Mario Werner

System Architectures and Techniques for Efficient, Secure, and Trusted Code Execution

DOCTORAL THESIS
to achieve the university degree of
Doktor der technischen Wissenschaften
submitted to
Graz University of Technology

Supervisor
Prof. Stefan Mangard (TU Graz)

Assessors
Prof. Stefan Mangard (TU Graz)
Prof. Ingrid Verbauwhede (KU Leuven)

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Graz, April 2020

Abstract

Contemporary devices rely on an increasing amount of software to implement their respective functionality. Hence, the software including its correct execution are key assets in modern systems. A huge portfolio of attack techniques, ranging from remotely mountable software attacks to local attacks that utilize physical access, endanger these assets and need to be mitigated. However, countermeasures deployed in current systems mainly focus on the prevention of pure software attacks. Attack types that (locally or remotely) exploit physical properties of a device are often neglected although they are applicable to mobile devices and threaten software in the emerging cloud computing, IoT, and Industry 4.0 contexts. The exploitation of side-channels (e.g., timing, power) and fault injection are two prominent examples of physical attacks. Unfortunately, modern processor architectures provide basically no support for protecting software against such physical attacks.

In this thesis, we work towards fixing this shortcoming of current architectures and present several novel techniques that enable secure software execution in the context of physical attacks. In particular, to protect the code, we showcase two hardware-supported Control-Flow Integrity (CFI) schemes which enforce that executed instructions are genuine and in correct sequence. Both techniques have been implemented in real processor designs, come with appropriate toolchain support, and are tested in simulation and/or on actual FPGA/ASIC hardware. Additionally, building upon such CFI schemes, a novel software-only technique for remote attestation has been developed. The new technique effectively incorporates all common existing approaches and can further be used for online licensing.

In the domain of protecting data against illegal access and tampering, two approaches for improving the memory subsystem are part of this thesis. Firstly, an open-source hardware framework for building as well as researching transparent memory encryption and authentication modules is presented. This framework can be used to protect data and code against disclosure and tampering via physical attacks. Secondly, a novel approach for building randomized set-associative CPU caches has been devised. Caches following this design approach have similar performance as contemporary designs but are considerably harder to attack via timing side-channel attacks (e.g., cache attacks).

Acknowledgements

This thesis was only possible because many great people supported me along the way. I am deeply grateful for receiving this kind of support and, in the following, want to thank those who accompanied me during my PhD studies.

First and foremost, I would like to thank my advisor Stefan Mangard for placing his faith in me by giving me the opportunity to pursue this degree. Thank you for introducing me to my field of research, for your continuous guidance, and for the freedom to experiment with new ideas. I also want to thank Ingrid Verbauwhede for agreeing to assess my thesis and for providing valuable feedback.

Thank you to all my amazing co-authors that gave me either the chance to collaborate on their research or actively contributed to mine. In particular, thank you Ferdinand Brasser, Hannes Groß, Manuel Jelinek, Daniel Kales, Maja Malenko, Pascal Nasahl, Sebastian Ramacher, Christian Rechberger, Ahmad-Reza Sadeghi, Roman Walch, and Samuel Weiser for the opportunity to work with you on exiting papers. Moreover, I am particularly grateful to Luca Benini, Alfio Di Mauro, Lukas Giner, Daniel Gruss, Frank K. Gürkaynak, Stefan Mangard, David Schaffenrath, Robert Schilling, Michael Schwarz, Thomas Unterluggauer, and Erich Wenger, with whom I wrote the papers discussed in this thesis. Thank you for your input, the fruitful discussions, and all your work.

Thank you also to all past and current colleagues at IAIK for interesting conversations during coffee/cake/lunch breaks and for the fun group events. I further wish to thank Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Unfortunately, we didn't manage to write a joint paper during my studies but I really enjoyed our discussions and highly appreciate your help with cryptographic problems. Furthermore, I am very grateful to Robert Schilling, Thomas Unterluggauer, and Erich Wenger. Robert, thank you for the great collaborations, for joining me on the compiler playground, and for ensuring that our hardware/software keeps working. Thank you, Thomas, for the amazing time at our office, for our joint papers, for challenging me during discussions, and for having an open ear for all kinds of crazy ideas. Erich, thank you for motivating me to start a PhD and for backing me during my first steps in the academic world.

Last, but not least, I wish to express my sincere gratitude to my friends and family. Thank you for the continuous support, for providing encouragement when needed, and for all the good times we shared. I am looking forward to many more exciting and fun years with you.

Mario

Table of Contents

List of Tables

List of Figures

Glossary

AD	Associated Data
AE	Authenticated Encryption
AEE	Authentic-Encrypted Execution
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
APE	Authenticated Permutation-based Encryption
ASIC	Application Specific Integrated Circuit
ASLR	Address Space Layout Randomization
ASM	ASseMbly
AXI	Advanced eXtensible Interface
BB	Basic Block
BIP	Bimodal Insertion Policy
CBC	Cipher Block Chaining
CETS	Compiler Enforced Temporal Safety
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
CFP	Control-Flow Path
CFT	Control-Flow Transfer
CIA	Code Injection Attacks
COOP	Counterfeit Object-Oriented Programming
cpb	cycles-per-byte
CPI	Code-Pointer Integrity
CPS	Code-Pointer Separation
CPU	Central Processing Unit
CRA	Code Reuse Attacks
CRC	Cyclic Redundancy Check
CSM	Continuous Signature Monitoring
CSR	Control and Status Register
CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial-of-Service
DEP	Data-Execution Prevention
DOP	Data-Oriented Programming
DoS	Denial-of-Service

DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
ECB	Electronic CodeBook
ECC	Elliptic-Curve Cryptography
EDA	Electronic Design Automation
EM	ElectroMagnetic
ESIP	Extraction of Software IP
ESSIV	Encrypted Salt-Sector Initialization Vector
FAIS	Fault Attack induced Instruction Skips
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GE	Gate Equivalent
GPIO	General-Purpose Input/Output
GPSA	Generalized Path Signature Analysis
HDD	Hard-Disc Drive
HDL	Hardware Description Language
I^2C	Inter-Integrated Circuit
I^2S	Inter-IC Sound
IDF	Index Derivation Function
IE	Infective Execution
IoT	Internet-of-Things
IP	Intellectual Property
IPC	Inter-Process Communication
ISA	Instruction-Set Architecture
JOP	Jump-Oriented Programming
JTAG	Joint Test Action Group
LLC	Last-Level Cache
LRU	Least Recently Used
LSB	Least Significant Bit
MAC	Message Authentication Code
MAIR	Memory Attribute Indirection Register
MISR	Multiple-Input Signature Register
MMU	Memory-Management Unit
MPU	Memory-Protection Unit
MPX	Memory Protection eXtensions
NVM	Non-Volatile Memory

NX	No-eXecute
OS	Operating System
PAT	Page Attribute Table
PC	Program Counter
PCB	Printed Circuit Board
PID	Process IDentifier
PL	Programmable Logic
PLC	Programmable-Logic Controller
PLRU	Pseudo-LRU
PS	Processing System
PSA	Path Signature Analysis
PTE	Page Table Entry
PTW	Page-Table Walker
RAM	Random-Access Memory
RMW	Read-Modify-Write
ROP	Return-Oriented Programming
SCE	Secure Code Execution
SCFP	Sponge-Based Control-Flow Protection
SDID	Security Domain IDentifier
SGX	Software Guard eXtensions
SHA	Secure Hash Algorithm
SME	Secure Memory Encryption
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TCB	Trusted Computing Base
TEC	Tamper Evident Counter
TEE	Trusted-Execution Environment
TLB	Translation-Lookaside Buffer
TMTO	Time-Memory Trade-Off
TOCTOU	Time-Of-Check Time-Of-Use
TPM	Trusted Platform Module
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XEX	Xor-Encrypt-Xor
XTS	XEX-based Tweaked-codebook with ciphertext Stealing

1 Introduction

Essentially every electronic device, which is built nowadays, is controlled by software that gets executed by one or more general-purpose processors. The reasons for this trend towards software and processor-based designs are manifold. For example, changing software is typically much simpler than updating specialized hardware that is built for one specific purpose. Relying strongly on software also makes it possible to easily fix errors and add features, even when the device is already deployed in the field. Furthermore, production and development costs can be reduced compared to custom hardware designs. These savings are achieved since general-purpose processors, as well as the majority of required peripherals, are widely available as cheap commercial off-the-shelf components. As the result, although the casing and the intended use case may differ, the architecture of modern devices—ranging from comparably small smart home gadgets to complex systems like self-driving cars—gets remarkably similar to a standard computer architecture. Therefore, the functionality of a device is primarily determined by the huge and highly complex software stacks, which are embedded into the device's firmware.

Unfortunately, exactly this huge amount of complex code is also a problem in practice. Writing software, which is both complex and correct is still very hard, especially when high performance and efficiency are needed. This is particularly problematic, because bugs in software not only degrade the user experience but also lead to all kinds of security problems. In fact, the majority of notable security incidents that we know of today were possible due to the exploitation of software bugs. However, even with perfectly bug-free application software, security of modern electronic devices is still an issue.

