	147	268	269	793	400	403	880	397	402	879
		factor	-	1.8x	1.8x	5.4x	2.7x	2.7x	6.0x	2.7x	2.7x	6.0x
	CC20	time	167	428	432	2787	596	630	2802	1157	1242	2874
		factor	-	2.6x	2.6x	16.7x	3.6x	3.8x	16.8x	6.9x	7.4x	17.2x
	ECDH	time	146	258	437	4217	544	565	4070	541	558	4070
		factor	-	1.8x	3.0x	28.9x	3.7x	3.9x	27.9x	3.7x	3.8x	27.9x
	ECDSA	time	1092	1704	1954	15765	3945	3834	18631	3883	3897	19654
		factor	-	1.6x	1.8x	14.4x	3.6x	3.5x	17.1x	3.6x	3.6x	17.1x
	EdDSA	time	60	124	156	1897	279	265	1759	280	290	1761
		factor	-	2.1x	2.6x	31.6x	4.7x	4.4x	29.3x	4.7x	4.8x	29.4x
	RSA	time	133	248	334	2901	588	605	2863	602	651	2870
		factor	-	1.9x	2.5x	21.8x	4.4x	4.5x	21.5x	4.5x	4.9x	21.6x
average factor			-	2.4x	2.7x	16.8x	3.9x	4.0x	17.3x	5.1x	5.3x	17.5x

Table 2: Performance measurements for the different CIPHERFIX variants and PRNGs. Each entry shows the geometric mean of the runtime overhead over all targets compared to the original, uninstrumented binary.

	AES	XS+	rdrand
FAST	2.4x	2.7x	16.8x
BASE	3.9x	4.0x	17.3x
ENHANCED	5.1x	5.3x	17.5x

Finally, the overhead is influenced by the general structure of the instrumented code, and the optimization capabilities of the binary rewriting framework. A framework operating at basic block level could perform better than our proof-of-concept implementation, which instruments each instruction in isolation to ease leakage analysis and debugging. For example, scratch registers may not need to be restored between usages, and instructions could be reordered to avoid saving status flags. This is particularly relevant as the compiler tends to interleave arithmetic instructions that have direct status flag dependencies with memory accesses (e.g., add-mov-adc).

A detailed overview over the observed memory accesses is given in Table 3 in Appendix B.

6.3 Security

In the following, we illustrate reasons for remaining collisions after applying the different variations of CIPHERFIX and evaluate its practical security.

6.3.1 Leakage sources

As we assume full path coverage of the implementation (see Section 7.2) and our taint tracking does not undertaint, all vulnerable instructions are identified and protected. Thus, the only remaining source of leakage are **collisions of the masks or the masked plaintexts**: With CIPHERFIX-BASE, the secrecy information is stored in a separate buffer. If the mask M is fully random and independent from the plaintext P , the masked plaintext \hat{P} becomes independent from P as well. However, the attacker can access both ciphertexts $C_{\hat{P}} = \text{Enc}_{\text{pt}}(\hat{P})$ and $C_M = \text{Enc}_{\text{mask}}(M)$, so they are able to detect whether \hat{P} or M appear repeatedly. If the data memory block is rarely changed and the number of protected bits is sufficiently low, a mask collision is possible and may leak information about the plaintext. A similar issue can occur with CIPHERFIX-FAST, which stores the secrecy information directly in the mask buffer by setting the mask to zero for public values. We can assume

that the attacker knows the ciphertext $C_0 = \text{Enc}_{\text{pt}}(0)$ of an unmasked zeroed data block, as memory usually is zero initialized. If they observe C_0 again after a write of P with mask $M \neq 0$, they can use that $C_0 = \text{Enc}_{\text{pt}}(P \oplus M)$ and thus $P = M$ to infer that $P \neq 0$. These leakages through masks or masked plaintexts are mostly relevant for 1-byte writes to variables in memory blocks with little other activity. With CIPHERFIX-ENHANCED, we enforce a minimum width of masked data, which further reduces the probability of mask collisions and other non-unique writes at the cost of a slightly higher overhead.

Another factor is the **quality of the PRNG used for mask generation**, for which we identified two primary criteria. First, the pseudorandomness should not correlate with the plaintexts: For example, simply incrementing the masks may lead to many collisions of the masked plaintexts in algorithms that use linear arithmetic. Second, deterministic PRNGs should have a sufficient cycle length, to keep an attacker from reliably triggering the same mask at the same address during the application’s runtime. Rdrand offers the fastest available solution for cryptographically secure pseudorandomness. However, given the subsequent memory encryption, the used PRNG does not necessarily need to be cryptographic, as long as it satisfies the above criteria and thus does not tend to generate repeating masks or masked plaintexts. XorShift128+ has a cycle length of $2^{128} - 1$ and passes all *BigCrush* tests of the *TestU01* suite [29], though it has some weaknesses [22]. Our custom one-round AES PRNG passes all *BigCrush* tests and seems to perform well in practice, but does not have a guaranteed cycle length. We leave this analysis to future work.

6.3.2 Observed collisions

To analyze potentially remaining ciphertext collisions, we extended the taint tracking to export a full trace of all memory writes alongside corresponding secrecy information. We then created a Pintool that generates a trace of all memory writes for an instrumented binary. As each original memory access may be replaced by multiple memory accesses during instrumentation, we inserted special marker instructions that denote the beginning and end of a particular instrumented memory access sequence. With this information, we align the traces using a custom evaluation tool, and proceed with checking whether there are repeated writes of the same secret value to the same address. As the dictionary attack builds upon the collision attack, finding no collisions implies security against all known ciphertext side-channel attack primitives. We found that using the same amount of test cases for our evaluation as for the initial taint tracking was sufficient, as due to the size and complexity of the evaluated targets systematic issues already appear during the first few executions.

We were able to confirm the suspected remaining leakages with our evaluation. For

example, there are several thousand collisions for CIPHERFIX-BASE and CIPHERFIX-FAST with the Mbed TLS AES-GCM target, which encrypts 16 KiB of plaintext using AES-NI and has 812 120 1-byte writes, which is 66% of its total writes. The observed collisions both included repeating masks and cases where applying a new mask to a new plaintext led to the same result. All colliding 1-byte writes were related to sequential writing into an array, e.g., when data is copied or buffers are cleared between different processing steps. The corresponding collisions had high temporal locality and the respective 16-byte blocks only appeared exactly two times, so while there is some leakage, its exploitability is limited. With CIPHERFIX-ENHANCED, all collisions disappeared. All observed collisions in the analyzed targets were for 1-byte writes, which suggests that restricting CIPHERFIX-ENHANCED to 1-byte writes (and possibly 2-byte writes) is sufficient. We encountered almost no 2-byte writes in our experiments. We further discuss the security impact of the collisions in CIPHERFIX-FAST in Section 6.4.

We did not see any relevant difference between the particular PRNGs: The number of collisions is roughly equal, and there was no 32-bit mask collision even for the targets with the highest number of instrumented writes. This suggests that they are all generally suited for generating masks for the evaluated primitives within the given constraints. Nevertheless, the decision for a particular PRNG should not be made easily, as is discussed in the next section.

6.4 Balancing Security and Performance

Each variant and PRNG comes with its own advantages and drawbacks. We point out some guidelines for choosing the best composition for a given use case.

6.4.1 Properties of the implementation

To determine the most suitable variant of CIPHERFIX, one should look at the properties of the given implementation. For example, symmetric primitives, which showed a huge amount of 1-byte writes in our evaluation, do not necessarily need to be hardened against ciphertext side-channels. With hardware extensions like AES-NI and CLMUL, we found that leakage is mostly restricted to copying of inputs and outputs between encryption rounds. Thus, if the same buffer is reused for multiple blocks, the attacker may occasionally learn that a particular plaintext block or parts of it repeat. Whether this is tolerable depends on the specific use case.

6.4.2 Choosing a CIPHERFIX variant

While CIPHERFIX-FAST has the least performance overhead, it has the additional risk of leaking whether the mask and the plaintext are equal, as described in Section 6.3.1. While we did not observe that particular scenario, we saw several 1-byte collisions in WolfSSL's X25519 cswap implementation. This suggests that CIPHERFIX-FAST and CIPHERFIX-BASE are dangerous even for algorithms with a very small number of 1-byte writes. Future work may develop a further variant that uses a merged mask/secret buffer but widens small writes to 4 bytes, to get both the performance benefit of CIPHERFIX-FAST and the protection of CIPHERFIX-ENHANCED. For deciding between CIPHERFIX-BASE and CIPHERFIX-ENHANCED, we generally recommend choosing the latter due to the better protection of 1-byte writes. While we observed a higher performance overhead, the difference was almost exclusively caused by the symmetric primitives which do a lot of 1-byte operations. Excluding the symmetric implementations from the geometric mean yields an overhead of 5.2x for CIPHERFIX-ENHANCED versus 4.9x for CIPHERFIX-BASE and the XorShift128+ PRNG.

6.4.3 Choosing a PRNG

Despite the high security guarantees, the considerable performance overhead of CIPHERFIX with `rand` suggests that this PRNG is not suitable for use with primitives that have a lot of vulnerable memory accesses. On the other hand, our custom AES PRNG is very fast and did not exhibit more collisions than the other PRNGs in our experiments, but is not well examined in terms of statistical properties and cycle length. Thus, as a compromise, we suggest using a fast PRNG that is well-analyzed and meets the criteria outlined in Section 6.3.1, such as XorShift128+, which only introduced a slightly higher overhead than AES. As a workaround for an insufficient cycle length or concerns that a high number of samples may expose weaknesses, the PRNG may be periodically reseeded with fresh entropy via instructions like `randseed`, e.g., each time before the hardened primitive is executed. Finally, a production-level implementation of CIPHERFIX may combine different PRNGs, like a fast one for hot code paths and `rand` elsewhere.

6.4.4 Practical impact of overhead

Note that we focused our performance analysis on isolated cryptographic primitives, which does not reflect their typical use case. Instead, they are usually embedded into a higher-level application like a network protocol, which limits the practical influence of a moderate overhead in a specific component. For example, in TLS, only the handshake is subject to asymmetric cryptography that needs to be hardened against ciphertext

side-channel attacks. The predominant part of the protocol's runtime, the symmetric encryption and transmission of the payload, may not need as much costly protection.

7 Discussion

We conclude our study with a discussion of some design decisions of CIPHERFIX, and point out possible angles for future work which may improve accuracy and performance.

7.1 Source Code vs. Binary Instrumentation

Instead of instrumenting binaries, the implementations could be hardened during compilation: As the compiler can freely adapt the code layout and is not restricted during register allocation, it can generate more efficient binaries. However, this comes with some obstacles. First, a source-based approach would need to be able to deal with handwritten assembly code, which is abundant in highly-optimized libraries like OpenSSL or libc. This assembly code is opaque to the compiler, but can be handled transparently by binary instrumentation.

A second obstacle is a leakage analysis that spans multiple libraries. At the beginning, the application developer would need to checkout the source code of all relevant dependencies, such that they can be recompiled with the appropriate protection. The compiler can then conduct a static data flow analysis that identifies all program points that may come in contact with secret data [8]. As we found during our experiments, a particular library may call a function in another library with secret parameters, so conducting a leakage analysis on a library in isolation is insufficient. This leaves two options: First, the leakage analysis can choose to protect the parameters of the entire outward facing API of a given library, such that all incoming function calls are assumed as passing secret data. However, this significant overapproximation is likely to neutralize the performance benefit of a compiler-based solution. Thus, as a second option, we may try to conduct the leakage analysis over all code bases at once. This is hindered by the fact that static analysis of a large code base like OpenSSL or libc is already difficult, and even more so when looking at several such code bases with different build systems and structure. At the very least, it would require lots of manual tuning by the application developer.

An alternative to binary rewriting that is worth exploring for a production-level implementation of CIPHERFIX is a hybrid approach combining dynamic analysis and compiler-based instrumentation: First, a dynamic analysis is conducted over all libraries as described in Section 4. However, the results are then not used to instrument the binaries

using SBI, but are sent back to the compiler. A suitable level for this is the intermediate representation (IR) of LLVM: The IR can be executed through a VM, enabling dynamic analysis. At the same time, it is abstract enough to still allow compiler optimizations between inserting the masking code and generating ELF binaries. Applying the analysis and instrumentation to IR also avoids the practical problems of dealing with large code bases, as those can be normally translated and linked into IR files. However, contrary to binary rewriting, this method still requires some effort from the library developer, and cannot straightforwardly deal with handwritten assembly code, that would need to be lifted to an equivalent IR representation first. Finally, advanced binary rewriting engines that generate symbolized reassemblable disassembly already offer performance similar to the compiler.

7.2 Analysis Coverage

Independent of the approach on instrumentation, we need to find all loads and stores that ever deal with protected data. Missing instructions during the secrecy analysis may lead to loading or storing invalid data, which can in turn cause functional incorrectness or crashes of the hardened binary. In constant-time implementations, there are no secret-dependent branches and memory accesses. However, it is useful to support some secret-independent control flow variation, e.g., for error handling or processing messages of varying length. As our analysis is dynamic, we have to rely on our inputs generating sufficient coverage, that is covering every possible execution path between classification and declassification of secrets. The secret tracking must not underapproximate (undertaint), as this may lead to missing leakages or instability due to instructions that cannot handle masked data. Overapproximation (overtainting) is acceptable to speed up leakage analysis, but may lead to unnecessary instrumentation and thus a higher runtime overhead. We found that few random inputs were sufficient to get the coverage needed for our analysis; however, one could also employ techniques like fuzzing to maximize the chances of finding all relevant code paths, especially when applying CIPHERFIX to non constant-time code. Fuzzing and a larger test case body would only impact the overhead of the offline analysis step.

Another approach for achieving full coverage is using a purely static analysis, which may be conducted either on binaries or as part of a pure compiler-based solution. However, even for the smaller exploitable primitives, we measured several tens of millions of executed instructions for a single dynamic analysis iteration, which poses a huge amount of instructions to analyze for a static analysis. To make this feasible, the static analysis would need to make some approximations, which would in turn increase the runtime overhead of the mitigation.

7.3 Alternatives to Masking

Our masking approach ensures that the written values are independent from the actual plaintexts. However, as mentioned in [31], instead of randomizing the values written to the same address, it is also possible to randomize the address itself. This approach would need a separate memory area for secret data. The area can, for example, be implemented as a queue with used and free space that is updated with each write. The original memory locations then point to the corresponding block in the secure memory area. The resulting memory overhead becomes a security parameter: The bigger the secure memory area, the lower the risk of collisions. In early experiments, we found that the instrumentation for this approach would have significantly higher overhead due to the management of the queue. It is better suited for narrow cases where code that deals with a well-defined data structure is hardened manually, e.g., the register save/restore during a kernel context switch. In our setting, we do not see an advantage of using randomized addresses instead of masking.

