UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Retrofitting Remote Control-Flow Attestation for ARM TrustZone

Erweiterung von ARM TrustZone um Kontrollfluss Attestierung

Bachelorarbeit

verfasst am
Institut für IT-Sicherheit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Finn Burmester

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

mit Unterstützung von
Jonas Sander, M. Sc.
Thore Tiemann, M. Sc.

Lübeck, den 2. Dezember 2022

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.

_____

Finn Burmester

Abstract

This thesis presents a hardware-assisted control-flow attestation scheme for TrustZone based on the existing debugging capabilities of the processor. Our approach requires no custom hardware design and just reuses components that are already present on most ARM-based SoCs. For monitoring the control-flow of applications in TrustZone and detecting unauthorized control-flow changes we use Coresight tracing. We successfully implemented a prototype that extends OP-TEE, a TrustZone-based trusted execution environment, with capabilities for control-flow attestation. The prototype consists of modules for generating the control-flow graph as a reference for valid control-flows, a measurement-engine based on Coresight and a verifier. Our evaluation shows that the runtime-overhead generally is reasonable for typical applications but could be improved with different debugging hardware.

Zusammenfassung

In dieser Arbeit wird ein hardwaregestütztes Kontrollfluss-Attestierungsverfahren für TrustZone vorgestellt, das auf den vorhandenen Debugging-Fähigkeiten des Prozessors basiert. Unser Ansatz erfordert kein spezielles Hardwaredesign und verwendet lediglich Komponenten, die bereits auf den meisten ARM-basierten SoCs vorhanden sind. Für die Überwachung des Kontrollflusses von Anwendungen in TrustZone und die Erkennung von nicht autorisierten Kontrollflussänderungen verwenden wir Coresight Tracing. Wir haben einen Prototyp implementiert, der OP-TEE, ein TrustZone-basiertes Trusted Execution Environment, um Fähigkeiten zur Kontrollfluss-Attestierung erweitert. Der Prototyp besteht aus Modulen zur Erzeugung des Kontrollflussgraphen als Referenz für gültige Kontrollflüsse, einer auf Coresight basierenden Mess-Engine und einem Verifizierer. Unsere Evaluierung zeigt, dass der Laufzeit-Overhead im Allgemeinen für typische Anwendungen vertretbar ist, aber mit anderer Debugging-Hardware verbessert werden könnte.

Contents

1

Introduction

The rise of cloud computing comes with many benefits, like access to (nearly) unlimited computing resources and cost optimization. But it also introduces new challenges. The operator of the cloud environment has complete control over the resources that they provide to their customers. So how can one make sure, that he does not exploit this power? How can it be assured, that the workload in the cloud is executed as intended? Similar problems exist in edge computing: For example, how can an electricity provider ensure that a firmware update to a deployed smart meter was executed correctly, without interference by a customer, who might be tempted to tamper with the device, to reduce his electricity bill? One solution to this is the isolation of security-critical software components. *Trusted Exectution Environments* (TEEs) like Intel SGX and ARM TrustZone provide just this and are available on many modern devices. By moving software into a TEE, it can not be tampered with anymore by the cloud operator. Additionally, we have static attestation, which allows a device to prove to a remote verifier, that the correct software was loaded in the TEE. But the software in the TEE still has interfaces to the untrusted environment it runs in and may take input parameters from outside. This leaves attack vectors open for runtime attacks, that exploit memory vulnerabilities in a program to manipulate the control flow and achieve arbitrary code execution. Even if the program is assumed to have no software vulnerabilities, it could still be manipulated by advanced side-channel attacks. To solve this, control-flow attestation has been proposed. It allows a device to prove to a remote verifier, that a program was executed as intended. Despite the existence of various control-flow attestation methods for different platforms, they have not yet been adopted by commercial applications due to the significant runtime overhead they incur. Therefore we want to explore the possible benefits of using hardware support to accelerate the attestation and the program tracing in particular, in terms of security and performance. We focus on the attestation of *Trusted Applications* (TAs) running on ARM TrustZone.

1.1 Contributions

For this thesis, we researched on the usage of hardware support for enabling *Control-Flow Attestation* (CFA). In particular, we focused on the ARM platform, which has long been

used mainly for mobile and low-performance devices but lately has gained traction in the cloud environment. We developed a CFA approach for security-critical applications running in TrustZone, so-called TAs. For measuring the control flow of a TA we use the hardware-based debugging framework Coresight [24]. We implemented a prototype on a low-end ARM board to show the feasibility of our approach and evaluated the performance and security of our system. As the software framework running in the TrustZone we use OP-TEE, an open-source TEE [32]. Our prototype consists of components for a full end-to-end CFA workflow:

- The Reference Measurement Generator creates reference measurements, representing a valid control flow for a program.
- The Measurement Engine is integrated into OP-TEE and is responsible for tracing the execution of a target application using Coresight.
- The Verifier compares the execution traces with the reference measurement and ensures the integrity of a program run.

We also discuss possible optimizations and starting points for future work. Other approaches like C-FLAT [1] often rely on binary instrumentation which is purely software-based or require custom hardware designs like LiteHAX [17]. There already are CFA solutions that use Coresight, e.g. LAHEL [6], but to the best of our knowledge, we are the first to investigate the applicability of Coresight-based tracing to protect the integrity of the execution of TAs in TrustZone.

1.2 Organization

We have already given a brief introduction to CFA and its relevance and described our own contributions. In chapter 2 we will discuss related works and introduce some important terms and concepts in more detail, to provide a solid foundation for the rest of this thesis. After that, we will explain our approach in chapter 3 and our concrete implementation in chapter 4. An analysis of the performance and the security guarantees of our solution will be provided in chapter 5. In chapter 6 we will summarize and discuss the results of our work, as well as possible starting points for future work.

2

Preliminaries

This chapter is used to introduce some terminologies, concepts and technologies that are used throughout this thesis. We also discuss related works, to give an overview of general approaches for CFA.

2.1 Related Work

There are already various works in the field of CFA [18][45][17][28][2] in general and also for ARM in particular [1][6][11]. In this chapter, we want to give a brief overview of the state of research and describe some different approaches. We will not only address approaches for CFA but also for *Control Flow Integrity* (CFI), as these two share many similarities. The main difference between them is the time of verification and the placement of the verifier in the system architecture. In CFI, the correct execution of a program is continuously verified and the verifier is located on the same device as the target. Contrary to this, in CFA the verifier is a remote party, that receives a proof about the correct execution of a target program.
C-Flat by Abera et al. is the original work, that proposed CFA [1]. As a reference for the valid control flow of an application, the applications *Control-Flow Graph* (CFG) is used. A CFG is a directed graph, that has basic blocks as nodes and control-flow transfers between these basic blocks as edges. A basic block is a sequence of code, that only has branches at the entry and exit. C-Flat uses software instrumentation to record the execution path of the target application. This means, that the software binary is modified in a way, that for each control-flow relevant instruction, e.g. a branch, the software jumps to a privileged piece of code, that saves the program counter and the branch target and then returns to the application. The measurement is a cryptographic hash, that is successively calculated over the CFG-nodes that were executed. The verifier can then verify the control flow by comparing the hash measurement with the valid reference value in its database. Special attention is needed for the handling of loops: when simply accumulating the whole execution path into one hash, this would lead to an exponentially growing amount of legal measurements, because every possible amount of loop iterations generates a new measurement. This is why loops are treated as subprograms and their execution path and iteration count are handled separately.

After C-Flat, there were many other publications in the field of CFA, which use different methods for attestation, to optimize the performance. One approach is to use hardware-based measurements of the control flow. A downside is, that systems based on this approach are very closely tied to the specific hardware platform and processor architecture and therefore lack flexibility in contrast to pure software solutions. On the other hand, this approach can massively increase the performance of the attestation, because it supersedes the very costly software instrumentation, which makes it worthwhile. LO-FAT [18], Atrium [45] and LiteHAX [17] use a custom *System on a chip* (SoC) design based on the RISC-V architecture and directly interface with the processor's pipeline, to intercept executed instructions. The prototype for LO-FAT is implemented on an FPGA and features a loop encoder for detecting loops, a hash engine for efficient hash calculation and dedicated memory for the execution traces. Atrium [45] is designed very similarly, but also attests the executed instructions themselves, not only the control flow. These approaches all have in common, that they need custom hardware designs. LAHEL [6] is designed for ARM and takes another approach, by using the existing Coresight tracing features. It is implemented as a custom IP core and therefore still requires a custom SoC design, but in contrast to LO-FAT, modifications to the CPU are not needed because of interfacing with Coresight. This makes the implementation easier and avoids possible problems with ARMs license agreement, which may prohibit modifications to the CPU cores. This approach is similar to [11], which builds on a SoC with an integrated FPGA. The programmable logic of the FPGA is used to build a custom hardware extension, that interfaces with Coresight. In [23] measurements are also obtained by using Coresight, but without the requirement for custom hardware designs. Very similar to our approach, on-device trace capturing is used. The main difference is, that this is a CFI monitoring system, whereas our solution provides CFA. Another aspect that differentiates our approach from the other Coresight-based approaches is, that our attestation target is running in a TEE, similar to the model of GuaranTEE [28], which attests applications in Intel SGX. We will examine LiteHAX and DIAT more closely, to provide an example of how a control-flow attestation system can be implemented.

LiteHAX [17] is a hardware-assisted runtime attestation scheme that also includes memory access operations for the attestation. In a one-time offline phase, the verifier generates the CFG by static and dynamic analysis. These are both forms of program analysis, where in dynamic analysis the code is executed or emulated, which is not the case in static analysis. The prototype implementation uses the angr framework [38] for this, which is also used by our implementation. The attestation on the prover side runs continuously and keeps track of control flow events and memory accesses. The information is obtained by a custom hardware module, that intercepts the CPU pipeline. The control-flow events are then encoded in a bitstream, while the data accesses are combined into a hash measurement. In specified time intervals or after reaching a certain length, intermediate attestation reports are sent to the verifier. The verifier keeps track of the previous execution state and continues with the attestation from there. To verify the execution path, the verifier uses context-sensitive control-flow analysis and checks if it matches the CFG. In context-sensitive analysis, the calling context is also considered, which gives better security guarantees. For the data-flow analysis, symbolic execution is used.

DIAT [2] is designed to secure the collaboration between autonomous drones and there-

fore aims to improve the attestation and verification overhead. One core idea is, that the attested software is divided into small modules, of which only the ones that are needed to serve a request are attested. This is called data-flow attestation and is implemented by the data flow monitor. Another important new concept is the usage of multisets and multiset-hashes, to represent the execution flow of the target. A multiset is a set, where elements can be included multiple times. The interesting thing about multiset-hash-functions is, that they can be incrementally calculated by adding new elements. The order in which elements are added does not matter and the same hash value is generated. Instead of storing and sending the whole sequence of control-flow events, each CFG edge is represented only once, together with its multiplicity, so recurring edges do not appear multiple times. This has the advantage, that storing the execution flow requires less memory and the hash can be calculated in parallel to the execution. The verification of the hash is also easier, compared to conventional hashes that are calculated in a nested way. The verifier only needs the CFG of the software. He can then compare the executed edges with the valid CFG edges, information about the number of loop iterations and possibly a verification policy. A disadvantage is, that attacks that only modify the order of CFG edges cannot be detected.

2.2 Trusted Execution Environment

A TEE is an isolated execution environment that is based on a *separation kernel*. The separation kernel can be a combination of hardware and software and enforces a partitioning of a computing platform into a *Rich Exectution Environment* (REE) and a TEE. This provides isolated execution, meaning the OS in the REE cannot tamper with applications running in the TEE, called TAs. The TEE often offers protection of the integrity and confidentiality of the runtime states of TAs and secure storage. It allows the execution of multiple TAs, that are isolated against each other as well as against the TEE itself. The software running in a TEE is not static and can be securely updated. Communication between the secure environment and the normal environment is only possible through defined interfaces that are provided by the TEE. An important feature of a TEE is remote attestation, which means it can prove its trustworthiness to a third party, by using some sort of measurement and a root of trust [34].

