UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Side-Channel Leakage Analysis of Randomized Implementations

Seitenkanalanalyse von randomisierten Programmen

Bachelorarbeit

verfasst am
Institut für IT Sicherheit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Kjell Dankert

ausgegeben und betreut von
Prof. Dr.-Ing. Thomas Eisenbarth

mit Unterstützung von
Jan Wichelmann

Lübeck, den 19. Juli 2024

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Kjell Dankert

Zusammenfassung

Seitenkanalangriffe auf kryptografische Bibliotheken in der Vergangenheit haben die Notwendigkeit einer automatisierten Analyse aufgezeigt, da diese Schwachstellen von Entwicklern nur schwer erkannt werden. Dies ist insbesondere bei Implementierungen der Fall, welche Randomisierung wie Blinding einsetzten. Bisherige Forschungsergebnisse haben gezeigt, dass dynamische Seitenkanalanalysen mithilfe statistischer Tests diese Schwachstellen in randomisierten Implementierungen effektiv finden können. Die meisten Tools benötigen dafür immer noch einen manuellen Arbeitsanteil, sodass Personen ohne Vorkenntnisse über Seitenkanäle diese Analyse nicht durchführen können. In dieser Arbeit stellen wir einen Algorithmus vor, welcher den Vergleich von mehreren Programmabläufen in Seitenkanalanalysen mithilfe der Konstruktion eines azyklisch gerichteten Graphen ermöglicht. Der vorgeschlagene Algorithmus verbessert die Fähigkeit, Schwachstellen zu finden, welche ansonsten durch andere Schwachstellen überdeckt werden würden. Wir zeigen, dass Microwalk mithilfe dieses Ansatzes in der Lage ist, weitere potenzielle Schwachstellen in kryptografischen Bibliotheken zu erkennen. Darüber hinaus implementieren und evaluieren wir einen Ansatz, kritische Instruktionen in randomisierten Implementierungen zu finden. Hierfür vergleichen wir die Verteilungen von Schwachstellen, die bei zufälligen und festen geheimen Eingaben beobachtet werden, mithilfe von statistischen Tests, ohne dass Analysten oder Entwickler manuell in diesen Prozess eingreifen müssen. Dabei zeigen wir, dass die Anwendung dieses Ansatzes innerhalb von Microwalk hilft, fehlerhafte Blinding-Implementierungen zu identifizieren. Wir demonstrieren jedoch auch, dass diese Technik nicht in der Lage ist, falsche Verwendungen von Blinding-Implementierungen in einigen Beispielen zu erkennen, da keine zusammengesetzten Schwachstellen detektiert werden können.

Abstract

Side-channel vulnerabilities within cryptographic libraries have proven the necessity of automatic side-channel analysis tools since side-channel leakages are challenging to identify, particularly in implementations that involve randomization. Previous research has indicated that statistical tests within dynamic side-channel analysis tools effectively assist the detection of side-channel leakages. However, most tools still require manual intervention from developers, which prohibits their practical usage without prior knowledge. We propose a trace alignment algorithm for differential side-channel analysis tools to align multiple execution traces within acyclic directed word graphs. This algorithm enhances the ability of differential side-channel analysis tools to find leakages covered by control flow leakages. With this approach, we demonstrate that Microwalk can detect additional potential side-channel vulnerabilities within common cryptographic libraries. Additionally, we implement and evaluate the approach to finding randomized leakages at the instruction level by statistically comparing distributions of leakages observed from random and fixed secret inputs, without requiring manual interventions of analysts or developers. We show that the application of statistical tests in the analysis of randomized implementations within Microwalk can help to identify incorrect blinding implementations. However, we also demonstrate that this technique might not detect some incorrect usages of blinding since it cannot detect composite leakages.

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, mich bei den Personen zu bedanken, welche diese Arbeit möglich gemacht haben. Zunächst gebührt dieser Dank dem ITS und insbesondere Professor Eisenbarth, welcher mich im vergangenen Jahr auf das Thema aufmerksam gemacht hat und mir die Möglichkeit anbot, durch das Bachelor-Projekt und der Bachelorarbeit einen Einblick in die Forschung zum Thema Seitenkanäle zu bekommen. Des Weiteren gilt mein Dank auch dir, Jan. Unsere wöchentlichen Besprechungen haben mir stets weitergeholfen und mich motiviert bis zur nächsten Woche neue Ergebnisse präsentieren zu können. Ich möchte auch meiner Familie danken, welche mich in dieser nicht ganz stressfreien Zeit immer unterstützt hat. Mein Dank gilt euch auch für das mehrfache Korrekturlesen dieser Arbeit. Frank, dir bleibt dies zumindest für die nächsten paar Jahre erspart, auch wenn es dich mit anderen Themen schlimmer hätte treffen können. Auch bei dir, Malin, möchte ich mich für die stundenlangen Telefonate bedanken, um über die englische Kommasetzung zu diskutieren, nur um erneut festzustellen, dass ein Komma an dieser Stelle mal wieder weggelassen werden kann. Ohne euch wäre diese Arbeit nicht möglich gewesen.

Contents

1

Introduction

Microarchitectural side-channel attacks have proven critical in real-world applications, enabling malicious users to extract information from cryptographic processes in shared processing environments (Irazoqui et al., 2014; İnci et al., 2016). As enterprises and individuals shift towards cloud computing, hardening cryptographic programs and libraries against side-channel attacks proves significant.

However, various attacks against common cryptographic libraries that already try to prevent side-channel attacks have demonstrated the necessity of finding side-channel leakages in applications and libraries efficiently and reliably (Aranha et al., 2020; Sieck et al., 2021; Weiser et al., 2020). In recent years, researchers have developed plenty of side-channel analysis tools that follow different approaches to support developers and analysts in finding side-channel leakages.

Differential side-channel analysis represents one of these approaches and relies on comparing execution traces collected by observing library invocations and executions. In deterministic settings, each difference in the execution trace indicates secret-dependent leakages that an attacker might be able to exploit. However, the naive comparison of the execution traces only finds leakages until different conditional branches are executed since the following entries within the traces no longer match. Therefore, it is necessary to realign execution traces beyond the scope of the conditional branches to reduce the number of false negatives in the analysis. Since the alignment of multiple execution traces is not trivial, most trace-based side-channel analysis tools compare all possible pairs of traces (Xiao et al., 2017; Weiser et al., 2018).

Microwalk, a differential side-channel analysis framework developed by the University of Lübeck, compares multiple execution traces simultaneously (Wichelmann et al., 2022). However, it does not realign execution traces beyond the scope of conditional branches.

Therefore, Microwalk might yield false negative results. This thesis aims to implement and evaluate an algorithm to realign execution traces within Microwalk to improve its potential to find leakages in cryptographic libraries.

Another problem in dynamic side-channel analysis is how to analyze randomized implementations that use techniques like blinding to harden them against side-channel attacks. Randomness is especially problematic for differential side-channel analysis tools since they assert that differences within traces indicate secret-dependent side-channel leakages. This implication is incorrect for randomized implementations, as differences in execution traces might be unrelated to secret inputs. However, recent research has indicated that detecting secret-dependent side-channel leakages within randomized implementations is possible by using statistical tests (Weiser et al., 2018).

We aim to adopt this approach and extend Microwalk's ability to discover side-channel leakages within randomized implementations.

1.1 Contributions of this thesis

This thesis proposes an algorithm to realign execution traces within a directed acyclic word graph using an iterative construction approach to increase the accuracy of differential side-channel analysis tools to discover leakages. Moreover, it evaluates the implementation of this algorithm into Microwalk by analyzing real-world cryptographic libraries. Additionally, the thesis implements a suggested leakage test proposed in DATA (Weiser et al., 2018) into Microwalk to extend its ability to analyze randomized library implementations. Furthermore, it evaluates the implementation of the leakage test on sample applications and presents limitations of finding leakages in randomized implementations using statistical tests.

1.2 Structure

This thesis commences by providing some background information in Chapter 2. It includes information about caches, microarchitectural side-channel attacks, dynamic side-channel analysis approaches and an introduction to Microwalk. Chapter 2 also presents research related to this thesis and summarizes the current state of the research in automated dynamic side-channel analysis. The following chapters present the two functionalities to improve Microwalk's side-channel analysis. Chapter 3 proposes an algorithm for aligning multiple execution traces within Microwalk to reduce false negative

results during the analysis, especially within randomized settings. The chapter also explains the implementation of the proposed algorithm and evaluates its application within Microwalk's analysis. Afterwards, Chapter 4 discusses an idea about finding leakages in randomized implementations using statistical tests within Microwalk. It also presents the implementation and evaluation of this idea to assess leakages in randomized implementations. Chapter 5 concludes the final results and gives an outlook on open questions and future work.

2

Background and related work

This chapter discusses basic concepts that include information about caches and microarchitectural side-channel attacks. Additionally, it presents different approaches to finding secret-dependent leakages used in side-channel analysis tools. This chapter concludes with a basic introduction to Microwalk's analysis approach and architecture.

2.1 Caches

As processors have gotten faster through the years, accessing the main memory has become a major bottleneck in modern computer architecture (Hennessy and Patterson, 2011). Caches have been developed to allow faster access to parts of the memory recently requested from the main memory. Faster access times are possible due to the lower microarchitectural complexity of caches compared to the main memory and the fact that they are closer to the processor cores. The spatial location of caches also provides another benefit since transferring data over smaller distances requires less energy.

Caches store memory blocks within cache lines with a fixed length that usually defaults to 64 bytes. If data from the memory is requested and not yet stored within the cache, the cache loads the block of data that contains the requested address from the main memory. It stores the requested data in one of its cache lines for future access. Caches, additionally, store an identifier called a tag, which uniquely determines the memory block stored within a cache line. The cache can identify whether a memory block is stored within a cache line by comparing the requested tag to the stored one.

2.1.1 Cache placement strategies

Platform-specific cache placement strategies determine the placement of data within cache lines (Hennessy and Patterson, 2011). These strategies can be grouped into three categories.