In a compiler-based setting, it is also possible to securely store data by interleaving it with random nonces. For example, each 16-byte block in AMD SEV can be split into two 8-byte halves, where the first half receives the payload, while the second half is treated as a nonce that is incremented on each write. Note that this has to be done in a single step, so the entire block may need to be buffered in a vector register, that is then written at once. This method guarantees that there are no collisions for 2^{64} writes to a given block, and has a higher locality of memory accesses, as no mask buffer is necessary. In addition, reads are almost as fast as for unprotected data, as no decoding is necessary. However, it has a high implementation complexity, as the compiler has to detect code that uses pointers to iterate over arrays and adjust such loops accordingly. Finally, the compiler needs to install logic for detecting unaligned accesses that may span multiple payload blocks, introducing a different kind of overhead. Nevertheless, interleaving may be worth exploring for programming languages that abstract away the memory layout of data structures and do not allow raw pointers.

7.4 Compatibility to CFI

Along with constant-time code and ciphertext side-channel mitigations, there are further mechanisms for ensuring secure code execution, an important one being control flow integrity (CFI) protection. For example, Intel and AMD provide the so-called control flow enforcement technology (CET), that detects unwanted control flow modifications through a shadow stack and by enforcing that indirect jumps and calls point to special `endbr64` instructions. Besides inserting direct jumps to the instrumentation section, which may be avoided by using a more sophisticated binary rewriting framework, our ciphertext

side-channel mitigation does not modify the control flow. Indirect branches still point to `endbr64` instructions, and the call stack is left untouched. Thus, CIPHERFIX is compatible with CFI mechanisms like CET.

8 Related Work

Dynamic taint analysis is a software analysis technique that is implemented in a variety of tools [15, 16, 18, 25, 27, 42]. Data flow based information tracking can support finding vulnerabilities in source or binary code. On the one hand, it can be used to increase the branch coverage of fuzzers like the AngoraFuzzer [14] or VUzzer [44] by checking on which bytes of secret inputs branching decisions are based. On the other hand, taint analysis can help to keep sensitive data always encrypted in memory through data protection tools like DynPTA [41] which is a compiler-based approach.

Automated analysis of side-channels in binaries focuses on finding non-constant-time behavior by analyzing leakages that can be modeled in different ways. There is a number of tools, which use DBI to observe leakages at runtime [51, 52] or detect secret-dependent accesses through symbolic execution [17, 49, 50]. Those existing tools for finding side-channel leakages do not cover the ciphertext side-channel attack vector, as it is not originated from a deviation in the behavior of memory accesses, but rather from the content of write accesses which affects the ciphertexts. However, they can be used to initially verify whether the code is constant-time, as non-constant-time code is even easier to attack than through the ciphertext side-channel.

Memory protection mechanisms implement the protection of sensitive data in memory. *Data space randomization* (DSR) [7] randomizes the representation of data that is stored in memory, with the aim of thwarting control flow hijacking attacks. This is done by instrumenting the code so that masks are added to or removed from variables before or after memory load and store operations. *CoDaRR* [43] extends DSR with a protection against leaking the masks that are used for DSR so that rerandomization prevents from recovering the secrets through attacking masks. These solutions are source code-based and thus not applicable for our tool.

Static binary instrumentation builds the basis for binary-level analysis and protection tools with different ways to insert additional code. The trampoline SBI approach is used by tools like *Detours* [24] and *PEBIL* [30] which relocate functions to newly-added `.text` and `.data` sections together with a redirection to these sections through 5-byte jumps. The technique is extended with inserting `int3` when a jump instruction does not fit in *BIRD* [38] and short 2-byte intermediate jumps in *DynInst* [11, 23]. In our work, we implemented an optimized combination of different jumps and `int3` to build a

lightweight static instrumentation. For a production-level implementation of CIPHERFIX, a sophisticated instrumentation framework should be used, but for our study, a custom tool tailored to the interaction with the dynamic analyses was easier integrated. Another way of coping with 5-byte jumps is *instruction punning*, as implemented in *LiteInst* [13] and *E9PATCH* [21]. This technique uses address offset bytes in a jump instruction to also encode instructions, so fewer bytes need to be overwritten. For our mitigation implementation, we did not employ instruction punning, as it introduces additional complexity and memory overhead due to the jump targets being scattered over a large memory area. *RetroWrite* [19] uses symbolization to generate reassemblable assembly that can be equipped with instrumentation passes and yields an optimized instrumented binary. Layout-agnostic binary rewriting can be performed with *Egalito* [54] that uses metadata to lift the program into a specialized intermediate representation. These approaches yield more efficient binaries, but need additional support for stripped binaries and some forms of inline assembly as used by libraries like OpenSSL, respectively.

9 Conclusion

In this work, we have presented a drop-in technique for automatically protecting binaries from leaking processed secrets through a ciphertext side-channel. Our approach comprises finding vulnerable code parts and then protecting them by preventing observable ciphertext changes based on secret data. The leakage localization technique combines dynamic binary instrumentation and dynamic taint analysis to protect only those memory accesses that deal with secrets or secret-derived data. The mitigation introduces randomness such that the plaintexts written to memory change for each write, leading to corresponding unique ciphertexts. We have shown that the highest security level of our proof-of-concept implementation can detect and mitigate all leaking memory accesses, with a very small probability of remaining leakage. Since there is no indication of fixes for existing or upcoming hardware, CIPHERFIX is a suitable approach for protecting software against the ciphertext side-channel.

Acknowledgements

We would like to thank Gregor Leander for a helpful discussion on the security of fast PRNGs, and the anonymous reviewers and our shepherd for their detailed comments and suggestions for improvement. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under grants 427774779 and 439797619, and by Bundesministerium für Bildung und Forschung (BMBF) through the ENCOPIA and SASVI projects.

References

- [1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. “Verifying Constant-Time Implementations”. In: *25th USENIX Security Symposium*. 2016. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>.
- [2] AMD. *AMD Secure Encryption Virtualization (SEV) Information Disclosure*. <https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1013.html>. 2021-08. (Visited on 2024-05-21).
- [3] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>. 2020. (Visited on 2024-05-21).
- [4] Dennis Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. No Starch Press, 2018.
- [5] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. “LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage”. In: *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020. DOI: 10.1145/3372297.3417268.
- [6] Daniel J Bernstein. *Cache-timing attacks on AES*. 2005.
- [7] Sandeep Bhatkar and R. Sekar. “Data Space Randomization”. In: *5th Detection of Intrusions and Malware, and Vulnerability Assessment Conference (DIMVA)*. 2008. DOI: 10.1007/978-3-540-70542-0_1.
- [8] Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. “Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484583.
- [9] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. “An Infrastructure for Adaptive Dynamic Optimization”. In: *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*. 2003. DOI: 10.1109/CGO.2003.1191551.
- [10] David Brumley and Dan Boneh. “Remote timing attacks are practical”. In: *Comput. Networks* 48.5 (2005), pp. 701–716. DOI: 10.1016/j.comnet.2005.01.010.
- [11] Bryan Roger Buck and Jeffrey K. Hollingsworth. “An API for Runtime Code Patching”. In: *Int. J. High Perform. Comput. Appl.* 14.4 (2000), pp. 317–329. DOI: 10.1177/109434200001400404.

- [12] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. “Fault Attacks on Encrypted General Purpose Compute Platforms”. In: *Seventh ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2017. DOI: 10.1145/3029806.3029836.
- [13] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. “Instruction punning: lightweight instrumentation for x86-64”. In: *38th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2017. DOI: 10.1145/3062341.3062344.
- [14] Peng Chen and Hao Chen. “Angora: Efficient Fuzzing by Principled Search”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018. DOI: 10.1109/SP.2018.00046.
- [15] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. “One Engine To Serve ‘em All: Inferring Taint Rules Without Architectural Semantics”. In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. 2019. URL: <https://www.ndss-symposium.org/ndss-paper/one-engine-to-serve-em-all-inferring-taint-rules-without-architectural-semantics/>.
- [16] James A. Clause, Wanchun Li, and Alessandro Orso. “Dytan: a generic dynamic taint analysis framework”. In: *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2007. DOI: 10.1145/1273463.1273490.
- [17] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure”. In: *ACM Trans. Priv. Secur.* 26.2 (2023), 11:1–11:42. DOI: 10.1145/3563037.
- [18] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. “DECAF++: Elastic Whole-System Dynamic Taint Analysis”. In: *22nd International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2019. URL: <https://www.usenix.org/conference/raid2019/presentation/davanian>.
- [19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00009.
- [20] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. “Secure Encrypted Virtualization is Unsecure”. In: *CoRR abs/1712.05090* (2017). arXiv: 1712.05090. URL: <http://arxiv.org/abs/1712.05090>.

- [21] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. "Binary rewriting without control flow recovery". In: *41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2020. DOI: 10.1145/3385412.3385972.
- [22] Hiroshi Haramoto, Makoto Matsumoto, and Mutsuo Saito. "Unveiling patterns in xorshift128+ pseudorandom number generators". In: *J. Comput. Appl. Math.* 402 (2022), p. 113791. DOI: 10.1016/j.cam.2021.113791.
- [23] Jeffrey K. Hollingsworth, Barton P. Miller, M. J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. "MDL: A Language and Compiler for Dynamic Program Instrumentation". In: *1997 Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1997. DOI: 10.1109/PACT.1997.644016.
- [24] Galen Hunt and Doug Brubacher. "Detours: Binary Interception of Win32 Functions". In: *3rd Usenix Windows NT Symposium*. 1999.
- [25] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation". In: *2011 Network and Distributed System Security Symposium (NDSS)*. 2011.
- [26] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf>. 2021. (Visited on 2024-05-21).
- [27] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. "libdft: practical dynamic data flow tracking for commodity systems". In: *8th International Conference on Virtual Execution Environments (VEE)*. 2012. DOI: 10.1145/2151024.2151042.
- [28] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *16th Annual International Cryptology Conference (CRYPTO)*. 1996. DOI: 10.1007/3-540-68697-5_9.
- [29] Pierre L'Ecuyer and Richard J. Simard. "TestU01: A C library for empirical testing of random number generators". In: *ACM Trans. Math. Softw.* 33.4 (2007), 22:1–22:40. DOI: 10.1145/1268776.1268777.
- [30] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snaveley. "PEBIL: Efficient static binary instrumentation for Linux". In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2010. DOI: 10.1109/ISPASS.2010.5452024.
- [31] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. "A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833768.

- [32] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. “Exploiting Unprotected I/O Operations in AMD’s Secure Encrypted Virtualization”. In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan>.
- [33] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. “CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel”. In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan>.
- [34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.43.
- [35] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *2005 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2005. DOI: 10.1145/1065010.1065034.
- [36] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. “CopyCat: Controlled Instruction-Level Attacks on Enclaves”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat>.
- [37] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. “SEVered: Subverting AMD’s Virtual Machine Encryption”. In: *11th European Workshop on Systems Security (EUROSEC)*. 2018. DOI: 10.1145/3193111.3193112.
- [38] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. “BIRD: Binary Interpretation using Runtime Disassembly”. In: *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2006. DOI: 10.1109/CGO.2006.6.
- [39] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *2007 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2007. DOI: 10.1145/1250734.1250746.
- [40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *The Cryptographers’ Track at the RSA Conference 2006 (CT-RSA)*. 2006. DOI: 10.1007/11605805_1.

- [41] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. “DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021. DOI: 10.1109/SP40001.2021.00082.
- [42] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks”. In: *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2006. DOI: 10.1109/MICRO.2006.29.
- [43] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. “CoDaRR: Continuous Data Space Randomization against Data-Only Attacks”. In: *2020 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2020. DOI: 10.1145/3320269.3384757.
- [44] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. “VUzzer: Application-aware Evolutionary Fuzzing”. In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017. URL: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/>.
- [45] Phillip Rogaway. “Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC”. In: *10th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. 2004. DOI: 10.1007/978-3-540-30539-2_2.
- [46] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *2010 IEEE Symposium on Security and Privacy (SP)*. 2010. DOI: 10.1109/SP.2010.26.
- [47] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. “Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484783.
- [48] Sebastiano Vigna. “Further scramblings of Marsaglia’s xorshift generators”. In: *J. Comput. Appl. Math.* 315 (2017), pp. 175–181. DOI: 10.1016/j.cam.2016.11.006.
- [49] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. “Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation”. In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai>.

- [50] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. “CacheD: Identifying Cache-Based Timing Channels in Production Software”. In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai>.
- [51] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. “DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/weiser>.
- [52] Jan Wichelmann, Florian Sieck, Anna Pättschke, and Thomas Eisenbarth. “Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications”. In: *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022. DOI: 10.1145/3548606.3560654.
- [53] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. “SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00080.
- [54] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. “Egalito: Layout-Agnostic Binary Recompile”. In: *2020 Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020. DOI: 10.1145/3373376.3378470.

A Static Instrumentation Example

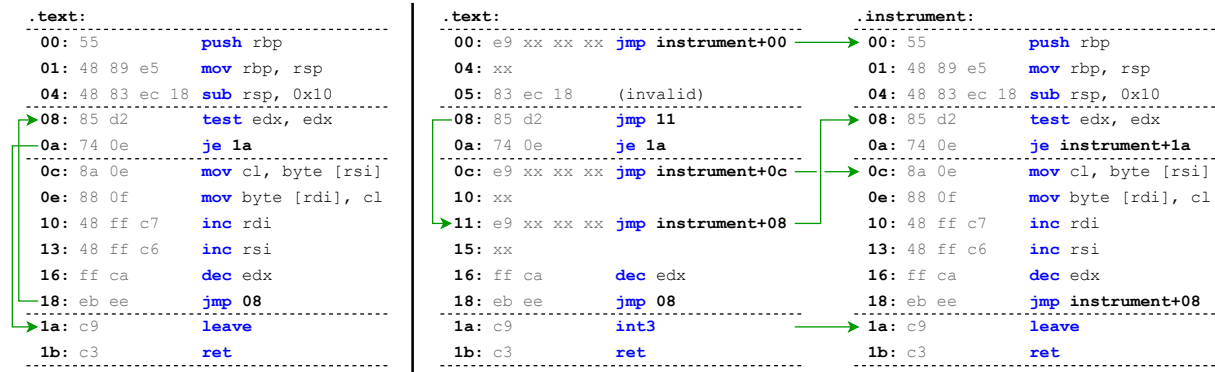


Figure 8: A simple memcpy implementation (left), and the resulting static instrumentation (right).