Remote Attestation

The parties involved in attestation are the *verifier*, the *attester* and the *target*. The attester wants to convince the verifier of the correctness of certain properties of the target. In our case, this usually means that the intended software binary is running on a certain hardware platform and has not been tampered with. As evidence for its claim, it passes a measurement of the target to the verifier. The measurement is an observation of the software state. An attestation protocol is also needed to ensure the authenticity and freshness of the measurement [13]. Usually, a challenge-response scheme is used. It is important that the measurement tool has access to the state of the target, while the measurement

tool's state is inaccessible to the target, so a corrupt target cannot tamper with the measurement [13]. Typically, remote attestation works like this:

1. A remote party (verifier) wants to initiate a secure communication with a TA (target). It sends a challenge to the client application of the TA.
2. The application passes the challenge to its corresponding TA.
3. The TA requests an attestation report from the attestation enclave (attester). The attestation enclave uses an existing measurement or generates a new one, that it concatenates with the provided challenge. Both are signed with a key that is only accessible from the TEE (attestation key), to prove that the software is running correctly on the intended hardware platform.
4. The TA passes the attestation report back to the client app, which then sends it to the verifier.
5. The verifier verifies the attestation report by checking the signature and the measurement. It then decides if it trusts the TA.

What is included in the measurement, how the measurement tool is implemented and when the measurement takes place depends on the type of remote attestation. A common example for the use of attestation on enclave platforms is the provisioning of an enclave with secrets: the attestation is initiated to prove to the verifier, that it indeed is communicating with the claimed enclave, that is running on genuine TEE-enabled hardware, in an isolated environment [42]. The enclave can then be provisioned with a communication key to establish a secure tunnel, so confidential data can be sent to the enclave [4]. The default way of remote attestation is binary attestation. This is a form of static attestation, where the measurement typically consists of a signed hash of the loaded software binary. There are several ways how and when the measurement can be generated:

- By hashing the binary file when the enclave is loaded, we get the guarantee that the correct binary was loaded and the enclave was initialized to a valid state.
- By hashing the corresponding memory pages (also mutable pages) after loading the enclave, we get the guarantee that the correct binary was loaded and the enclave was initialized to a valid state.
- By hashing the read-only memory pages each time a measurement is requested, we get the guarantee, that the correct binary was loaded and that the loaded code was not manipulated until the time of measurement.

Because of the form of measurement, the guarantees we get from that are not ideal. What we can ensure is, that the enclave code is correct and is indeed running in the intended environment. But this static form of attestation can not detect runtime attacks and thereby does not guarantee the integrity of the execution of the target.

ARM TrustZone

TrustZone is a security feature, that comes with ARM processors and can be used as the hardware base for a TEE platform. Since ARM processors are very common, especially in mobile and embedded devices, security solutions on these devices are often based on

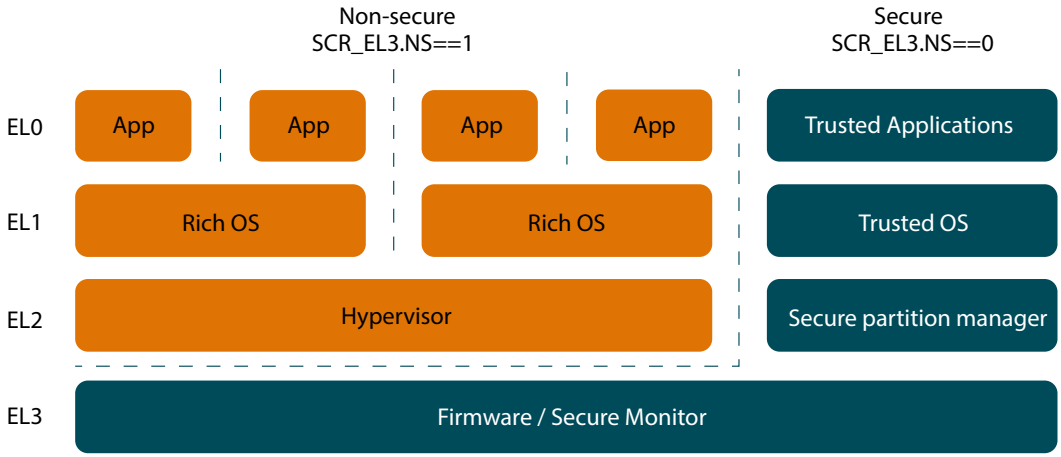


Figure 2.1: Exception Levels in secure and normal world (since ARMv8.4-A); adapted from [25].

TrustZone. Examples of this are Samsung Knox [35] or the Android Keystore [5]. TrustZone provides a framework for implementing secure system architectures by enforcing isolation at the hardware level. The whole system is partitioned into two domains, the normal world and the secure world. Each physical CPU core provides two virtual cores, one for the secure world and one for the normal world. The design also includes the main system bus and the peripherals that are attached to it. System components belong to one of the two worlds, represented by the NS-bit that they have set. NS=1 is the non-secure state, NS=0 the secure state. Non-secure components can only access resources in the normal world. Secure components are able to access resources from both the secure and the normal world. The virtual CPU cores share the physical core in a time-sliced fashion. Depending on the currently active virtual core, the NS-bit is set and determines which system resources it can access. In ARM CPUs, the different levels of privileges that software is run in are called Exception Levels. A higher Exception Level means higher privileges. A visualization of this concept can be seen in Figure 2.1. Typically, there are three Exception Levels (EL0-EL2) in the normal world. The secure world has Exception Levels EL0 and EL1, but no EL2 (since ARMv8.4-A it has EL2). On the highest privilege level EL3 in the secure world runs the Secure Monitor. It offers context-switching between the two worlds and makes communication between the two worlds possible via *Secure Monitor Call* (SMC) instructions. The *Memory management unit* (MMU) is also TrustZone-aware and provides two virtual MMUs. Each world has its own set of translation tables and in general, each physical memory location belongs either to the secure world, the normal world, or is shared between both worlds.
The kind of software environment that is run in the secure world is up to the developer. While it is possible to run only simple software libraries in the secure world, to provide services for normal world applications, it is also possible to run a full OS in the secure world. This has the advantage, that multiple TAs can run concurrently while being isolated by the MMU.
TrustZone itself is not a TEE. The security guarantees it achieves depend on the security

hardware on the SoC and on the software that is deployed in the secure world [10]. Unless TrustZone is considered in the design of the whole SoC, the system may not be secure. For example, *Direct memory access* (DMA) attacks are possible if the device lacks a System MMU or Address Space Controller [39]. By default, TrustZone does not enable Secure Remote Execution, which means the outsourcing of applications to a remote platform with guarantees, that the application is executing as expected [41]. Still, it could be possible to build such a scheme with additional software and hardware. By combining Secure Boot with a per-device-unique secret, that is sealed in dedicated cryptographic hardware. The remote attestation could then be implemented in the secure signed software.

How Does TrustZone Compare to Other System Security Architectures?

Intel Software Guard Extensions (SGX) is a CPU extension that allows the secure and isolated execution of enclaves on Intel processors. Enclaves are isolated against the main OS as well as against each other. This is both implemented on the hardware and microcode level. SGX supports memory encryption and thereby protects the enclave memory pages. It also supports remote attestation of enclaves. For this purpose, SGX CPUs are provisioned with unique device root keys at manufacture time, which establish a trust relationship of each SGX platform with the manufacturer (Intel). CPUs with TrustZone are not provisioned with device root keys, like SGX and do not support remote attestation by default. Furthermore, it is possible to run an entire operating system in TrustZone, while only individual applications are run in SGX.
AMD SEV-SNP is a technology by AMD that allows the isolated execution of virtual machines, which are protected from the hypervisor and also from each other. In contrast to TrustZone, which is part of the CPU design, SEV-SNP uses the AMD security processor, which is an ARM core on the main CPU die to provide its security services. It is based on memory encryption, protects the CPU register states and also ensures the integrity of the main memory. Remote attestation is also a feature of SEV-SNP CPUs. [3].
Unlike SGX and SEV-SNP, TrustZone does not support memory encryption. Instead, the on-chip memory of the SoC is often used for security-critical data, because it is harder to access by hardware attacks (e.g. cold boot). Another conceptual difference is that TrustZone considers peripherals, which is not the case for SGX and SEV-SNP.
RISC-V itself specifies no security architecture like TrustZone. Still, there are software-defined TEE solutions like MultiZone [29] or Keystone [26] that are based on the physical memory protection unit.
Confidential Compute Architecture (CCA) is the new security architecture since ARMv9-A, which was announced in 2021. It features the Realm Management Extension, which introduces a new kind of isolation environment called a realm, that also supports memory encryption. Realms can protect complete virtual machines and containers and also have native support for attestation. To be backward-compatible, realms work along TrustZone [7].

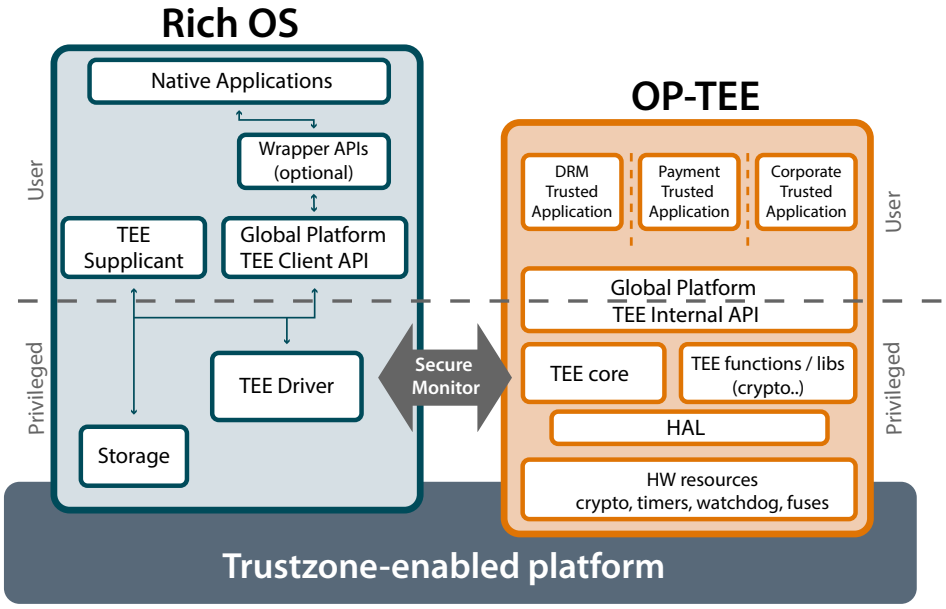


Figure 2.2: Overview of the OP-TEE architecture; adapted from [36].

OP-TEE

OP-TEE [32] is an open-source TEE that is designed for ARM TrustZone and supposed to be run side-by-side with a non-secure Linux Kernel. It is compatible with the GlobalPlatform TEE Client API Specification, which specifies the communication between TAs and client applications on TrustZone-like platforms [20]. Because the code is open-source [1] and the project is still actively maintained, it is often used for research on TrustZone.

The switching of worlds is done using SMC instructions. When a switch from normal world to secure world is requested, a SMC instruction is executed, which causes the processor to enter monitor mode, where the Secure Monitor handles world switching. The Secure Monitor can also be reached from an IRQ or FIQ exception, which are the different types of interrupts on ARM. As the Secure Monitor, OP-TEE uses Trusted Firmware-A, which is a reference implementation by ARM [44]. The OP-TEE OS is the OS that runs in the secure world. The Linux kernel driver for OP-TEE manages the allocation of memory that is shared between the non-secure and secure world and is also responsible for passing SMC parameters from the user-mode client to the TA. An overview of the components of OP-TEE can be found in Figure 2.2. The following API services are available to TAs:

- Trusted Storage API for Data and Keys
- Cryptographic Operations API
- Time API
- Arithmetical API

Code for cryptographic operations runs in kernel mode inside the TEE core. The Crypto API can be extended with other cryptographic services and modified to make use of the

[1] GitHub Repository for OP-TEE OS: https://github.com/OP-TEE/optee_os

cryptographic features of the specific hardware. OP-TEE offers secure storage, meaning that the confidentiality and integrity of the data are guaranteed. The data can either be stored in the file system of the REE or on a Replay Protected Memory Block (RPMB) if the hardware supports it. TAs can be either stored signed and encrypted on the REE file system or they can be linked to the TEE core blob, which allows them to load earlier. TAs run in user mode and have to be single-threaded. OP-TEE also has virtualization support, so that one OP-TEE instance can run TAs from multiple virtual machines (in the REE) in an isolated way. Secure Boot can be used with the authentication framework in Trusted Firmware-A [32]. A feature that was recently introduced to OP-TEE is remote attestation. It is implemented by a *Pseudo Trusted Application* (PTA), which is a concept for adding new functionality to the trusted OS. The PTA offers an interface that can be used by any TA to request a measurement of itself.
OP-TEE is not a real OS, but more of a software library which is why it relies on the OS in the REE for scheduling. TAs offer services to their client applications and are only executed for a specific service request. The interface between the TAs and the client applications consists of commands that can be invoked by the client applications.