Direct-mapped caches use the tag of the requested memory block to calculate the cache line to store data. Given the tag of the requested memory block t, the cache line size of s, and the number of cache lines n, the used cache line is often calculated with $t/s \bmod n$. The direct usage of the tag means that the same requested memory block is always saved within the same cache line, as each cache line i can only contain blocks with starting addresses a that satisfy $t/s \bmod n \equiv i$.

Fully associative caches use another placement strategy to circumvent the inflexibility of direct-mapped caches. They try to be more flexible by allowing every cache line to store every memory block. If data is requested from the cache, the requested tag is compared to all tags inside the cache and the data is returned if a matching cache line is found. Else, the data is loaded into the cache from the main memory. Therefore, it is not possible to calculate which memory blocks will be stored in a single cache line.

However, nowadays, most caches use a layered combination of both approaches called set-associative caches. These caches consist of a direct-mapped cache that maps memory block addresses to cache sets instead of cache lines. Each of those cache sets is then fully associative, allowing them to store every memory block from this set in different cache lines. It combines the efficient data fetching from direct-mapped caches and the flexibility of fully associative caches. Due to the direct-mapped association with cache sets, it is possible to infer possible memory blocks that could be saved within the observed cache set.

2.1.2 Cache hierarchies

Caches are organized hierarchically to facilitate even quicker memory access (Hennessy and Patterson, 2011). Cache hierarchies typically consist of three layers with different access times and capacities. L1 caches are closest to the processor cores and provide quick access to a small amount of memory. Due to its limited capacity, data is evicted from the cache if new data is stored and no cache line is available. The memory management unit inserts the evicted data into a larger L2 cache with slightly larger access times. Typically, each processor core has its own L1 and L2 cache. Processor cores usually share an additional L3 cache. It serves the same purpose for the L2 caches as the L2 caches do for the L1 caches. Due to its larger capacity and its shared access from multiple processors, it is slower than

the L1 and L2 caches. Nevertheless, accessing data from the L3 cache is still faster than accessing it directly from the main memory. If requested data is not stored within the caches, it is retrieved from the main memory instead.

Common computer architectures like x86 hide this cache hierarchy from the software components. Since the computer architecture manages memory automatically, developers only use a single address space to address memory. This method eases the development of programs and the efficient usage of memory in multiprocessor environments. However, it also causes memory accesses to have different execution times.

2.2 Microarchitectural side-channel attacks

In contrast to most other attacks, side-channel attacks try to infer information about processed data by observing the nonfunctional characteristics of a program (Wichelmann et al., 2018). There are a lot of different characteristics that could be exploited, such as execution times and power consumption of computers during computations. However, this thesis will focus on a subset called microarchitectural side-channels. Attacks on microarchitectural side-channels exploit different hardware features present in current computer architectures to gain information about data of other processes. More specifically, shared caches and branch prediction units are used to gain information about another running process on the same system.

The following sections present different microarchitectural side-channels relevant to the analysis of Microwalk. An attacker could exploit them to infer information about processed data.

2.2.1 Memory accesses

Side-channel attacks often rely on observing memory access patterns to infer information about other processes (Wichelmann et al., 2018). As caches have different access times, an attacker can learn which cache was accessed to retrieve the requested data. Using cache attacks, such as Flush+Reload (Yarom and Falkner, 2014) or Prime+Probe (Osvik et al., 2006), an attacker can manipulate cache line entries and infer possible addresses of accessed memory locations. If these memory accesses are secret-dependent, which means that accessed memory addresses correlate to the secret, an attacker might be able to infer secrets from cryptographic processes. Therefore, programs with secret dependent memory accesses are potentially vulnerable to side-channel attacks. Since an attacker

might infer information about secret keys, critical cryptographic implementations should avoid secret-dependent memory accesses.

Flush+Reload attacks exploit shared memory between processes (Yarom and Falkner, 2014). The victim's and the attacker's process must run simultaneously and share some memory for this attack to work. Memory is shared between processes, for example, if they use the same shared library. Using a cache flush instruction like cflush for the x86 architecture, an attacker can evict the shared memory of the victim's process. If the victim then tries to access a secret-dependent memory address from the shared memory address range, its memory block is loaded into the cache from the main memory. After that, the attacker accesses all memory blocks within the shared memory and measures its access times. The attacker can identify accessed memory blocks since they have lower access times due to being already stored within the cache. If the accessed memory addresses depend on a secret value, the attacker might learn something about the secret.

Figure 2.1 shows a simple Flush+Reload (Yarom and Falkner, 2014) implementation that an attacker could use to gain information about a victim's secret input parameter. In this case, lookup.c is a dynamically linked shared library that provides a lookup table for pre-calculated function values. It is assumed that a single cache line consists of 64 bytes, so each entry of the aligned lookup table has its own memory block. The attacker begins by clearing all the shared memory. It then waits for a lookup in the victim's process. After that, the access time for each potential lookup is measured and reported to the attacker. Using this information, the attacker can reconstruct the victim's secret parameter since memory blocks associated with the key will have faster access times.

Prime+Probe (Osvik et al., 2006) also exploits shared caches to gain information about memory accesses of another process. However, it does not require shared memory between the attacker's and victim's processes. Instead, Prime+Probe exploits the eviction behavior of caches by filling it with its own data. It then waits for the victim to access different memory addresses. If the victim's process accesses the memory, it evicts some of the attacker's data from the cache. In the next step, the attacker can check the access times to its data. If some memory accesses are quicker, an attacker can infer address ranges that correlate with cache lines that the victim's process accessed. The resolution of memory address ranges depends on the platform-specific cache placement strategies presented in Section 2.1.1. Prime+Probe's accuracy is worse than Flush+Reload's since it can only determine all memory blocks correlating to the same cache line. However, its application is more diverse since it does not require sharing memory with the victim's process.

```c
#include <stdint.h>

__attribute__((aligned(64)))
uint8_t LOOKUP_TABLE[256][64] = { ... };
```

lookup.c

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

extern uint8_t LOOKUP_TABLE[256][64];

int secretFunc(uint8_t input, uint8_t output[64]) {
    output = LOOKUP_TABLE[input];
    return 0;
}

int main(int argc, char **argv) {
    char* p;
    uint8_t input = strtol(argv[1], &p, 10);
    printf("%u\n", LOOKUP_TABLE[input]);
}
```

victim.c

```c
#include <stdio.h>
#include <stdint.h>
#include <x86intrin.h>

extern uint8_t LOOKUP_TABLE[256][64]; // 256 * 64 bytes

// Flushes the given memory address from the cache.
static void clflush(void *ptr) {
    __builtin_ia32_clflush(ptr);
    _mm_lfence();
}

// Gets the current time stamp counter value from the processor.
static uint64_t rdtsc() {
    _mm_lfence();
    return __rdtsc();
}

int main() {
    for(int i = 0; i < 256; ++i) {
        clflush(LOOKUP_TABLE[i]);
    }

    ... // Waits for the victim to perform a lookup.

    for(int i = 0; i < 256; ++i) {
        uint64_t timeStart = rdtsc();
        uint8_t tmp = LOOKUP_TABLE[i][0];
        uint64_t timeEnd = rdtsc();
        uint64_t time = timeEnd - timeStart;
        printf("%d: %ull\n", i, time);
    }
}
```

attacker.c

Figure 2.1: Flush+Reload cache timing attack

$$x^n = \begin{cases} x \cdot (x^2)^{(n-1)/2}, & 2 \nmid n \\ (x^2)^{n/2}, & 2 \mid n \end{cases}$$

(a) Defintion for $n > 0$

```
1: y ← 1
2: while n > 1 do
3:     if 2 ∤ n then
4:         y ← x · y
5:         n ← n − 1
6:     x ← x²
7:     n ← n/2
```

(b) Algorithm

Figure 2.2: Control flow leakage through Square-and-Multiply

2.2.2 Control flow

Another side-channel that an attacker could exploit is observing the control flow of a program (Wichelmann et al., 2018). As branches within the execution of programs change the sequence of executed instructions, an attacker can infer which branch has been executed. These attacks are possible, as information about a single branch might leak through execution time differences or other memory accesses. Especially in the case of loops, the instruction cache, which is typically a small part of the cache hierarchy described in Section 2.2.1, might leak information about the number of iterations. If control flow depends on the secret data, an attacker might infer information about the secret. Consequently, secret-dependent branches should be avoided in cryptographic libraries.

A typical example of leakage through control flow is exponentiation by squaring, typically called the Square-and-Multiply algorithm shown in Figure 2.2. Due to its simple definition shown in Figure 2.2a and its applicability in the Diffie-Hellman Key Exchange and RSA encryption, naive modular exponentiation implementations might use it. However, the naive implementation in Figure 2.2b is vulnerable to side-channel attacks. The third line introduces a branch that depends on the current value of n. Depending on which branch was executed, an attacker might infer the last bit of n in each iteration and reconstruct the original value of n. The Diffie-Hellman Key Exchange and RSA encryption use a secret key as the exponent in calculations. Consequently, naive implementations with conditional control flow might be vulnerable to side-channel attacks.