In particular, with the increasing use of mobile devices (e.g., smart phones, tablets) and the rise of cloud computing, the Internet-of-Things (IoT), and

Industry 4.0, more and more software gets executed in completely unknown environments. Trust becomes a serious issue in such scenarios given that even foundational invariants (e.g., code is executed as programmed) can easily be violated by the respective device user/owner. Without further protection, a cloud provider, for example, can access the data of its users or tamper with the performed computations. After all, cloud providers control the most privileged software of the system (e.g., hypervisor, operating system) in addition to having direct physical access to the machine including sub-components like Hard-Disc Drives (HDDs) and Random-Access Memory (RAM). Moreover, on contemporary hardware, even completely unrelated third parties are able to mount attacks due to co-location of different applications on the same hardware, which is common in cloud computing.

At the other end of the spectrum, IoT and Industry 4.0 device vendors face very similar challenges given that their products are operated on-premise of the respective customers. Malicious users can exploit the implied physical access to arbitrarily tamper with a device and its software. In an industrial application, for example, this endangers all of the vendor's software Intellectual Property (IP) that is deployed in the shipped Programmable-Logic Controllers (PLCs) and enables the production of counterfeited machinery. Exactly the same threat scenario also holds true for IoT device vendors that, for instance, specialize in smart home equipment. Finally, without proper authentication of code and data, even supply chain attacks that install arbitrary malware on genuine hardware are facilitated.

Problem Statement

The modern computing landscape is changing rapidly towards mobile devices, cloud computing, Industry 4.0 and the IoT. In such applications, correct and secure execution of software—one of the most valuable assets of every modern device—is of utmost importance. Unfortunately, ensuring that code is executed in this way is hardly possible on contemporary general-purpose processor architectures.

In more detail, most widely used architectures have been optimized for threat models that only consider software-based attacks. In such a scenario, access control mechanisms (e.g., memory management/protection units, privilege modes) are, for instance, sufficient to ensure that the integrity of code is maintained and that secrets cannot be accessed arbitrarily. The hardware aspects, on the other hand, are largely neglected under the assumption that all components—including the processor—behave according to the functional specification.

This simplistic approach is already problematic in purely software-based attack scenarios as it does not include side-channel attacks (e.g., for stealing data from co-located cloud users). However, solely relying on functional correctness is clearly insufficient as soon as a device is operated under the physical control of an untrusted third party. Adversaries with direct access can, in addition to mounting software-based attacks, also physically tamper with a device (e.g.,

inject faults, probe signals on a bus, measure power consumption). Flipping a single bit in an opcode of an instruction, for example, can already be sufficient to bypass a security check and completely compromise a system. Data in RAM is similarly vulnerable given that one changed bit, for instance in a page table, is typically enough to circumvent all common access control schemes.

Summarizing, current computing architectures—although commonly deployed in such settings—are not sufficiently prepared for malicious environments that allow physical attacks. The used processor cores lack necessary hardware-support to enforce the correct execution of instructions and the utilized memory subsystems fail to provide the desired confidentiality and authenticity properties.

Motivation and Related Work

To provide additional context, the following section elaborates in more detail on the challenges in terms of security that modern devices face. In particular, we introduce the most relevant nomenclature as well as attack strategies and briefly summarize currently deployed countermeasures.

Software Attacks

The term software attack, in this thesis, denotes all attack types that solely rely on the functional behavior of a system and can exclusively exploit bugs (e.g., missing or wrong checks) in the implementation and design. In our attack model for software attacks, we assume that the targeted device is operated in a protected environment under nominal conditions and is behaving according to its functional specification. Physical access and interaction with the system is permitted but restricted to its official input and output interfaces. As a result, both local (e.g., via a tampered USB stick), as well as remote exploitation over the Internet, is possible in a software attack. Moreover, functional system simulation (e.g., in a virtual machine or instruction set simulator) is sufficient to evaluate and reproduce such a software attack. The basic goal of attackers that perform software attacks is to alter a device's behavior, or even to gain full control over the device.

Software attacks are currently the prevailing attack type for embedded and mobile devices, as well as for workstations and servers. Considering that software bugs are a very common problem, a wide range of attack techniques and countermeasures have been developed over time.

Historically, code-injection attacks have been the most direct approach to achieve arbitrary code execution. However, since Data-Execution Prevention (DEP) is supported on most common platforms using the No-eXecute (NX) bit of the Memory-Management Unit (MMU), code-injection attacks are hardly possible anymore. Code-reuse attacks that alter the control flow of a program, like for instance return-to-libc [Ner01; Sol97], Return-Oriented Programming (ROP) [Sha07], Jump-Oriented Programming (JOP) [Ble+11], and Counterfeit Object-Oriented Programming (COOP) [Sch+15], have been found as effective

alternative attack techniques. As countermeasures against code-reuse attacks, mainly cheap probabilistic methods like Address Space Layout Randomization (ASLR) and stack canaries [Cow98] are widely used. More sophisticated techniques based on Control-Flow Integrity (CFI) [Aba+09], on the other hand, are often too coarse grained [Car+15; Dav+14; Gök+14] or too expensive to gain adoption. Moreover, even when the control flow is properly enforced, data-only attacks like Data-Oriented Programming (DOP) [Hu+16] are still possible.

Taking a step back and looking at the software security problem more analytically shows that the key to basically every commonly used software attack technique is the lack of memory safety [Sze+13]. The main reason for this problem is that C and C++, the most widely used system programming languages, do not enforce memory safety by default. Unfortunately, also retrofitting full memory safety, using, for example, a combination of SoftBound [Nag+09] and Compiler Enforced Temporal Safety (CETS) [Nag+10], seems to be too slow for widespread adoption. Partial memory safety, like provided by SafeStack, Code-Pointer Integrity (CPI), and Code-Pointer Separation (CPS) [Kuz+14], can therefore be used as compromise between performance overhead and protection.

Modern processor architectures currently only give little support to address the memory safety issue effectively. Intel's MPX is the only widely deployed hardware extension which supports, for example, bounds checking with fine granularity. However, performance-wise the advantage of using MPX compared to software-only approaches is still negligible [Ole+17]. The currently predominant software security approach supported by hardware are Trusted-Execution Environments (TEEs) [Mae+18] like Intel's SGX [Hoe+13; McK+13] and ARM's TrustZone [ARM09]. However, these TEEs are more or less comparable to yet another privilege level of the processor and do not address the original memory safety problem. Subsequently, using such a TEE requires to partition an application into secure and insecure parts but does not prevent attacks within the individual partitions [Bio+18; Lee+17; SWG19].

Large software vendors therefore approach the problem of software security by implementing bug bounty programs (e.g., Facebook [Fac], Google [Goob], Microsoft [Mic]) that encourage users to search and responsibly disclose problems in their products. Subsequently, security updates are provided based on the reported vulnerabilities, given that the respective product is still supported by the manufacturer. Unfortunately, this is not always the case. Especially in the smart phone sector (e.g., Android) the lack of longtime support by many vendors is a serious issue, which leaves a multitude of devices [Gooa] deliberately vulnerable to software attacks [Det].

Devices which are built solely on open-source software often do not have this problem, assuming that a strong community supports the respective gadget. However, also open-source software has its issues. Although many vendors rely on free software components in their products, comparably little resources are invested in maintenance and security audits by the respective commercial users. This is critical given that even a single bug in a commonly used open-source library can have a devastating impact. The simple buffer over-read CVE-2014-

0160 [Cor13] in the OpenSSL library, which is more widely known as Heartbleed, is a prime example for this problem. Interestingly, since Heartbleed, several other security vulnerabilities have been branded with fancy names and logos (e.g., Shellshock [Cor14], Dirty COW [Cor16]). However, note that the risk potential of a vulnerability is not necessarily related to its media coverage. Many other vulnerabilities, based on sometimes even more dangerous but less known bugs, can be exploited too.

In summary, independent of the specific reasons, regular reports on data breaches (e.g., LinkedIn [Sil12], Sony [Cor11; Pic14], Yahoo [CIS16], Equifax [Off18], Marriott [Roo19]), large scale cyber-attacks (e.g., DDoS against Dyn [Inc16] using the Mirai [Ann16] botnet), and hacked devices (e.g., iOS jailbreaks [Wik], WiiU [fai14], Playstation 4 [fai15]) clearly show that the problem of secure software in the context of software attacks is far from solved on current architectures.

Physical Attacks

Unfortunately, having correct software is only a necessary but not sufficient requirement for building an overall secure system. For correctness, software relies on the fact that it is executed by the processor exactly in the intended way. Under normal operation conditions, as in our software attack model which idealizes the hardware, this is assumed to be true despite the fact that hardware bugs are common too. Unfortunately, the situation changes dramatically as soon as an adversary has physical access to the respective device. In such a potentially malicious environment, an attacker can perform physical attacks by tampering with the device in every possible way.

In this thesis, we use the term physical attack to denote attack types that exploit this reality and take into account that hardware is far from ideal in practice. The physical attack model complements the previously defined software attack model by extending it with unrestricted access to a device including its internal building blocks and its physical properties (e.g., timing behavior, power consumption, all kinds of emanation). Adversaries that operate within this model are, in the strongest interpretation, only limited by physics and their own abilities—they are allowed to perform any measurement/modification they can achieve. However, unrestricted capabilities are far too strong in practice. Researchers, therefore, often focus on finding solutions in more realistic subsets of the model by restricting the attack capabilities to certain operations (e.g., reading, writing), and by introducing security boundaries (e.g., signals on a chip cannot be arbitrarily probed).