2.3 Coresight

Coresight is the architecture for hardware-assisted debugging on ARM systems. It provides means for invasive and non-invasive debugging. While invasive debugging includes techniques such as single-stepping through programs, for this work we are more interested in non-invasive debugging, which only allows observation of code execution and has low performance impact [30].
The basic components of a tracing system can be categorized as sources, links and sinks. The most important types of sources are the *Embedded Trace Macrocell* (ETM) and *Program Trace Macrocell* (PTM). PTMs are found in older systems before ARMv8 and provide only instruction tracing. ETMs provide instruction tracing as well as data tracing, although the availability of data tracing depends on the implementation. Instruction traces include information that is needed for the reconstruction of the execution path of a program, like jump targets or if a branch was taken or not. Data traces include information about data accesses of the processor. A funnel is a link component. It collects traces from multiple sources and combines them into a single trace stream. There are also multiple types of trace sinks. A *Trace Memory Controller* (TMC) is a device, that captures trace data into memory. The different configurations are *Embedded Trace Buffer* (ETB), *Embedded Trace Fifo* (ETF) and *Embedded Trace Router* (ETR). The ETB and ETF both have an SRAM attached, that can have a maximum capacity of 64KB. The ETR can store the trace in system memory, allowing longer tracing. The *Trace Port Interface Unit* (TPIU) routes the trace data out of the chip, so it can be accessed by an external debugger. All tracing components can be configured by software on the target itself, by writing to memory-mapped configuration registers, or from an external debugger. An example of a basic tracing system can be seen in Figure 2.3. The system has two processors, each with multiple cores. Each core has an ETM and the ETMs are merged by a cascade of funnels, that end up in an ETF as a trace sink.

Trace Format

A trace stream contains trace information in a compressed binary format. It can include trace frames from multiple trace sources, that are identified by unique IDs. The frames contain packets, that encode different event types. The most important packet types are:

- P-headers: These encode a sequence of atoms. An E atom represents an instruction with a fulfilled condition code test and a N atom is an instruction that failed its condition codes test. A P-Header has a size of one byte and can encode up to 15 E atoms (with a counter) and one N atom.
- Branch Packets: These are emitted when an indirect branch instruction is executed. It contains the branch target address in a compressed format. Only the address bytes that are different from the last branch packet are explicitly stated. Because of this, the branch packets are often only one byte large, instead of the maximum 8 bytes. Branch packets are also used for indicating exceptions [19].

Together with some other information like synchronization packets, this is enough to reliably reconstruct the execution of a program.

2.4 Runtime Attacks

There are several types of attacks that may be used by an attacker to gain control over a target program and manipulate its control flow for malicious intents. Directly injecting malicious code into an application is not possible anymore on modern systems, since there are memory protection features like data execution prevention, which prevent code execution from the heap or stack. Instead, attackers resort to other techniques, that do not need code injection. A prominent class of attacks are code-reuse attacks. These allow an attacker to reach arbitrary code execution by repurposing existing code. With *Return-oriented programming* (ROP) an attacker exploits a memory vulnerability like a buffer-overflow in a program, to overwrite code-pointers on the stack and divert the control-flow to a sequence of so-called gadgets, which form a new malicious program [37]. These gadgets are short instruction sequences, that may be located anywhere in the application code. They need to end with a return instruction, so they can be chained together by the attacker. A similar type of attack is Jump oriented Programming. This does not require return instructions, but instead works with indirect branches [12]. These attacks work on many processor architectures, including ARM.
Non-control data attacks corrupt program variables to cause unexpected program-flows that can for example lead to privilege escalation. A target could be a variable that controls the entry to a privileged program section.
Data-oriented programming [21] is similar to ROP as it also re-uses existing code and chains instruction sequences together to perform arbitrary computations. However, in contrast to ROP it does not change the control flow. Instead, variables are manipulated so that instructions perform unintended operations that can for example lead to information leakages.

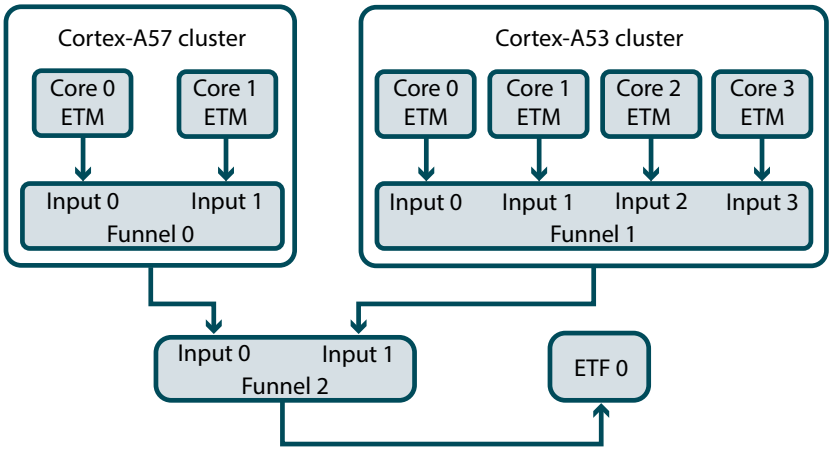


Figure 2.3: Basic Tracing Setup; adapted from [31].

3

Approach

3.1 General Considerations

When it comes to control-flow attestation there are some basic questions, that define the approach:

- How do we generate the reference measurements for defining valid program runs?
- Do we need to keep a database of (a representation of) all valid program runs?
- Do we need additional (custom) hardware for the attestation?
- How is the measurement engine implemented?
- How is the measurement engine isolated from the target?
- How much information do we include in the attestation report?
- How do we balance the load? More complexity for the appraiser or the attester?

These are all important questions, and no matter which approach you choose for each aspect of the attestation system, there will always be some kind of tradeoff. In the next section, we will first define the threat model, that motivates our approach. Then we will give a high-level overview of our CFA approach. After that, we have a closer look at general considerations for the different stages of CFA.

3.2 Threat Model

We want to protect TAs in OP-TEE against a malicious attacker.
The separation provided by OP-TEE and TrustZone is assumed secure. Also, we assume that the prover uses secure boot and only boots OP-TEE with our attestation extension. We assume that the prover and verifier share a symmetric key for authentication therefore, the attestation reports are integrity protected and authenticated. We consider only the secure world as trusted, the REE with Linux is considered untrusted. We assume an attacker that has complete control over the REE and may try to launch runtime attacks using the public interfaces of OP-TEE. For example, he could pass malicious parameters to a TA for a code reuse attack.

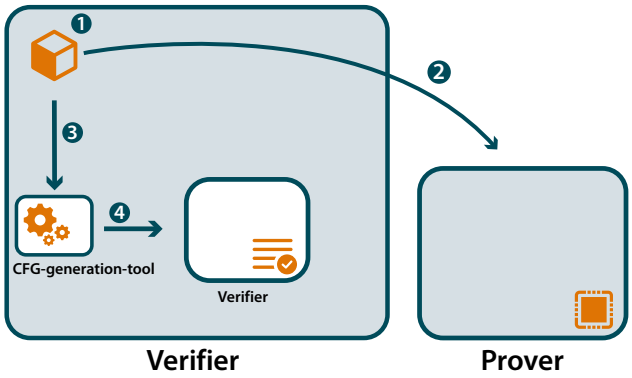


Figure 3.1: Offline pre-processing phase.

3.3 Attestation Process

Our approach works like this: First, we have an offline pre-processing phase, see Figure 3.1. The owner builds a TA that he wants to be attested (1) and deploys it to OP-TEE (2). He configures the prover with the UUID of the new TA that is to be attested. Then he feeds the TA binary to the CFG-generation-tool to create the CFG (3). The CFG is exported and stored (4). When everything is set up, the verifier generates a challenge c to ensure the freshness of the attestation report and transmits it to a client application on the prover device. The client application acts as a proxy and invokes the TA (1), see Figure 3.2. When OP-TEE detects that the UUID of the invoked TA is selected to be attested, it passes the control to the prover, which starts the tracing (2). The TA executes the requested command, and when it returns, the prover stops the attestation and collects the trace t (3). After the execution, the client application passes the challenge to the prover and requests the attestation report, see Figure 3.3. The prover calculates the *Message Authentication Code* (MAC) of the trace and returns the attestation report $r := (t, MAC(t, c))$ (4). The client application can then pass r to the verifier. The verification phase can be seen in Figure 3.3. The verifier receives r and first checks the integrity and authenticity of r by validating the MAC and ensuring that the correct challenge c was used. When it's valid, he loads the trace t. The pre-computed CFG is then used, to verify the control flow measurement. If t does not comply with the CFG, the verifier will report this.

3.4 How to Obtain the Measurements?

The main approaches for collecting information about the control flow of a program are the following.

Instrumentation
The software binary is modified, so that each time an instruction that can modify the control flow is reached, the program jumps to a piece of code, called a trampoline function. This code collects information like the jump target or if a branch was taken or not and

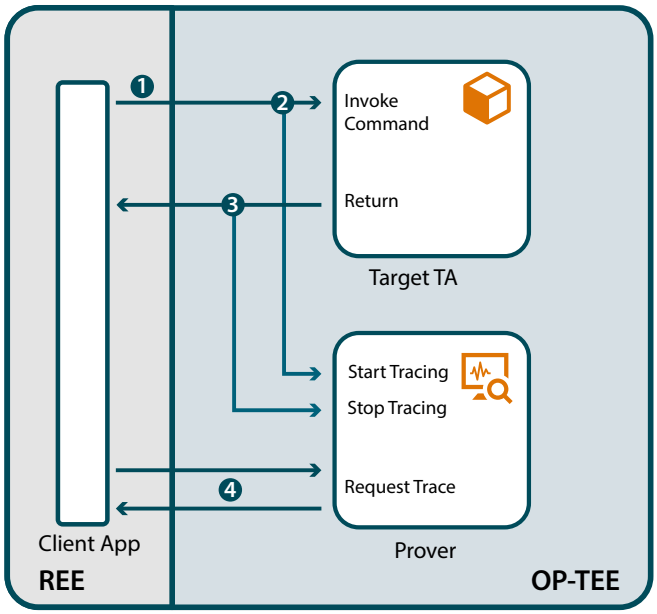


Figure 3.2: Online phase.

adds it to the current measurement. This approach has the disadvantage, that it induces a high overhead on the target because it is solely implemented in software. In normal programs, control-flow modifying instructions like branches and returns occur quite frequently, and each time the additional code of the trampoline function has to be executed. The advantage is, that it can theoretically be used on any hardware, right away.

Hardware-based
Another common approach is to use custom hardware designs. The information about the control flow can for example be obtained by directly interfacing with the instruction pipeline of the processor. The resulting data can then be processed by custom hardware

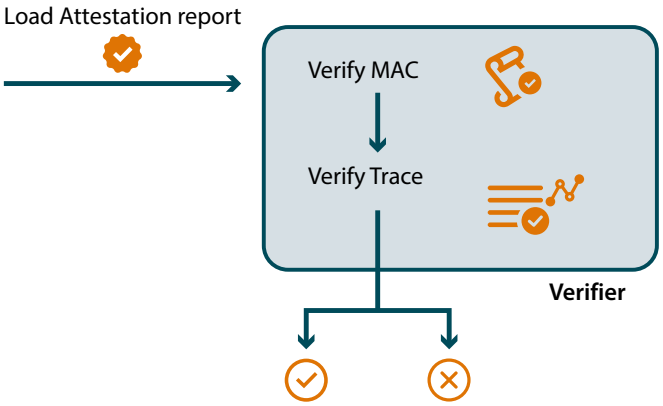


Figure 3.3: Verifying phase.

modules. While reaching better performance, this approach has the disadvantage, that custom hardware is needed to use the attestation.

Having seen these approaches and their disadvantages, the idea was to try an approach that would potentially combine the advantages of both, but without the disadvantages. Modern processors often come with hardware-based debugging and tracing capabilities: Using the tracing we could reliably get information about the program flow, but without inducing additional load on the attestation target. If we could use the tracing without using additional hardware like a debugging device, we would even have a solution that works right away, on any hardware! As the focus of this work was supposed to be on ARM platforms, Coresight was the way to go.