```c
if (x % 2 == 0) {
    y = TABLE[a];
} else {
    y = TABLE[b];
}
```

(a) Original vulnerable code

```c
int mask = -(x % 2);

res = ~mask & TABLE[a] | mask & TABLE[b];
```

(b) Constant-time implementation

Figure 2.3: Constant-time example written in c

2.3 Side-channel prevention

Since developers cannot always avoid secret-dependent control flow or memory accesses, different techniques have been established to harden cryptographic libraries against side-channel exploitation.

2.3.1 Constant-time programming

Constant-time programming is a paradigm for eliminating conditional operations within implementations. It has become popular after Kocher demonstrated that timing attacks can leak information about performed operations since different operations might not always take the same amount of time (Kocher, 1996). Constant-time programming can harden programs against side-channel attacks by eliminating conditional control flow. It works by effectively performing computations from multiple branches and discarding all but one of the results depending on the original condition of the branches.

Figure 2.3 shows an example of the application of constant-time programming. Figure 2.3a, implemented in c, is vulnerable to side-channel attacks since it involves secret-dependent branches. Figure 2.3b uses bit masks constructed from the condition of the branch to compose the result from both data accesses, each performed in one of the branches. Since the mask's construction in the first line does not leak any information about `x`, an attacker cannot infer any information by observing the program's execution. All observations would register data accesses to `TABLE[a]` and `TABLE[b]` so that an attacker cannot conclude which lookup is discarded.

Constant-time programming is a powerful technique to harden library implementations against side-channel attacks. However, it also introduces a performance penalty if computations are expensive or plenty of different branches are possible since all of them are being computed instead of a single one. Yet, it is still an effective technique to eliminate secret-dependent control flow.

2.3.2 Blinding

Another strategy used to prevent the leakage of sensitive information is called blinding. Blinding is a technique to compute functions without revealing information about the input or output values (Chaum, 1983). It is used within cryptographic implementations to harden them against side-channel attacks since leakages within the computation do not reveal any relevant information. Blinding randomizes some inputs before critical code sections in a particular way so that randomness is removed from the result afterwards.

Given a compatible computation $y = f(x)$ with $x \in X$ and $f : X \to Y$, blinding introduces randomness by choosing a random permutation $E : X \to X$ on the input space of f as encoding and its associated decoding $D : Y \to Y$, which removes the randomness from the result afterwards. The value y can be computed by calculating $y = D(f(E(x))) = (D \circ f \circ E)(x)$ instead of applying f to x directly. Since E is unknown to the observer, they cannot infer information about x by observing the computation. Observed leakages are randomly distributed and cannot be used to extract information about the original input values.

For example, blinding can be applied to RSA's modular exponentiation by choosing a random $r \in [1, N)$ with $\gcd(r, N) = 1$ (Kocher, 1996). Instead of directly encrypting the plaintext m, it can be encoded first by applying $E(x) = x \cdot r \bmod N$. In the final step, blinding removes randomness by applying the decoding $D(x) = x \cdot (r^{-1})^e \bmod N$, which yields the following blinding implementation:

$$(D \circ f \circ E)(m) = (D \circ f)(m \cdot r \bmod N) = D((m \cdot r)^e \bmod N) = (m \cdot r \cdot r^{-1})^e \bmod N = f(m).$$

Kocher argues that an attacker cannot infer any information about the message m using the described blinding implementation as long as r and $(r^{-1})^e \bmod N$ are unknown to the attacker (Kocher, 1996). Since calculating $(r^{-1})^e \bmod N$ or reusing previous values might be vulnerable to timing attacks, he also proposes a method to derive new values for r and $(r^{-1})^e \bmod N$ from previous ones in a secure way.

2.4 Dynamic side-channel leakage analysis

In recent years, multiple side-channel leakage analysis tools have been developed, which use different methods to find and localize secret-dependent control flow and memory accesses. These tools can be categorized into static and dynamic analysis approaches (Weiser et al., 2018). These approaches fundamentally differ in how they gain information about the analysis target. Static approaches produce statements about the target that apply to all input parameters to prove the absence of secret-dependent side-channels without actually executing it. Instead, dynamic approaches use instrumentation to infer information about the program by its executions. Due to this approach, dynamic approaches do not guarantee to find all side-channel leakages within the code base, as this ultimately depends on whether vulnerable code sections are executed. This approach also causes problems with randomized applications as dynamic side-channel leakage analysis tools rely on statistical tests to verify whether leakages are random.

The following sections present multiple dynamic approaches to the problem of finding side-channel leakages dynamically. This overview is limited to dynamic approaches since the objective of this thesis is ultimately the integration of functionality within Microwalk, which also uses a dynamic approach based on trace diffing.

2.4.1 Taint tracking

Taint tracking is one of the most intuitive ways to detect secret-dependent leakages. It allows one to annotate different inputs with labels and track their usage within the program. Taint tracking detects secret-dependent memory accesses and branches as they depend on values marked as secret-dependent.

ct-grind (Langley, 2010) is a tool that tracks the usage of secret-dependent input values by providing a patch for the Valgrind debugging framework. ct-grind prevents secret inputs within the program from being initialized and tracks their usage with memcheck. It can detect whether memory accesses or control flow relies on this data by propagating the undefined values throughout the program. ct-grind reports found leakages with an instruction-level granularity.

However, using the approaches from ct-grind, any masking or blinding of the input values is marked as secret-dependent. Both implementations would report those leakages, even if the implementation is not flawed. Irazoqui et al. investigated this by also collecting cache traces for the affected code locations (Irazoqui et al., 2017). They estimated dependencies between cache traces and input values with mutual information analysis and were able to reduce false positive results.

2.4.2 Fuzzing

Another method related to the field of side-channel analysis is fuzzing. Fuzzing is a method to create different input values automatically that produce different execution paths. ct-fuzz (He et al., 2020) and DifFuzz (Nilizadeh et al., 2019) use this method to generate two different inputs repeatedly. These tools pass the inputs to the analysis target, which records potential leakages using dynamic instrumentation. If the two generated leakage reports differ, they argue that an attacker could observe the program's execution using different inputs and inevitably learn something about them. The analyses of both tools prove efficient for deterministic applications since these approaches often find vulnerabilities within a short time.

2.4.3 Statistical tests

One of the rudimentary solutions to detect secret behavior is measuring the execution times of specific parts of an algorithm. dudect (Reparaz et al., 2017) uses this approach by performing multiple measurements for different input values. Statistical tests can find significant differences between the measured times of input values. Reparaz et al. argue that input-dependent control flow causes significant differences in the measured execution times. However, they cannot localize leakages since they do not record any information about the instructions.

While Zankl et al. also use statistical tests to detect side-channel leakages, their approach differs (Zankl et al., 2017). Focusing specifically on libraries that perform modular exponentiation, they use statistical methods to determine the correlation of the exponent to the number of executions of single instructions. If the exponent correlates with the number of executions, evidence for control flow side-channel leakages is found and reported with instruction-level granularity.

2.4.4 Trace diffing

Trace-based or differential leakage analysis tools generate traces of accessed addresses and branches within different executions using dynamic instrumentation. Differences between execution traces might indicate exploitable leakages.

STACCO (Xiao et al., 2017) uses this approach. It focuses its analysis on TLS implementations and Bleichenbacher attacks. It traces branches within the process of two random TLS packets and compares these control flow traces with regular diff tools. Moreover, STACCO also reports whether an attacker could use these leakages within cache or page attacks by calculating the cache line and page of the exact leakage address.

DATA (Weiser et al., 2018), which stands for differential address trace analysis, is another tool that performs differential side-channel analysis using dynamic instrumentation. It aims to facilitate the analysis of x86 binaries and collects memory accesses and branches in traces during the execution with different input parameters. Using a custom diff algorithm, DATA detects leakages between two different traces. DATA repeatedly traces instructions whose target addresses differ in execution traces with random and fixed inputs. After that, a statistical test compares the distributions of target addresses and the number of accesses within traces for both input sets. With this strategy, DATA can detect leakages in programs that use randomization to harden them against side-channel attacks. However, quantifying leakages still needs manual intervention, as the analysis requires the development of a specific leakage model that correlates inputs with found leakages. Nevertheless, it is the only tool extensively tested with randomized implementations to detect secret-dependent leakages efficiently.

2.5 Microwalk

Microwalk is a trace-based microarchitectural leakage detection framework that quantifies and localizes side-channel leakages using dynamic instrumentation and statistical methods. It currently supports the analysis of x86 (Wichelmann et al., 2018), RISC-V (Wichelmann et al., 2023) and Node.js (Wichelmann et al., 2022) libraries and applications. Microwalk aims to be a tool for developers that easily integrates with existing Continuous Integration (CI) pipelines to help developers detect vulnerabilities within their code. Moreover, it also allows the analysis and verification of closed-source libraries and applications (Wichelmann et al., 2018).