Note also that software attacks, which operate in a subset of our physical attack model, are commonly not considered to be physical attacks. They are considered as an individual attack class distinguished by the type of exploited vulnerabilities or the needed type of access to the target—classical physical attacks require physical access whereas software attacks do not. However, these simple classifications fail to be sufficient given that more recent attacks manage to introduce/exploit similar effects as classical physical attacks using software.

Considering that also the needed countermeasures are similar, we consider these new attack types to be physical attack variants, denote them as software-controlled physical attacks, and selected our attack models accordingly.

In literature, physical attacks are often restricted/distinguished based on the characteristics of the performed attacks [Sta10] (e.g., active vs. passive, invasive vs. non-invasive) or using special nomenclature (e.g., fault attacks, side-channel attacks). Fault attacks are, for instance, active physical attacks that intentionally violate presumed invariants of a device to introduce faults into the performed operations. Commonly used attack vectors for injecting faults are, for example, supply voltage [Aum+02], maximum frequency [AK97], the allowed temperature range [Sko02], and the tolerated amount of injected photoelectric [SA02] or ElectroMagnetic (EM) [Sam+02] energy. Especially in the context of processors and software execution, fault attacks have been shown to be very powerful. Adversaries can, for example, use precisely timed voltage and clock glitches to deterministically skip and repeat instructions [KH14; KSV13].

Countermeasures [Bar+04] against fault attacks, are typically either sensor or redundancy based. Hardware sensors try to detect the event of the fault injection. Adding sensors is easy but has the disadvantage that unorthodox injection techniques, for which no sensors have been integrated, can stay undetected. Redundancy-based schemes, on the other hand, try to detect or even correct the actual fault, *i.e.*, the effect of the injection. Therefore, redundancy-based approaches work against any potential fault injection technique. In theory, approaches which are also used for soft-error detection [Gol+06] can be used in the context of fault detection as well. However, it has to be noted that, unlike random soft errors, fault attacks can be very controlled [Sel+15]. Simple approaches like direct duplication are therefore not sufficient. Furthermore, implementations in pure software are also hardly possible given that not a single instruction can be executed reliably without special protection.

Side-channel attacks [Sta10] are also attack types that are typically considered as passive physical attacks. The basic idea of side-channel attacks is to infer information about internally processed data solely by observing related externally visible signals during the operation. In the simplest case, this external signals can be some data bus over which confidential information is transmitted into another microchip. However, as soon as cryptography is used, most probably power consumption [KJJ99] or EM radiation [Agr+02] are observed. Using the captured information, subsequently, the internal state is revealed using statistical techniques and signal processing [MOP07]. As countermeasures against side-channel attacks, typically various types of hiding [MMS01] (reduces the signal-to-noise ratio) and masking [GP99] (splits and randomizes information into shares) are used.

Unfortunately, state-of-the-art processor architectures feature very little protection against physical attacks. In fact, only the latest high-end Intel (as part of SGX [Gue16]) and AMD (SME [KPW16]) processors consider physical attacks at all. Both companies recognize the previously described trust problem in cloud scenarios and support transparent RAM encryption. This is sufficient to address, for example, cold boot attacks [Hal+08]. However, fault attacks and side-channel

attacks, on the other hand, are still not really considered a problem by most vendors. Subsequently, other physical attacks like, for instance, the reset glitch attack on the Xbox 360 [fre] or the NAND mirroring attack on the iPhone [Sko16] are still widely applicable.

Software-controlled Physical Attacks. To make matters worse, recent attacks even show that mounting a physical attack does not necessarily require physical access to a device.

Rowhammer-based [Kim+14] fault attacks, for example, exploit that fast and highly controlled accesses to selected memory cells in modern Dynamic Random-Access Memory (DRAM) can cause bit flips in neighboring memory cells. After reverse engineering the involved mapping functions, adversaries can craft access patterns that trigger this behavior to inject faults into DRAM. Note that such attacks do not necessarily require native [SD15] code execution capabilities but can also be mounted remotely using JavaScript [GMM16] or via the network [Lip+18a; Tat+18]. Also devices (e.g., a Field Programmable Gate Array (FPGA) [Wei+19b]) in the same System-on-Chip (SoC) can be used for fault injection and DRAM with error-correcting codes is vulnerable [Coj+19] too. Finally, these faults can be used to mount Denial-of-Service (DoS) attacks on SGX [Jan+17] and for escalating privileges by modifying page tables [SD15] or instruction encodings [Gru+18]. In terms of countermeasures [Lou+19], using more conservative refresh rates as well as probabilistically refreshing neighboring rows [KNQ15] appear to be the most reliable approaches.

Other examples for software-controlled fault attacks are the recent voltage (e.g., Plundervolt [Mur+20], V0LTpwn [Ken+19], Voltjockey [Qiu+19a; Qiu+19b]) and clock (e.g., CLKSCREW [TSS17]) glitch attacks against TEEs. These attacks exploit that TEEs, like SGX and TrustZone, should be secure even with a compromised Operating System (OS) running in ring 0 on x86 or in privilege level 1 on ARM. However, operating systems are typically in charge of controlling the Dynamic Voltage and Frequency Scaling (DVFS) [Wei+94] capabilities of modern processors. An attacker with control over the OS can, hence, misuse the DVFS for overclocking/undervolting the processor to inject faults into TEE computations.

Side-channel attacks [Spr+18] are a rather active research field of the software-security community too. Modern mobile devices feature a multitude of different sensors (e.g., accelerometer, gyroscope, ambient light) that can be accessed from software and facilitate sensor-based side-channels attacks. Such attacks, for instance, permit to track a user by fingerprinting sensors [Dey+14], to infer the location of a device [Han+12], and to log key as well as pin inputs [CC11; Spr14] without requiring any special privileges. Typical approaches to counter sensor-based side-channels attacks [Spr+18] are stricter permission systems and limiting the sampling frequency. However, the overall problem is still unsolved given that both strategies only make exploitation harder at the cost of reduced usability/functionality for genuine applications.

Finally, in the context of processors and software execution, timing-based

side-channel attacks [Koc96] have to be mentioned. Cache attacks, probably the most famous type of software-controlled timing side-channel attacks on processors, were extensively studied by the community in recent years resulting in a sizeable portfolio of attacking techniques [Ber05; GBK11; Gru+16b; OST06; YF14] and countermeasures [WL07; WL08]. Unfortunately, industry opted to ignore these types of attacks for a long time due to the cost of the mitigations. However, recent advances in transient execution attacks (e.g., Meltdown [Lip+18b], Spectre [Koc+19]), which often exploit cache covert channels for data exfiltration, definitely increased interest in cache attack countermeasures.

Contribution and Outline

In this thesis, we tackle the problem of secure and trusted software execution in malicious environments that include physical access to a device. While the primary focus of our work is the prevention of physical attacks, also software attacks are mitigated whenever possible. To make progress towards the ambitious goal of building more secure systems, our approach is to augment contemporary system architectures with minimal hardware extensions. These hardware extensions serve as security-anchor on which further countermeasures can be built efficiently in software. Following this methodology, we developed several techniques that can be used to build efficient general-purpose computing architectures that are hardened against physical tamping.

We structured our contributions into two main parts based on to the protection the individual approaches provide. Part I focuses on extensions to the core of a processor, which ensure that code is always executed in the correct sequence. These extensions either prevent the successful execution of wrong code by design or permit to establish trust into a device by making its software execution locally and/or remotely attestable. In Part II, on the other hand, protecting arbitrary code and data against direct modification as well as indirect exposure (e.g., via side-channel attacks) are the central topic. The approaches in this context are mainly modifications to the memory subsystem and can be deployed completely independent of our processor core extensions.

The following paragraphs outline the contributions in the individual parts and chapters in more detail. Additionally, an illustration of the presented techniques in relation to the affected attack classes is provided in Figure 1.1. Note, however, that a connection between a scheme and an attack category does not necessarily imply that all attacks within the respective category are mitigated.

Part I: Providing Control-Flow Integrity and Attestation. In Chapter 2, we present our first contribution [WWM15] on code protection by bringing CFI into the context of fault attacks. For this work, we analyzed requirements for successful fault detection and evaluated various building blocks according to the determined criteria. The resulting countermeasure is effectively an extension of an old soft-error detection technique into the security context. Last but not least, we have shown the practicability of the concept using a demonstrator

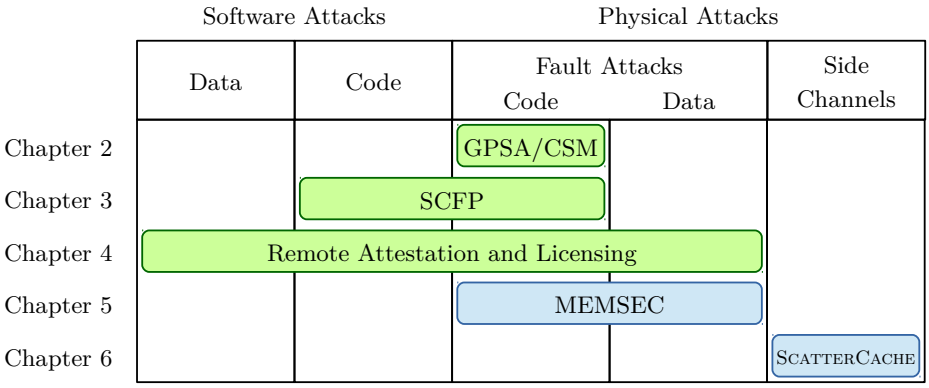


Figure 1.1: Graphical mapping between the presented techniques and the attack classes which are affected by the countermeasure.

that comprises a marginally modified ARM Cortex-M3 clone and a suitable C compiler toolchain based on LLVM.