3.5 Verifier

In this section, we focus on our approach to verifying attestation reports.

Concept

To be able to verify an attestation report, we need some kind of reference, that shows how a valid execution of the target program would look like, to compare it to the captured execution trace. We chose to use the CFG of the program as a reference for the verification. A CFG is a directed graph, that has basic blocks as nodes and control-flow transfers between these basic blocks as edges. A basic block is a sequence of code, that only has branches at the entry and exit. The big advantage over other approaches is, that we don't have to generate and keep a huge database of every possible execution path, so the memory requirements for the verifier are reduced. Another advantage is the flexibility we gain when it comes to the time at which the verification takes place. While hash-based approaches can only verify the trace after the complete program execution, our approach allows us to also verify partial execution traces. So instead of verifying the whole program run after the execution, it would be possible to monitor the program either in real-time or in regular intervals, for improved security. Though this is reserved for future work and not implemented yet, our approach would allow implementing this more easily. A disadvantage is that the size of the trace we generate and need to store on the attester device grows proportionally to the execution time of the TA, which can become a problem for a resource-constrained attester. Another disadvantage of the CFG approach is the complexity of precise CFG recovery. Also, depending on the application and the recovery technique, some control-flow paths may be not recovered, leading to false positives (falsely detected control-flow violations). Our goal is to reduce the complexity of verification as much as possible by offloading the main effort to the offline pre-processing phase and the CFG-recovery. The verifier should then simply compare the stored reference with the measurement.

Requirements for the Control-Flow Graph (CFG)

There are two main properties, that define a CFG recovery technique:
Soundness: Each edge in the graph represents a legal control-flow transfer. This is very important to ensure, that only legal program runs pass the verifier. An example for a sound CFG would be an empty graph, though this is obviously not what we want.
Completeness: All legal control flows are represented in the graph. A less complete CFG leads to a higher false positive rate for our verifier. A trivial example for a complete CFG would be a complete directed graph [38].
For CFA both properties are equally important because neither a verifier that constantly reports false control-flow violations nor a verifier that overlooks a runtime attack can be trusted.
We also want our CFG to be context-sensitive. This means, that the calling context is considered in the analysis. If a certain function is called at multiple call sites, it should be represented with multiple nodes in the graph, so that the return edges are correct. If there was only one node in the graph for this function, then it would have multiple outgoing edges, thereby allowing unintended control flows. By using a context-sensitive CFG, we have stricter control-flow enforcement.

Verification

The implementation of our verifier is very simple. We wrote the verifier in C, which gives us the flexibility to transform our CFA approach to a CFI approach, by moving it into OP-TEE. The verifier works like this: First, a reference CFG and attestation report are loaded. The verifier checks if the MAC for the trace is correct and proceeds if it is valid. Using the control-flow information from the attestation report, we navigate through the CFG. The CFG gives us constraints and enforces that only a valid control flow leads to a valid verification result. As the trace contains the information if a conditional branch was taken or not, we know which edge we need to take in the CFG to follow the trace. For each indirect branch, there is a branch packet in the trace, that contains the target address of that branch. We compare this address with the expected target address from the CFG. Because we know where we are in the CFG, we can verify that both source and target of the branch are correct. If the trace address and the expected address do not match, we report a control-flow violation. We also report a control-flow violation, when we find unexpected trace. This could for example be a branch packet, for which there is no edge in the CFG.

3.6 Protecting the Attestation Report

Our attestation protocol uses MAC-based challenge-response authentication. An attestation report consists of a trace and an HMAC-SHA256 that is calculated over the trace and the challenge. The MAC ensures the integrity of the trace and also authenticates the attester. The verifier and the attester share a secret symmetric key, that only they can access. Therefore, only these parties are able to create valid MACs, so no one can tamper with the attestation reports. The reason we use a MAC and not a digital signature is that

it is more efficient to calculate and provides all the security properties that we need: integrity and authentication. As our hardware platform is a low-end embedded system we need to be mindful of our resources. The MAC does not provide non-repudiation, but for our use case, this is not needed.

4

Implementation

After explaining our approach in the last chapter, we would like to go into more detail about the implementation in this chapter. This could be particularly useful for future work.

4.1 Selecting the Hardware

In practice, Coresight tracing turned out to be a little bit more complicated than expected. There are two main factors that determine if you can use Coresight on a certain hardware platform:

- Coresight components on the SoC
- documentation of these components and the corresponding memory addresses

Another factor that we needed to consider, was the compatibility with TrustZone and OP-TEE, as this is where the TAs that we want to attest run. Due to the nature of the ARM ecosystem, not every ARM SoC has all the components that are needed for on-device tracing. ARM designs IP cores, which are building blocks for SoC designs, and licenses these to SoC manufacturers. All the Coresight components are also IP cores, and it's up to the manufacturer, which he wants to license and include in his SoC design. This leads to a very diverse landscape of boards, that provide debugging and tracing capabilities to varying degrees.
We tried several boards, of which only one did work for our use-case. We tested the Raspberry Pi 3B, as this is a well-known single-board computer, that is also very cheap and is supported by OP-TEE. Its Broadcom BCM2837 SoC features a ARM Cortex A53 CPU, which supports Coresight. According to the official documentation for the CPU, it contains an ETM and a CTI [14]. Unfortunately there is no documentation about the debugging capabilities of the Broadcom SoC to be found. Online research and usage of the Coresight hardware discovery tools provided by the Coresight Access Library (CSAL) [15] did also yield no results. As there was no way to discover or configure the Coresight tracing components, it must be assumed that they are not implemented on the SoC, so we did not consider this board.
We also tried using the Tinkerboard 2S. It has a Rockchip RK3399 SoC, that is supported

by OP-TEE and is supposed to support Coresight. The documentation for the SoC states, that it features a "Full Coresight debug solution", so this looked quite promising. After experimenting a bit with *Coresight Access Library* (CSAL), it turned out that the board could be used for Coresight tracing, but only with an external debugger, what is in conflict with our goal of not using additional hardware. The reason for this was, that the SoC does not have a Coresight sink device, which is needed for on-device tracing, what we wanted to use.
The board that we chose was the STM32MP157DK-1, which is a development board by ST-Microelectronics. Although it is a bit dated and does not have very powerful hardware, it is supported by OP-TEE, has all the necessary components for Coresight and is very well documented. It features an ETM for each core, an ETF for trace storage and a *Cross Trigger Interface* (CTI) for advanced trace controlling.

4.2 Software for Coresight Tracing

For on-device tracing there are not that many options in terms of software for controlling the tracing. In Linux there are perf and the sysfs-interface. Both are only available on Linux, because they rely on some kernel functionality, so these were not an option, as we wanted to have the measurement engine in the secure world, where only OPTEE-OS, but no Linux runs. This left us with CSAL, which is developed by ARM and can run on both Linux and bare-metal [15].
The library offers some abstractions and functionality for Coresight and makes it easier to configure the tracing system. To get started with tracing, a custom configuration file, that describes the available tracing hardware and how it is interconnected, was needed by the library. This was relatively easy to create, because the documentation for the board [40] contained all the physical addresses for the memory-mapped configuration registers and also detailed schematics, showing the connection between the tracing components. Before we could trace anything, we also needed to configure the debug authentication interface. This is controlled by the Boot and security and OTP control (BSEC) and can only be set from the secure processor state. On our device there are several options that control the behaviour of the debug system, concerning the TrustZone state. For enabling on-device trace-capturing, we need to set dbgen to enable debug and trace features in general, niden to enable trace and performance monitoring, spniden to enable trace and performance monitoring in secure mode and dbgswenable to enable software access to debug components. The security implications of these settings are discussed in section 4.5.

4.3 First Traces

We conducted the first tracing tests in Linux, to determine its applicability for attestation. For tracing a specific program, trace filters have to be configured in the ETM. These specify conditions on which the ETM starts or stops tracing. First, it can be specified if you want to trace in secure mode, non-secure mode or both. It is also possible to trace

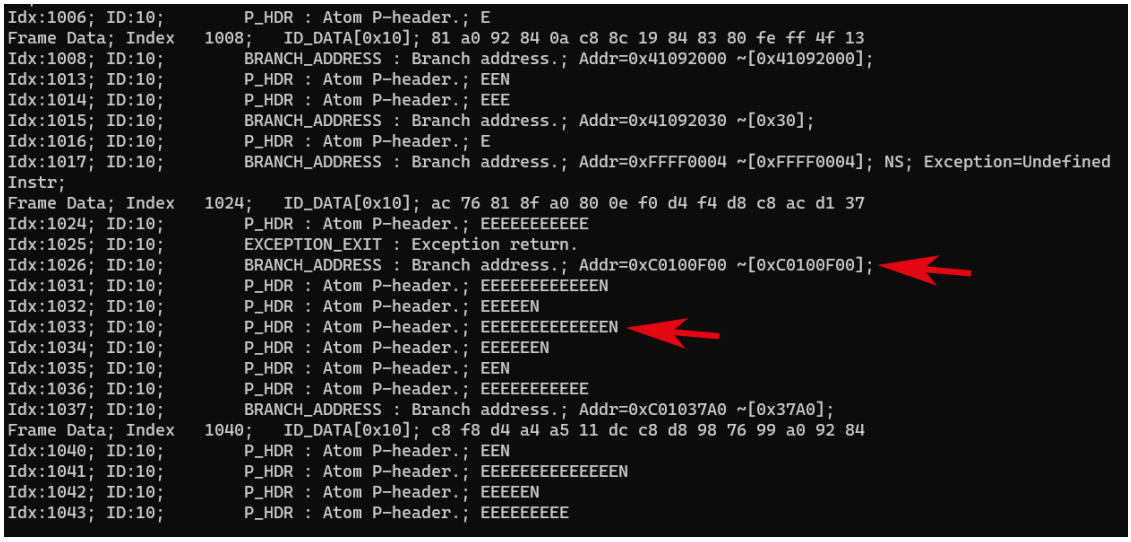


Figure 4.1: Trace packet output

user-mode or kernel-mode only or both. Then there are two general options. First, it is possible to program the ETM to only trace instructions for a specific context id. As OP-TEE has no concept of context ids, and we eventually wanted to trace TAs in OP-TEE, instead we opted for tracing based on the address range. As the address filter works with virtual addresses, the load address of the binary and the offsets of the program regions that are to be traced needed to be determined to configure the tracing. This can be done using objdump. To make sure that no other programs or security features interfere with our tracing, we disabled *Address Space Layout Randomization* (ASLR) and scheduled the target program on a core that we excluded from Linux scheduling. The raw trace that is read from the ETF is in a compressed format and can be decoded in multiple ways. The first way is to only unpack the raw trace packets. This leads to an output like in Figure 4.1, where you can see the atom packets and the branch addresses. This can be done by Open CSD [33], and only the trace file is needed. Another option is to fully decode the trace, so that you get a sequence of the executed instructions. This can also be done using Open CSD or the commercial debugging tool ARM DS-5 [8]. An example output that we produced can be seen in Figure 4.2. You can see the instruction addresses and the actual instructions that were executed. For this stage of decoding, the binary of the traced program needs to be provided to the debugging tool. We found that the packet decode output like in Figure 4.1 is the best fit for our use-case, as it provides all information we need for CFA, omits one stage of decoding and does not require the binary of the traced program. Using the atom packets and the indirect branch addresses (highlighted with an arrow in Figure 4.1) we can reliably recover the control-flow.

Overcoming Coresight Configuration Challenges

Implementing Coresight-based tracing is not trivial. If the synchronization sequence at the beginning of a trace is missing, trace packets get lost which leads to an incomplete trace that is unusable for later verification. The synchronization sequence consists

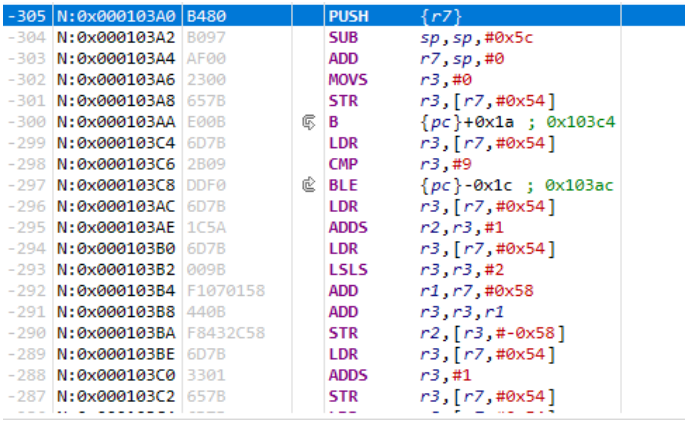


Figure 4.2: Decoded trace in ARM DS-5

of ASYNC and ISYNC packets. On the ST-Board, we encountered an issue that lead to missing ASYNC packets. As troubleshooting approaches proposed by ARM did not solve the issue, we developed a workaround that allows us to get complete traces.