Most of Microwalk's analysis relies on the comparison of execution traces. Each trace consists of multiple trace entries that contain observed jumps (including calls and returns), memory accesses, and memory allocations. A call tree aligns these entries within the traces for an efficient comparison. See Section 3.1 for further information about the construction of the call tree. The analysis modules then traverse the tree and examine differences between trace entries and splits within the tree. Using the number of test cases accessed in each node, Microwalk quantifies found leakages using statistical methods (Wichelmann et al., 2022).

Microwalk's architecture is based on a custom analysis pipeline (Wichelmann et al., 2018) shown in Figure 2.4. Its pipeline consists of multiple stages in which different modules are available. The analysis commences with generating or loading different test cases. Microwalk then passes these test cases to the trace generation stage in which applications

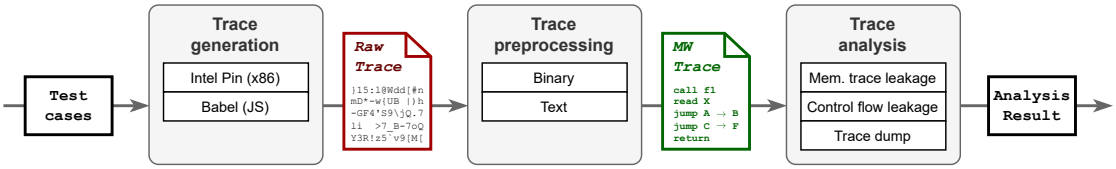


Figure 2.4: Microwalk Pipeline (Wichelmann, 2022)

are instrumented to record memory accesses and control flow in execution traces. The preprocessor optimizes them to process them efficiently in later stages. At last, these traces are processed in different analysis modules to identify leakages within the targeted library and applications. This pipeline approach allows Microwalk to be easily extendable, as extensions for different languages or platforms only require the development of compatible tracers. It also allows one to develop analysis modules specifically crafted to target different analysis use cases.

3

Alignment of multiple execution traces

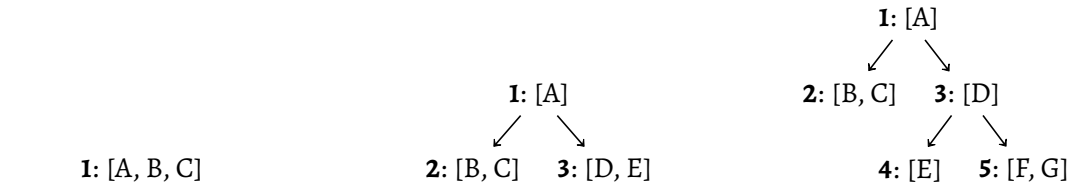
This chapter discusses the development of a data structure to efficiently compare multiple execution traces while constantly realigning the traces across the scope of control flow. We discuss this implementation of this approach into Microwalk and describe changes necessary to adapt the previous alignment implementation. This chapter also verifies the effectiveness of this approach by analyzing the call graphs constructed in various sample scenarios. Finally, we evaluate the application of this construction approach by analyzing common cryptographic libraries.

3.1 Call tree construction

Since Microwalk's approach to finding microarchitectural side-channel leakages within applications is based on detecting differences between execution traces, it is necessary to facilitate the efficient comparison between different traces. Microwalk achieves this by constructing a call tree that encodes differences of execution traces in its underlying structure (Wichelmann et al., 2022).

The call tree is implemented using a radix tree structure often used to compare text sequences efficiently. Each node contains a part of the sequence and references to subsequent nodes. The radix tree splits in case sequences deviate from each other so that its paths encode the sequences inserted into the radix tree.

Microwalk adapts this structure to encode execution traces as sequences of trace entries. A trace entry contains information from the memory accesses, memory allocations and jumps within a program's execution. This information is later used during the analysis to localize and quantify leakages. These sequences are extracted from the execution traces



(a) Addition of $[A, B, C]$ (b) Addition of $[A, D, E]$ (c) Addition of $[A, D, F, G]$

Figure 3.1: Construction of the call tree using the traces $[A, B, C]$, $[A, D, E]$ and $[A, D, F, G]$

and are added to the radix tree individually. Therefore, each node of the radix tree contains a list of trace entries and a list of pointers to the following nodes within the tree.

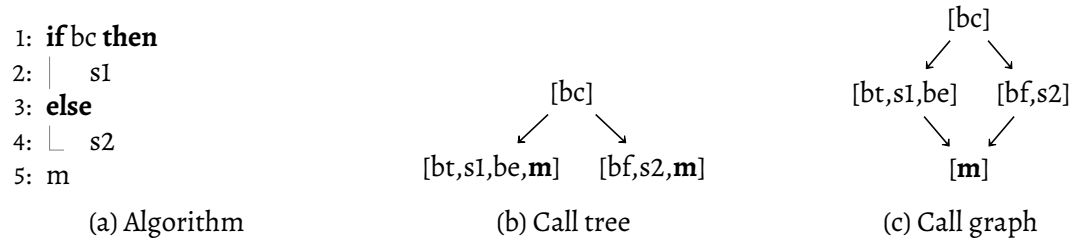
However, there is an exception to splitting the radix tree in case of deviations within execution traces. Multiple memory accesses with different target addresses are saved within the same trace entry since they should not cause splits within the radix tree. Microwalk also handles function calls differently. To support the analysis of nested function calls, Microwalk translates each function call to a single trace entry and adds it to the radix tree as usual. However, this trace entry also contains a child radix tree that encodes the execution trace of this function call. The call tree, therefore, represents a nested radix tree.

The first addition of an execution trace into the call tree would create a single node that contains the whole sequence of call tree entries. However, in later additions, inserted execution traces might deviate from the preceding sequences. In this case, two new nodes are created that represent the execution traces beginning from the point of execution where they start to deviate. Consequently, each deviation in the execution traces creates a split within the call tree.

The example in Figure 3.1 shows the iterative construction of a call tree using three test cases with their corresponding traces. The first trace contains entries $[A, B, C]$, the second one $[A, D, E]$ and the third one $[A, D, F, G]$.

As this type of visualization will be used in the rest of this chapter, some remarks on the notation of this figure: Each node in this graph presents a node of the radix tree. A node can have an optional numerical name to reference them within explanations. Additionally, it contains a list of trace entries identified by letters. Each outgoing edge represents a pointer to a subsequent node.

In the first step, the sequence $[A, B, C]$ is added, which produces a single node 1 that contains all entries as shown in Figure 3.1a. After that, another execution trace containing A, D and E is added. This sequence splits the node 1 during construction since entries B and D differ. Node 2 is created, which contains the rest of the entries of node 1. In addition to that, the construction adds a new node 3 that contains the rest of the sequence that



(a) Algorithm (b) Call tree (c) Call graph

Figure 3.2: Example for the alignment of conditional branches

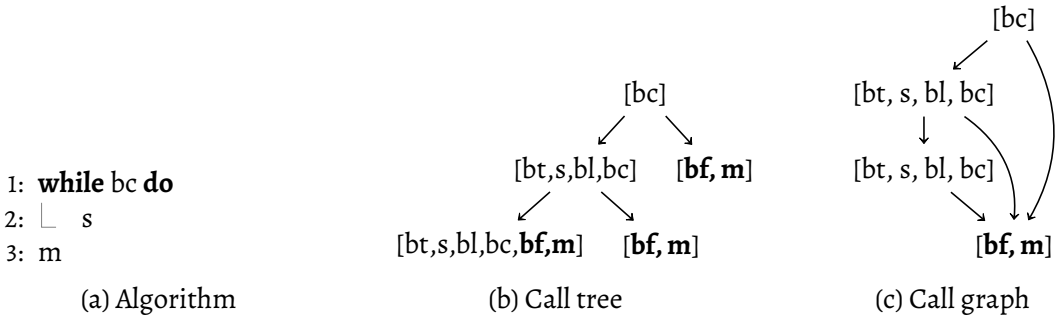
Legend: bc: branch condition, bt: jump to if block, bf: jump to else block, be: jump to skip else block, s1: statement in if branch, s2: statement in else branch, m: rest of the algorithm

was added to the radix tree. The resulting radix tree is shown in Figure 3.1b. Figure 3.1c shows the addition of the sequence $[A, D, F, G]$. Since this sequence differs from $[A, D, E]$ at the second position, node 3 is split. Node 4 is created and contains the rest of the sequence added in the second round. Additionally, the algorithm creates another node 5 that contains all remaining entries of the third sequence. The construction process, therefore, constructs a radix tree with two deviations at the nodes 1 and 3.

3.2 Problem

Microwalk's current methodology for identifying deviations in execution traces involves the iterative construction of a radix tree. It splits the radix tree into multiple branches in case of different conditional or indirect jumps in different execution traces. This approach is problematic since each deviation might cover later leakages after the conditional or indirect branch. More specifically, leakages originating within conditional branches are not discovered later. This is caused by the fact that execution traces which split due to different conditional or indirect jumps are kept independent since radix trees do not merge if they encounter an identical entry later on. It is, therefore, required to realign the branches in the call tree to reduce the number of false negative results in the analysis stage.

Figure 3.2 shows an example of the problem of missing leakages in case of deviations within the call tree. It represents a simple case where execution traces deviate after evaluating the condition of the branch bc in Figure 3.2a. Microwalk's current approach would construct a call tree similar to the one shown in Figure 3.2b. Each branch of the tree would be constructed independently due to the deviation caused by bc, thus producing two entries for m. Each entry for m within the call tree is analyzed individually, which might result in false negative results. For example, this might be the case if m represents



(a) Algorithm (b) Call tree (c) Call graph

Figure 3.3: Example for the alignment of branches within loops

Legend: bc: branch condition, bt: jump to the loop, bl: jump to the start of the loop, bf: jump to the end of the loop, s: statement inside the loop, m: rest of the algorithm

a memory access instruction and the target address is modified in both s1 and s2. As shown in the algorithm in Figure 3.2a, m is the same instruction for each execution as the branches merge in line five at the end of the if-statement. A better approach would be to merge the entries of m to form a graph as shown in Figure 3.2c. In this case, the leakage of the conditional control flow would no longer shadow potential leakages within m.

There are a few more cases where such a merge could be helpful to reduce the number of false negatives. Most notably, Figure 3.3 shows a loop where the number of iterations depends on the loop's condition. The current analysis would miss leakages after the loop due to the deviations caused by the loop's condition since it constructs a call tree as illustrated in Figure 3.3b. However, it would be best to merge branches within the call tree if they contain the same sequence of entries as shown in Figure 3.3c. In this case, this is satisfied by the jump bf and the rest of the algorithm m, which are performed by all executions. If we merge branches, Microwalk can perform its analysis independently on the number of iterations of the loop.

3.3 Idea

For Microwalk's new approach, we construct a directed acyclic word graph instead of a radix tree to additionally encode the program's control flow correctly since we can use this information to facilitate a more precise analysis. More specifically, we want to merge branches that split due to control flow within executions to facilitate a more accurate analysis of leakages for the following code sections.

Our approach to enable merges of different branches relies on keeping track of splits caused by indirect and conditional branches within execution traces. We track trace entries added to the graph after such splits as possible merge points since branches might

merge into them again. As the graph is constructed iteratively, we can track merge points in one iteration and use them in later ones to prevent searching the graph for the correct merge point in later iterations.