Our second contribution [Wer+18], presented in Chapter 3, builds on our previous work and extends the fault attack countermeasure with sponge-based Authenticated Encryption (AE) techniques. By doing so, we gain a code protection scheme, named Sponge-Based Control-Flow Protection (SCFP), which not only is able to detect fault attacks but also delivers a configurable amount of confidentiality and authenticity. SCFP with a sufficiently large sponge, furthermore, also can be used to protect code against software attacks. This is possible given that SCFP is actually very similar to CFI techniques that are used to prevent code-reuse attacks. To demonstrate the viability of the concept, we extended a RISC-V processor with SCFP support and developed a corresponding LLVM-based toolchain.

Finally, in Chapter 4 we describe how schemes like SCFP, which decrypt code in execution order, can be used as foundation for remote attestation schemes that establish trust in computations performed on remote compute nodes. In this chapter we show that common static and path remote attestation schemes can be built purely in software using SCFP as hardware security anchor. Additionally, by taking advantage of the inherent balancing needed for encrypted code execution, a novel graph attestation mode as well as an online licensing approach are proposed.

Part II: Counteracting Physical Attacks on the Memory System. In Chapter 5, we present our first contribution [Wer+17] in the data protection domain—MEMSEC, an open-source framework for building transparent memory encryption and authentication modules. The developed encryption/authentication modules feature an AXI4 bus interface and can be used in FPGA and Application Specific Integrated Circuit (ASIC) designs to protect data in memory against tampering. The framework is written in VHDL and enables easy prototyping of different encryption and authentication modes. As ciphers,

AES, Prince, and Ascon are used in different modes (e.g., ECB, CBC, XTS) of operation. Evaluation has been performed under real-world conditions using a Xilinx Zynq SoC FPGAs where the entire memory traffic of ARM processors running Linux is processed.

Last but not least, in Chapter 6, we elaborate on our cache attack countermeasure ScatterCache [Wer+19b]. In the novel ScatterCache design, we combine lightweight cryptographic primitives with per-way addressing logic, which yields a randomized and skewed set-associative cache. The resulting cache is drop-in compatible with existing set-associative caches, but has the added benefit of complicating eviction-based attacks considerably. The design additionally features an optional software-controlled tweak, *i.e.*, the Security Domain IDentifier (SDID), that enables ScatterCache-aware operating system to control if data, while shared memory in RAM, should also be shared in the cache. Performance evaluation has been performed using the gem5 full system simulator and a custom-made cache simulator.

Finally, we conclude this thesis in Chapter 7 by summarizeing our achievements, discussing opportunities for future work, and by presenting all our (co-)authored publications.

Part I

Providing Control-Flow Integrity and Attestation

Executing code correctly is next to impossible on contemporary processors when physical attacks are considered given that even a single bit flip is sufficient to completely change the semantic of a program. Furthermore, neither local nor remote support for retrospectively detecting if any tampering has been performed is available.

We address these two problems in this first part of the thesis and present two processor core extensions that provide fine grained CFI. Additionally, building upon the second CFI scheme, a novel remote attestation technique is discussed. The content of Part I is largely based on two publications [Wer+18; WWM15] that have been extended based on two internal manuscripts. The enumeration below maps the individual chapters to the respective papers, clarifies my contributions, and acknowledges the work of my collaborators.

- Chapter 2 is primarily based on the following publication that was presented at CARDIS 2015 in Bochum (Germany):

  Mario Werner, Erich Wenger, and Stefan Mangard. "Protecting the Control Flow of Embedded Processors against Fault Attacks." In: *Smart Card Research and Advanced Applications – CARDIS*. 2015, pp. 161–176. DOI: 10.1007/978-3-319-31271-2_10

  I am the main author of this paper, wrote the majority of the text, performed all experiments, and implemented the GPSA hardware extension as well as all software components. Erich Wenger contributed to the text and programmed the used ARMv7-M processor along with Thomas Unterluggauer. Stefan Mangard supplied the original idea and supported the project in many discussions. Conceptually, this paper is a followup to my master's thesis but shares neither the implementation nor text.

- Chapter 3 is primarily based on the following publication that was presented at EuroS&P 2018 in London (United Kingdom):

  Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. "Sponge-Based Control-Flow Protection for IoT Devices." In: *European Symposium on Security and Privacy – EuroS&P*. **Best Paper Award**. 2018, pp. 214–226. DOI: 10.1109/EuroSP.2018.00023

  Again, I am the main author of this paper and developed the main concepts (e.g., instruction stream encryption, software handling, RISC-V ISA extension). Furthermore, I wrote the majority of the text, performed all experiments in HDL simulation, and developed the core components of the software toolchain. Thomas Unterluggauer contributed to the project with text and, together with Stefan Mangard, in many productive discussions. David Schaffenrath designed *Remus*, our RISC-V processor with SCFP support, as part of his master's thesis at the Institute for Integrated Systems (IIS) at ETH Zurich. Therefore, also Frank K. Gürkaynak, Germain Haugou, Beat Muheim, and Prof. Luca Benini—David's supervisors and colleagues at ETH Zurich—contributed to the HDL design too. Finally, Robert Schilling supported me in maintaining the software toolchain, and

Florian Mendel and Christoph Dobraunig provided guidance on the involved cryptographic primitives and modes.

Additionally, from an internal manuscript, chip area and power consumption results for the *Patronus* ASIC were added to this thesis. The area results have been provided by Frank K. Gürkaynak and the power measurements were performed by Alfio Di Mauro, both from ETH Zurich.

- Chapter 4 is entirely based on an unpublished manuscript of which I am the main author. I contributed the main part of the text, the idea for graph attestation and licensing as well as the HDL simulation results. Thomas Unterluggauer contributed to the text, the software implementation and the analysis.

2

Protecting the Control Flow of Embedded Processors against Fault Attacks

Fault attacks are a very active field of research since the seminal publication of the Bellcore attack [BDL97] in 1997. The basic idea of such a fault attack is to operate devices outside their specified operation conditions in such a way that faults occur during critical operations. For instance, unless countermeasures are implemented, observing only few faulted executions of a keyed cryptographic primitive can already be sufficient to reveal the used secret key. A comprehensive introduction to fault attacks and countermeasures for cryptographic primitives can be found in [JT12].

However, while most of the research on fault attacks focuses on attacking and securing cryptographic primitives, it is important to point out that securing a cryptographic primitive is not sufficient to secure a system. For example, the Xbox 360 has not been compromised because of a fault attack on a cryptographic primitive. It has been attacked successfully because it was possible to use a glitch on the reset line to make the system bypass the signature check of the loaded software [fre]. In case of such attacks on the control flow of the executed software, often a single successful fault induction is sufficient to compromise the security of a system completely (e.g., by branching to an administrative function, by obtaining root privileges, or by skipping all kinds of security checks). Attacks on the control flow also allow to bypass certain countermeasures for cryptographic computations. In [WWM11] for example, techniques for multiple fault inductions are discussed to first induce a fault in a cryptographic computation and then to bypass the comparison with a redundant computation.

When we started to work on this topic, only very few publications dealt with the challenge of securing the control flow of software against fault attacks and none of them seemed to be suitable for wide deployment. The software-based approaches [LHB14; Sch+10], for instance, either only allowed to detect integrity violations at a coarse level of granularity—where some modified, missing, or repeated instructions stay undetected—or introduce huge overheads. The hardware-based approaches [Aro+06; RCS02; RS05] looked more promising but still incur significant complexity and overheads (e.g., one signature per basic block)—in particular when larger programs are protected.

Contribution. In this chapter, we discuss how we improved upon related work in this regard and present an efficient hardware-supported technique to ensure control-flow integrity, even in the presence of fault. Our technique builds upon Generalized Path Signature Analysis (GPSA) that has been introduced in the context of soft errors by Wilken and Shen [WS88; WS90]. We, therefore, investigate the requirements for fault detection in the setting of fault attacks and adapt the scheme of Wilken and Shen accordingly. Furthermore, we present an implementation of the resulting countermeasure using state-of-the-art hardware (*i.e.*, an ARM Cortex-M3 compatible processor) and software (LLVM compiler infrastructure).