The basic blocks of the original code are separated by dashed lines, control flow edges are marked with arrows. The first basic block has sufficient space for a direct 5-byte jump to the instrumentation code. The second basic block only has 4 bytes, but the third basic block offers space for two 5-byte jumps, so the second basic block gets a 2-byte jump to the third basic block (offset 11) and from there a 5-byte jump to the instrumentation code. For the fourth basic block, all remaining space in the other basic blocks is already consumed, so it has to use an `int3` instruction. Execution that ends up at the beginning of any of the original basic blocks is always redirected to their counterparts in the `.instrument` section.

B Evaluation Results

Table 3: Memory accesses that have to be instrumented. Writes are split by their size, whereby $\#n$ denotes the number of n -byte writes. % instr. reads/writes shows the respective total percentage of instrumented accesses. Each target was iterated 10 times.

Target	# reads	instr. reads		# writes	instrumented writes						
		#	%		#1	#2	#4	#8	#16	#32	%
<u>libsodium</u>											
EdDSA	648 453	448 415	69	441 736	4681	0	0	372 600	6180	1160	87
SHA512	200 328	82 722	41	104 000	810	0	0	58 718	4800	784	62
<u>MBEDTLS</u>											
AES	1 887 551	1 403 255	74	1 237 457	812 120	0	42	20 715	30 256	304	70
Base64	195 458	16 020	8	128 552	23 599	0	0	5130	0	0	22
CC20	1 737 111	1 487 956	86	1 105 221	641 280	0	250 910	217	60 140	10 068	87
ECDH	37 328 410	3 454 726	9	18 773 246	0	0	881 397	1 566 188	0	1 172 058	19
ECDSA	7 120 602	3 301 437	46	3 748 086	14 240	10	260 673	1 447 753	7674	123 806	49
RSA	21 203 381	12 012 577	57	12 068 011	1950	10	360 804	7 303 398	1243	122 320	65
<u>OpenSSL</u>											
ECDH	4 799 917	390 344	8	2 532 111	2750	0	2550	248 691	62	470	10
ECDSA	12 041 083	5 463 996	45	6 950 318	2329	0	524 025	2 671 708	1492	3762	46
<u>WolfSSL</u>											
AES	3 661 782	1 550 484	42	288 454	13 150	0	90 630	60 427	10 234	0	60
CC20	2 603 432	1 547 406	59	994 267	320 320	0	476 140	25 893	20 020	0	85
ECDH	2 317 955	1 753 953	76	1 916 549	1248	0	10 752	1 409 475	20	0	74
ECDSA	19 969 154	11 606 148	58	9 519 250	721	0	543 354	5 292 431	258 140	1584	64
EdDSA	1 213 466	694 483	57	884 122	4711	0	11 560	568 368	40	82	66
RSA	2 350 077	1 886 260	80	1 193 204	1351	0	106 801	753 096	20 580	46 176	78

Obelix: Mitigating Side-Channels through Dynamic Obfuscation

Publication

Jan Wichelmann, Anja Rabich, Anna Pätchke, and Thomas Eisenbarth. *Obelix: Mitigating Side-Channels through Dynamic Obfuscation*. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

Contribution

Main author.

Outline

1	Introduction	249
2	Background	251
3	Defenses Against Attacks on TEEs	254
4	OBELIX Design	258
5	Single-Stepping Resistant Code Blocks	263
6	Implementation	270
7	Evaluation	274
8	Discussion	276
9	Related Work	278
10	Conclusion	279
	References	280
A	SEV-Step Measurements	288

B	Meta-Review	289
---	-----------------------	-----

Obelix: Mitigating Side-Channels through Dynamic Obfuscation

Jan Wichelmann, Anja Rabich, Anna Pättschke, and Thomas Eisenbarth.

Universität zu Lübeck

Trusted execution environments (TEEs) offer hardware-assisted means to protect code and data. However, as shown in numerous results over the years, attackers can use side-channels to leak data access patterns and even single-step the code. While the vendors are slowly introducing hardware-based countermeasures for some attacks, others will stay unaddressed. This makes a software-level countermeasure desirable, but current available solutions only address very specific attack vectors or have a narrow leakage model.

In this work, we take a holistic view at the vulnerabilities of TEEs and design a tool named OBELIX, which is the first to protect both code and data against a wide range of TEE attacks, from cache attacks over single-stepping to ciphertext side-channels. We analyze the practically achievable precision of state-of-the-art single-stepping tools, and present an algorithm which uses that knowledge to divide a program into uniform code blocks, that are indistinguishable for a strong attacker. By storing these blocks and the program data in oblivious RAM, the attacker cannot follow execution, effectively protecting both secret code and data. We describe how we automate our approach to make it available for developers who are unfamiliar with side-channels. As an obfuscation tool, OBELIX comes with a considerable performance overhead, but compensates this with strong security guarantees and easy applicability without requiring any expert knowledge.

1 Introduction

With ongoing digitization, there is a growing demand for hardware-enforced security that cannot be bypassed by privileged malicious parties on the same system. This applies to moving sensitive data processing to the cloud, but also to local applications which verify the user's identity or enforce the validity of software or content licenses. Owners of sensitive workloads may want to protect both their secret data and their code. Processor vendors recognized these demands and designed a variety of so-called *Trusted Execution Environments* (TEEs) that use hardware access control and cryptography to restrict access

to code and data to its owner. Examples are Intel SGX [28] for user-space processes, and AMD SEV [5] and Intel TDX [30] for whole-VM protection.

However, in recent years, there have been numerous examples of how gaps in the threat model and implementation issues can be exploited to tamper with the executed code or extract secret data. For example, all currently available TEEs are known to be vulnerable to microarchitectural side-channels like TLB and cache attacks [6, 24, 45]. For simplicity, we refer to those as *timing side-channels*. An attack method that is more specific to TEEs is *single-stepping* [13, 59], which allows an attacker to precisely measure the execution of single instructions. While those attacks do not allow direct access to the protected data, the attacker gets valuable information about the TEE's execution state, which may be used to partially derive the currently processed data and code, e.g., through counting instructions [41] or measuring their execution time [12, 46]. Another TEE-related attack vector are *ciphertext side-channels* [36, 37], where the attacker frequently reads encrypted memory and observes whether the ciphertext at a specific address changes, breaking common constant-time primitives.

For each of those attacks, there are numerous proposed countermeasures and tools, which focus on protecting the secret data [21]. Only few hardware-based methods were deployed by CPU vendors yet, so software-level countermeasures are necessary. For example, timing attacks can be mitigated by writing constant-time code or linearizing existing code [8]. Recently, with AEX-Notify [18], a hardware-assisted single-stepping countermeasure for Intel SGX was introduced, but that does not apply to other TEEs in the field. Some libraries employ custom-crafted constant-time code to defend against timing and single-stepping attacks, but that code still remains vulnerable to ciphertext side-channel attacks [36]. At the time of writing, the only available automated countermeasure for ciphertext side-channels is Cipherfix [58], which relies on dynamic taint tracking and binary rewriting and thus lacks the stability needed for practical deployment. Writing robust code that is immune to all known attack vectors requires expert knowledge, takes a lot of resources and still is not guaranteed to be side-channel free. Finally, it is not straightforwardly possible to protect the code *itself* against extraction by the attacker, which may be desirable if it, for example, contains secret algorithms.

In this work, we take a broad view at security issues on TEEs. We classify currently known attack vectors against commonly available TEEs, and discuss the suitability of existing software-level defenses, finding that each only addresses a subset of attacks. Subsequently, we show how those attack vectors can be averted with a *single*, modular drop-in countermeasure that does only require minimal action from the user. Contrary to other mitigation approaches, our approach named OBELIX does not only aim to protect secret data, but also the executed algorithms, against all relevant side-channel vulnerabilities. OBELIX takes and advances the ideas from Obfuscuro [2], to build a

dynamic obfuscation engine which runs within any TEE and can reliably protect code and data against extraction. Based upon an evaluation of the *practical* capabilities of a single-stepping attacker, we design an algorithm to partition machine code into a set of uniform code blocks, which are indistinguishable for said attacker. By storing the code blocks and the associated data in an ORAM, we effectively prevent the attacker from learning anything about the executed code and data. We present a proof-of-concept implementation for OBELIX as an LLVM compiler extension, and apply it to a number of programs.

1.1 Our Contribution

We present OBELIX, a drop-in obfuscation engine that automatically mitigates a wide range of TEE-related attacks. For that, we:

- discuss existing countermeasures and determine the properties needed for a generic mitigation;
- evaluate Linear ORAM and Path ORAM for their suitability within an untrusted client setting with small amounts of data;
- analyze an attacker’s ability to distinguish instructions through single-stepping;
- design an algorithm to split a program into uniform code blocks which are indistinguishable to an attacker;
- show how to combine these components into a single comprehensive countermeasure.

The source code of our proof-of-concept implementation is available at <https://github.com/UzL-ITS/obelix>.

2 Background

2.1 Trusted Execution Environments

Trusted Execution Environments (TEEs) offer hardware-based isolation for protecting a program’s execution against privileged adversaries on the same system. Starting with small user-space enclaves in Intel SGX [19] with at most 96MB of storage, modern TEEs are nowadays able to protect whole virtual machines (VMs). Common features of TEEs are memory encryption, hardware-enforced memory access prevention and runtime attestation. However, their implementation differs greatly. For example, Intel

SGX reserves a small amount of physical memory for the enclave, which is integrity protected, features fresh ciphertexts (only SGX version 1 [31]) and cannot be accessed by any other party than the running enclave. In contrast, the current version of AMD SEV [5] only actively prevents unauthorized write accesses to enclave memory, while relying on its (deterministic) memory encryption for averting illegitimate reads.

2.2 Side-Channel Attacks

TEEs are designed to protect against architectural attackers, i.e., malicious privileged processes on the same system which try to directly tamper with the execution. However, the threat models of many TEEs exclude *side-channel attacks* [28]. A side-channel attacker does not access private data directly, but tries to derive the data by observing seemingly unrelated channels. A simple example is measuring the execution time of the program to learn about the length of the processed data. Due to many performance optimizations and shared resources, the microarchitecture of CPUs exhibits lots of side-channel leakages. A prominent example are cache attacks, where the attacker learns whether victim data was recently accessed by manipulating and observing the cache state [1, 6, 45]. Other examples are attacks through the TLB [24] or contention of execution ports [4]. For simplicity, we refer to those as *timing attacks*. Software vulnerability to side-channel attacks is mostly rooted in its handling of secret data. If a program makes a secret-dependent memory access or executes a secret-dependent chunk of code, the attacker can use one of the above side-channel attack primitives to learn which data or code was accessed and thus reconstruct the secret.

2.3 TEE-Specific Side-Channel Attacks

While there are already numerous side-channel vulnerabilities on common systems, TEEs with their unique threat model are particularly susceptible. Various attacks [10, 20, 23, 40, 51] show how leakage can be used to recover secrets from executions deemed protected. Not only the parts of the execution inside the TEE but also control mechanisms from outside the protected memory region, like page management, can leak information to the attacker. SGX uses protected areas for the enclave page tables, but these can still be stealthily monitored via the cache [14, 57], or the access rights can be manipulated [61] by the OS. However, an attacker can not only leak secrets by observing microarchitectural effects of unprotected implementations. The encrypted memory of VM TEEs is not completely protected from the hypervisor, and the strong attacker scenario allows for very precise and fine-grained observations that enable further attacks.

2.3.1 Ciphertext side-channel attacks

The ciphertext side-channel [36, 37] is an example of a more structural leakage. Ciphertext side-channels are unique to systems that use deterministic memory encryption in combination with read access for a hypervisor to the encrypted memory. While TEEs like Intel SGX version 1 only protect a small amount of memory and can thus keep fresh nonces needed for non-deterministic encryption, this does not scale for TEEs which manage gigabytes of memory, like VMs. Thus, TEEs like Intel SGX version 2 [31] and AMD SEV employ deterministic memory encryption, where the same plaintext written to the same address results in the same ciphertext. However, while Intel SGX prohibits unauthorized reads, AMD SEV allows the malicious hypervisor full read access to the ciphertexts. If an implementation now does multiple write accesses to the same address depending on a few secret bits, the attacker can label the few resulting (and repeating) ciphertexts and infer the secret [36, 37]. This attack is unique in a way, as constant-time implementations, which are usually considered the standard countermeasure for many side-channel attacks, are particularly vulnerable. There are no publicly stated plans of AMD for introducing an effective prevention of ciphertext reads, and it is unclear whether other upcoming TEEs like Intel TDX [30] and ARM CCA [38] are immune to this class of attacks. Thus, protecting against ciphertext side-channels is left to software.

2.3.2 Single-stepping attacks

Another TEE-specific attack vector is *single-stepping*. As the TEE threat model explicitly considers privileged attackers, they can configure the CPU to precisely interrupt the enclave after every instruction. With dedicated frameworks such as SGX-Step [13] or SEV-Step [59], they can conduct very fine-grained measurements. For example, by measuring the latency of instructions, it is possible to tell apart certain opcodes and alignments [12, 46]. Another powerful primitive is instruction counting to learn loop iteration counts [11] or the length of small intra-cache line secret-dependent branches [41]. Finally, single-stepping allows for precisely applying cache attacks against the data accessed by a single instruction, enabling exploits of small vulnerabilities with as few as one measurement [55, 59], which is impossible with ordinary cache attacks due to their comparatively low temporal resolution.

2.4 ORAM

To defend against timing attackers in the cloud, one needs to conceal the memory access patterns. One such solution is Oblivious RAM (ORAM) [22]. ORAM was originally designed for large databases which do not fit in a trusted local client and thus are stored

on an untrusted server. However, ORAM can also be adapted to aid side-channel defenses for both hiding code and data [2, 48, 49]. A straightforward ORAM technique is Linear ORAM, where the client sequentially accesses all data blocks, discarding all but one. While easy to implement, this approach doesn't scale for large amounts of data. Thus, numerous other algorithms were proposed. One of particular interest for side-channel defense is Path ORAM [56], where the data is stored in a tree. On each access, the *path* containing the searched data is sent to the client. The client then replaces the address of the searched data by a random one, and writes the entire path back into the tree. Data which could not be written back (because all nodes in the path are already used) is kept in a local *stash* and only written back at the next access. The mapping of addresses to leafs in the tree is stored in a local *position map*, which may be implemented as a Path ORAM as well, leading to recursively nested Path ORAMs. This way, the protocol offers a logarithmic complexity.