There is a single difference to the previous construction approach. If the current test case deviates from previous execution traces, we use the information of potential merge points to merge branches again instead of keeping them independent. We achieve this by searching for a similar already discovered trace entry in the other branch and merging both branches into that entry if it exists. We also save performed merges to keep track of which splits still require to be merged.

However, there are a few caveats to this kind of construction. Firstly, we need to prohibit merges across different function calls, as these would not correctly represent the program's control flow and could cause unexpected results in the analysis of nested function calls. Secondly, we also require splits to be merged in the reverse order as they have appeared since the program's conditional blocks are constructed hierarchically. As blocks of nested conditionals might end before the same instruction, we allow merges of multiple splits within the same entry as long as they are still possible according to the rules above. Since our goal is ultimately the correct control flow representation and not to maximize the alignment of execution sequences, we always try to merge splits within their first possible merge point. Section 3.4.4 explains the reason for this in more detail.

Our new approach for Microwalk's call graph construction aims to be backwards compatible with previous analysis modules. Additionally, it facilitates a quantitative comparison to Microwalk's call tree construction approach.

3.4 Implementation

Despite our aim to encode execution traces within a new data structure, we do not require any changes to its underlying trace entry and analysis structure, which facilitates backwards compatibility with previous Microwalk versions.

3.4.1 Memorization

Our new construction approach relies on memorization to store merge points for different branches that split due to conditional control flow. The algorithm keeps track of actual and possible splits within all function calls using triples, which are stored in a stack-like data structure to represent the call stack.

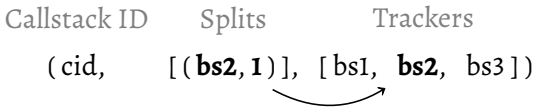


Figure 3.4: Triple inside the branch split stack

An example of such a triple is shown in Figure 3.4. Each triple consists of an associated call stack ID, the actual splits within the current function call and potential splits that might occur in future iterations, further called trackers. Before each conditional or indirect jump within the execution trace, we insert temporary branch split nodes into the directed acyclic word graph, which contains all observed jumps at this point. These temporary nodes simplify the management of splits and merges within the implementation. However, we are required to remove them before the analysis since they would break backwards compatibility. If we visit a branch split node in an insertion, we add this node to the list of trackers in the current call stack triple. If this branch split node contains more than a single jump after adding the jump entry of the current trace, we also add this branch split node to the split list in the triple with its position in the tracker list. This index is later used to clean the tracker list efficiently if branches are merged successfully.

During construction, we use the tracker list from the current function call to store possible merge points for the following iterations. Merge points for a trace entry consist of a pointer to its parent node and the index it is stored within its parent's sequence. We use this information to know at which position to split the parent node if we merge into this merge point. Merge points are registered if the algorithm does not find a matching trace entry within the current position in the graph. These points are associated with all branch split nodes inside the current tracker list and are efficiently stored within a hash map.

3.4.2 Merge process

During the construction of the graph, merges might happen if we do not find a matching trace entry at the current position. If this is the case, the algorithm checks whether a matching entry is found in the map of potential merge points. If it is not, construction continues as usual. Otherwise, we merge the current branch into the found merge point. To achieve this, we split the node of the found trace entry at its position in the sequence using the information provided from the hash map that stores potential merge points. The node that begins with the found merge point is added as a successor to the current node. After that, we resume the sequence's insertion at the found merge point.

(a) Before insertion of **B** in **5**

(b) After insertion of **B** in **5**

Figure 3.5: Node splitting within merge processes

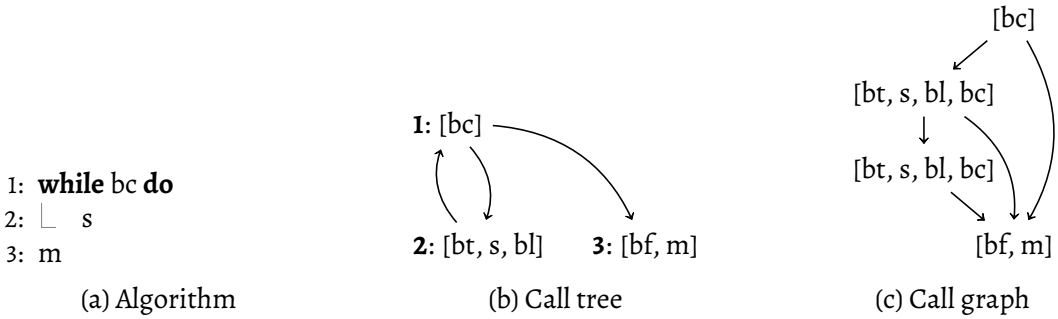
An example of the merge process is shown in Figure 3.5. It is assumed that nodes 2 and 5 have the equivalent parent node 1 and the split was caused by different conditional branches. If we insert a trace entry similar to *B* into 5, we will find *B* in the map storing potential merge points. We then split node 2 at the second position and add the newly created node 6 as successor to node 5. By performing this operation, we merged the branches represented by node 2 and 5. Finally, we can continue adding the current sequence into the graph in node 6.

However, since we merged branches, we need to clean the stack keeping track of splits within the graph during each insertion. For that, the algorithm checks how many branch splits could be merged into the found merge point by using the split list inside the triple. This is achieved by finding the last branch split node in the list associated with the found merge point. We then use its position within the list and the associated tracker index to remove all branch splits from the split and the tracker list beginning at the respective index. This way, it is ensured that the hierarchy of branch splits within the execution traces is handled correctly.

3.4.3 Preventing cycles

Nevertheless, cycles would still be a problem that could arise in the merge process. An example of this problem is shown in Figure 3.6. Figure 3.6b shows the cyclic merge from node 2 into 1, which is also a predecessor of 2. These merges might happen if a merge point is chosen that was already visited by insertions of previous sequences that also visited the current node. This means there is a path between the found merge point and the current node, which causes cycles if merges are performed.

Fortunately, it is easy to prevent this problem since we already store which nodes were visited when adding different execution traces. We can use this information by calculating the intersection of the execution traces that visited the merge point and the current node.



(a) Algorithm (b) Call tree (c) Call graph

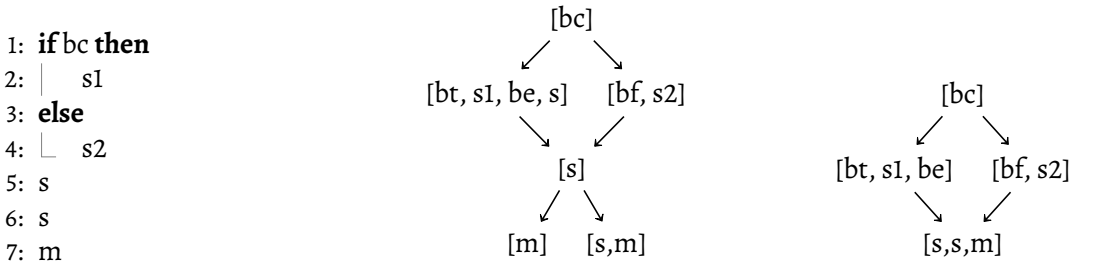
Figure 3.6: Example for cyclic merges within the call graph

Legend: bc: branch condition, bt: jump to the loop, bl: jump to the start of the loop, bf: jump to the end of the loop, s: statement inside the loop, m: rest of the algorithm

If the intersection is not empty, there is a path within the graph from the merge point to the current node. In that case, we would have to create new trace entries and ignore the merge point instead, which produces a graph like the one shown in Figure 3.6c.

3.4.4 Alignment with similar instructions

Since we require multiple memory addresses to be stored within the same trace entry, we cannot determine possible merge points by comparing trace entries. Instead, we merge splits by checking the similarity of a trace entry to its potential merge point. Similarity is based on custom hash codes and comparators, which use different properties of the trace entries to determine whether a pair of entries is similar. Most of the implemented hash codes compare all properties present. However, in the case of memory accesses, we omit the target addresses from this hash code to match merge points with different target addresses. A custom hash function has the advantage of being able to use highly optimized built-in data structures and optimizing access times within the hash map. These implementations also handle hash collisions using the specified comparators in our implementation. Since the algorithm merges based on similarity, we have to take care of later instructions that might overwrite the correct merge point. Moreover, this is also the case if instructions are executed multiple numbers of times.



(a) Algorithm (b) Merging into the second s (c) Merging into the first s

Figure 3.7: Example for incorrect similarity merges

Legend: bc: branch condition, bt: jump to if block, bf: jump to else block, be: jump to skip else block, s1: statement in if branch, s2: statement in else branch, s: statement after the conditional block, m: rest of the algorithm

Figure 3.7 shows an example that causes issues if we do not handle this problem. Figure 3.7a presents an algorithm that executes conditional instructions before continuing with the executions of s and the rest of the algorithm m. It is assumed instructions s are deemed similar by the algorithm, thus producing the same similarity hash code. During the construction of the first branch, the first and second occurrences of s in the algorithm shown in Figure 3.7a would be registered as possible merge points. However, as both instructions would be deemed similar, the merge point association of the first occurrence would be overwritten by the second. If the second branch is constructed and trying to find merge points, it only discovers the second instruction and merges the branches at this entry as shown in Figure 3.7b. However, since the merge point represents the second instruction followed by m, the algorithm splits this node another time since it cannot find a matching trace entry s. This graph, though, does not represent the control flow in the algorithm. The correct merge should happen in the first instruction of s, so the algorithm constructs a graph like the one shown in Figure 3.7c. Only the first discovered merge point for each similar trace entry should be stored in the hash map to prevent such behavior. If we follow this approach, we merge branches into the first occurrence of each trace entry, which is the desired behavior.

3.5 Verification

We constantly checked the algorithm against simple examples during the implementation process. These samples can easily be verified without delving deep into real-world libraries, as they produce complex call graphs that are difficult to check for correctness. Figure 3.8 shows sample cases in which Microwalk's new call graph construction provides advantages compared to the old one. All examples specifically include leakages originating

Scenario	Tree	Graph
(b) if-case	4	4, 8
(c) if-else-case	4	4, 11
(d) for-case	5	5, 12
(e) nested-case	4, 7	4, 7, 13

(a) Found leakages identified by their line numbers using the call tree and call graph