4.4 Tracing OP-TEE

Another problem that we needed to tackle was how to integrate CSAL into OP-TEE. CSAL is also designed for bare-metal and does not have many OS dependencies, which made this simpler. Still, we needed to integrate it into the OP-TEE build-system and do some fine-tuning, especially concerning the access to the memory-mapped registers.

We added a routine to OP-TEE that runs at startup. It adds the register-addresses for the Coresight components to OP-TEEs virtual address space and registers the tracing hardware with CSAL. The main tracing functionality is built into the attestation PTA. This is responsible for controlling the tracing. We modified the code of the OP-TEE kernel, so each time a TA command is invoked, OP-TEE checks if the UUID of the TA is registered for attestation. If this is the case, the attestation PTA is called to start tracing. The ETM is configured to only trace secure world code in user mode. This suffices, as we configured OP-TEE to only run one TA at a time. In the discussion, we will talk about the support for parallel execution of TAs. When the TA returns, the attestation PTA is called another time to stop the tracing and to retrieve the trace from the ETF buffer and load it into secure memory. The client application that invoked the TA can then request the attestation report from the attestation PTA. For our prototype, we disabled ASLR in OP-TEE, because this simplifies the attestation. However, by including information about the memory layout in the attestation report, we could also support ASLR.

Tracing Longer Programs

As the ETF buffer on our board has only 8kB, additional work was needed to trace program runs that produce longer traces. For this, we needed to configure the Coresight CTI, so an interrupt is triggered when the buffer overflows. We also needed to register an interrupt

handler in OP-TEE, that reads the trace to secure memory and resets the ETF buffer. As we have restricted OP-TEE to one core, it is ensured that no trace is lost, because the target application can not run while the interrupt is handled.

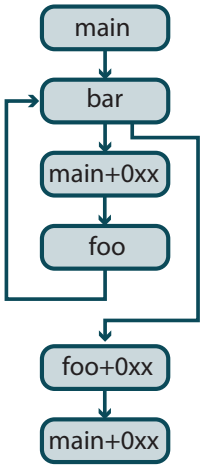
CFG Generation with Angr

For recovering the CFG we chose the Angr framework [38]. It is also used by other CFA approaches, like LiteHax [17]. Angr is a powerful library for binary analysis, that can work with many different architectures, including ARM. This is achieved by lifting the binary code to an intermediate representation, called VEX. A big challenge in CFG recovery are indirect branches. These are branches, where the target is loaded from a register or from memory. In contrast to direct branches, where the target address is encoded in the instruction, the control-flow transfers for these branches can not easily be resolved by static analysis. Angr offers two options for the CFG-generation.
CFGFast uses static analysis but can also resolve some basic indirect branches like those resulting from switch statements and function returns. Basically, the analysis starts from the entry point and builds the graph from there, by adding new edges and nodes for branch targets at the end of a basic block. Due to its static nature, some transitions may not be found by this analysis, because they can only be resolved at runtime. It also lacks context-sensitivity. **CFGEmulated** uses a different approach to recover the CFG. It combines several techniques, to refine the recovered graph step by step, namely forced execution, backwards slicing and symbolic execution. *Forced execution* ensures, that for conditional branches always both possible paths are followed. *Symbolic execution* is used together with a constraint solver, to recover indirect jumps in a dynamic way. *Backward slicing* is a technique that is used to add context to the analysis. If an unresolved indirect jump is in a function, that is called from multiple call sites in the program, the node for the function will be split into multiple nodes and symbolic execution will be leveraged, recovering the target for each node separately. As long as there are unresolved indirect jumps, Angr will try to apply one of the techniques, until there is no change in the resulting CFG anymore. A drawback of CFGEmulated is, that it is significantly slower than CFGFast [38].
We chose to use the CFGEmulated option because we are interested in the best possible completeness and soundness. The time needed for the generation is not a real problem, because the CFG only needs to be computed once, in an offline pre-processing phase.
The context-sensitivity in Angr can be configured by a parameter, specifying the number of callers to keep on the call stack for a *basic block* (BB). We want to have a high value because this results in a more sound graph. An example showing the effect of the context-sensitivity is given in Figure 4.3 for the code in Listing 4.3. With the static CFG generation or a context-sensitivity level of 0, there is no context for the call to the basic block that represents the bar() function. Both main and foo will be valid exits, no matter if the basic block was entered coming from main() or foo(). If we use a higher context-sensitivity level like 1, the calling context will also be considered in the analysis, so if the BB is entered, coming from main, the only exit goes to main. This is why a basic block may be represented by multiple nodes in the graph. The drawback is, that a higher value also results in a larger CFG size. We found that a context-sensitivity higher than 15 does not change

```
void bar(){}

void foo()
{
    bar();
}

void main()
{
    bar();
    foo();
}
```



(1) With context-sensitivity 0

(2) With context-sensitivity 1

Figure 4.3: Comparison of different context-sensitivity levels for CFG analysis in Angr.

the resulting CFG anymore for the programs in our evaluation.

Export

To modularize the CFG-recovery and the verifier, we needed a way to export the CFG. For this, we use the built-in GEXF-export in Angr, though we had to enrich the export with additional information for duplicate nodes (due to context-sensitivity) and for conditional branches. GEXF is an XML format for storing complex graph structures, which is exactly what we want. By using XML, we ensure that the stored graph can be easily processed by our verifier, as there are many XML parsing libraries available.

4.5 How to Make the System Secure

Creating a secure attestation system with TrustZone and Coresight is not trivial. Efforts for creating a secure system can be split into two tasks. First, we describe general considerations for securing OP-TEE [16][32][40]. After this, we also talk about securing the tracing system. For our prototype, we made no efforts to secure OP-TEE, but we still consider it very important and thus want to discuss this process.

Securing OP-TEE

There are various hardware requirements for a secure OP-TEE configuration:

Random Number Generator A random number generator is needed as a secure source for cryptographic key material. The STM32MP157 has a hardware random number generator. It is possible to assign it to the secure world only.

Hardware-unique Key For secure operation a *Hardware unique key* (HUK) is required. This is a key that is unique for each individual board and should only be accessible from the secure world. It is needed to give the device a unique identity and to derive other cryptographic keys from, like for secure storage. The STM32MP157 does not directly come with such a HUK but has the required hardware to implement it. *One-Time-Programmable Memory* (OTP) can be used to store a key, that can only be read by the secure world.

DDR Firewall For extending the TrustZone security state to the DDR memory, our board has a *Trustzone address space controller* (TZC). This device filters accesses to the RAM and can create up to 9 separate regions, that can be either assigned to the secure or normal world.

Device Bus Access Policies To protect other devices on the bus, the STM32MP157 has an *Extended Trustzone protection controller* (ETZPC). This can be used to assign peripherals to the secure or the non-secure world. It is also used to partition ROM and SRAM into secure and non-secure regions.

Secure Boot To ensure that the device is in a trusted state and can only boot our OP-TEE instance in the secure world, secure boot needs to be configured. A private/public key pair needs to be created. A hash of the public key is then provisioned to the device and stored in OTP. The private key is used to sign the firmware images, that should be loaded on the device. The boot chain can be seen in Figure 4.4. The processor starts in secure mode. First, the ROM code loads the first stage boot-loader (FSBL) and authenticates it. The FSBL then authenticates and loads the next stage. In the secure world, this is OP-TEE. Then the normal world is initialized by first loading U-Boot, which in turn loads the Linux OS.

Securing the Tracing System

It is very important that the target application for the tracing has no access to the tracing components. The non-secure world should also have no access. Otherwise, it would be very easy to tamper with the attestation, by pausing the tracing or inserting fake data into the trace buffer. TAs in OP-TEE have by design no direct access to physical memory and are restricted to their own virtual address space. Instead, the tracing is only controlled by our CFA PTA, which runs in kernel mode in the secure world. So our target can not manipulate the Coresight tracing and we are left with securing the tracing hardware from the Non-secure world. Depending on the hardware platform this can be either very easy or not possible at all. The component for securing peripherals on our board, the ETZPC, does not offer the possibility, to restrict access to the tracing components. Therefore it can not be used to build a secure attestation system using Coresight. However, in general, it is possible to secure access to Coresight devices, just like with any other peripheral. For example on the Juno ARM Development Platform SoC, which is a reference design, the NIC-400 (a network interconnect, that connects peripherals to the main system bus) can be used to configure security attributes for almost all memory-mapped peripherals,

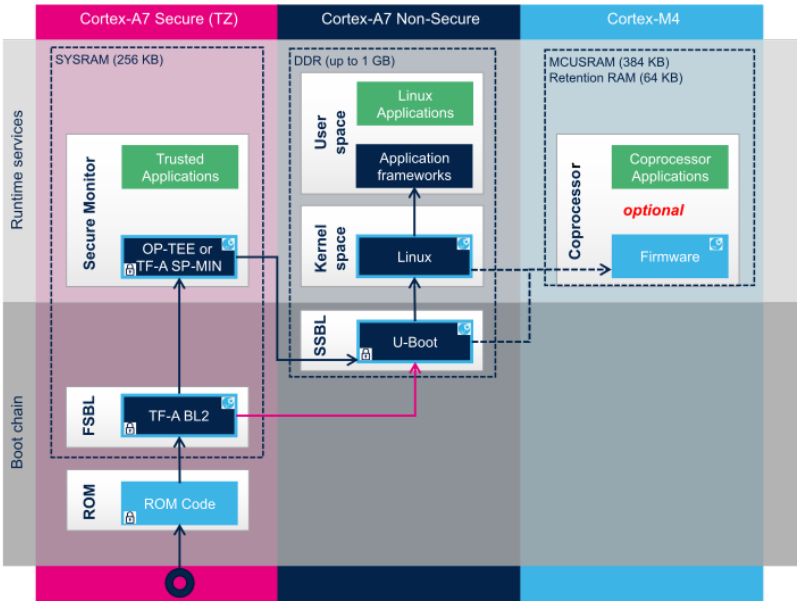


Figure 4.4: STM32MP157 boot chain; taken from [9].

including the Coresight components [22].
Another aspect of security is external access to the board via the debug port. By setting the deviceen bit in the BSEC register to 0, we can disable external access to the debug subsystem. However, the debug port is also connected to the AXI system interconnect and can thereby access all other resources, including memory. Because we need to set dbgen and spiden to 1, as described in section 4.2, the external debugger connected to the debug port can initiate both non-secure and even secure transfers, and thereby compromise our whole system.

5

Evaluation

In this chapter, we evaluate our CFA approach and our proof-of-concept implementation. First, we will demonstrate our prototype with a simple example, to show how it works. Then we will discuss some performance and overhead metrics, and finally, we will examine what kind of security guarantees we can get from using our approach and how our solution compares to others in terms of functionality. All tests in this chapter were done by using the setup in Figure 5.1. All tasks that can be associated with the role of the verifier (steps 1 and 4 in the figure), were run on an external Linux VM with an i7-6700K CPU and 12 GB of memory. The target and the prover were located in an OP-TEE instance on the STM32MP157A-DK1 board (steps 2 and 3 in the figure).