3 Defenses Against Attacks on TEEs

In the following, we describe the attacker capabilities in a TEE setting and identify gaps in existing hardware-assisted and software-level defense mechanisms. In line with this, we show that an existing software defense, Obfuscuro [2], can be broken through modern single-stepping techniques.

3.1 Attacker Model

In line with the standard TEE attacker model, we assume a privileged attacker, who acts as a malicious hypervisor with read access to enclave memory. The memory itself is encrypted. The attacker has no way of decrypting the memory or modifying it, and execution state such as register values is considered safe. The TEE is susceptible to single-stepping attacks, i.e., the attacker can deterministically interrupt the enclave with precise instruction granularity. This single-stepping capability allows for counting the instructions that are executed in the enclave as well as precise measurements of their individual cache usage, execution latency and, in some cases (integer division), even operand-dependent latency. We evaluate the practically feasible attacker capabilities in this regard for two exemplary systems in Section 7. Attack vectors requiring transient execution are considered out of scope. Adversaries attempting to leverage attacks from within a TEE on the hypervisor or neighboring TEEs are not within the attacker model. Finally, we assume that the code running in the enclave is bug-free and does not exhibit software-level vulnerabilities.

3.1.1 Goals of the Attacker

We consider two major adversarial objectives: First, they are interested in the (secret) data the program processes. Second, they want to learn which algorithms are executed in the enclave, and, by extension, maybe even extracting parts of the executed code. In fact, the logic running in the enclave may be intellectual property that should be kept confidential, or the owner may not want to expose implementation details. The attacker tries to collect execution traces through various side-channels and reverse engineer the program's control and data flow, aiming to deduce which kinds of algorithms are deployed, and perhaps even identify specific modules.

3.2 Hardware-Assisted Single-Stepping Prevention

Most side-channel attacks are considered out-of-scope by TEE vendors and responsibility is shifted to the user (e.g., Intel SGX [28, Sec. Protection from Side-Channel Attacks]). However, for single-stepping, a few countermeasures are under development. AEX-Notify [18] makes Intel SGX enclaves interrupt-aware and can thereby hinder instruction-granular observations. However, AEX-Notify does not get applied on all older processors [27], and has not yet been thoroughly scrutinized by the research community. During the evolution of the newer virtual machine TEEs, single-stepping protection gets partially factored in the design process. For Intel TDX, single-stepping is restricted by randomly limiting the number of instructions executed after an interrupt through the TDX module [30, Sec. 17]. At the time of writing, no countermeasures are proposed against single-stepping for AMD SEV and ARM CCA.

3.3 Software Leakage Defenses

Due to limited availability of hardware-based security mechanisms, software-level approaches are necessary. For the three main attack classes, timing side-channels, single-stepping and ciphertext side-channels, numerous manual and (semi-)automated countermeasures were proposed. In the following, we broadly classify them and show how they can protect against algorithm fingerprinting and exfiltration of secret data. The results are summarized in Table 1.

3.3.1 Constant code and data patterns

Constant control flow and constant data access patterns (also called constant-time code) are popular building blocks for secret-independent code. While the executed code is assumed to be publicly known, the secret data being processed must be protected. The idea

Table 1: Defenses against algorithm inference (Alg. inf.) and secret data exfiltration (Sec. data exf.) through the attack classes timing leakage \odot , single-stepping \paw and ciphertext side-channels \star . Checkmarks \checkmark show protection against the given leakage class, crosses \times that no protection is achieved, and a diamond \diamond indicates that protection depends on the granularity of the defense. OBELIX combines oblivious code and data patterns with ciphertext freshness to protect against both algorithm inference and secret data exfiltration.

Defense	Alg. inf.		Sec. data exf.		
	\odot	\paw	\odot	\paw	\star
Constant code and data patterns	\times	\times	\checkmark	\checkmark	\times
Oblivious code and data patterns	\checkmark	\diamond	\checkmark	\diamond	\times
Ciphertext freshness	\times	\times	\times	\times	\checkmark

of constant-time code is that for any pair of different secret inputs the program executes the same instructions and accesses the same memory addresses. An attacker collecting an execution trace via timing side-channels like cache leakage thus should not be able to use the trace to derive information about the secret inputs. If implemented properly [41], constant-time code also helps against single-stepping attacks, as the instructions that are executed and the observable memory accesses are always the same and independent of the secret.

A related approach is randomization, where independent noise is added to decorrelate the observed trace from the secret inputs. However, this only works for certain algorithms (e.g., exponent blinding for RSA) and requires high manual effort. There is also a risk of leaking the random value while applying it to the input or removing it from the output.

3.3.2 Oblivious code and data patterns

While constant-time code reliably hides secret data from the attacker, it is trivial to extract the control flow and use that to infer which algorithms are currently running. In order to also hide the executed code, *oblivious* control flow and data access patterns are necessary. We define oblivious control flow as follows: Given a program that is divided into a set of single-entry/single-exit *blocks*, the attacker cannot distinguish which block is currently executed. This directly requires that an attacker also cannot learn the block's identity via monitoring the data accessed by it. In contrast to constant control flow, code rewritten to have oblivious control flow may still contain secret-dependent branches, but those are invisible to the attacker.

The efficacy of oblivious code against attacks depends on the block structure and the

surrounding controller logic. For example, due to their low temporal resolution, typical cache attacks cannot distinguish blocks whose size does not exceed the size of a cache line, even if the blocks have slightly varying instruction counts. However, with more powerful single-stepping attacks, the attacker can precisely count the instructions in a block and even measure their individual latency. To combat such attacks, the blocks must be carefully crafted to exhibit a uniform profile (Section 4.3).

3.3.3 Ciphertext freshness

Ciphertext side-channel leakage is caused by deterministic memory encryption in TEEs. When secrets are repeatedly written to the same address (e.g., in a typical constant-time swap pattern), the attacker can learn those by observing whether the ciphertext changes. By forcing freshness of the ciphertexts on each write, the leakage can be eliminated. There are three ways of introducing freshness on software level: First, one could XOR the data before each write with a random mask, such that the ciphertext is independent of the secret. Second, one could interleave the data with random fresh values (e.g., counters) which are updated on every write. A third method is address rotation, which takes advantage of the address-dependent tweak values used in memory encryption. When a variable is copied to a new memory location on each write, it results in a new ciphertext even if the variable's contents are not changed. All methods require heavy instrumentation and bookkeeping to automatically apply them to existing programs [58]. Note that ciphertext freshness is only necessary for secret data, as the code usually is written to a random memory location once and does not change during runtime.

3.4 Obfuscuro

A solution that partially addresses the issue of code inference is Obfuscuro [2]. However, it is susceptible to single-stepping and ciphertext side-channel attacks. Following the oblivious code approach, Obfuscuro greedily divides the program into a number of code blocks which start with a memory access and have a fixed length of 64 bytes, which is the size of an x86 cache line. The code blocks are fetched from a Path ORAM and copied into a fixed memory region called *scratchpad*. Data obliviousness is achieved through another Path ORAM, which holds 64-byte data blocks. For each code block, exactly one data block is fetched from the ORAM and copied into a data scratchpad. This way, a cache side-channel attacker only sees accesses to oblivious memory and the fixed scratchpads. Though the code blocks generated by Obfuscuro may have varying instruction counts and latencies, the limited temporal resolution of a cache attack prevents the attacker from effectively exploiting this.

3.4.1 Single-stepping attack

With single-stepping, temporal resolution is vastly increased, and the attacker can distinguish blocks by counting instructions. The instruction counts allow to assign labels to individual code blocks, helping identification of branches and the underlying algorithm. If the algorithm is partially known and contains secret-dependent branches, the attacker may be able to extract secret information. Further information is gained by additionally measuring instruction latencies. We evaluate the observable latencies in Section 7.2 and show that even for fixed instruction counts, leakage persists.

3.4.2 Other issues

Besides the vulnerability to single-stepping, Obfuscuro has a number of other issues. First, it only focuses on Intel SGX, which was complemented by VM-based TEEs after the paper's publication. Thus newer threat models are missing, e.g., it does not take into account ciphertext side-channels, which can leak the identity of a code block even quicker (Section 4.4). Then, it makes several assumptions that greatly simplify implementation complexity but reduce practical relevance of the security and performance evaluation. For example, Obfuscuro makes a single memory access at the beginning of a block, which the attacker can distinguish as being a load or store by monitoring the instruction latency, or, if possible, by checking whether the ciphertext of the data scratchpad changes. Another restriction is lack of support for position-independent code, which is non-trivial due to code being copied to a scratchpad before execution, breaking relative branches.

To summarize, while the Obfuscuro approach is a good first step at side-channel proof code obfuscation in TEEs, it does not protect against strong adversaries and has several practical limitations.

4 OBELIX Design

In the taxonomy of countermeasures in Section 3 we observed that there is a lack of hardware-based mitigations, and typical software techniques only protect against specific attack classes. We show how we can combine common mitigation approaches in a single drop-in tool, called OBELIX, which defends against all attack classes through specifically hardened oblivious code. In this section, we describe the general design of OBELIX, which includes its execution model, realization of oblivious accesses and defenses against the different attacks. Then, in Section 5, we show how we can generate uniform code blocks which are indistinguishable by the attacker. Finally, in Section 6, we explain how we can apply OBELIX to a program.

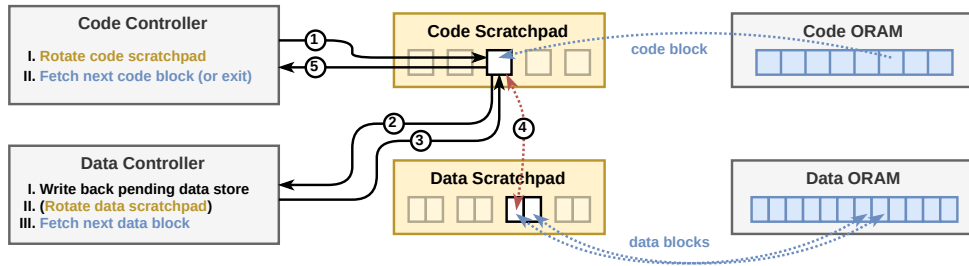


Figure 1: OBELIX execution engine overview. The code controller fetches the next code block and starts executing it (①). Then, possibly multiple times, the code block may request a certain memory address from the data controller (② ③) and access it (④). Finally, the code block redirects the execution back to the code controller to fetch the next block (⑤).

4.1 Execution Model

The core of OBELIX is its execution engine, which is depicted in Figure 1. Its design is inspired by Obfuscuro [2], but has several key conceptual and technical differences. During compile time, the program is decomposed into structurally uniform code blocks, which are intended to be indistinguishable by a side-channel attacker. Similarly, data is divided into equally sized blocks, which can happen transparently at runtime.

We use two ORAMs for storing code and data blocks, respectively. Each ORAM is managed by a dedicated controller. When execution enters the code controller, it fetches the next code block from the code ORAM, copies it into the code scratchpad, and ① starts executing it. If the code block wants to read or write data, it ② jumps into the data controller, which copies the desired memory location into the data scratchpad and ③ jumps back. Then, the code block ④ accesses the data in the data scratchpad. If the code block does not need data at the given time, dummy data is copied to the scratchpad instead. Each request to the data controller has a flag marking whether the code block wants to write to the given memory location; if that flag is set for the *previously* fetched data, the data scratchpad will be written back into the data ORAM first. Finally, execution ⑤ leaves the code block and jumps to the code controller, in order to fetch the next block. Given a suitable ORAM implementation and constant-time controllers, a timing attacker gains no direct information about the original location of the accessed code or data. The code and data scratchpads themselves are protected by the TEE’s memory encryption. This way, we achieve security against timing attackers trying to exfiltrate data or infer the executed code.

Multiple data accesses per block Steps ② to ④ may be repeated several times within a code block. If the instrumented application wants to store data, multiple data controller accesses per code block become mandatory, as we cannot interchange loads and stores at the same offset within a code block without leaking the nature of the access to a timing

attacker. Conveniently, this design can also better represent the load-to-store ratio, which typically heavily favors loads, reducing the number of total executed code blocks and thus expensive code ORAM queries. To be indistinguishable by an attacker, code blocks are generated in a way that they always have the same sequence of loads and stores.

4.2 Choosing an ORAM Engine

Most ORAM schemes described in literature are designed for handling millions of entries in large databases stored on an external server. In addition, they often assume that the ORAM controller (client) is trusted, i.e., that the attacker only resides on the server/network side. For TEEs, due to their vulnerability to various local side-channel attacks, the ORAM controller is observable by the adversary as well. The controller thus needs to be suitably hardened, likely by implementing it in a constant-time fashion. In the following, we briefly discuss two popular ORAM schemes, Linear ORAM and Path ORAM, and which approach we take for our code and data controllers.

A simple and secure approach is *Linear ORAM* [22], where we iterate over the entire data set and retrieve the desired data via a constant-time selection primitive. Let N be the number of blocks. As we access all blocks, the complexity of Linear ORAM is $\mathcal{O}(N)$. Note that there are numerous possible optimizations, like dividing highly repetitive data (e.g., uniform code blocks) into arrays of indexes into pre-computed maps, achieving compression without compromising on security.

Another simple but asymptotically much faster approach is *Path ORAM* [56], with a complexity of only around $\mathcal{O}(\log N)$. However, this protocol and others building on top of it only apply to the aforementioned client/server scenario, and the client is inherently non-constant-time. Most problems are caused by the path fetch, where an entire path is copied into a local buffer, and then written back into the tree. For ORAM bucket size B , the fetch results in $B \log N$ block accesses and writes. When writing back the path, we must not leak which entries are written back, which are copied into the stash, and which nodes are filled with dummy data. This means that we need to touch every buffer entry for every path node in the worst case, which results in $(B \log N)^2$ accesses. For a recommended bucket size of $B = 4$ [56], this means that the break-even point to Linear ORAM is at around $N \approx 1900$, not counting other factors like bad spatial locality of accesses, the high cost of frequent memory stores, and the linear scan of the position map.

Data controller For each data access, we need to always fetch *two* adjacent blocks, the first being the one actually pointed at and the second for supporting an unaligned access

to the first. The block size S_{data} is bounded by $8 \leq S_{\text{data}} \leq 32$, as the maximum non-vectorized memory access width on x86 is 8 bytes, and two blocks must fit into a single 64-byte cache line to not leak unaligned accesses to a cache attacker. For these parameters and reasonable N , Linear ORAM outperforms Path ORAM by a wide margin.