To the best of our knowledge, this work is the first to actually implement a Control-Flow Integrity (CFI) scheme based on GPSA. Our prototype implementation is primarily software-oriented, increases the processor size by merely 6.4 %, and detects every fault on the instruction-stream with 99.9 % probability within three cycles. The runtime overhead of the protected applications ranges from 2 % to 71 %. The majority of this overhead is caused by our software-heavy handling of conditional branches. Subsequently, cryptographic primitives, which typically have a low number of conditional branches, can be protected with comparably low costs.

Outline. This chapter is organized as follows. Section 2.1 gives an introduction on control-flow integrity and the existing work of Wilken and Shen. Section 2.2 presents how we adapt the scheme to the setting of fault attacks. Our prototype implementation is discussed in Section 2.3 and the corresponding evaluation results can be found in Section 2.4. Finally, the chapter is concluded in Section 2.5.

2.1 Control-Flow Integrity in Fault-Tolerant Computing

The detection of faults in the control flow of a program requires to include redundant information about the control flow into the program. The concept of GPSA by Wilken and Shen [WS88; WS90] is a very efficient technique to add this redundancy. In the following subsections, we first define the problem of control-flow integrity and then discuss the concept of GPSA.

2.1.1 Control-Flow Integrity

The control flow of a program refers to the order in which its instructions, branches, and function calls have to be executed. Two instruction types can be distinguished in this context. First, sequential instructions, like arithmetic and memory operations, that only have indirect influence on the execution sequence. They are executed in strictly sequential order and have exactly one subsequent instruction. Second, control-flow instructions, like `branch` and `call`, that alter the execution sequence directly. Control-flow instructions have one or more subsequent instructions and can select which one is executed next.

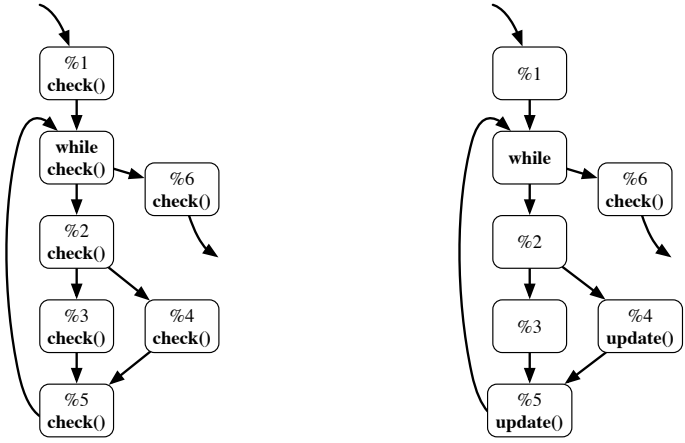
A program is typically structured into code fragments which consist out of an arbitrary number of sequential instructions (zero or more) followed by up to one control-flow instruction. Such fragments are denoted as basic blocks. A basic block is a strictly sequential piece of code which can only be entered at the first and exited after the last instruction. All basic blocks of a program form the so-called Control-Flow Graph (CFG). The edges in a CFG are always directed and describe in which way the control flow can be transferred from one basic block to another. Ensuring CFI during the execution of a program means that all instructions in a basic block are executed by the processor as defined in the original program (*i.e.*, no instructions are skipped or altered) and that no new connections are added to the control-flow graph (*i.e.*, no other branches are done than those defined at compilation time).

Control-flow integrity does not include the protection of the decision which path is taken in a CFG. This requires protecting the integrity of the data that is used for the decision. However, it is important to note that data integrity cannot be achieved without control-flow integrity. CFI is the basis for further countermeasures, like data integrity. For example multiple computations and comparisons can be done to ensure data integrity and the techniques for CFI make sure that all these operations are indeed executed by the processor.

2.1.2 Derived Signatures

Derived signatures are a common technique in fault-tolerant computing to detect violations of the integrity of the control flow. The basic idea of a derived signature is to add a small piece of hardware to the processor executing the software that should be protected. Upon the execution of each instruction, the hardware updates a checksum based on the executed instruction and the corresponding control signals of the decoder. In the literature on CFI in fault-tolerant computing, such a checksum is called “derived signature”. It is important to note that it is not a cryptographic signature. Nevertheless, in order to be consistent with the existing literature, we also use the term derived signature to denote a checksum that is calculated in hardware based on a sequence of executed instructions.

In order to check such a derived signature when a program is executed, it is necessary to have corresponding reference values. Derived signatures depend on the executed instructions and the initial value of the signature. As both are known at compilation time, reference values can be calculated when a program



(a) Basic block checking **(b)** GPSA updates + checking

Figure 2.1: Signature based checking methodologies.

is created. Typically, the reference values are embedded into the program by instrumenting the binary, either during compilation or in a post-processing step.

Derived signatures can for example be checked at the end of every basic block. This is shown in Figure 2.1a. The figure shows a control-flow graph with six basic blocks, labelled %1 to %6 that include a while loop. At the end of each basic block a signature check is done and therefore a reference value for each basic block has to be added to the program. This is for example done in [Aro+06], [RCS02] and [RS05]. However, this leads to a significant overhead, which can be avoided when using generalized path signatures. Furthermore, there is no protection for the connections of the basic blocks.

2.1.3 Generalized Path Signature Analysis

In [Nam82], the so-called Path Signature Analysis (PSA) has been introduced. PSA checks the integrity not only for a basic block, but along paths through a control-flow graph. This significantly reduces the overhead. Wilken and Shen in [WS88; WS90] extended PSA into GPSA in order to optimize the overhead.

The basic idea of GPSA is to insert signature updates into the program code in such a way that independent of the used paths in a CFG, the signature value at a given instruction is always the same. This idea is illustrated in Figure 2.1b. At the end of basic block %4, there is an update that makes sure that the signature at the beginning of %5 is the same independent of the fact whether it is reached via %3 or %4. The update at the end of %5 ensures that the signature at the beginning of the while loop is independent of the fact whether it is reached via %1 or %5. The values that need to be stored in the program code to do the updates are called justifying signatures [WS88] and they are calculated at compilation time—just like the reference values for the checks.

The concept of GPSA optimizes the number of total justifying signatures in a CFG and can also be extended to protect function calls. In case of function calls, there is an additional justifying signature necessary for each function call. For details, please refer to [WS88].

GPSA does not require to have a check in every basic block and allows to place signature checks at arbitrary positions in the program. These checks are denoted as vertical signature checks. At minimum, it is necessary to insert one signature check at the end of the program as it is done in Figure 2.1b. Depending on the application, a trade-off has to be made between between runtime and memory overhead on the one hand and the detection latency on the other hand.

2.1.4 Continuous-Signature Monitoring

Wilken and Shen proposed Continuous Signature Monitoring (CSM) as an alternative concept to the manual placement of signature checks and to solve the latency problem of vertical signature checks. The idea of CSM is to check the signature, or at least parts of it, on every executed instruction. Implementing CSM on top of GPSA is therefore as simple as checking h bits of the $|S|$-bit runtime signature on every executed instruction.

It has been proposed to use spare bits in the instruction encodings or to embed reference information into the error-correction/detection bits of the memory system. However, these approaches are not applicable to most modern processor architectures given their dense instruction encodings and the lack of error detecting memory.

2.2 Control-Flow Integrity in the Setting of Fault Attacks

Fault attack detection is per se very similar to the detection of soft errors. The main difference between the two is the fault model. Soft errors occur randomly at a low frequency. Fault attacks on the other hand can be very controlled. When comparing different checksums for derived signatures, it is therefore important to not only look at average detection probabilities but to also keep the worst case scenario in mind.

This section elaborates on the required properties that are needed in order to make the schemes of Wilken and Shen ready for fault attacks. We define functional requirements for both, the signature and the update function, which make single faulty instructions detectable with certainty. We further show that the actual function selection has an huge impact on the detection capabilities. The best of the evaluated functions can detect up to 7 faulty bits in the instruction stream across two cycles with certainty.

2.2.1 Signature Function Selection

The calculation of derived signatures can be modeled using a compression function f which is used in a Merkle-Damgård-like mode of operation. The next signature $S_{j+1} = f(S_j, I_j)$ is calculated based on the preceding signature S_j and the current instruction I_j. Collisions across multiple iterations of the signature function are unavoidable given that the signature value S has fixed size. However, choosing a signature function with specific properties can at least provide certain worst-case guarantees.

Functional Requirements

The signature function f needs the following properties in order to make a single faulty instruction $I_j \oplus \Delta_{I_j}$ detectable with certainty, independent of the actual error Δ_{I_j} and the number of faulty bits $HW(\Delta_{I_j})$.

- *Reliability*: Every error in the instruction stream ($\Delta_{I_j} \neq 0$) has to result in a signature error ($\Delta_{S_{j+1}} \neq 0$) given that the original signature was correct ($\Delta_{S_j} = 0$). Note that this requirement can only be fulfilled if $|S| \geq |I|$.

$$S_{j+1} \oplus \Delta_{S_{j+1}} = f(S_j, I_j \oplus \Delta_{I_j}), \quad \forall \Delta_{I_j} \neq 0 \rightarrow \Delta_{S_{j+1}} \neq 0 \tag{2.1}$$

- *Error preservation*: An error, absorbed into the signature $S_j \oplus \Delta_{S_j}$, must not be eliminated by an error-free sequence of inputs ($\Delta_{I_j} = 0$). This requirement allows to arbitrarily delay the checking of a signature. Consequently, the number of necessary signature checks can be reduced.

$$S_{j+1} \oplus \Delta_{S_{j+1}} = f(S_j \oplus \Delta_{S_j}, I_j), \quad \forall \Delta_{S_j} \neq 0 \rightarrow \Delta_{S_{j+1}} \neq 0 \tag{2.2}$$

- *Non associativity*: The order in which instructions I_j, I_k are absorbed by f must have an influence on the resulting signature value.