```
1  uint16_t ifc(uint8_t input) {
2    uint16_t c = 0;
3    int a = 0;
4    if (input % 2 == 0) {
5      a++;
6    }
7    c |= array[1];
8    c |= cases[a] << 8;
9    return c;
10 }
```

(b) if-case

```
1  uint16_t ifelsec(uint8_t input) {
2    uint16_t c = 0;
3    int a = 0;
4    if (input % 2 == 0) {
5      c |= array[0];
6      a++;
7    } else {
8      c |= array[1];
9    }
10   c |= array[2];
11   c |= cases[a] << 8;
12   return c;
13 }
```

(c) if-else-case

```
1  uint16_t forc(uint8_t input) {
2    uint16_t c = 0;
3    int a = 0;
4    int bound = input >> 6;
5    for(uint8_t i = 0;
6        i < bound;
7        i++) {
8      c += indexAccess[i];
9      a++;
10   }
11   c |= array[0];
12   c |= cases[a] << 8;
13   return c;
14 }
```

(d) for-case

```
1  uint16_t nestedc(uint8_t input) {
2     uint16_t c = 0;
3     int a = 0;
4     if (input % 2 == 0) {
5        c |= array[0];
6        a++;
7        if (input >> 7 % 2) {
8           c |= array[1];
9           a++;
10       }
11    }
12    c |= array[2];
13    c |= cases[a] << 8;
14    return c;
15 }
```

(e) nested-case

Figure 3.8: Leakage analysis verification using Microwalk's call graph construction

within conditional branches to check whether our algorithm facilitates the analysis of these leakages. All executions that follow the same execution path yield the same memory accesses. These leakages cannot be found by Microwalk's previous radix tree construction approach, because they are covered by the control flow leakage above. However, analysis results shown in Figure 3.8a show that our new approach to constructing a directed acyclic word graph detects those leakages.

3.6 Evaluation

We expect our new algorithm to find at least as many leakages as the old one without introducing new false positives since the old radix tree is a special case of our graph approach. If no conditional branches are merged during construction, the same call tree as the old one is constructed. In all other cases, conditional branches have been merged, that combine previously separated execution paths. Due to the increased number of test cases in these merged execution paths, the new algorithm might be able to detect differences if they are not detected with the original approach as shown in Figure 3.8. If they are already detected, we use additional information about the test cases that visited this node to estimate the amount of the leaked information more accurately.

Table 3.9 compares both approaches using the same traces for mbedTLS's RSA signing with a 2048-bit key and WolfSSL's ECDSA signing with secp192r1. This benchmark was performed with eight test cases on a Linux Ubuntu 22.04.3 with an Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz and 8GiB system memory @ 2667 MHz. All libraries were built using their preferred build tool and compiled using gcc v11.4.0. Microwalk was built in Release configuration using .NET v8.0.102 and MSBuild v17.8.5.

As shown in Table 3.9a, the new algorithm provides an advantage over the previous construction of the call tree since it reduces the number of independent execution paths. The new approach finds all the leakages previously detected by the old one. Moreover, these results prove that Microwalk's previous analyses might have missed leakages that were found using the new approach. The benchmark of mbedTLS also shows that only a few merges within the call graph might significantly improve the number of potential leakages.

The benchmark also includes the construction times of each approach as presented in Table 3.9b. One remarkable discrepancy is the difference between the construction times for the first trace in both approaches. The overhead is caused by the memorization described in Section 3.4.1. If branch trace entries are encountered during the construction of the first trace, each branch is added to the tracker list until the call stack is dropped

Target	Number of merges	Call tree leakages	Call graph leakages
wolfSSL ECDSA	8975	11	35
mbedTLS RSA	15	3	7

(a) Comparison of Microwalk's analysis results using its trace representations

Target	Iteration								Post	Overall
	1	2	3	4	5	6	7	8		
wolfSSL ECDSA	1417	837	557	750	677	636	544	493		5914
	28756	1080	883	1071	880	819	729	639	382	35242
mbedTLS RSA	22	22	5	5	4	7	1	1		72
	75	8	7	5	6	9	1	1	1	117

(b) Time comparison between Microwalk's call tree and call graph construction for each iteration, their post-processing duration and the overall construction times in ms

The bottom time in each cell corresponds to the analysis using the call graph as presented in Chapter 3

Table 3.9: Benchmark of selected libraries

at the end of its function. Each trace entry constructed afterwards is inserted into the map, which stores possible merge points. As every trace entry is constructed in the first iteration, this produces high overhead since plenty of entries have to be inserted into the map, which eventually has to be resized. Additionally, the number of insertions heavily depends on the depth of branches within the code bases since it causes the tracker list to grow, which multiplies the number of insertions into the map. Therefore, it is currently impossible to analyze complex libraries efficiently with high branch depths. However, it might be possible to use further information about an instruction's basic block provided by dynamic instrumentation tools like Intel Pin to improve the algorithm's performance by keeping track of less potential merge points. Such optimization techniques could be relevant in future work to improve Microwalk's analysis performance.

As described in Section 3.4, our new approach introduces temporary branch split nodes before conditional or indirect branches. These temporary nodes are removed from the graph in a post-processing step. As shown in Table 3.9b, this post-processing step takes additional time. Nevertheless, this cost is negligible to the cost of constructing the graph itself. It might also be possible to remove these branch split nodes entirely so that the necessity of the post-processing step will be inapplicable.

We conclude that the new graph construction algorithm provides a significant advantage over the previous one, as shown in this evaluation. However, optimizations are required to improve its performance in analyzing complex libraries. Furthermore, potential leakages that were found should be verified in future work to notify the developers of the libraries in case these leakages are critical.

4

Statistical leakage test

In this chapter, we discuss the implementation of statistical tests to find partially randomized leakages. This chapter also verifies the effectiveness of this approach in finding leakages within randomized implementations that use blinding to prevent side-channel attacks by analyzing various sample targets. Finally, we present the advantages and limitations of this approach by using the results of the analysis.

4.1 Idea

The side-channel leakage model for implementations without randomness is straightforward, as any deviation within execution traces might lead to leakages. However, it is not as evident in the context of randomized implementations. Randomness might cause deviations that are not related to the input values which would result in false positive results. Tools for dynamic side-channel analysis have difficulties determining whether leakages are secret-dependent in randomized implementations. This is caused by the fact that those tools can only test a few inputs and have to remove randomness from the results by using statistical methods.

The reason above explains why automated side-channel leakage analysis tools often do not support randomness in their analyses. However, DATA has shown that the analysis of randomized implementations is possible using statistical tests (Weiser et al., 2018). The approach from DATA is to consider leakages for single instructions and compare their distributions of accessed addresses and the number of occurrences within execution traces for fixed and random inputs. An attacker might be able to infer information about an input value if it is possible to distinguish between the leakages of a single input and the leakages caused by different inputs. As shown in further research, DATA can analyze

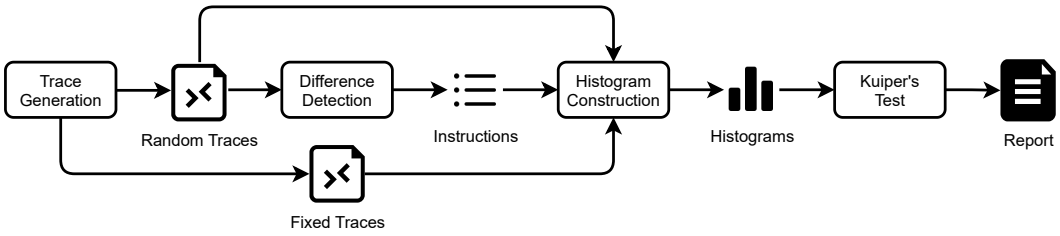


Figure 4.1: Leakage test steps

randomized implementations and correctly determine leakages within them (Weiser et al., 2020).

As Microwalk aims to be a general-purpose tool for microarchitectural side-channel leakages, we adopt DATA's approach to detecting randomized leakages. More specifically, we implement DATA's leakage test within an additional Microwalk analysis module to discover secret-dependent leakages that are randomized. We refrain from implementing DATA's quantification since it requires the manual development of a leakage model specific to the analysis target.

4.2 Leakage test

DATA's approach to detecting secret-dependent leakages in randomized applications consists of multiple steps (Weiser et al., 2018). These steps are visualized in Figure 4.1. Firstly, we generate traces for the two input sets described in Section 4.1. In the next step, we use the set of random inputs to detect differences between these execution traces, which are handed to the histogram construction. This step uses the information from the traces to construct multiple histograms for each instruction that was found to cause differences in the difference detection step. The histogram construction creates histograms for each input set that are passed on to Kuiper's test. This statistical test finally tests whether the distributions described by the histograms are significantly different so that an attacker can infer information about the inputs. The following sections describe the implementation of these steps into Microwalk in more detail and discuss some minor adaptations to DATA.

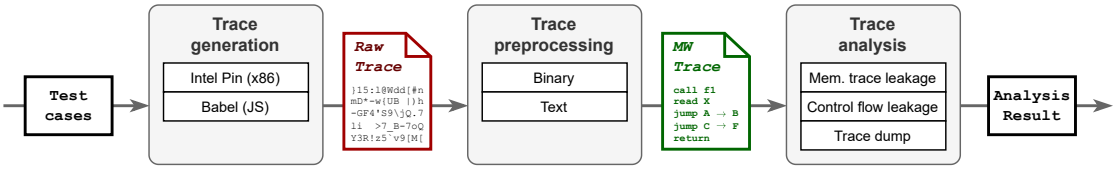


Figure 4.2: Microwalk Pipeline (Wichelmann, 2022)

4.2.1 Trace generation

As mentioned in Section 4.1, the approach is based on generating traces for random inputs and comparing them with traces for a single fixed input. However, due to its pipeline approach, Microwalk does not yet support tracing inputs multiple times without manually copying the input file a few times.

As mentioned in Section 2.5 and shown in Figure 4.2, Microwalk's pipeline starts with loading test cases (Wichelmann et al., 2018). Each test case produces a trace entity exchanged between different stages and stores intermediate results. A test case ID uniquely identifies these trace entities.

The idea to support multiple traces per test case is to create more trace entities per test case within the pipeline. For this, a secondary ID is introduced, uniquely identifying the repetition of a single test case. This way, trace entities from the same test case can be identified by their secondary ID and grouped by their test case ID. As zero is the default value for secondary IDs in scenarios without repetitions, old test environments are still compatible with this approach. This is achieved by omitting the default secondary ID value from the trace file naming convention. Backwards compatibility within the trace modules is facilitated by providing multiple entry points for dynamic instrumentation to pass on secondary IDs to the tracer. These traces are then used within the leakage test module depending on their assigned IDs.

This contradicts the approach from DATA since we already record traces for both input sets during the initial trace stage (Weiser et al., 2018). DATA's approach intends to record traces at different times during its analysis to minimize the number of instructions observed during the instrumentation. However, this approach is impractical for Microwalk's current architecture, as it would result in circular dependencies and require bidirectional communication between pipeline stages. To implement DATA's approach, we would have to extend the pipeline, which would increase Microwalk's configuration complexity and, therefore, would decrease the analysts' user experience. Nevertheless, adjustments to the infrastructure to optimize the trace collection for the leakage test could be interesting to explore further.

4.2.2 Difference detection

At this stage, we aim to detect potentially leaking instructions to consider within the statistical test. We detect differences by traversing the call graph constructed by the randomized data set as described in Chapter 3. During the traversal of the call graph, we record instructions that caused splits within the call graph and memory accesses with multiple target addresses.

As already described in Section 4.2.1, this contrasts with DATA's approach, as DATA uses these differences to instrument the target library again to collect shorter traces for the input sets.

4.2.3 Construction of histograms

DATA's approach to detecting leakages in randomized applications includes the construction of histograms for different input sets, which are used to estimate the distributions of accessed memory addresses and the number of leakages within test cases (Weiser et al., 2018). In this step, Microwalk requires the information from the previous steps about potentially leaking instructions, as histograms are only constructed for these instructions.

We then search for these instructions within execution traces and collect information to build the following histograms for each potentially leaking instruction:

- The number of accesses for each target address
- The number of leakages in a single test case at this particular instruction
- The number of accesses for each target address sorted by its access count
- The number of unique target addresses in a single test at this particular instruction

Note that the original publication of DATA at USENIX Security 2018 only included the generation of the first two histograms (Weiser et al., 2018). After the publication, they added the last two histograms to improve their leakage test further (Fraunhofer AISEC, 2018). This is the reason why we decided to implement them as well.

We also align corresponding histograms from both input sets to prepare the collected data for the following steps. This is required as both input sets might have different accessed memory addresses and we would like to compare the same corresponding address within the statistical test. This means that keys that do not exist in the other histogram are initialized with zero to facilitate the comparison of values with the same key. The algorithm also collects the number of samples used to construct the histograms to facilitate calculations performed within Kuiper's test.

4.2.4 Kuiper's test

In the next step, these histograms are compared for similarity based on Kuiper's test. The Kuiper's statistical test can be used to verify whether two data samples originate from the same unknown base distribution (Kuiper, 1960). While the Kuiper's test is closely related to the Kolmogorov-Smirnov test, Weiser et al. argue that it yields better results in scenarios where no assumptions about distributions can be made (Weiser et al., 2018).

Kuiper's test first calculates the empirical distribution function from the aligned histograms constructed for the fixed and random input sets, called F_f and F_r. It then calculates the Kuiper's statistic

$$V = \sup_x [F_r(x) - F_f(x)] - \inf_x [F_r(x) - F_f(x)].$$

Afterwards, it compares V to the significance threshold V_{st} for a given significance level α using the number of samples n_r and n_f collected in the previous step, which is defined as

$$V_{st} = \frac{Q_{st}^{-1}(1-\alpha)}{C_{st}(n_r, n_f)}.$$

Q_{st} and its inverse, which associates the test with a significance threshold, are calculated approximately using the following equation (Lanzante, 2021)

$$Q_{st}(\lambda) = 2\sum_{i=1}^{\infty}(4i^2\lambda^2 - 1)e^{-2i\lambda^2}.$$

C_{st} links the test to the number of samples in the histograms to the test and can be estimated (Stephens, 1970) with

$$N(n_r, n_f) = \sqrt{\frac{n_r \cdot n_f}{n_r + n_f}} \qquad \text{and} \qquad C_{st}(n_r, n_f) = N(n_r, n_f) + 0.155 + \frac{0.24}{N(n_r, n_f)}.$$

If V exceeds the significance threshold V_{st}, the distributions of the histograms can be assumed to be different. If this is the case for any histogram provided in Section 4.2.3, an attacker might be able to differentiate between leakages of fixed and random inputs and inevitably learn something about secret inputs.

This test produces false positives with a probability of $1 - \alpha$ (Weiser et al., 2018), which defaults to 0.01% within Microwalk's leakage tests. However, false negatives can occur if the constructed histograms do not sufficiently estimate the base distribution of leakages at a particular instruction (Weiser et al., 2018). This might be the case if the sample sizes of histograms are too low, which can be fixed by increasing the number of test cases within the analysis.

Finally, the results from Kuiper's test could be aggregated with results from other Microwalk modules to omit instructions that do not leak any information.

4.3 Evaluation

This section applies the statistical leakage test to multiple specifically constructed targets with different leakages. This approach has been chosen to verify the effectiveness of the test. It is explicitly not evaluated against real-world libraries, as verification of found results would be beyond the scope of this thesis. However, an evaluation would be interesting for future studies.

This evaluation considers four targets for memory access leakages and control flow leakages in its analysis. All of them perform the same calculations but leak different values. Each target has a set of 32 random inputs, out of which the first one is traced 32 times to form the fixed input set.

1. **Leakage**: Leaks the input value directly
2. **Blinding**: Applies blinding correctly and only leaks the blinded value
3. **Mistake**: Applies blinding incorrectly and leaks parts of the input value through the blinded value
4. **Mask**: Applies blinding correctly and leaks the mask used for blinding as well as the blinded value

Figure 4.3 and Figure 4.4 show the leakage analysis of these targets for memory access leakages and control flow leakages, respectively. The tables present found leakages related to the source code below and list Kuiper's statistics V for some of the histograms described in Section 4.2.3. Figure 4.3a show the relevant Kuiper's statistics for the histograms of accessed addresses in their sorted and unsorted form. Figure 4.4a lists Kuiper's statistic for the number of executions at a particular instruction in a single test case. Kuiper's statistics that exceed the threshold, which means that the histograms are significantly different, are underlined and marked in bold. These values would cause the leakages to be reported, as it has to be assumed that they are secret-dependent.

Target	Line number	Accessed addresses	Sorted accessed addresses	Threshold
Leakage	9	**0.9375**	**0.9375**	0.5902
Blinding	15	0.3125	0.375	0.5902
Mistake	22	0.53125	**0.625**	0.5902
Mask	29	0.1875	0.4375	0.5902
	30	0.3125	0.375	0.5902

(a) Statistical memory leakage results