Code controller Code blocks have a much larger size $S_{\text{code}} \geq 64$, making Path ORAM more attractive. However, as explained above, N needs to be sufficiently large to overcome the penalty from the frequent path write-backs and memory stores. In our evaluation, we did not encounter a program where Path ORAM performed well, so we use an optimized Linear ORAM for the code controller as well.

4.3 Protecting Against Single-Stepping

Unlike typical timing attacks (e.g., cache side-channels), single-stepping offers a very fine-grained view of execution. First, an attacker can simply single-step each code block and count instructions [41]. As we showed in Section 3.4, filling each block until a fixed size (64 bytes) is reached leads to easily distinguishable code blocks. In OBELIX, we address instruction counting by dividing each code block into a fixed number of *instruction slots*. Moreover, we ensure that loads and stores are always placed at the same offsets within a block's instruction list. This way, the attacker always observes the same instruction counts.

A stronger attack is additionally measuring the latency of every executed instruction [12]. The resulting latency profile of a code block leads to a unique fingerprint, allowing the attacker to reconstruct a sequence of executed blocks. This sequence reveals the structure of the control flow, which may in turn expose the underlying algorithm or allow inferring secret data if the original implementation was not constant-time. OBELIX assigns each instruction slot a certain *instruction class*, which only contains instructions that are indistinguishable for the given measurement setup.

However, it is possible that a fixed latency pattern is still insufficient for full protection against single-stepping attackers: In some cases, the alignment of an instruction influences its measured latency due to peculiarities of the CPU frontend [46]. This can be avoided by placing each instruction slot at a fixed alignment, and filling the remainder of each slot with an instruction that does not expose the aforementioned latency variations, like a multi-byte no-op.

4.4 Preventing Ciphertext Side-Channel Attacks

With the previously described countermeasures, even precise timing measurements cannot distinguish individual code blocks. However, several current and proposed full-VM TEEs employ deterministic memory encryption, i.e., at a fixed memory location a given plaintext always yields the same ciphertext. Deterministic encryption is problematic for the code and data scratchpads, as an attacker can easily assign a label to each executed code block by observing the ciphertext of the code scratchpad. Similarly, the attacker can track data blocks by keeping label/ciphertext pairs, and updating the ciphertext part whenever they notice a change of the data scratchpad following a store from the code block.

As described in Section 3.3.3, there are three methods for ensuring ciphertext freshness: Adding a random mask, interleaving with a counter, and rotating store addresses.

4.4.1 Protecting code

For the code scratchpad, masking is not an option, as code must be provided in plain form to be executable by the processor. Interleaving is disadvantageous as well, as the code would need to be modified to jump over or incorporate the counters, reducing efficiency of code blocks. The last option, rotating the location of the code scratchpad for each block, has neither of those disadvantages and is thus best suited for protecting the executed code. While there may be eventual repetition of code ciphertexts, a sufficiently large pool of locations greatly reduces the probability that a certain code block ends up twice at the same address [36]. Aside from that, we found that picking a different code scratchpad location for each next block actually improves performance due to decreasing machine clears from self-modifying code (see Section 8.1).

4.4.2 Protecting data

Since we allow the program to modify data, we need to protect both the scratchpad and the ORAM. Masking and interleaving both increase memory usage by 100%. However, rotating addresses of large memory objects is even more costly, so it is only suitable for small regions, i.e., the data scratchpad. Effective masking requires a continuous supply of randomness, which was found to be quite costly [58]. For this reason, we recommend protecting larger amounts of data with interleaving. In a Linear ORAM implementation, this means splitting up data into chunks of $\frac{E}{2}$ bytes for encryption block size E , and putting counter values of $\frac{E}{2}$ bytes in between, such that each encryption block consists of a data chunk and a counter. Each time we write back into the ORAM, we increment every counter, so we get new ciphertexts for every ORAM entry. Interleaving guarantees

that a ciphertext can only repeat after at least $2^{8 \cdot \frac{E}{2}} = 2^{4E}$ iterations (i.e., 2^{64} for 16-byte encryption blocks), contrary to masking, where a fast non-cryptographic random number generator may lead to accidental collisions.

5 Single-Stepping Resistant Code Blocks

As shown in the execution engine overview in Figure 1, we divide the code into same-sized blocks, which are obviously fetched and copied into a fixed code scratchpad at run-time. While this protects against timing attackers with limited resolution, single-stepping offers instruction granularity and hence allows precisely counting and measuring the instructions executed within a block. We thus need to devise a way to generate uniform blocks, which exhibit a fixed structure and latency pattern that even a single-stepping attacker cannot distinguish. At the same time, the uniform blocks should not increase performance overhead compared to the naïve greedy block creation approach.

There are two major ways to approach this. First, one could modify the instruction scheduler in the compiler to emit a suitable instruction sequence, in a similar approach as for Very Large Instruction Word (VLIW) architectures. However, this requires writing a new instruction scheduler or at least heavily modifying the existing one, making this approach very complex and hard to maintain. We thus picked the other approach, which instead of ensuring that the generated machine code conforms to a certain pattern, computes a *block pattern* that is optimized for the existing machine code. The machine code is then post-processed and divided into blocks in an instrumentation pass.

To achieve this, we first show how we can group instructions into classes, which contain instructions mutually indistinguishable from each other by single-stepping. We then first discuss the high-level layout of code blocks, and conclude with the uniform pattern generation algorithm.

5.1 Classifying Instructions

As a first step, we need to determine which instructions fall into the same classes, i.e., can be used interchangeably without leaking their identity to the single-stepping attacker. To minimize the number of classes and avoid a lot of dummy instructions per block, we focus on the base instruction set, i.e., scalar arithmetic and memory accesses, and disable more volatile extensions like vector arithmetic. The general purpose instructions cover all common use cases.

```

-----
00: mov r14, r15; jmp 40;
40: imul r14, r15; jmp 80;
80: imul r14, r15; jmp c0;
c0: mov r14, r15; jmp 100;
100: dec counter; jnz 00;
-----

```

Figure 2: Microbenchmark for the mov and imul instructions. We execute each instruction two times in an ABBA pattern, always aligned to a cache line, to avoid bias from other CPU components.

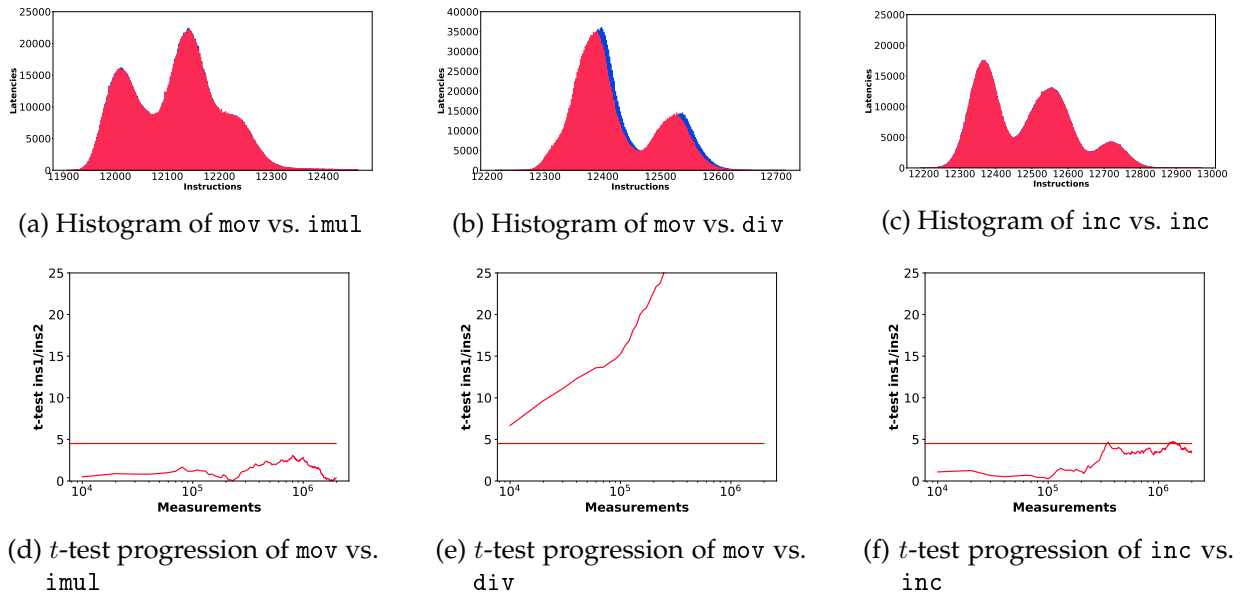


Figure 3: Histograms and t -test progressions for some instruction latency experiments on Intel SGX.

5.1.1 Measurement setup

While resources like `uops.info` offer very precise latency information, this kind of precision is in practice not achievable by an attacker, who has a lower temporal resolution due to noise from the enclave entry/exit context switches. We thus devise an own microbenchmark specifically for single-stepping. For each pair of instructions, we generate a standardized gadget which is designed to minimize noise from other CPU components, and measure it 1 000 000 times. An example gadget for the x86 `mov` and `imul` instructions is shown in Figure 2.

5.1.2 Results

After composing the instruction gadgets, the microbenchmark is loaded into an SGX enclave (respectively, an application running inside an AMD SEV VM). Our test systems are two machines, first an Intel Core i7-9750H CPU (Coffee Lake) with 16 GB of RAM, running Ubuntu 22.04.3 LTS with a custom kernel version 5.9, and an AMD EPYC 7763 (Zen3) with 128 GB of RAM, running Ubuntu 22.04.3 LTS with a custom kernel 5.19 with SEV-Step patches. Both systems were thoroughly configured to have as little noise as possible, e.g., by disabling dynamic frequency scaling and memory prefetchers, isolating cores from the kernel scheduler, disabling SMT, and minimizing overall system load. This way, we emulate a strong attacker who is able to conduct measurements with the maximum possible precision. We single-step the benchmark, recording 1 000 000 samples for every executed instruction.

As visible in Figure 3, the Intel SGX instruction measurements are quite noisy and produce multiple peaks in the histograms (for AMD, see Figure 7 in the appendix). Most of the variation is caused by the context switches to/from the enclave, where, among other activities, the entire execution state is saved/restored from memory. As a method to get a quantification of the attacker’s capabilities to distinguish the latency distributions of two instructions, we employ Welch’s t -test with a threshold of 4.5 [50]. If $|t| < 4.5$, we conclude that the instructions are not distinguishable for 1 000 000 measurements. To find out whether t converges, we plot its progression with an increasing amount of measurements.

Figure 3a and Figure 3b show the histograms for a comparison of `mov reg, reg` vs. `imul reg, reg` respectively `div reg, reg`. Per `uops.info`, on our Coffee Lake CPU, `mov` has a latency of 0.25, `imul` a latency of 3 to 4, and `div` a latency of 5 to 89. As is apparent in the histograms, `imul` is nearly indistinguishable from `mov`, while `div` clearly deviates. This is supported by the t -tests: For `imul`, t stays well below the threshold (Figure 3d), while for `div` it immediately deviates (Figure 3e), expressing clear distinguishability by an attacker. We conducted those measurements for several types of arithmetic instructions, and identified two classes with instructions which were indistinguishable for our sample count: The first one, named `class1`, contains all standard arithmetic like addition, multiplication and bit shifts, effective address computations and (conditional) register/immediate moves. The second one, `class2`, contains only the division instructions. However, as indicated by the `uops.info` measurements, those exhibit an operand-dependent latency range, so they may leak some information on their operands. If this leakage is not tolerable, we recommend avoiding such instructions until security features like Intel DOIT [25] become available, which enforce data operand-independent execution times. As a temporary workaround, we could remove and emulate such instructions by common arithmetic.

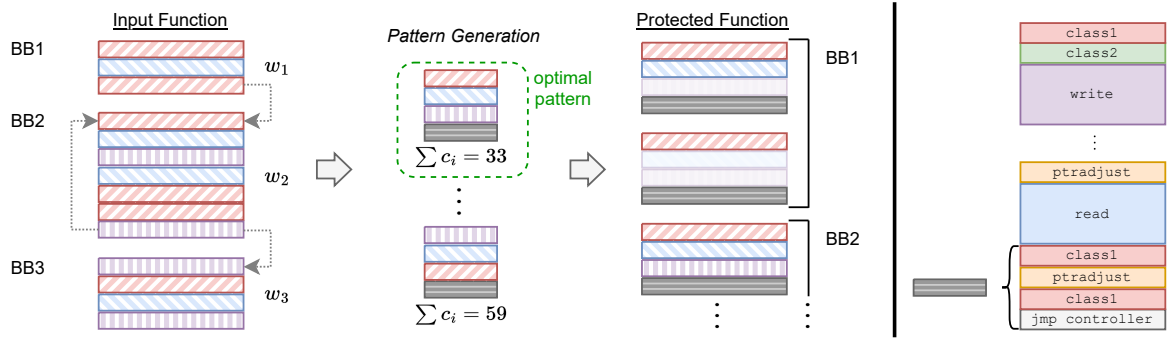


Figure 4: The function instrumentation process (left) and an example code block (right). Using the basic block profile from the input function, we compute costs for different pattern candidates, which are generated by a genetic algorithm. The basic blocks are then translated according to the optimal pattern, yielding a protected version of the function.

Note that, while `mov` and `imul` turned out indistinguishable in our experiments, they *should* be distinguishable by an attacker doing enough measurements, as they have a slightly different latency. In fact, with far more than the million measurements in our experiment, a difference may eventually become apparent. However, gathering so many measurements for a real target is very difficult in practice, especially if the target deploys application-level mitigations which prevent repeated execution. This is supported by another observation, that noise causes the t -test to first signal a leakage, but then drop below the threshold again. This effect is apparent in Figure 3d, where t briefly approaches 4.5 and then drops to almost 0 again. In our sanity check where we compared two identical `inc` instructions (Figure 3c and Figure 3f), the t -test falsely signaled a leakage.

We conclude that, while it is impossible to *prove* that an attacker cannot distinguish two instructions from the identified equivalence classes if given an even better optimized experiment environment and sufficient measurements, we deem it very unlikely in practice. If the user decides that this risk is too high, they can still increase the number of classes and narrow the latency interval of the contained instructions.

5.2 High-Level Block Layout

A code block always executes from its beginning, and concludes with an unconditional jump to the code controller at its end. The exit jump is preceded by a single (conditional) move instruction which determines the address of the next block to execute. Between the entry and exit points the block can have an arbitrary sequence of application instructions and memory accesses.