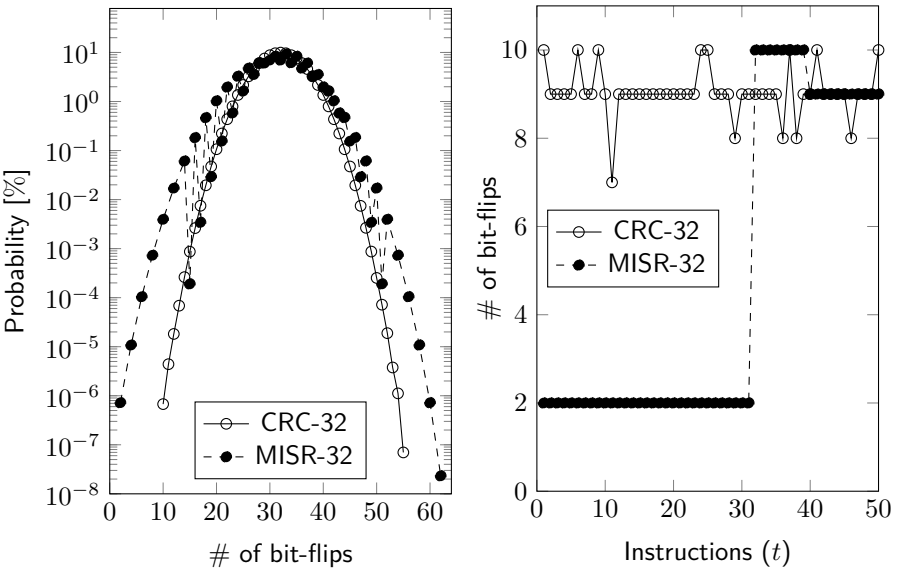
$$\forall I_j \neq I_k \rightarrow f(f(S_j, I_j), I_k) \neq f(f(S_j, I_k), I_j) \tag{2.3}$$

- *Invertibility*: Depending on the concrete implementation of the scheme, invertibility may also be a requirement. The signature function should therefore be invertible in S given S_{j+1} and I_j. Our implementation for example uses this property to be able to place signature updates at arbitrary places along a path through the CFG. A different implementation, which enforces that signature updates are only performed at merging points in the CFG, would be able to cope without this property.

$$S_j = f^{-1}(S_{j+1}, I_j), \quad \forall S_{j+1}, \forall I_j \tag{2.4}$$

Choosing the Signature Function

Classical choices for checksums in the setting of fault-tolerant computing are Cyclic Redundancy Check (CRC) and Multiple-Input Signature Registers (MISRs)



(a) Probability density function for $q(j, 1)$. **(b)** Min. # of bit-flips for collision.

Figure 2.2: Comparison between CRC-32 and MISR-32.

with various polynomials. MISRs as well as CRCs fulfill the mentioned requirements. However, they are not equally suited when fault attacks with high control over the injected fault are considered.

For the evaluation of different signature functions, we evaluated the number of bit-flips required to introduce a fault on one instruction Δ_{I_j} and to compensate it with a fault on a subsequent instruction $\Delta_{I_{j+t}}$. The sum of the bit-flips required for both faults $q(j,t) = HW(\Delta_{I_j}) + HW(\Delta_{I_{j+t}})$ is a measure for the attack complexity. The quality function q has been chosen in this way to take into account that exact knowledge of the injected fault is needed in order construct and subsequently inject the compensating fault. Average as well as worst-case performance is important when fault attacks are considered.

A comparison between the signature functions CRC-32 and MISR-32 (identical polynomial) based on the probability density function of $q(j,t)$ at $t = 1$ is shown in Figure 2.2a. CRC-32 as well as MISR-32 have an expected number of bit-flips of 32. The expected value for $q(j,t)$ in general is identical to the degree of the reduction polynomial for both MISR and CRC codes. Performance in the average case is therefore identical which makes them equally suited for soft-error detection.

The worst-case performance on the other hand is different. The comparison in Figure 2.2b ($min(q)\ \forall\Delta_{I_j}, \forall\Delta_{I_{j+t}}$) shows that the CRC-32 is superior to the MISR-32 regarding worst-case performance. The used CRC enforces that at least 7 bit-flips are needed in order to construct a collision. The MISR on the other hand can already be defeated using 2 bit-flips within the first 31 instructions.

This weakness is caused by the simple structure of the MISRs which makes them not suited as signature functions in the fault attack context. A more extensive comparison between various polynomials regarding worst-case performance can be found in Table 2.1.

2.2.2 Update Function Selection

The second function which is required in GPSA and CSM implementations is the so-called update function. This function is needed in order to balance the various paths through the control-flow graph. The update function u calculates the next signature $S_{j+1} = u(S_j, J_j)$ based on the preceding signature S_j and a justifying signature constant J_j. The update function has to fulfill the following requirements in order to be usable for GPSA.

- *Full control*: All possible signature values S_{j+1} have to be constructible given an arbitrary S_j and a justifying signature J_j. Note that, the size of J must be larger or equal to S ($|J| \geq |S|$) to modify each bit in S.

$$S_{j+1} = u(S_j, J_j), \quad \forall S_{j+1}, \forall S_j, \exists J_j \tag{2.5}$$

- *Error preservation*: An error, absorbed into the signature $S_j \oplus \Delta_{S_j}$, must not be eliminated by an error-free justifying signature ($\Delta_{J_j} = 0$). It would otherwise not be possible to arbitrarily delay the actual checking.

$$S_{j+1} \oplus \Delta_{S_{j+1}} = u(S_j \oplus \Delta_{S_j}, J_j), \quad \forall \Delta_{S_j} \neq 0 \rightarrow \Delta_{S_{j+1}} \neq 0 \tag{2.6}$$

- *Invertibility*: Given S_{j+1} and S_j, it must be possible to efficiently compute the justifying signature J_j.

$$J_j = u^{-1}(S_j, S_{j+1}), \quad \forall S_j, \forall S_{j+1} \tag{2.7}$$

A simple function which fulfills all those requirements is the binary `xor` function.

Table 2.1: Performance ($min(q)$ $\forall t = [1, 50], \forall \Delta_{I_j}, \forall \Delta_{I_{j+t}}$) of different polynomials. The polynomials are given in reversed representation.

Type	Polynomial	$min(q)$	t
CRC-8	`0xAB`	2	11
CRC-16-ARINC	`0xD405`	4	10
CRC-16-CCITT	`0x8408`	4	1
CRC-16-CDMA2000	`0xE613`	4	32
CRC-16-DECT	`0x91A0`	2	15
CRC-16-T10-DIF	`0xEDD1`	4	7
CRC-16-DNP	`0xA6BC`	2	9
CRC-16-IBM	`0xA001`	4	1
CRC-32	`0xEDB88320`	7	11
CRC-32C (Castagnoli)	`0x82F63B78`	8	2
CRC-32K (Koopman)	`0xEB31D82E`	6	34
CRC-32Q	`0xD5828281`	8	2
MISR-32	`0xEDB88320`	2	1

2.3 Prototype Implementation

We implemented GPSA and CSM on the basis of a state-of-the-art microprocessor architecture (an ARM Cortex-M3 compatible processor) and modern compiler technology (LLVM). The resulting implementation supports all C language features and common programming practices. Our prototype implementation is therefore not only a theoretic construct, but practically usable. The implementation supports separate compilation of C files and enables the use of static libraries. It also allows to randomize the signature values of identical programs (diversity) on different devices. This makes it harder to extend attacks against an individual towards multiple devices.

In this section, we discuss the necessary hardware modifications (which are minimal), the necessary modifications of the to-be-protected software, and elaborate on the modifications of the toolchain.

2.3.1 Hardware Architecture

The presented implementation is based on a clone of the ARM Cortex-M3 microprocessor architecture. Its performance-to-energy ratio makes this processor an interesting candidate for many embedded application areas, including smartcard applications. Hence, they are often used in malicious environments. The processor uses 3 pipeline stages to implement the ARMv7-M instruction set that supports both Thumb and Thumb-2 instructions.

As depicted in Figure 2.3, the processor was extended with a memory-mapped signature monitor which is tightly integrated into the design. This monitor automatically computes $|S| = 32$-bit signatures absorbing the 16–32-bit large Thumb and Thumb-2 instructions I_j. The CRC-32C code has been implemented as signature function based on the analysis presented in Section 2.2.1. Via the memory interface, the monitor enables the Central Processing Unit (CPU) to perform signature updates, signature replacements, and signature assertions. Assertion failures can either trigger an interrupt or reset the system.