```
uint8_t lookup[32];

void init(void) {
    for (int i = 0; i < sizeof(lookup); ++i)
        lookup[i] = i;
}

uint8_t leakage(uint8_t input) {
    return lookup[input % 32];
}

uint8_t blinding(uint8_t input) {
    input = input % 32;
    uint8_t randomness = rand() % 32;
    uint8_t masked_output = lookup[input ^ randomness];
    return masked_output ^ randomness;
}

uint8_t mistake(uint8_t input) {
    input = input % 32;
    uint8_t randomness = rand() % 16;
    uint8_t masked_output = lookup[input ^ randomness];
    return masked_output ^ randomness;
}

uint8_t mask(uint8_t input) {
    input = input % 32;
    uint8_t index = rand() % 32;
    uint8_t randomness = lookup[index];
    uint8_t masked_output = lookup[input ^ randomness];
    return masked_output ^ randomness;
}
```

(b) Memory target source code

Figure 4.3: Memory access leakages in randomized settings

Target	Line number	Number of accesses	Threshold
Leakage	3	**0.9375**	0.5902
Blinding	12	0.1875	0.5902
Mistake	21	**0.8125**	0.5902
Mask	30	0.25	0.5902
	33	0.375	0.5902

(a) Statistical control flow leakage results

```
1  uint8_t leakage(uint8_t input) {
2      uint8_t output = 0;
3      for (int i = 0; i < input; i++)
4          output++;
5      return output;
6  }
7
8  uint8_t blinding(uint8_t input) {
9      uint8_t randomness = rand();
10     uint8_t masked = input ^ randomness;
11     uint8_t masked_output = 0;
12     for (int i = 0; i < masked; i++)
13         masked_output++;
14     return masked_output ^ randomness;
15 }
16
17 uint8_t mistake(uint8_t input) {
18     uint8_t randomness = rand() % 64;
19     uint8_t masked = input ^ randomness;
20     uint8_t masked_output = 0;
21     for (int i = 0; i < masked; i++)
22         masked_output++;
23     uint8_t output = masked_output ^ randomness;
24     return masked_output ^ randomness;
25 }
26
27 uint8_t mask(uint8_t input) {
28     uint8_t randomness = rand();
29     uint8_t masked = input ^ randomness;
30     for (int i = 0; i < randomness; i++)
31         masked++;
32     uint8_t masked_output = 0;
33     for (int i = 0; i < masked; i++)
34         masked_output++;
35     return masked_output ^ randomness;
36 }
```

(b) Control flow target source code

Figure 4.4: Control flow leakages in randomized settings