5.2.1 Instruction slots

To ensure a fixed block length and avoid leakage due to varying instruction alignment, we partition each code block into a number of *instruction slots*, as explained in Section 4.3. In practice, every instruction slot has a size of 8 bytes and holds one instruction of up to 7 bytes and one multi-byte no-op filling up the remainder of the slot. Most instructions from the x86 base instruction set fit into 7 bytes; notable exceptions are moves with a 64-bit immediate and address expressions with 32-bit displacements, which we split into a sequence of equivalent shorter instructions before instrumentation. We verified that the multi-byte no-op instructions recommended by the CPU vendors are indistinguishable through single-stepping.

5.2.2 Position-independent code

Modern programs usually employ position-independent code, which means that the code and data can be mapped to an arbitrary virtual address during startup. To access a global variable on x86-64, a *RIP-relative addressing mode* is used, i.e., the address is computed dependent on the current instruction pointer. However, this becomes a problem when the code is dynamically copied into a scratchpad at runtime. We handle this by introducing a special instruction class, called *ptradjust*, which adds the difference between the addresses of the code scratchpad and the original code to the pointer. If an instruction uses a RIP-relative addressing mode, it is translated to be followed by such a pointer adjustment.

5.2.3 Memory accesses

A code block may contain multiple memory accesses. Consistent with the uniformity requirement, each access is located at a fixed position within the block and always performs the same type of memory access (load or store). A memory access consists of the following components: We first need to load the address of the memory location we want to access, and the offset to which we want to return in the code block. Both are stored in general-purpose registers reserved for the code and data controllers, as we want to avoid polluting the application's stack. We then jump into the data controller, which performs the ORAM fetch and copies the desired memory into the data scratchpad. Subsequently, we execute the memory access instruction, where we have replaced the address operand by an access to our data scratchpad. With careful optimization, we managed to fit a memory access into 24 bytes, i.e., three instruction slots. We also ensured that all instructions involved in a memory access have a consistent latency.

5.2.4 Code block size

The code block exit point is directly preceded by three fixed instruction slots, which are needed for emulating conditional jumps. Thus, with the exit jump, the code block end takes four instruction slots. If we assume that a function both loads and stores data, we need another 6 slots for memory accesses. This means that without any other instructions any block already needs 10 slots, which are 80 bytes. The code block size directly impacts performance, as larger blocks mean fewer costly code ORAM fetches, but may also come with an increased number of dummy memory accesses. Security is also affected: In theory, the block size and pattern can be selected in a way that the function's biggest basic block can be encoded into a single code block. However, this may leak a lot about the function's internal structure, so smaller blocks are desirable. We found that a block size of 160 bytes (20 instruction slots) is a reasonable upper bound.

5.3 Generating an Efficient Block Pattern

Given the overall block structure, we now need to determine a good sequence of `classX` instructions and memory accesses. For this, we first compute profiles for the protected functions. These are then used in the second step to estimate the runtime cost of converting a function to each of the generated code block pattern candidates, so we can choose an optimal candidate in the end. The instrumentation process is illustrated in Figure 4.

5.3.1 Getting profiles for basic blocks

To generate a profile for a function, we traverse its basic blocks and then their machine instructions, assigning each one of the following classes: `load` (memory reads), `store` (memory writes), `ptradjust` (for converting RIP-relative pointers to absolute ones), and `class*` for the various instruction latency classes. Basic blocks are also given a weight, which is higher if they appear in a loop or child function.

5.3.2 Cost function

To estimate the efficiency of any pattern candidate, we devised a function that takes a pattern candidate and the weighted basic block profiles, and then simulates the basic block translation as it would occur in the actual instrumentation. Nearly all runtime cost is caused by ORAM fetches, so we want to both minimize the number of executed code blocks and dummy memory accesses. For each basic block i , we thus count the number of resulting code blocks b_i and the number of dummy loads l_i and stores s_i which need to be generated due to mismatches between the original code and the block

pattern. Given weight w_i , the cost for instrumenting basic block i is then computed as $\text{cost}(i) = w_i \cdot (b_i + l_i + 2s_i)$. Stores are weighted double due to the higher runtime cost of writing to memory. That cost function balances the number of code blocks and memory accesses, but may be adjusted to better reflect workloads that have lots of code or handle large amounts of data.

5.3.3 Computing the pattern

Given the cost function, we can now look for a pattern that minimizes the estimated runtime cost when applied to the given basic blocks. The pattern search has a few constraints: First, there is a fixed number of *instruction slots* per code block which need to be filled. The pattern must contain all instruction classes present in the profiles. Then, for good runtime performance, we want to avoid inserting too many dummy instructions. As a final constraint, the algorithm should not take too much time to run, to avoid slowing down the compilation.

Brute-force search One approach for finding a suitable pattern is a brute-force search: For example, for a small block size of 96 bytes (12 slots) and both read and write accesses, we need to distribute two memory accesses and two `classX` instructions. The resulting search space is reasonably small and can be fully scanned easily. However, for larger block sizes, the search space quickly grows, making brute-force search infeasible in the generic setting.

Genetic algorithm As a more efficient alternative, we devised a simple genetic algorithm. Initially, we generate a population of P random pattern candidates. Then, all candidates are validated, i.e., we ensure that mandatory instruction classes and the suffix is present. If a mandatory class is missing, we write it into one or more random slots. After computing the cost function for each candidate, we take the top t candidates with the lowest costs and store them in a set T . We then create a copy \tilde{T} which contains slightly mutated versions of the candidates from T with random insertions, deletions and modifications. Subsequently we crossover T and \tilde{T} to a new population \tilde{P} by combining random prefixes and suffixes of the respective elements. We then set $P := \tilde{P} \cup T \cup \tilde{T}$, so we keep both the top candidates in their original and slightly mutated versions. Finally, we replace duplicates by entirely new random candidates and repeat the process.

We limit the genetic search both by number of generations (10 000) and total runtime (10 seconds). The number of top candidates was set to $t = 15$, leading to a population size of $|P| = 2t^2 + 2t = 480$. We also fine-tuned several other parameters like the mutation rates. In our experiments, we seldom reached the generation cap, usually running a few thousand generations. In fact, for all targets which we evaluated, our algorithm

converged within the first few hundred generations, and even running it for a way longer time did not yield a better result.

As a result, we now have a code block pattern that fits the given function and minimizes the number of ORAM queries for dummy accesses.

6 Implementation

As we work towards a generic TEE protection framework with as little manual intervention as possible, we need to automate almost all instrumentation steps. OBELIX initially asks the user to mark the functions that need protection. Everything else is taken care of by a number of LLVM 17 compiler passes. First, we propagate the function markers to all child functions in the respective call tree. Then, we identify data which is accessed within said call tree, so we can generate initialization calls for the data controller. Finally, after computing a code block pattern for the functions to be instrumented, we rewrite the machine code of the marked functions to fit the pattern, and add a jump to the code controller entry point at the top of the parent function. In total, we added 9710 lines to LLVM.

In this section, we describe the aforementioned instrumentation steps and the implementation of the code and data controllers.

6.1 Marking Functions Needing Protection

As we only want to protect particularly sensitive portions of the program, we ask the user to mark the corresponding functions with a dedicated attribute (Figure 5). This is the only manual step required by OBELIX. Given an annotated parent function, we want to instrument its entire call tree, to avoid leaving secure mode in between and thus leaking control flow. We cannot use an optimal pattern for every child function, as this would allow the attacker to distinguish them during oblivious execution. Instead, we need to compute a single pattern for the entire call tree. To enable this, we create a copy of each child function which is then associated with the call tree parent function. We leave annotations on all copied child functions, such that they are processed together with the parent and later loaded into the same code ORAM.

Calls to external functions Calls to functions not residing in the same library are more challenging, and currently not supported by our implementation. The other library may be compiled with a different compiler and/or not support OBELIX at all. Thus, upon encountering a call to an external function, we would need to temporarily switch back

```
library.c:
[[clang::obelix]]
int func1(void *buf, int n) {
    ...
}

main.c:
[[clang::obelix("extern")]]
int OBELIX(func1)(void *buf, int n);

int main(...) {
    buffer = ...;
    int result = OBELIX(func1)(buffer, 5);
}
```

Figure 5: Library function annotated with a custom attribute (C23 syntax) that indicates to the compiler that the function should be protected (top). For calls from the application, the user can annotate the function prototype with a specialized version of the attribute (bottom), allowing the compiler to insert the necessary data ORAM initialization call for the `buffer` variable. The `OBELIX` preprocessor macro adjusts the function name to allow the user to use both the original and the protected versions of the `func1` function.

from oblivious to normal execution, and resume oblivious execution when the external function returns. This switch would temporarily break obliviousness as the attacker now knows the precise location in the program, and what external function it relies on. By introducing a random number of dummy blocks around the context switch we could again establish a secure state, assuming that there are only very few external calls in the function. Due to the loss of obliviousness and introduction of unprotected code into a secure context we advise against external calls, and recommend moving those outside the protected call tree.

6.2 Initializing the Data ORAM

The functions in the protected call tree work with various types of data: They may take pointers to input data, write output buffers, access global values or store local variables on the stack. Those addresses must be present in the data ORAM. We automate this by scanning the function parameters, analyzing the stack frame layout and detecting accesses to global variables, and then generating an ORAM insertion call for each pointer. This works as long as we can determine the size of each object at compile time, and those objects do not contain pointers to further objects. If we want to support nested objects, we need static points-to and type analysis. At the time of writing, due to a breaking change beginning with LLVM 15 that removed type information from pointers in IR, only older LLVM versions are supported by corresponding static analysis tools. As a

temporary workaround for the missing analysis and to ensure that OBELIX is stable even when a used memory object is not contained in the data ORAM, we created a fallback in the ORAM fetch logic that detects missing addresses and lazily inserts them on demand, at the cost of briefly violating obliviousness due to changing the size of the ORAM.

6.3 Ciphertext Side-Channel Protection

To defend against ciphertext side-channel attacks, we use the methods described in Section 4.4. The entire logic exists in the data and code controllers, and can thus easily be enabled without changes to the code blocks, depending on whether the TEE is vulnerable to ciphertext side-channels or not. We rotate the code scratchpad each time we fetch a block. For our benchmarks, we use a code block size of 160 bytes and ten memory pages (40 960 bytes) for hosting the scratchpad. This leads to 160 possible 256-byte aligned locations. The size of the pool of possible scratchpad locations has little impact on performance once it is fully allocated, so it can be easily expanded if desired.

To protect data, we rotate the data scratchpad similar to the code scratchpad, and apply interleaving to the data ORAM. For performance, we chose a data block size of 16 bytes, so the data scratchpad, which holds two blocks, fits in an AVX2 vector register. The data ORAM itself is a single contiguous array of blocks, where the mapping of original pointers to ORAM indices is kept in a separate list. As the encryption block size is 16 bytes, each data block is divided into two halves and interleaved with 8-byte counters, which are incremented on each store. Since we must update both the counter and the data at the same time, we use `vpunpck*` instructions to merge counter and data into a single vector register, which is then written back. This way, we never store the same plaintext at the same location.

6.4 Controller Implementation

We complement the compile-time instrumentation with a runtime library, which contains the code and data controllers and the ORAM implementations. The controller code is a mix of fully constant-time C and assembly code to ensure that it does not leak parts of the protected execution state or the addresses which are fetched from the ORAMs. The ORAM operations are vectorized to maximize throughput, using blending with a mask instead of `cmov`. We defer oblivious write-back operations until the next fetch, reducing load on the memory system.

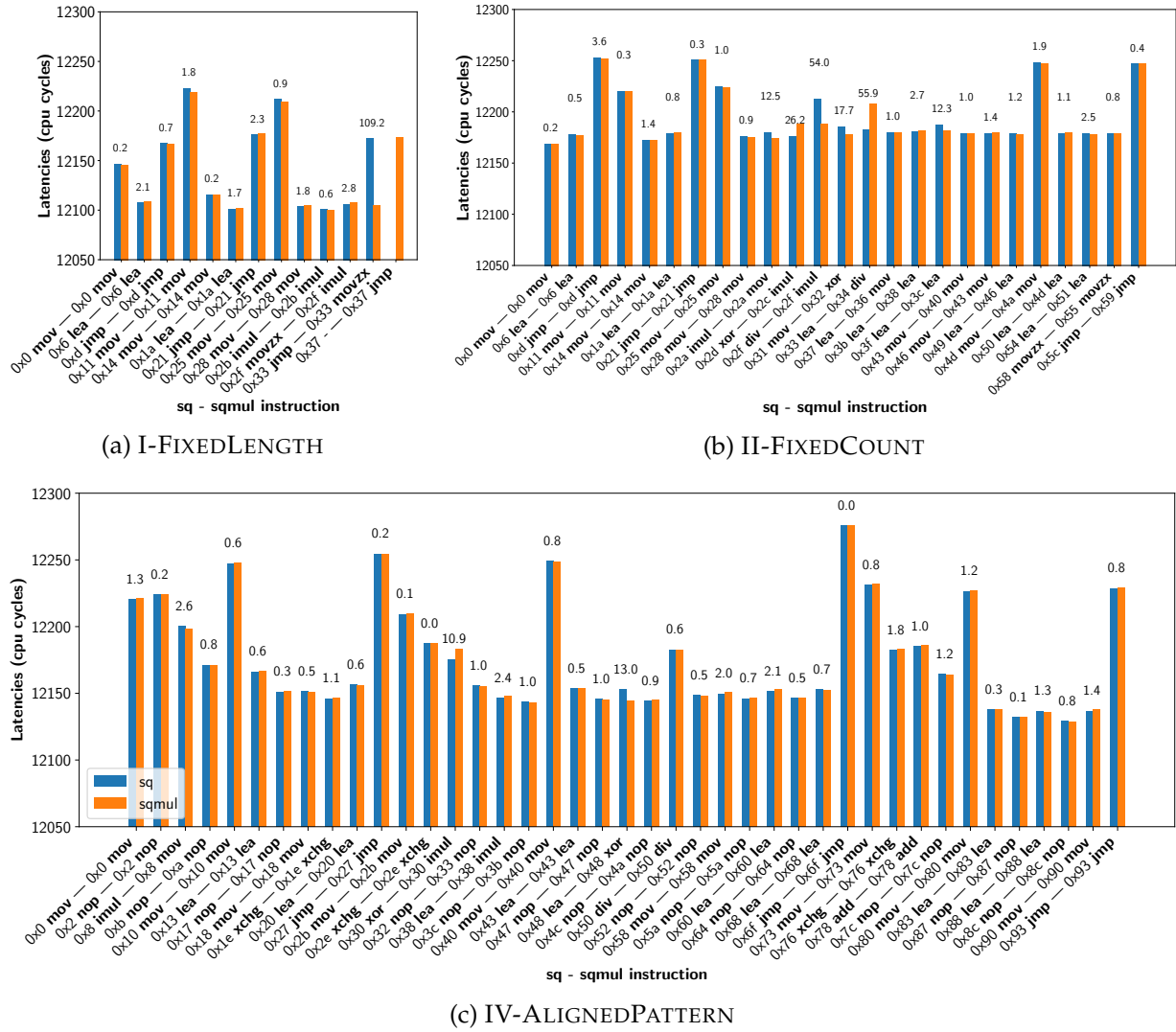


Figure 6: Single-stepping measurements of two code blocks from a square & multiply function. The blue bars represent a “square, mod” code block, while the orange bars map to “square, multiply, mod”. The numbers above the bars are the corresponding t -test result.