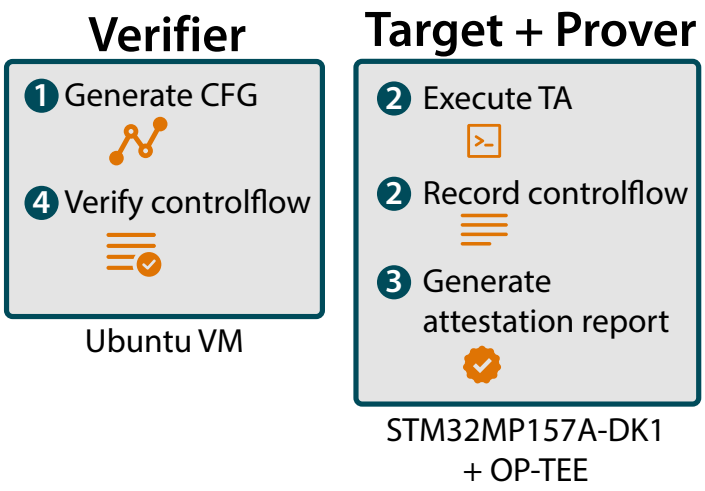


Figure 5.1: Our setup for testing the prototype

Listing 5.2: TA code for the demonstration

```
int val = 0;

void foo() {
        val = val + 1;
}

void bar() {
        foo();
        val = val + 5;
}

int test() {
        val = val + 3;
        return 1;
}

static TEE_Result test_0(uint32_t param_types,
        TEE_Param params[4])
{
        int x[2]={};
        for (int i=0; i<2; i++) {
                x[i] = i+1;
                foo();
        }
        bar();
        test();
        return TEE_SUCCESS;
}
```

5.1 Demonstration

To show how our prototype works, we walk through the whole process of tracing and verifying. We create a small TA and the associated Client App and add it to OP-TEE. Our example code (see Listing 5.2) is very straightforward. We have a short loop and also some (nested) function calls. The actual TA code contains some additional boilerplate code that we omit for clarity. For our example, we only trace our actual code of interest, which is shown in Listing 5.2, not the whole TA address space. This makes the demonstration more clear. We set the tracing range accordingly and configure the filter conditions, so that only user-mode code, running in the secure world is traced. We also set the UUID of the TA in OP-TEE, so that only this specific TA is attested. Using, the TA binary, we build the CFG with our CFG-recovery tool. The output of the CFG-generation tool is a GEXF file, that contains our graph and can later be loaded by the verifier. After building OP-TEE, we can start the testing. We execute our client app, which in turn invokes our TA. The tracing starts automatically and stops when the TA returns. After the execution, we can

Listing 5.3: Excerpt of the verifier output for the demonstration program

```
....
....
We wait for a jump now
set new expected jump target to 40014f
Idx:34; BRANCH_ADDRESS : Branch address.; Addr=0x4001014F ~[0x4F];
Indirect Jump is ok
....
....
*****************************************
*****************************************
No controlflow violations found
*****************************************
*****************************************
```

retrieve the attestation report, by calling the attestation PTA. We get a trace file and a file containing the MAC for the trace. We copy the files over to the device hosting our verifier. Then we run the verifier on the trace and CFG. We get a detailed output of the verification process and at the end a message, that tells us if there were any control-flow-violations or other errors during the verification. In Listing 5.1 you can see an excerpt from the verifier output. The verifier expects an indirect branch to a certain address. The branch address found in the trace matches this address and the verification can continue. Because the control flow was correct in our example run, the verifier states that no violations were found at the end.

5.2 Performance

While most other CFA approaches either have the target running on a Linux environment or on a microcontroller, our approach is aimed at attesting TAs in OP-TEE. Thus, the applications and environments for each are quite different. This is why we concentrate on the timing overhead for the different parts of our attestation system and the network overhead (trace size), instead of a direct comparison with other approaches. To analyze the overhead, we will follow the modeling in [23]. We identified the following tasks in our attestation system, that contribute to the overall attestation overhead:

- T_{cfg} - Generation of the CFG (offline)
- T_{init} - Initialization of the Tracing System (online)
- T_{tc} - Trace Collection Interrupt Routine (online)
- T_{leave} - Trace Collection of the rest of the trace after the execution and cleanup (online)
- T_{rt} - get the attestation report (Trace+MAC) (online)
- T_{ver} - Verification of the attestation report (online)

The time for the execution of the attested application itself is denoted by T_u. We will focus on T_{init}, T_{tc} and T_{leave}, as these directly interfer with the execution time of T_u.T_{cfg}, T_{rt} and

T_{ver} are considered later.

Tracing Overhead

For longer program runs, most of the timing overhead can be attributed to T_{tc}. While *init* and *leave* are only invoked once for each program run, *tc* may be invoked for an arbitrary number of times. The accumulated execution time of *tc* can be expressed as $c \cdot T_{tc}$ where c is the number of times that *tc* is invoked. c directly correlates with the size of the trace that results from the execution of u. The more trace is generated by the ETM the faster the ETF buffer overflows and the more often *tc* needs to run to drain the buffer.

Performance Measurements

We measured the performance of our system with several different types of applications. First, we used some synthetic assembly sequences to precisely evaluate the effect of certain instruction sequences on the trace size and tracing overhead. Then we tested some example TAs provided by OP-TEE, which we slightly modified for our testing. These applications are simple and mainly rely on the OP-TEE standard libraries. We also chose a more complex workload, in the form of a deep neural network, that is partly running in OP-TEE.

Experiment 1: Synthetic Performance Measurements

By analyzing the specification of the ETM signal protocol we found the instruction sequences that produce the largest trace sizes and thereby the worst overhead. We verified our assumptions by embedding assembly code in a test TA and measuring trace size and execution time overhead. The best case, in terms of trace size, is a long sequence of normal instructions or conditional instructions that pass their condition, as these are encoded in the most efficient way. For such a sequence, up to 16 executed instructions are represented in a one-byte packet by a counter value.

```
mov     r0, r0
mov     r0, r0
...
```

The instruction sequence above results in the following trace. Each line represents one packet and the amount of Es represents the number of instructions that are encoded in one packet.

```
Idx:170; ID:10; P_HDR : Atom P-header.; EEEEEEEEEEEEEE
Idx:171; ID:10; P_HDR : Atom P-header.; EEEEEEEEEEEEEE
Idx:172; ID:10; P_HDR : Atom P-header.; EEEEEEEEEEEEEE
```

Very bad for the trace size are multiple successive N atoms or alternating E and N atoms. This leads to a trace where in the worst case one instruction is represented in a separate one-byte packet. We used the following code to test this.

```
mov     r0, #1
cmp     r0, #2
```

```
it     eq
addeq     r1, r1, #1
it     eq
addeq     r1, r1, #1
...
```

Because the condition for the addeq instruction always fails, we have alternating E and N atoms, which leads to low compression.

```
Idx:144; ID:10; P_HDR : Atom P-header.; NE
Idx:145; ID:10; P_HDR : Atom P-header.; NE
Idx:146; ID:10; P_HDR : Atom P-header.; NE
Idx:147; ID:10; P_HDR : Atom P-header.; N
Idx:148; ID:10; P_HDR : Atom P-header.; E
```

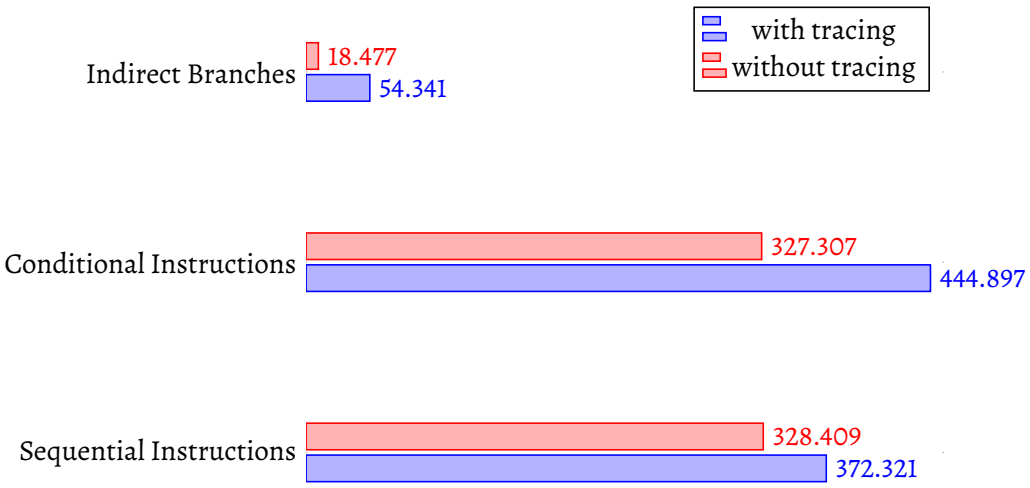


Figure 5.4: Execution time for different instruction sequences. Time in ms.

Indirect branch packets may also have a large size, as they include the address of the branch target. Because of address compression, they are often only one byte large. We used the following code to test the overhead for many successive indirect branches. The code shows a loop that iterates 50 times and uses an indirect branch in each iteration.

```
mov     r1, #1
label_0: cmp     r1, 50
beq label2
add     r1, r1, #1
bl       label_1
b    label_0
...
...
mov     r0, r0
mov     r0, r0
label_1: bx             lr
label_2: mov   r0, r0
```

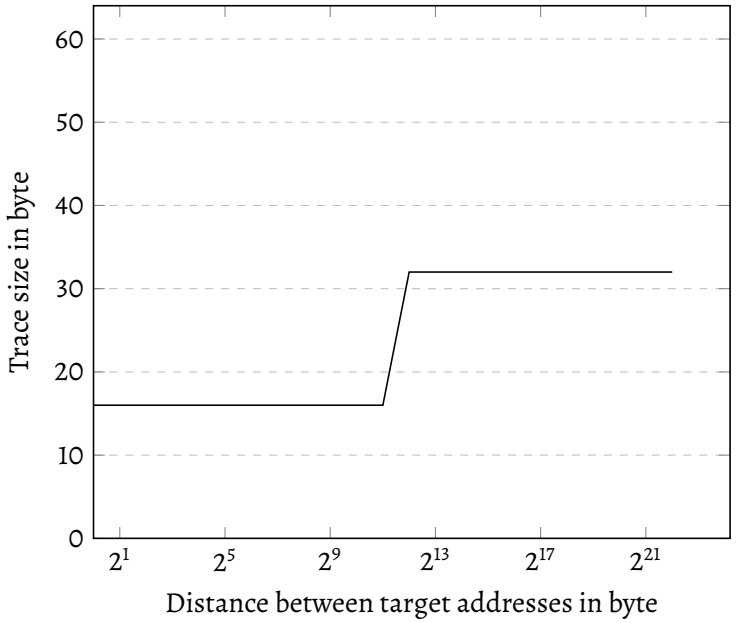
In the resulting trace, we can see that each branch results in a separate package, where only the last 8 bit of the address are encoded.

```
Idx:183; ID:10; BRANCH_ADDRESS : Branch address.; Addr=0x400101D6 ~[0x56];
Idx:184; ID:10; P_HDR : Atom P-header.; EEN
Idx:185; ID:10; P_HDR : Atom P-header.; EEE
Idx:186; ID:10; BRANCH_ADDRESS : Branch address.; Addr=0x400101D6 ~[0x56];
```

To compare the overhead of these different cases, we created different TAs with 500,000 instructions for each of the assembly sequences.

In Figure 5.4 we can see that the sequential instructions have the smallest overhead of only 13.37%. The conditional instructions lead to an overhead of 35.92% and the indirect

Figure 5.5: Effect of the distance between branch addresses on the trace size



branches to the largest overhead of 194.10%. For a case where the target addresses of indirect branches are very different, the overhead can become larger. To show this effect, we traced a small assembly program, see Listing 5.6 , where we inserted a sequence of NOP instructions in the middle, that we scaled up. It can be clearly seen, that indirect branches to addresses which are further apart lead to larger traces, see Figure 5.5. For the attestation this causes larger overhead, as the trace needs to be saved more often The trace size and execution time are not deterministic. Preemption of TAs and exceptions may lead to slightly different results for each invocation because the exceptions are represented in the trace.

In another experiment, we traced a simple loop with a function call (see Listing 5.2) and scaled up the loop iterations to see the effect on the tracing overhead. In Figure 5.7 you can see how the overhead becomes less until it stabilizes at around four. The overhead is given as a multiple of the original execution time. The same plot, but with a higher resolution and a focus on the first 100,000 iterations can be seen in Figure 5.8.

```
int k = 0;

void test() {
    k++;
}

void my_test_fun(int n) {
        for (int i=0;i<n;i++) {
        test();
   }
}
```

Listing 5.6: Assembly Code for showing the effects of longer distance between return branch addresses on the trace size

```
main: bl foo
      b end

[[NOP BLOCK]]

foo: push {lr}
     bl bar
     pop {lr}
     bx lr
bar: bx lr
end:
```

Experiment 2: OP-TEE Example TAs

For testing the overhead of the attestation in a realistic setting, we first tested typical use cases for OP-TEE. As OP-TEE is often used for cryptography-related tasks, we chose examples from this area. For each example we measured:

- the execution time with attestation: $T_{att} = T_u + T_{init} + c \cdot T_{tc} + T_{leave}$
- the execution time without attestation: T_u
- the timing overhead $O = (T_{att}/T_u) \cdot 100 - 100$

The first application we tested is a modified version of an example TA, that comes with OP-TEE. It generates a 256-bit RSA key-pair and signs the string "test". Here we could only measure a small overhead of 4.67%. Another application we tested creates a secure object and stores it in the secure storage of OP-TEE. The overhead for the tracing is 9.88%. The low overhead also results from the usage of API functions in the TAs which are executed in kernel mode and not traced by us. An overview of our results can be seen in Table 5.9.