```
Require: input
1: mask ← rand()
2: unmask ← reverse[mask]
3: masked ← lookup[input ⊕ mask]
4: output ← masked ⊕ unmask
```

Figure 4.5: Leakages of the mask and the masked value

Direct leakage of the input value leads to significant differences between the histograms of the two input sets. Each trace in the fixed input set accesses an address for the same number of times. This number varies within the random input set and causes different probability distributions. The same behavior can be observed in the third scenario. Since these targets implement blinding incorrectly and leak a single bit of data, the number of accessed addresses varies more in the random input set. The observed difference is deemed significant within both scenarios.

However, the two other scenarios implementing blinding correctly also find differences in the histograms, but the differences are not significant enough. This is true for the second scenario, where only the masked value leaks. Since it is impossible to differentiate between the mask and the input value within the leaked information, an attacker cannot infer any information about the secret input. The fourth scenario is examined more broadly in the following section.

4.3.1 Limitations

Due to the analysis approach applied to each potentially leaking instruction independently, Microwalk and DATA cannot find any leakages composed of multiple instructions. They consider each of these instructions harmless if they are analyzed individually. However, it might be possible that an attacker might infer secret data by combining the information from them.

An example of this is shown in Figure 4.5, which represents scenario four in the previous section. Assuming the data lookups in lines two and three cause leakages about the mask and the masked value, an attacker could use this information to recover the unmasked value. For example, if both leakages contain the mask m and masked value v, the attacker can reconstruct the original value by calculating $m \oplus v$.

Unfortunately, it would be crucial to catch those cases during the analysis of randomized implementations since this case covers correct blinding implementations that leak information. For example, Microwalk would miss that any leakages of the mask besides leakages of the masked value in the same execution are critical. Microwalk would not

report any vulnerabilities in the implementation since each leakage is not vulnerable independently. These leakages might be detectable by performing the leakage test for all subsets of potentially leaking instructions. Due to the number of $2^n - 1$ non-empty subsets with n being the number of leaking instructions, this approach is unpractical for real-world scenarios with plenty of potential leaking instructions. It might be possible to catch these kinds of composite leakages by combining all histograms from Section 4.2.3 for all instructions that were assumed not to be leaking information. In scenarios with a lot of leakages, however, it is likely that the number of critical instructions is too low and does not influence histograms in a significant way. Reducing the amount of information in such aggregated histograms and applying the statistical test afterwards might be relevant for future studies.

We conclude that Microwalk's statistical leakage test detects partially randomized leakages in sample scenarios. Although, it misses other leakages that are composed of multiple leaking instructions. However, these observations should be verified on real-world applications in future studies since the amount of data might influence the effectiveness of this test.

5

Conclusion

In this thesis, we have implemented and evaluated two techniques to improve Microwalk's side-channel analysis. Moreover, this thesis also established some open questions that could be investigated further in future work.

Firstly, we developed an approach to realign multiple execution traces within an acyclic directed word graph to improve the accuracy of finding leakages within differential side-channel analysis tools. Since Microwalk did not realign execution traces beyond the scope of control flow, we adapted its previous representation of execution traces to support realigning them. We have shown that this technique improves the analysis of leakages originating from instructions within conditional branches that were previously not found. An evaluation has demonstrated that Microwalk detects more potential leakages within typical cryptographic libraries than it did with the previous approach.

Due to performance limitations within the current implementation, the call graph construction cannot currently be used to analyze complex libraries. However, general optimizations within the algorithm and the usage of additional information from dynamic instrumentation tools during the trace generation could improve the performance of the call graph construction. Using the new call graph construction approach, Microwalk has found multiple potential leakages in real-world libraries that could be subject to future research to verify whether they are critical and report them to the developers if they are. It could also be interesting to compare Microwalk to other differential side-channel analysis tools that rely on the pairwise comparison of execution traces.

Secondly, we adopted DATA's leakage test and implemented it into Microwalk, extending its potential to detect randomized leakages. We have shown in sample scenarios that it detects incorrect blinding implementations. The results from the evaluation also illustrate that Microwalk can statistically filter leakages unrelated to the secret. However,

we have also demonstrated that this approach does not detect all kinds of randomized side channels since it fails to detect composite leakages.

Since we have shown that statistical tests at the instruction level do not find composite leakages, it remains an open question of how differential side-channel analysis tools can detect them efficiently without the manual intervention of analysts. Nevertheless, the current implementation might be able to analyze real-world cryptographic libraries. Therefore, whether Microwalk can detect critical leakages in real-world applications and how it compares to other tools capable of analyzing randomized implementations could be subject to further research.

Bibliography

Aranha, D.F., Novaes, F.R., Takahashi, A., Tibouchi, M., and Yarom, Y. (2020). LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20. Virtual Event, USA: Association for Computing Machinery, pp. 225–242. ISBN: 9781450370899. DOI: 10.1145/3372297.3417268. URL: https://doi.org/10.1145/3372297.3417268.

Chaum, D. (1983). Blind Signatures for Untraceable Payments. In Advances in Cryptology, (Chaum, D., Rivest, R.L., and Sherman, A.T., eds.). Boston, MA: Springer US, pp. 199–203. ISBN: 978-1-4757-0602-4.

Fraunhofer AISEC (Aug. 2018). *DATA*. https://github.com/Fraunhofer-AISEC/DATA. [Online; accessed July 10, 2024].

He, S., Emmi, M., and Ciocarlie, G. (2020). ct-fuzz: Fuzzing for Timing Leaks. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 466–471. DOI: 10.1109/ICST46399.2020.00063.

Hennessy, J.L. and Patterson, D.A. (2011). Computer Architecture, Fifth Edition: A Quantitative Approach. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Chap. 2. ISBN: 012383872X.

İnci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. (2016). Cache Attacks Enable Bulk Key Recovery on the Cloud. In Cryptographic Hardware and Embedded Systems – CHES 2016, (Gierlichs, B. and Poschmann, A.Y., eds.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 368–388. ISBN: 978-3-662-53140-2.

Irazoqui, G., Cong, K., Guo, X., Khattri, H., Kanuparthi, A., Eisenbarth, T., and Sunar, B. (Sept. 2017). Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. English. CoRR *abs/1709.01552*.

Irazoqui, G., Inci, M.S., Eisenbarth, T., and Sunar, B. (2014). Wait a Minute! A fast, Cross-VM Attack on AES. In Research in Attacks, Intrusions and Defenses, (Stavrou, A., Bos, H., and Portokalidis, G., eds.). Cham: Springer International Publishing, pp. 299–319. ISBN: 978-3-319-11379-1.

Kocher, P.C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Advances in Cryptology — CRYPTO '96, (Koblitz, N., ed.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 104–113. ISBN: 978-3-540-68697-2.

Kuiper, N.H. (1960). Tests concerning random points on a circle. In Nederl. Akad. Wetensch. Proc. Ser. A, vol. 63. 1, pp. 38–47.

Langley, A. (Sept. 2010). *ctgrind*. https://github.com/agl/ctgrind. [Online; accessed April 4, 2024].

Lanzante, J.R. (2021). Testing for differences between two distributions in the presence of serial correlation using the Kolmogorov–Smirnov and Kuiper's tests. International Journal of Climatology *41*, 6314–6323. DOI: https://doi.org/10.1002/joc.7196. URL: https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/joc.7196.

Nilizadeh, S., Noller, Y., and Pasareanu, C.S. (2019). DifFuzz: Differential Fuzzing for Side-Channel Analysis. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 176–187. DOI: 10.1109/ICSE.2019.00034.

Osvik, D.A., Shamir, A., and Tromer, E. (2006). Cache Attacks and Countermeasures: The Case of AES. In Topics in Cryptology – CT-RSA 2006, (Pointcheval, D., ed.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–20. ISBN: 978-3-540-32648-9.

Reparaz, O., Balasch, J., and Verbauwhede, I. (2017). Dude, is my code constant time? In Design, Automation Test in Europe Conference Exhibition (DATE), 2017, pp. 1697–1702. DOI: 10.23919/DATE.2017.7927267.

Sieck, F., Berndt, S., Wichelmann, J., and Eisenbarth, T. (2021). Util::Lookup: Exploiting Key Decoding in Cryptographic Libraries. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, pp. 2456–2473. ISBN: 9781450384544. DOI: 10.1145/3460120.3484783. URL: https://doi.org/10.1145/3460120.3484783.

Stephens, M.A. (1970). Use of the Kolmogorov–Smirnov, Cramér–Von Mises and Related Statistics Without Extensive Tables. Journal of the Royal Statistical Society: Series B (Methodological) *32*, 115–122. DOI: https://doi.org/10.1111/j.2517-6161.1970.tb00821.x. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1970.tb00821.x.

Weiser, S., Schrammel, D., Bodner, L., and Spreitzer, R. (Aug. 2020). Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, pp. 1767–1784. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., and Sigl, G. (2018). DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, (Enck, W. and Felt, A.P., eds.). USENIX Association, pp. 603–620. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser.

Wichelmann, J. (Nov. 2022). *Microwalk Pipeline*. https://github.com/microwalk-project/Microwalk/blob/02904d4afc191454dffa8791dbb2431c78d9f293/resources/images/MicrowalkPipeline.drawio. [Online; accessed June 15, 2024].

Wichelmann, J., Moghimi, A., Eisenbarth, T., and Sunar, B. (2018). MicroWalk: A Framework for Finding Side Channels in Binaries. In Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, pp. 161–173. ISBN: 9781450365697. DOI: 10.1145/3274694.3274741. URL: https://doi.org/10.1145/3274694.3274741.

Wichelmann, J., Peredy, C., Sieck, F., Pätschke, A., and Eisenbarth, T. (2023). MAMBO–V: Dynamic Side-Channel Leakage Analysis on RISC–V. Lecture Notes in Computer Science, Springer Nature Switzerland, pp. 3–23. ISBN: 9783031355042. DOI: 10.1007/978-3-031-35504-2_1. URL: http://dx.doi.org/10.1007/978-3-031-35504-2_1.

Wichelmann, J., Sieck, F., Pätschke, A., and Eisenbarth, T. (2022). Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, pp. 2915–2929. ISBN: 9781450394505. DOI: 10.1145/3548606.3560654. URL: https://doi.org/10.1145/3548606.3560654.

Xiao, Y., Li, M., Chen, S., and Zhang, Y. (2017). STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17. Dallas, Texas, USA: Association for Computing Machinery, pp. 859–874. ISBN: 9781450349468. DOI: 10.1145/3133956.3134016. URL: https://doi.org/10.1145/3133956.3134016.

Yarom, Y. and Falkner, K. (Aug. 2014). FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA: USENIX Association, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

Zankl, A., Heyszl, J., and Sigl, G. (2017). Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In Smart Card Research and Advanced Applications, (Lemke-Rust, K. and Tunstall, M., eds.). Cham: Springer International Publishing, pp. 228–244. ISBN: 978-3-319-54669-8.