6.5 Correctness

To ensure that our protection preserves program semantics, every replacement of an instruction must have the same architectural behavior. When entering the controllers, status flags and register values are saved and later restored. Finally, dummy instructions are selected to be effective no-ops with zero side-effects: For instruction class `class1`, the instruction `add reg, 0` would have a suitable latency, but affects status flags. On the other hand, `lea reg, [reg+0]` has the same latency, but does not have any such side-effects. For `class2`, it is more difficult to find a dummy instruction, as there is no no-op instruction with similar latency to `div`, which affects status flags. As a workaround,

Table 2: Benchmark of several example programs with the different OBELIX variants on our AMD machine. All times are given in microseconds, the overheads are factors relative to the original execution time (orig).

Target	orig (μ s)	I-FIXEDLENGTH		II-FIXEDCOUNT		III-FIXEDPAT.		IV-ALIGNEDPAT.		V-CIPHERTEXT	
		time	factor	time	factor	time	factor	time	factor	time	factor
<u>small</u>											
matmul	0.053	37	703x	38	720x	38	706x	35	660x	38	708x
modexp	0.217	49	229x	50	231x	37	169x	35	163x	37	171x
<u>mbedtls</u>											
aes	0.134	401	2983x	403	3010x	196	1463x	192	1424x	217	1607x
base64	0.785	363	454x	440	559x	413	526x	397	506x	425	541x
cc20	0.455	2045	4526x	2068	4806x	1802	4208x	1799	4205x	1869	4403x
ecdh	890.6	90M	101 019x	90M	102 910x	68M	77 361x	69M	78 439x	78M	88 852x
rsa	1061.9	50M	47 107x	59M	55 959x	48M	45 716x	48M	44 694x	83M	78 750x

div instructions are enclosed with infrastructure that ensures that flags are correctly preserved, making class2 instructions and their dummy equivalents take two instruction slots.

7 Evaluation

We now evaluate the security and performance of OBELIX. We use the same processors as in our latency analysis in Section 5.1.

7.1 OBELIX Variants

In order to understand the characteristics of different security levels, we devised five variants of OBELIX, which can be selected when compiling the target program. The first variant is named I-FIXEDLENGTH, where we build code blocks greedily by putting a memory read and a write at the beginning and then adding as many instructions until the code block is filled. This is the Obfuscuro approach. The second variant II-FIXEDCOUNT is similar, but uses a fixed instruction count instead of filling the code block, to thwart instruction counting attacks. We set the instruction count to 10 for a code block size of 160 bytes (20 8-byte slots).

The variant III-FIXEDPATTERN is the first to actually enforce the pattern generated by our algorithm in Section 5.3, and thus is resistant to Nemesis-style [12] latency measurement attacks. Variant IV-ALIGNEDPATTERN adds no-op padding to ensure that all instructions are aligned at 8-byte boundaries, to eliminate the risk of alignment-based timing

differences [46]. V-CIPHERTEXT uses the same code blocks, but additionally enforces ciphertext freshness.

7.2 Security Evaluation

We built a small example program, called `modexp`, which computes a modular exponentiation of integers with a standard leaky square & multiply algorithm. The given implementation has a secret-dependent branch, either performing a “square, mod” operation, or “square, multiply, mod”. For different OBELIX variants, we analyze how an attacker can try to distinguish the resulting code blocks after 10 000 executions. Given the constant-time ORAM implementation, they are not able to use timing attacks to find out which block is currently executed, and the ciphertext side-channel countermeasures prevent straightforward labeling. Thus, the attacker may resort to single-stepping, to count instructions or tell them apart by measuring their latency differences. Figure 6 shows the result for the aforementioned operations. We did not observe penalties from instruction alignment [46] in our examples, so we left out the measurements for III-FIXEDPATTERN, which look identical to IV-ALIGNEDPATTERN.

We see that for variant I-FIXEDLENGTH, which builds blocks greedily until they are filled (the Obfuscuro approach), the different instruction counts are immediately apparent to an attacker, which confirms our findings in Section 3.4. Variant II-FIXEDCOUNT addresses this by using a fixed instruction count per block. However, the `div` is placed at another offset within the block, allowing the attacker to distinguish them via a latency measurement. Finally, with IV-ALIGNEDPATTERN, instructions are assigned to fixed slots with identical alignment, and the *t*-test reports no leakage for `div`. Due to noise, we observe false-positive differences between `xor` and `imul`, which disappear when conducting more measurements.

We thus conclude that, within the limits discussed in Section 5.1, an attacker is not able to distinguish code blocks which use a fixed and aligned code block pattern, making OBELIX secure in our attacker model.

7.3 Performance

To analyze the performance impact of OBELIX, we applied it to a number of targets. We first evaluated two small example programs, `modexp` and `matmul` from Obfuscuro. To show that OBELIX works with a large real-world library, we applied it to a representative set of cryptographic primitives from MbedTLS [39]. We ran 1000 measurements per target and computed the mean execution times and overheads. The results are summarized in

Table 2. Additionally, Table 3 shows the compilation time, code size and memory usage overhead of (the most expensive) variant V-CIPHERTEXT.

As is visible in the results, performance overhead is rather high, especially for large programs. We found that the main contributors to the overhead are the ORAM queries and machine clears due to self-modifying code detection (see Section 8.1). Clearly, a large program which handles lots of data also leads to more expensive ORAM queries per code block and per memory access, and thus a higher overhead. We also note that OBELIX variants with a fixed block pattern often perform better than the Obfuscuro-equivalent baseline I-FIXEDLENGTH. This is mostly due to memory accesses placed at more convenient locations, leading to comparatively fewer code blocks.

As ORAM queries are a main bottleneck, finding a more efficient ORAM implementation is highly desirable. For example, it was shown that ORAM queries can be sped up notably by offloading them to an FPGA [43]. We leave development of faster side-channel resistant ORAM algorithms to future work.

The compilation time directly depends on the parameters used for the genetic algorithm and the number of call trees which need an individual optimal code block pattern. While we used a rather high threshold of 10 seconds for the pattern search, we found that it can be safely reduced to 2 seconds without any loss of quality for the non-asymmetric examples, reducing the compilation overhead by 80%. The binary size depends on the number and size of code blocks. At runtime, memory usage is additionally influenced by the size of the data ORAM and the overhead introduced by the ciphertext side-channel protection.

We conclude that the performance overhead may be too high for large and complex programs such as asymmetric cryptography. However, for medium-sized programs which are only executed on occasion or run asynchronously (such as a license check running on an end user's system), the overhead can be justified given the broad security guarantees of OBELIX. By underlining the high usefulness of ORAM for side-channel protection, we hope to inspire further research to identify efficient ORAM schemes.

8 Discussion

8.1 Circumventing Self-Modifying Code Detection

Both Intel and AMD CPUs guarantee that self-modifying code (SMC) is executed correctly, i.e., the CPU always runs the latest version of the machine code that exists in memory. To achieve this, the CPU vendors deploy various mechanisms, e.g., checking

Table 3: Build time, code size and memory usage of the example programs protected with variant V-CIPHERTEXT. Build time and code size are given for the entire library binary, while memory usage is estimated for the isolated primitives.

Target	Build time (s)	Code size (KB)	Memory usage (KB)	
			original	instrum.
<u>small</u>	0.1	6.4		
matmul	2.5	11.1	2204	2216
modexp	2.2	12.7	2204	2240
<u>mbedTLS</u>	5.9	697		
aes	31.5	769	2472	2712
base64	31.5	732	2392	2504
cc20	28.1	864	2528	2768
ecdh	29.0	1915	2752	5916
rsa	53.5	2818	2788	6384

whether pending stores touch addresses currently present in the pipeline (AMD) or in the instruction cache (Intel). These are barely documented and cause severe penalties, as upon detecting SMC the entire CPU pipeline gets flushed (machine clear) [47].

As we overwrite the code scratchpad with the next code block, we trigger an SMC condition. Memory fences and serializing instructions (as recommended by the documentation) after the code scratchpad store do not fix this issue, which we suspect is due to out-of-order execution and possibly prefetching. We found that the code scratchpad rotation from our ciphertext side-channel protection significantly improves performance. On AMD, we could further reduce the observed penalties by adding a number of dummy `div` instructions before jumping into the new code block, to prevent the CPU from executing it out-of-order before the store has completed. Still, on AMD, SMC machine clears are responsible for up to 50% of the observed overhead in small targets; on Intel, it is up to 90%. We leave a thorough analysis of the SMC detection mechanisms and suitable workarounds to future work. Circumventing the performance penalties of SMC detection is an interesting research question, the answer to which may help speed up both OBELIX-like code hardening frameworks and common just-in-time compilers.

8.2 Integrity Protection

Another attack class relevant for TEEs are *fault injection attacks* which try to corrupt the data, code or computations within the enclave. While we did not include them in our proof-of-concept implementation of OBELIX, its modular structure and the clear separation of concerns is particularly suited for implementing effective countermeasures.

Attacks against memory integrity such as Rowhammer [33] bypass the CPU protections and flip bits directly in physical memory. While Intel SGX checks memory integrity, AMD SEV fully relies on its memory encryption to prevent targeted modifications. However, even if the attacker is not able to place their own plaintext, they can still tamper with data in order to break cryptographic implementations such as RSA-CRT [32]. OBELIX can address this by introducing an own layer of integrity checks in the ORAM controllers and the code blocks.

Preventing undervolting attacks like Plundervolt [42] which target the integrity of computations in the processor itself is more involved, but possible. For example, in the Plundervolt paper the authors stated that they could fault multiplication instructions, but not simpler arithmetic like addition or shifts. To harden OBELIX against such attacks, one could systematically identify such vulnerable instructions, and replace them in the controllers and code blocks by sequences of instructions which are less susceptible to faulting. In combination with a memory integrity protection as discussed above and the built-in control flow obliviousness which complicates targeting vulnerable code sections, this should greatly reduce the attack surface.

8.3 Transient Execution Attacks

While we considered transient execution attacks such as Spectre [15, 34] out-of-scope, OBELIX already provides good protection against such attacks: Due to partitioning the code into branch-free blocks, which are pulled from an ORAM in a rather expensive operation, the attacker has little opportunity to trigger speculative execution of code blocks. The indirect branches necessary to jump between the code block and controller are vulnerable to Spectre-BTB, but this can be addressed by `endbr64` instructions at the controller entry points, the code block entry point and in the code block memory accesses. `endbr64` was introduced with Intel CET and enforces that indirect jumps always end at such an instruction, greatly reducing the number of gadgets reachable by manipulating pointers of indirect jumps (even speculatively [26, p. 17.3.8]). This leaves hardening the controller, which can be done using conventional methods.

9 Related Work

A number of publications propose *single-stepping countermeasures* to thwart fine-grained measurements of enclaves and their state. Proposals like T-SGX [53] or MoLE [35] try to protect leaking executions from being interrupted, but they depend on TSX, which is known to introduce vulnerabilities and was therefore disabled on lots of processors

via a microcode update [29]. Varys [44], Déjà-Vu [17] and HyperRace [16] check AEX to prevent single-stepping, but with a limited scope. AEX-Notify [18, 27] successfully deployed a hardware-assisted way of making enclaves interrupt-aware to mitigate single-stepping attacks. However, it is limited to SGX. For Intel TDX, the TDX module ensures that single-stepping is no longer precise and deterministic [30] by allowing to execute a random number of instructions after an interrupt, but this only reduces the accuracy of stepping and does not prevent side-channel leakage in general. Nothing similar to the TDX module has yet been proposed for AMD SEV or ARM CCA. Therefore, other ways to protect the code running in TEEs are needed to ensure that secrets are not leaked during execution.

Compiler-based side-channel mitigations have been proposed for various protection levels. SGX-Shield [52], Klotski [62] and deterministic multiplexing [54] only hide accesses to code at page level granularity and thereby do not protect against cache attacks. Tools that result in data-oblivious execution include Constantine [8] as well as SGX-specific approaches like Obliviate [3], Raccoon [48], ZeroTrace [49] and DR.SGX [9]. However, they do not aim to protect the executed code itself, so attackers can still infer information about the used algorithms. As discussed in Section 3.4, Obfuscuro [2] has the goal of code and data obliviousness, but does neither protect against single-stepping attacks nor ciphertext side-channel leakage. A Nemesis countermeasure for specific embedded targets has been proposed by Winderix et al. [60], who equalize secret-dependent branches by aligning basic blocks in a way such that their latency profiles become indistinguishable. This has also been adapted in a thorough analysis and mitigation approach [7]. However, their work is tailored to embedded targets without critical optimizations such as caches or out-of-order execution, as they require deterministic instruction timings. On more complex processors, an attacker can mount a cache attack to observe which part of a balanced branch is executed or which memory address is accessed, bypassing the latency trace protection.

10 Conclusion

In this work, we have showcased OBELIX, a compiler-based drop-in software-level defense against various side-channel based attacks. Our approach is based on oblivious code execution and data accesses by using Linear ORAM together with uniform code blocks which are indistinguishable even for strong side-channel attackers in a TEE scenario. We have shown that OBELIX successfully safeguards implementations against different attack classes. Due to its modular structure, OBELIX can be easily extended with defenses for other attack classes in the future. In summary, even for application develop-

ers without expertise in side-channel defense, OBELIX provides a way to automatically protect implementations against all relevant attack classes.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd for their valuable feedback. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under grants 427774779 and 439797619, and by Bundesministerium für Bildung und Forschung (BMBF) through the ENCOPIA and SAM-Smart projects.