Experiment 3: Neural Network

Another application we used for benchmarking is DarkneTZ [27]. This is an application based on the Darknet Deep Neural Network framework which runs several layers of the network in OP-TEE to achieve certain security guarantees. This is a challenging task, as it is quite a heavy workload for our low-end hardware. For testing, we used the inference for image classification. In our configuration, seven of the 11 layers are running in an OP-TEE TA. We measured an overall overhead of 1.57% for the tracing, see Table 5.10.

Other Attestation Overhead

We also examined the time T_{rt} for retrieving the trace and calculating the MAC. For a trace with a size of 22,000 byte, we measured 15ms execution time. This could probably be improved by using the cryptographic accelerator on our board, but we did not use this yet

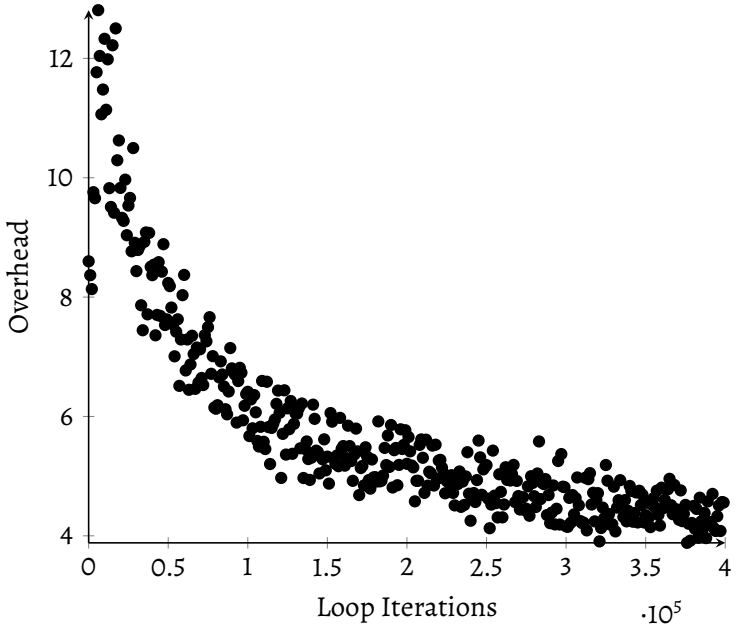


Figure 5.7: Tracing Overhead for a simple loop with a function call. The loop iterations are scaled up from 0 to 400,000. For the code, see Listing 5.2.

because OP-TEE has no driver support for this hardware component. For the verification T_{ver} we measured a time of 3.37 ms. As the verifier runs on a separate device with potentially more capable hardware, this measurement is not really representative. For T_{cfg} we could measure up to 30 minutes for the example TAs with a context-sensitivity of 15. Because the CFG-recovery is only executed once, this can be neglected.

How to Lower the Overhead

We showed that the performance overhead is reasonable for the real applications that we tested. In general, however, it can become larger, depending on the code of the traced application. There are several possibilities for lowering the performance overhead. One potential enhancement could be the use of DMA. This could improve the time needed for saving the tracing buffer to memory because the CPU would not be involved anymore. We tried this approach, but unfortunately, DMA access to the ETF buffer is not possible on our board. Another, even more promising improvement would require different tracing components. By usin+g an ETR, the traces could be directly written to memory. This would reduce the attestation overhead significantly. As we currently have no access to a board with this hardware, we leave this to future work (see Section 6.3).

About the Trace Size

The trace size for a program cannot be calculated in advance, as it depends on the input and can also be influenced by external events like exceptions, that appear in the trace. As we currently do not use any means of trace compression, the traces can become very big,

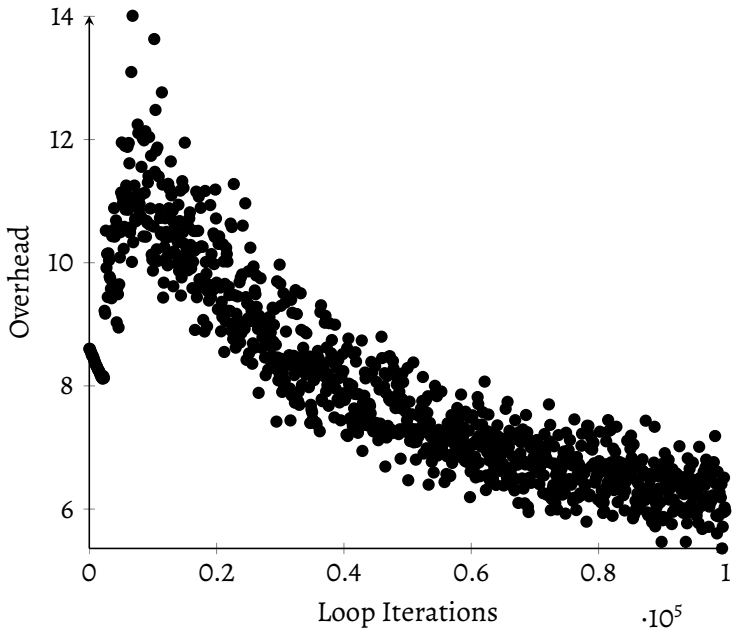


Figure 5.8: High resolution plot for the code in Listing 5.2 with the focus on the first 100,000 loop iterations.

Table 5.9: Attestation overhead for some example TAs provided by OP-TEE

	Original (s)	Traced (s)	Overhead	Trace Size (kB)
Crypto	1.437	1.504	4.67%	2.313
Secure Storage	0.37	0.406	9.88%	10.438

which can be a problem for boards with a small memory, like ours. This might also be problematic if they are sent over the network. A possible solution could be to use preprocessing of the trace on the prover side, but this would require trace decoding and impose additional overhead on the prover. Solving this remains open for future work.

5.3 Security Guarantees

In the following section, we will describe the security guarantees, that our CFA approach can give.
Due to the use of Coresight, we can obtain precise measurements of the control flow

Table 5.10: Attestation overhead for DarkneTZ

	Original (s)	Traced (s)	Overhead	Trace Size (kB)
Overall	1474.722	1497.946	1.57%	26691.921

of a TA in the form of indirect branch targets and information about taken conditional branches. We use that information, to determine the source and target of each indirect branch. Together with the context-sensitive CFG that we recovered, we can reliably detect violations of control-flow integrity.
One type of attack that can be detected are ROP attacks, which we introduced in 2.3. Control-flow transitions to unexpected CFG-nodes or to addresses within CFG-nodes can be easily detected as they do not conform to the legal control-flow of the program. Due to the context-sensitive CFG, we also enforce that the call- and return-addresses match.
The same applies for Jump oriented Programming. This kind of attack manipulates indirect jumps and also results in an illegal control-flow measurement. Non-control data attacks can in principle be detected with our approach. If an unintended (privileged but legal) control-flow path was taken or if a loop was executed an unexpected amount of times, this is visible in the attestation report. However, our current implementation of the verifier cannot detect such attacks. Because we only attest the control flow, data-oriented programming attacks cannot be detected as these work by solely manipulating data, without exhibiting illegitimate control flows.
Attacks that replace the target application code are ruled out because OP-TEE checks the integrity of TA binaries before loading them. To ensure the integrity of OP-TEE itself, we assume that secure boot is used. The integrity and authenticity of the attestation report are also protected, by using a MAC.

CFG Recovery

Because the CFG-analysis is usually used for binary analysis and not for control-flow verification, it does not fully suit our needs. One big issue is the missing information about the number of loop iterations. The CFG simply allows infinite iterations, so some kinds of attacks that manipulate the loop variable to iterate an unexpected amount of times, can currently not be detected. The addition of loop information could be considered for the next iteration of the prototype.

5.4 Comparison to Other Approaches

To conclude with the evaluation, we compared some aspects of our attestation solution with other approaches. The results can be seen in Table 5.11. Some approaches do not describe their verification approach in detail, so we could not evaluate every aspect of every approach.

Table 5.11: Functional comparison of our solution to other approaches

	CFA / CFI	Protection of Indirect Branches	Protection of Control-Flow Order	Context Sensitivity	Data Accesses Attested	Transparency	Custom Hardware Design Needed	Target Protected by TEE	Hardware-assisted	Architecture
Our Solution	CFA	●	●	●		●		●	●	ARM
C-Flat [1]	CFA	●	●	●						ARM
LOAFAT [18]	CFA	●	●	—		●	●		●	RISC-V
ATRIUM [45]	CFA	●	●	—		●	●		●	RISC-V
LiteHAX [17]	CFA	●	●	●	●	●	●		●	RISC-V
LAHEL [6]	CFA	●				●	●		●	ARM
Coresight CFI [23]	CFI	●	●	●		●			●	ARM
DIAT [2]	CFA	●								ARM
GuaranTEE [28]	CFA	●	●					●		Intel x86
ScaRR [43]	CFA	●	●	●						Intel x86
ReCFA [46]	CFA	●	●	●						Intel x86

● = provides property; empty = does not provide property; — = unknown

6

Conclusion

In the following sections, we will discuss some aspects of our approach and summarize our work. We conclude with possible starting points for future work.

6.1 Summary

In this work, we showed that Coresight tracing can be used to build a CFA-system on ARM platforms, without the need for additional hardware. We successfully implemented a prototype that extends OP-TEE, a TrustZone-based TEE, with CFA-capabilities. The prototype consists of modules for generating the CFG as a reference for valid control flows, a measurement engine based on Coresight and a verifier. We evaluated the performance of our approach and found that the runtime overhead generally is reasonable. The size of the attestation report could be a problem in some cases, as this can become pretty large. However, with the programs we tested in our experiments, the size was still reasonable. We also showed the security guarantees we get from our solution. Because we use the order, the source and the target for indirect jumps in our verification, most relevant runtime attacks like ROP can be reliably detected.

6.2 Discussion

A great advantage of our approach over other approaches for CFA is, that there is no need for additional hardware or custom SoC designs. Instead, we repurpose the existing Coresight tracing system for our use case. This makes our solution easy to implement on existing boards. However, by repurposing Coresight for attestation, we inherit a certain overhead in terms of trace size, that could be avoided by building a custom tracing solution. Another thing we noticed is that it can be hard to use Coresight tracing in a secure way. As it is originally intended to be used for debugging purposes, some SoCs lack fine-granular security configurations for Coresight and TrustZone. However, generally it is possible to create a secure system with a suitable SoC. In our implementation, we currently do not support the parallel execution of TAs. In principle, however, this would be possible by distinguishing the different applications by context IDs that are recorded in the trace. In

addition, the execution of all OP-TEE TAs would have to be actively interrupted when the ETF buffer is read out in order to save the trace. While in some theoretical worst-cases the overhead can become very big, we found that for actual TAs the overhead of our attestation approach is still reasonable. Because TAs are usually not very complex applications the trace size is often not that big. While we have shown that attestation is possible with our hardware, we think that it is generally preferable to use other hardware like an ETR to allow more efficient tracing.

6.3 Future Work

Due to the limited scope of a bachelor thesis, there is still much room for improvement regarding our attestation solution. For simplification, we decided to disable ASLR in OP-TEE. As it is obviously better to have this security feature turned on, the next iteration of our prototype should be able to cope with ASLR. Another simplification that we introduced, was restricting our system to one core only. While this made it easier, to implement the interrupt routine for saving the trace, it would be nice when our system could also handle multiple cores. It could also be interesting to experiment with other Coresight hardware, like the ETR. This could potentially reduce the attestation-induced runtime overhead massively, as the traces could be directly stored in memory, thereby also omitting the need for interrupts. Depending on the ETM-version, Coresight is also able to trace memory accesses. The usage of this additional data could potentially allow for more fine-grained attestation and enable the detection of so-called non-control-data attacks. However, this would result in much bigger traces and also require significant changes to our verifier. In our current implementation, we do not enforce the number of loop iterations. As this allows some types of runtime attacks, where loop counter variables are manipulated, we are interested in adding these capabilities to the verifier. However, this would only work for loops with a static number of iterations, not for loops with data dependencies. A potential simple solution could also be loop-unrolling in the compiler, as this would not require any changes on the side of the verifier.