To support the automatic computation of derived signatures, the CPU only had to be modified to forward the currently executed instruction to the monitor. To perform continuous signature monitoring, the fetch unit of the processor was modified. The signature bits are stored in a block at the end of the program. The base address of this signature block has been embedded into the interrupt vector table, similar like the initial stack pointer. At start-up, the base address is automatically initialized. During run-time, the fetch unit always loads the instructions in combination with the reference values. An instruction is only forwarded to the decode stage, once both the instruction and the reference value are valid.

2.3.2 Source Code Modifications

All signature modifications are performed in software, which in turn are monitored by the derived-signature monitor in hardware. Performing the necessary software

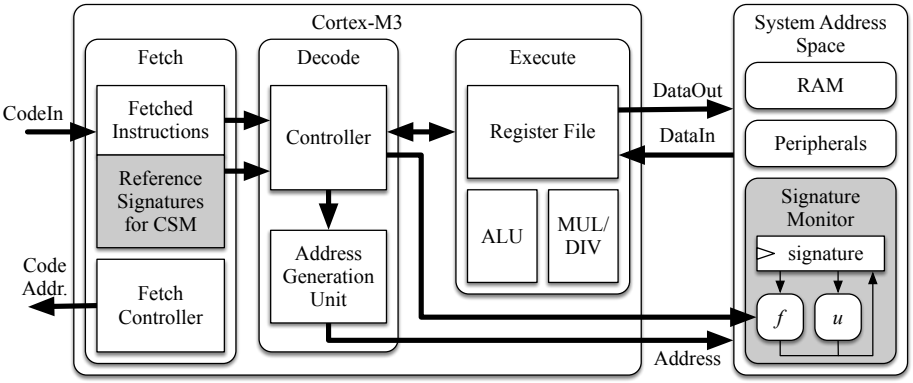


Figure 2.3: Simplified processor architecture with grey-shaded modifications.

transformations manually is a challenging and error prone task. It is clearly favorable to automatically perform the transformations within the tool-chain, which makes the whole instrumentation process transparent for the programmer. Consequently, modifications of the application C source code are minimal. In the best case, a to-be-protected software does not have to be modified at all.

The programmer can insert vertical signature checks as `assert_signature()` function calls into critical sections of the program. All remaining work is performed by the compiler which automatically replaces these function calls with actual signature checks. The use of function calls for the annotation has the advantage that `clang`, LLVM's C front end, can be used without any modification.

Assembly code on the other hand requires a little more work (as usual). The programmer has to place signature updates by hand when branches, loops, and function calls are encountered. However, no actual derived signature calculation has to be performed by the programmer. Additionally, if the programmer forgets a signature update, the toolchain will automatically notify her.

2.3.3 Software Modifications

Related work [Aba+09; RCS02; RS05; ZS13] usually performs the software transformations either during compilation or by applying a dedicated post-processing tool after linking. In this work, both techniques are combined in order to generate a protected executable. The compiler is responsible to insert signature updates based on GPSA and to insert signature assertions. A post-processing tool consecutively computes the derived signatures and patches the executable with signature update and reference values.

LLVM Compiler Modifications. The compiler has been built using the LLVM compiler infrastructure which already has great support for the targeted ARMv7-M architecture.

A machine function pass has been added to the ARM back-end in order to perform the following transformations:

- *Insertion of asserts*: Every call to the `assert_signature()` function is replaced by an actual vertical signature check. A signature check is performed as a memory-write operation of the expected signature to a certain pre-defined monitor address and is composed of three instructions. (LOAD address, LOAD value, STORE value)
- *Insertion of signature updates*: Signature updates are inserted to make the runtime signature independent of the executed path through the control-flow graph. The placement of signature updates is performed efficiently by computing the spanning tree of a function's undirected control-flow graph. Signature updates are, similar to checks, a write of the justifying signature to the memory mapped monitor.

The smart placement of the machine function pass in the optimization pipeline allows us to reuse much of the original compiler's functionality and therefore benefit from the available optimizations as well. Register allocation is for example still handled using stock LLVM functionality.

An additional component which had to be adapted is the run-time library. The compiler relies on its functions for standard operations (e.g., clearing memory) or to perform computations which are not natively supported by the processor. It was therefore necessary to instrument this library with justifying signature updates to generate a working GPSA-hardened program.

Post Processing Tool. As a result, the compiler generates a binary with all necessary signature updates/assertions that still lacks the correct signature constants. The signature values can only be computed once the program is linked and all instructions have been finalized. The compiler never has access to this information in a traditional separate-compilation design-flow. We therefore perform the derived signature calculation using a post-processing tool.

A recursive disassembling [ZS13] approach was used to recover the control flow and the location of the signature constants. LLVM's disassembling machinery simplifies this step considerably. Based on the control flow it is possible to identify the constant pools (*i.e.*, constant islands) in the binary. Tracking the monitor's addresses using data-flow analysis techniques consecutively reveals the location of the instructions which modify the signature values.

The actual calculation of the derived signatures relies on all this recovered information. The signature values are computed by initializing each function with a random initial signature, and consequently flooding the control-flow graph of each function. As a result, all justifying signatures, assertion constants, and reference signatures for the CSM are embedded into the executable.

Another feature of the post-processing tool is its static code analysis functionality of the binary. Only correctly instrumented binaries pass the derived signature calculation. Error messages notify a programmer about wrongly instrumented assembly code.

2.4 Evaluation

As this is the first published, practical implementation of both GPSA and CSM in the context of fault attacks, we are excited to report performance results based on qualitative characteristics as well as practical benchmarks.

2.4.1 Error-detection Coverage

Based on the previously stated requirements on the signature functions, every single fault on the instruction stream changes the runtime signature with certainty. Using vertical checks, any runtime-signature error can be detected. On the contrary, CSM checks h bits of the runtime signature per cycle. Therefore, the probability to detect an error is $1 - 2^{-h}$. As any error propagates within the signature register, the probability to detect an error is way beyond 99.9 % after 3 checks of $h = 4$ bits.

If an attacker targets two instructions, she could possibly hide the error by colliding the signature value. It was shown in Section 2.2.1 that the attacker has to flip 32 bits on average or 8 bits in his best case when a CRC-32C is used as signature function. Even using advanced attack setups, the probability for introducing a fault with precise bit-flips across multiple cycles is very low.

2.4.2 Error-detection Latency

An error can only be detected at the time of the vertical signature check when GPSA is used without CSM. It is up to the programmer to insert these vertical checks next to the critical pieces of code. This allows to perform very controlled checks and consecutively reduces overhead. However, it is possible that, due to bad check placement, vertical signature checks by itself detect an error once it already has been exploited.

CSM solves this problem given that it checks parts of the signature register after every executed instruction. With an increasing probability any error is detected after a few iterations.

2.4.3 Monitor Complexity

One of our design goals was to only introduce minimal hardware overhead. All operations beside derived signature calculation are performed entirely in software. We evaluated the monitor complexity after synthesis for UMC's 130 nm Low Leakage process using Cadence 2009 tools. The standard cell library for this process comes from Faraday. Without the monitor, our processor is 36,957 Gate Equivalent (GE)[1] large. Adding the monitor for GPSA increases the size of the processor by only 1469 GE, respectively by less than 4 %. Adding support for CSM additionally increases the size of the fetch unit which results in a total core size of 39,319 GE. The modifications to support GPSA and CSM therefore are minimal and account to merely 6.4 % hardware overhead.

[1] 1 GE conforms to the area of a 2-input NAND gate with driving strength 1.

2.4.4 Memory Overhead and Processor-Performance Loss

Memory overhead and processor-performance loss highly depend on the executed program. These characteristics are mainly determined by the number of branches, function calls, and vertical signature checks.

Qualitatively speaking, a single signature update costs around 10 bytes of memory and 6 cycles in our software-centered implementation. A function call costs around 14 bytes of memory and 10 cycles. Using CSM, the introduced redundancy is proportional to the size of the code within the `text` section of the executable. For $h = 4$ per 16-bit Thumb instruction, up to 25 % of redundant Non-Volatile Memory (NVM) has to be added.

For a quantitative, empirical evaluation, we tested multiple programs: a coremark benchmark (one iteration), an AES-256 roundtrip (encryption followed by decryption with check), and a 160-bit Elliptic-Curve Cryptography (ECC) example performing a scalar multiplication with optional ASseMbly (ASM) optimized finite-field arithmetic. The coremark benchmark has been optimized for speed (`-O2`) given that this yields the best performance. The crypto algorithms have been optimized for size (`-Os`). Additionally, link-time garbage collection (`-ffunction-sections -fdata-sections` and `-Wl,-gc-sections`) has been used to preserve only the absolutely necessary code and data segments. A synthesizeable VHDL model of the hardware, evaluated using Cadence NC Sim, has been used to execute the benchmarks.

The raw numbers and the relative overhead in terms of runtime as well as Random-Access Memory (RAM) and NVM size are summarized in Table 2.2. The evaluation was performed in two steps. First, our GPSA implementation is compared against the unmodified LLVM backend which is used as baseline. Second, CSM is compared with the GPSA version given that it extends GPSA's checking capabilities.

RAM. The RAM overhead of GPSA is below 10 % in all evaluated programs. For coremark it is even merely 3 %. This overhead is solely a side effect of the increased register pressure during function calls. The additional live variables force the compiler to spill more values and therefore slightly increase the memory usage on the stack. Using CSM on top of GPSA introduces no additional RAM overhead given that the code itself stays absolutely unchanged.