References

- [1] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *2nd ACM Symposium on Information, Computer and Communications Security*. Singapore, 2007. ISBN: 1595935746. DOI: 10.1145/1229285.1266999.
- [2] Adil Ahmad, Byungill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. “OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX”. In: *26th Annual Network and Distributed System Security Symposium (NDSS)*. 2019. URL: <https://www.ndss-symposium.org/ndss-paper/obfuscuro-a-commodity-obfuscation-engine-on-intel-sgx/>.
- [3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. “OBLIViate: A Data Oblivious Filesystem for Intel SGX”. In: *25th Annual Network and Distributed System Security Symposium (NDSS)*. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_06A-2_Ahmad_paper.pdf.
- [4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Taveri. “Port Contention for Fun and Profit”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00066.
- [5] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>. 2020. (Visited on 2024-05-21).
- [6] Daniel J Bernstein. *Cache-timing attacks on AES*. 2005.

- [7] Marton Bognar, Hans Winderix, Jo Van Bulck, and Frank Piessens. “MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling”. In: *8th IEEE European Symposium on Security and Privacy (EuroS&P)*. 2023. DOI: 10.1109/EUROSP57164.2023.00045.
- [8] Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. “Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484583.
- [9] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostinen, and Ahmad-Reza Sadeghi. “DR.SGX: automated and adjustable side-channel protection for SGX using data location randomization”. In: *35th Annual Computer Security Applications Conference (ACSAC)*. 2019. DOI: 10.1145/3359789.3359809.
- [10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostinen, Srdjan Capkun, and Ahmad-Reza Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *11th USENIX Workshop on Offensive Technologies (WOOT)*. 2017. URL: <https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser>.
- [11] Jo Van Bulck, David F. Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes”. In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019. DOI: 10.1145/3319535.3363206.
- [12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. In: *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018. DOI: 10.1145/3243734.3243822.
- [13] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *2nd Workshop on System Software for Trusted Execution (SysTEX@SOSP)*. 2017. DOI: 10.1145/3152701.3152706.
- [14] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. In: *26th USENIX Security Symposium*. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck>.

- [15] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *28th USENIX Security Symposium*. 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/canella>.
- [16] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, Xiaofeng Wang, Ten-Hwang Lai, and Dongdai Lin. “Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018. DOI: 10.1109/SP.2018.00024.
- [17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. “Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu”. In: *2020 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2017. DOI: 10.1145/3052973.3053007.
- [18] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. “AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/constable>.
- [19] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: <http://eprint.iacr.org/2016/086>.
- [20] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. “CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 171–191. DOI: 10.13154/tches.v2018.i2.171-191.
- [21] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. “A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries”. In: *2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023. DOI: 10.1145/3576915.3623112.
- [22] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs”. In: *J. ACM* 43.3 (1996), pp. 431–473. DOI: 10.1145/233551.233553.
- [23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. “Cache Attacks on Intel SGX”. In: *10th European Workshop on Systems Security (EUROSEC)*. 2017. DOI: 10.1145/3065913.3065915.

- [24] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *27th USENIX Security Symposium*. 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>.
- [25] Intel. *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html>. 2023.
- [26] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals*. 2023.
- [27] Intel. *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*. <https://cdrdv2-public.intel.com/819680/architecture-instruction-set-extensions-programming-reference.pdf>. 2024. (Visited on 2024-05-21).
- [28] Intel. *Intel Software Guard Extensions*. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf. 2016. (Visited on 2024-05-21).
- [29] Intel. *Intel Transactional Synchronization Extension (Intel TSX) Disable Update for Selected Processors*. <https://cdrdv2.intel.com/v1/dl/getContent/643557>. 2023.
- [30] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*. <https://cdrdv2-public.intel.com/733575/intel-tdx-module-1.5-base-spec-348549002.pdf>. 2023. (Visited on 2024-05-21).
- [31] Intel. *What Technology Change Enables 1 Terabyte (TB) Enclave Page Cache (EPC) size in 3rd Generation Intel® Xeon® Scalable Processor Platforms?* <https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html>.
- [32] Chong Hee Kim and Jean-Jacques Quisquater. “Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures”. In: *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop (WISTP)*. 2007. DOI: 10.1007/978-3-540-72354-7_18.
- [33] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014. DOI: 10.1109/ISCA.2014.6853210.

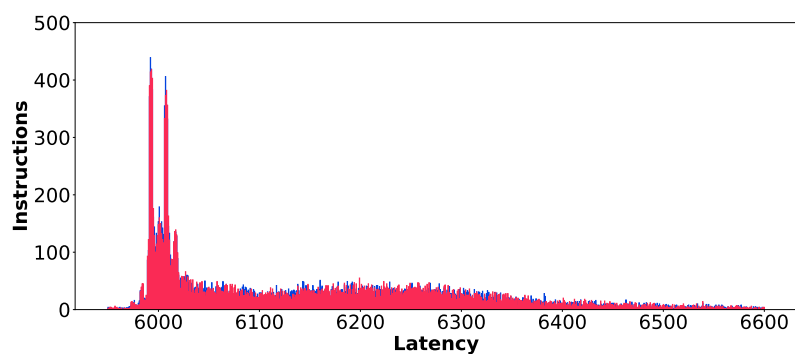
- [34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00002.
- [35] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. “MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape”. In: *2022 Annual Computer Security Applications Conference (ACSAC)*. 2022. DOI: 10.1145/3564625.3568002.
- [36] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. “A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022. DOI: 10.1109/SP46214.2022.9833768.
- [37] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. “CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel”. In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan>.
- [38] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. “Design and Verification of the Arm Confidential Compute Architecture”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2022. URL: <https://www.usenix.org/conference/osdi22/presentation/li>.
- [39] Mbed-TLS. <https://github.com/Mbed-TLS/mbedtls>. (Visited on 2024-05-21).
- [40] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *19th International Cryptographic Hardware and Embedded Systems Conference (CHES)*. 2017. DOI: 10.1007/978-3-319-66787-4_4.
- [41] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. “CopyCat: Controlled Instruction-Level Attacks on Enclaves”. In: *29th USENIX Security Symposium*. 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat>.
- [42] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00057.

- [43] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. "TRUSTORE: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA". In: *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020. DOI: 10.1145/3372297.3417265.
- [44] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference (USENIX ATC)*. 2018. URL: <https://www.usenix.org/conference/atc18/presentation/oleksenko>.
- [45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *The Cryptographers' Track at the RSA Conference 2006 (CT-RSA)*. 2006. DOI: 10.1007/11605805_1.
- [46] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/puddu>.
- [47] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks". In: *30th USENIX Security Symposium*. 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/ragab>.
- [48] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". In: *24th USENIX Security Symposium*. 2015. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane>.
- [49] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. "ZeroTrace: Oblivious Memory Primitives from Intel SGX". In: *25th Annual Network and Distributed System Security Symposium (NDSS)*. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_02B-4_Sasy_paper.pdf.
- [50] Tobias Schneider and Amir Moradi. "Leakage assessment methodology - Extended version". In: *J. Cryptogr. Eng.* 6.2 (2016), pp. 85–99. DOI: 10.1007/S13389-016-0120-Y.
- [51] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *14th Detection of Intrusions and Malware, and Vulnerability Assessment Conference (DIMVA)*. 2017. DOI: 10.1007/978-3-319-60876-1_1.

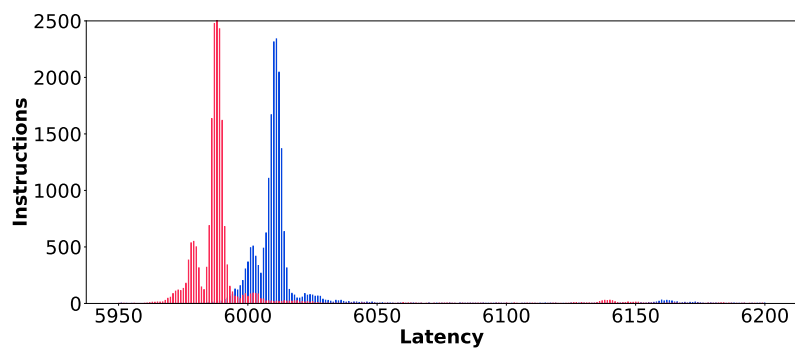
- [52] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs”. In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017. URL: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/sgx-shield-enabling-address-space-layout-randomization-sgx-programs/>.
- [53] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs”. In: *24th Annual Network and Distributed System Security Symposium (NDSS)*. 2017. URL: <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/>.
- [54] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. “Preventing Page Faults from Telling Your Secrets”. In: *2016 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 2016. DOI: 10.1145/2897845.2897885.
- [55] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. “Util: Lookup: Exploiting Key Decoding in Cryptographic Libraries”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021. DOI: 10.1145/3460120.3484783.
- [56] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *J. ACM* 65.4 (2018), 18:1–18:26. DOI: 10.1145/3177872.
- [57] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134038.
- [58] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. “Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software”. In: *32nd USENIX Security Symposium*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/wichelmann>.
- [59] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. “SEV-Step A Single-Stepping Framework for AMD-SEV”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 180–206. DOI: 10.46586/TCHES.V2024.I1.180-206.

- [60] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. “Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks”. In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021. DOI: 10.1109/EUROSP51992.2021.00050.
- [61] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. 2015. DOI: 10.1109/SP.2015.45.
- [62] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. “Klot-ski: Efficient Obfuscated Execution against Controlled-Channel Attacks”. In: *2020 Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020. DOI: 10.1145/3373376.3378487.

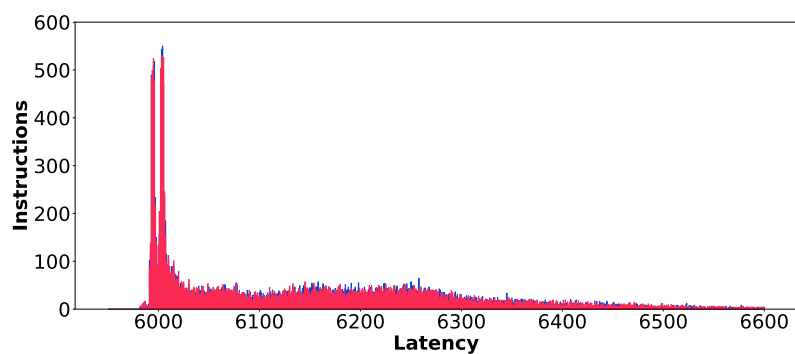
A SEV-Step Measurements



(a) Histogram of `mov` vs. `imul`



(b) Histogram of `mov` vs. `div`



(c) Histogram of `inc` vs. `inc`

Figure 7: Histograms for our instruction latency experiments on AMD SEV, analogous to those depicted in Figure 3 for Intel SGX.

B Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1 Summary

This paper presents Obelix, a framework to mitigate side-channel attacks in trusted execution environments (TEEs). By leveraging a linear oblivious RAM model and enforcing uniform code blocks, Obelix can prevent attackers from gaining insights into both executed code and accessed data.

B.2 Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

B.3 Reasons for Acceptance

1. This paper addresses a long-known issue. Recent work has revealed numerous vulnerabilities that attackers can exploit to tamper with the executed code or secret data in trusted execution environments. This paper aims to protect both the code and data against relevant side-channel vulnerabilities in TEEs.
2. The paper provides a valuable step forward in an established field. There are numerous existing countermeasures, but most of them focus on protecting the secret data, and only a few are deployed into the hardware by vendors. Obelix takes a holistic view of the vulnerabilities and designs solutions that can protect both code and data against a wide range of TEE attacks at the software level.
3. The paper creates a new tool to enable future science. Obelix is implemented as an LLVM compiler extension, and the authors will make it publicly available, thereby facilitating future research.

B.4 Noteworthy Concerns

While Obelix offers valuable security enhancements, its deployment could lead to considerable overhead. This work evaluates that with micro-benchmarks, but it lacks thorough evaluations of real-world, full-fledged applications.

Part III

Appendix

About the Author

Education

2012 Abitur

at Kolleg St. Thomas Vechta

2012 – 2015 Bachelor of Science *Informatik*

at Universität zu Lübeck

2015 – 2017 Master of Science *Informatik*

at Universität zu Lübeck

2017 – 2024 Doctoral Studies

at Universität zu Lübeck



Peer-Reviewed Publications

- Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *International Journal of Parallel Programming (IJPP)*, 2018.
- Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *34th Annual Computer Security Applications Conference (ACSAC)*, 2018.
- Luca Wilke, Jan Wichelmann, Mathias Morbitzer, Thomas Eisenbarth. “SEVurity: No Security Without Integrity - Breaking Integrity-Free Memory Encryption with Minimal Assumptions”. In: *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
- Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. “Util::Lookup: Exploiting Key Decoding in Cryptographic Libraries”. In: *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.
- Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. “undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation”. In: *Workshop On Offensive Technologies (WOOT)*, 2021. (best paper award)

- Jan Wichelmann, Sebastian Berndt, Claudius Pott, and Thomas Eisenbarth. “Help, My Signal has Bad Device! - Breaking the Signal Messenger’s Post-Compromise Security Through a Malicious Device”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) - 18th International Conference*, 2021.
- Jan Wichelmann, Florian Sieck, Anna Pättschke, and Thomas Eisenbarth. “Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications”. In: *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.
- Sebastian Berndt, Jan Wichelmann, Claudius Pott, Tim-Henrik Traving, and Thomas Eisenbarth. “ASAP: Algorithm Substitution Attacks on Cryptographic Protocols”. In: *2022 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2022.
- Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. “A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP”. In: *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.
- Jan Wichelmann, Christopher Peredy, Florian Sieck, Anna Pättschke, and Thomas Eisenbarth. “MAMBO-V: Dynamic Side-Channel Leakage Analysis on RISC-V”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) - 20th International Conference*, 2023.
- Jan Wichelmann, Anna Pättschke, Luca Wilke, and Thomas Eisenbarth. “Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software”. In: *32nd USENIX Security Symposium*, 2023.
- Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. “SEV-Step: A Single-Stepping Framework for AMD-SEV”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2024.
- Jan Wichelmann, Anja Rabich, Anna Pättschke, and Thomas Eisenbarth. “Obelix: Mitigating Side-Channels Through Dynamic Obfuscation”. In: *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

Awards

2018 Bester Master-Abschluss des Studienjahres 2017/2018

im Studiengang Informatik

2021 Best Paper Award at WOOT 2021

for “undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation”

2022 Walter-Dosch-Lehrpreis der MINT-Sektionen 2022

für das Praktikum “Cybersecurity” im gleichnamigen Modul