Bibliography

[1] Abera, T., Asokan, N., Davi, L., Ekberg, J., Nyman, T., Paverd, A., Sadeghi, A., and Tsudik, G. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM, 2016, pp. 743–754. DOI: 10.1145/2976749.2978358 (cit. on pp. 2, 3, 38).

[2] Abera, T., Bahmani, R., Brasser, F., Ibrahim, A., Sadeghi, A., and Schunter, M. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/diat-data-integrity-attestation-for-resilient-collaboration-of-autonomous-systems/ (cit. on pp. 3, 4, 38).

[3] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. In: *White Paper, January*, 2020 (cit. on p. 8).

[4] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for CPU based attestation and sealing. In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. 7. ACM New York, NY, USA. 2013 (cit. on p. 6).

[5] *Android Keystore*. https://source.android.com/docs/security/features/keystore. Accessed: 2022-11-03 (cit. on p. 7).

[6] Arias, O., Sullivan, D., Shan, H., and Jin, Y. LAHEL: Lightweight Attestation Hardening Embedded Devices using Macrocells. In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*. IEEE, 2020, pp. 305–315. DOI: 10.1109/HOST45689.2020.9300257 (cit. on pp. 2–4, 38).

[7] *Arm Confidential Compute Architecture*. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed: 2022-04-24 (cit. on p. 8).

[8] *Arm DS-5 Debugger User Guide*. https://developer.arm.com/documentation/100953/0529. Accessed: 2022-11-25 (cit. on p. 21).

[9] *Boot chain overview - STM32MP157*. https://wiki.st.com/stm32mpu/wiki/Boot_chain_overview. Accessed: 2022-10-16 (cit. on p. 26).

[10] *Building a Secure System using TrustZone Technology*. https://community.arm.com/cfs-file/___key/telligent-evolution-components-attachments/01-2057-00-00-00-00-53-99/PRD29_2D00_GENC_2D00_009492C_5F00_

trustzone__5F00__security__5F00__whitepaper.pdf. Accessed: 2022-11-25. 2009 (cit. on p. 8).

[11] Certes, J. and Morgan, B. Remote Attestation of Bare-Metal Microprocessor Software: A Formally Verified Security Monitor. In: *Database and Expert Systems Applications - DEXA 2021 Workshops - BIOKDD, IWCFS, MLKgraphs, AI-CARES, ProTime, AISys 2021, Virtual Event, September 27-30, 2021, Proceedings*. Ed. by G. Kotsis, A. M. Tjoa, I. Khalil, B. Moser, A. Mashkoor, J. Sametinger, A. Fensel, J. M. Gil, L. Fischer, G. Czech, et al. Vol. 1479. Communications in Computer and Information Science. Springer, 2021, pp. 42–51. DOI: 10.1007/978-3-030-87101-7_5 (cit. on pp. 3, 4).

[12] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., and Winandy, M. Return-oriented programming without returns. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. Ed. by E. Al-Shaer, A. D. Keromytis, and V. Shmatikov. ACM, 2010, pp. 559–572. DOI: 10.1145/1866307.1866370 (cit. on p. 11).

[13] Coker, G., Guttman, J. D., Loscocco, P. A., Herzog, A. L., Millen, J. K., O'Hanlon, B., Ramsdell, J. D., Segall, A., Sheehy, J., and Sniffen, B. T. Principles of remote attestation. In: *Int. J. Inf. Sec.* 10(2):63–81, 2011. DOI: 10.1007/s10207-011-0124-7 (cit. on pp. 5, 6).

[14] *Cortex A53 documentation*. https://developer.arm.com/documentation/ddi0500/e/introduction/compliance/debug-architecture. Accessed: 2022-10-06 (cit. on p. 19).

[15] *CSAL*. https://github.com/ARM-software/CSAL. Accessed: 2022-10-06 (cit. on pp. 19, 20).

[16] Czerwinski, R. *Using OP-TEE to Authenticate IoT Devices*. https://www.pengutronix.de/de/blog/2021-02-09-optee-product.html. Accessed: 2022-10-16 (cit. on p. 24).

[17] Dessouky, G., Abera, T., Ibrahim, A., and Sadeghi, A. LiteHAX: lightweight hardware-assisted attestation of program execution. In: *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*. Ed. by I. Bahar. ACM, 2018, p. 106. DOI: 10.1145/3240765.3240821 (cit. on pp. 2–4, 23, 38).

[18] Dessouky, G., Zeitouni, S., Nyman, T., Paverd, A., Davi, L., Koeberl, P., Asokan, N., and Sadeghi, A. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In: *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. ACM, 2017, 24:1–24:6. DOI: 10.1145/3061639.3062276 (cit. on pp. 3, 4, 38).

[19] *Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.5*. https://developer.arm.com/documentation/ihi0014/q/?lang=en. Accessed: 2022-04-24 (cit. on p. 11).

[20] *GlobalPlatforms TEE Client API Specification Version 1.0*. https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf. Accessed: 2022-04-19 (cit. on p. 9).

[21] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Com-

puter Society, 2016, pp. 969–986. DOI: 10.1109/SP.2016.62. URL: https://doi.org/10.1109/SP.2016.62 (cit. on p. 11).

[22] *Juno ARM Development Platform SoC Technical Reference Manual*. https://www.st.com/resource/en/reference_manual/rm0436-stm32mp157-advanced-armbased-32bit-mpus-stmicroelectronics.pdf. Accessed: 2022-10-16 (cit. on p. 26).

[23] Kuzhiyelil, D., Zieris, P., Kadar, M., Tverdyshev, S., and Fohler, G. Towards Transparent Control-Flow Integrity in Safety-Critical Systems. In: *Information Security - 23rd International Conference, ISC 2020, Bali, Indonesia, December 16-18, 2020, Proceedings*. Ed. by W. Susilo, R. H. Deng, F. Guo, Y. Li, and R. Intan. Vol. 12472. Lecture Notes in Computer Science. Springer, 2020, pp. 290–311. DOI: 10.1007/978-3-030-62974-8_17 (cit. on pp. 4, 29, 38).

[24] *Learn the architecture - Introducing CoreSight debug and trace*. https://developer.arm.com/documentation/102520/0100. Accessed: 2022-11-23 (cit. on p. 2).

[25] *Learn the architecture: TrustZone for AArch64*. https://developer.arm.com/documentation/102418/0101/TrustZone-in-the-processor. Accessed: 2022-04-19 (cit. on p. 7).

[26] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., and Song, D. Keystone: an open framework for architecting trusted execution environments. In: *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. Ed. by A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer. ACM, 2020, 38:1–38:16. DOI: 10.1145/3342195.3387532 (cit. on p. 8).

[27] Mo, F., Shamsabadi, A. S., Katevas, K., Demetriou, S., Leontiadis, I., Cavallaro, A., and Haddadi, H. DarkneTZ: towards model privacy at the edge using trusted execution environments. In: *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*. Ed. by E. de Lara, I. Mohomed, J. Nieh, and E. M. Belding. ACM, 2020, pp. 161–174. DOI: 10.1145/3386901.3388946 (cit. on p. 34).

[28] Morbitzer, M., Kopf, B., and Zieris, P. GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments. In: *CoRR* abs/2202.07380, 2022. arXiv: 2202.07380. URL: https://arxiv.org/abs/2202.07380 (cit. on pp. 3, 4, 38).

[29] *MultiZone*. https://hex-five.com/. Accessed: 2022-11-07 (cit. on p. 8).

[30] Ning, Z., Wang, C., Chen, Y., Zhang, F., and Cao, J. Revisiting ARM Debugging Features: Nailgun and Its Defense. In: *IEEE Transactions on Dependable and Secure Computing*:1–1, 2021. DOI: 10.1109/TDSC.2021.3139840 (cit. on p. 10).

[31] Ning, Z. and Zhang, F. Ninja: Towards Transparent Tracing and Debugging on ARM. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, pp. 33–49. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning (cit. on p. 12).

[32] *OP-TEE Documentation*. https://optee.readthedocs.io/en/3.16.0/. Accessed: 2022-03-15 (cit. on pp. 2, 9, 10, 24).

[33] *Open Coresight Trace Decode Library*. https://github.com/Linaro/OpenCSD. Accessed: 2022-10-26 (cit. on p. 21).

[34] Sabt, M., Achemlal, M., and Bouabdallah, A. Trusted Execution Environment: What It is, and What It is Not. In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357 (cit. on p. 5).

[35] *Samsung Knox Security Solution Whitepaper*. https://images.samsung.com/is/content/samsung/p5/global/business/mobile/SamsungKnoxSecuritySolution.pdf. Accessed: 2022-11-03 (cit. on p. 7).

[36] *Securing Edge IoT devices with OP-TEE*. https://www.iwavesystems.com/news/securing-edge-iot-devices-with-op-tee/. Accessed: 2022-05-03 (cit. on p. 9).

[37] Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. Ed. by P. Ning, S. D. C. di Vimercati, and P. F. Syverson. ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313 (cit. on p. 11).

[38] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157. DOI: 10.1109/SP.2016.17 (cit. on pp. 4, 17, 23).

[39] Stajnrod, R., Yehuda, R. B., and Zaidenberg, N. J. Attacking TrustZone on devices lacking memory protection. In: *J. Comput. Virol. Hacking Tech.* 18(3):259–269, 2022. DOI: 10.1007/s11416-021-00413-y (cit. on p. 8).

[40] *STM32MP157 Documentation*. https://www.st.com/resource/en/reference_manual/rm0436-stm32mp157-advanced-armbased-32bit-mpus-stmicroelectronics.pdf. Accessed: 2022-10-06 (cit. on pp. 20, 24).

[41] Subramanyan, P., Sinha, R., Lebedev, I. A., Devadas, S., and Seshia, S. A. A Formal Foundation for Secure Remote Execution of Enclaves. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM, 2017, pp. 2435–2450. DOI: 10.1145/3133956.3134098 (cit. on p. 8).

[42] Swami, Y. SGX Remote Attestation is not Sufficient. In: *IACR Cryptol. ePrint Arch.*:736, 2017. URL: http://eprint.iacr.org/2017/736 (cit. on p. 6).

[43] Toffalini, F., Losiouk, E., Biondo, A., Zhou, J., and Conti, M. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 2019, pp. 121–134. URL: https://www.usenix.org/conference/raid2019/presentation/toffalini (cit. on p. 38).

[44] *Trusted Firmware-A*. https://www.trustedfirmware.org/projects/tf-a/. Accessed: 2022-11-03 (cit. on p. 9).

[45] Zeitouni, S., Dessouky, G., Arias, O., Sullivan, D., Ibrahim, A., Jin, Y., and Sadeghi, A. ATRIUM: Runtime attestation resilient under memory attacks. In: *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*. Ed. by S. Parameswaran. IEEE, 2017, pp. 384–391. DOI: 10.1109/ICCAD.2017.8203803 (cit. on pp. 3, 4, 38).

[46] Zhang, Y., Liu, X., Sun, C., Zeng, D., Tan, G., Kan, X., and Ma, S. ReCFA: Resilient Control-Flow Attestation. In: *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 2021, pp. 311–322. DOI: 10.1145/3485832.3485900 (cit. on p. 38).

Acronyms

ASLR *Address Space Layout Randomization*

BB *basic block*

CCA *Confidential Compute Architecture*
CFA *Control-Flow Attestation*
CFG *Control-Flow Graph*
CFI *Control Flow Integrity*
CSAL *Coresight Access Library*
CTI *Cross Trigger Interface*

DMA *Direct memory access*

ETB *Embedded Trace Buffer*
ETF *Embedded Trace Fifo*
ETM *Embedded Trace Macrocell*
ETR *Embedded Trace Router*
ETZPC *Extended Trustzone protection controller*

HUK *Hardware unique key*

MAC *Message Authentication Code*
MMU *Memory management unit*

OTP *One-Time-Programmable Memory*

PTA *Pseudo Trusted Application*
PTM *Program Trace Macrocell*

REE *Rich Exectution Environment*
ROP *Return-oriented programming*

SMC *Secure Monitor Call*
SoC *System on a chip*

TA *Trusted Application*
TEE *Trusted Exectution Environment*
TMC *Trace Memory Controller*
TPIU *Trace Port Interface Unit*
TZC *Trustzone address space controller*