NVM. Overhead on the NVM side ranges from 29 % for the AES test case to 79 % for ECC. This overhead is composed of the actual signatures (justifying + reference) and the added code for signature updates and vertical checks. In this software-centered implementation, the majority of the overhead is code. The signatures account for 25 % NVM overhead at most.

The NVM overhead of CSM over GPSA on the other hand is purely signature based. Only minor optimization potential remains.

Runtime. The most remarkable figure in this evaluation is probably the runtime overhead. The overhead of GPSA ranges from 2 % for optimized ECC to

Table 2.2: Empirical Results for GPSA and CSM regarding RAM, NVM, and runtime overhead. Additionally, the NVM overhead solely for justifying and reference signatures is given.

Program	**RAM** Byte	**NVM** Byte	**Runtime** Cycle	**Justifying Signatures**	**Reference Signatures**
Baseline					
Coremark	2,444	9,384	547,294	—	—
AES-256	248	3,212	48,581	—	—
ECC	444	4,036	4,251,697	—	—
ECC w/ ASM	400	4,824	2,836,180	—	—
Overhead of GPSA (Relative to Baseline)					
Coremark[a]	2.3 %	69.0 %	56.7 %	23.5 %	0.1 %
AES-256[b]	9.6 %	29.0 %	36.7 %	10.8 %	0.5 %
ECC[c]	9.0 %	78.9 %	33.3 %	24.5 %	0.3 %
ECC w/ ASM [c]	8.0 %	53.5 %	1.9 %	16.3 %	0.2 %
Overhead of CSM with $h = 4$ bit (Relative to GPSA)					
Coremark[a]	—	22.2 %	8.9 %	—	22.2 %
AES-256[b]	—	19.3 %	6.5 %	—	19.3 %
ECC [c]	—	21.9 %	7.6 %	—	21.9 %
ECC w/ ASM [c]	—	22.6 %	0.4 %	—	22.6 %

[a] One vertical signature check before and one after the benchmark.
[b] One vertical signature check after every round of AES.
[c] One vertical signature check after every processed bit of the scalar.

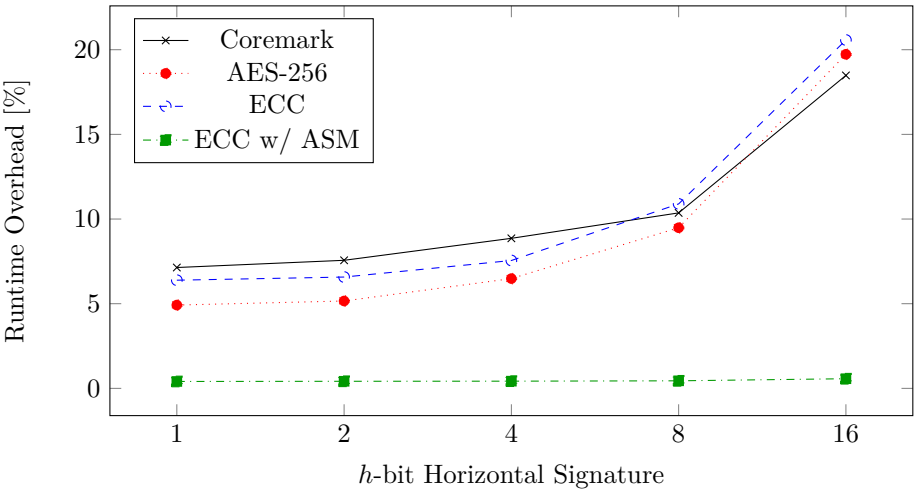


Figure 2.4: Runtime overhead of CSM with different horizontal signature sizes (h-bit). (Relative to GPSA)

57 % for coremark. The software-centered approach taken in this implementation is again one of the reasons for these high values. Each GPSA operation takes between 6 and 10 cycles. Adding more hardware support could bring this values down to around 2 cycles. However, even without additional hardware much better results can be achieved. The 31.4 % difference between the ECC programs show that there is still a lot of optimization potential on the compiler side as well. Implementations of cryptographic primitives should be protectable at hardly any cost given that their control flow is typically very sequential.

Enabling CSM on top of GPSA implies an additional overhead of up to 9 %. However, this is rather low considering that horizontal signatures with 4-bit (25 % redundancy per instruction) are used. Figure 2.4 shows how the runtime overhead of CSM scales in dependence of the horizontal signature size h. Most astonishing is probably that the overhead is still below 21 % even at 100 % redundancy (16-bit per instruction). The processor's Harvard architecture and the combination of 32-bit bus and 16-bit instruction set makes this possible.

2.5 Conclusion

We extended the CFI concepts of Wilken and Shen from the soft error to the fault attack context in this work. To achieve this goal we not only analyzed the functional requirements for derived signature calculation, but also performed an evaluation of actual signature functions. Using a CRC with suitable polynomial, any error in a single cycle and at least 7 bit-flips, spread across two cycles, can be detected with certainty.

We further practically implemented the derived signature based GPSA and

CSM techniques for a state-of-the-art processor. Additionally, a toolchain for this platform has been created utilizing the LLVM compiler infrastructure. This toolchain incorporates all necessary transformations and is completely transparent for the programmer. As a result, arbitrary C programs can be protected by simple compilation. The design's low hardware overhead and the good detection capability indicates that the combination of GPSA and CSM is well suited to protect the control flow in the context of fault attacks.

However, GPSA is capable of providing even stronger guarantees. In the following chapter, we extend the provided protection of GPSA into the software attack context. Moreover, we evolve CSM to not only check that the code and its control flow are genuine but to prevent the controlled execution of manipulated code in the first place.

3

Sponge-Based Control-Flow Protection for IoT-Devices

Fault attacks against processors, as thematised in the previous chapter, are a huge threat to any embedded device and need to be mitigated. However, modern Internet-of-Things (IoT) devices—ranging from consumer products in smart home environments, over sensor nodes in modern cars, to control units in critical infrastructures—face numerous additional security challenges that have to be addressed too. The prevalent Internet connection of IoT devices, for example, gives rise to remote software attacks on their exposed interfaces. Attackers can try to find and exploit software vulnerabilities in these interfaces to take control over IoT devices via code injection or code reuse attacks, like Return-Oriented Programming (ROP) [Sha07] and Jump-Oriented Programming (JOP) [Ble+11].

To prevent these types of software attacks, different countermeasures have been proposed, such as Data-Execution Prevention (DEP), return stack protection [Cow98; FPC09; Kuz+14], software diversification (e.g., Address Space Layout Randomization (ASLR) [PaX01; Sha+04]) and Control-Flow Integrity (CFI) [Aba+09]. To protect the authenticity and confidentiality of Intellectual Property (IP), encryption and authentication of software binaries and Random-Access Memory (RAM) can be used. Finally, to counteract physical fault attacks on the control flow of the processor, countermeasure similar to our Generalized Path Signature Analysis (GPSA) scheme (see Chapter 2) have to be deployed.

Unfortunately, current embedded devices hardly implement any of these countermeasures. Moreover, existing countermeasures work well for their original purpose in isolation, but for each of them, some of the attacks on IoT devices remain feasible due to the vast amount of different attack vectors. As a result, a variety of different countermeasures have to be deployed which can inhere

significant overheads that are impractical for lightweight embedded devices. Finally, the security analysis of combinations of countermeasures can also become highly complex.

Novel countermeasures, that by themselves can counteract a combination of software and physical attack vectors, are needed for such hostile environments. SOFIA [Cle+16; Cle+17b] was the first presented approach that fits this category of countermeasures. By encrypting, authenticating and chaining blocks of instructions using a stream cipher and Message Authentication Code (MAC), SOFIA yields CFI as well as confidentiality and authenticity of software. However, although SOFIA is reasonable efficient, memory and runtime overhead are far from optimal due to the execution-block oriented design. Also the reset triggered by the dedicated MAC verification—while beneficial for immediate error detection—is a single point of failure.

Contribution. As an alternative approach to SOFIA, we present Sponge-Based Control-Flow Protection (SCFP), a hardware-supported CFI scheme that enforces the confidentiality and authenticity of software at execution time. SCFP is based on our experience with GPSA and is well suited for IoT devices due to its low memory and runtime overhead. The involved hardware extension continuously decrypts and authenticates instructions at the latest possible point before the processor's decode stage. SCFP relies on cryptographic sponge constructions to encrypt and authenticate software binaries with instruction-level granularity.

The use of sponge-based authenticated encryption in SCFP yields fine-grained control-flow integrity and thus prevents code reuse attacks. By keeping the software encrypted throughout all memory, SCFP completely thwarts code injection attacks from within software, and effectively protects the IP of software vendors. By decrypting instructions right before the decode stage of the processor, SCFP resists tampering with code in memory, physical attacks on memory like rowhammer [GMM16; Kim+14], and fault attacks that manipulate control flow or instruction encodings. SCFP supports interrupt handling and is thus compatible with operating systems.

SCFP also offers strong fault resistance without requiring an explicit verification step that can potentially be bypassed using controlled faults. In particular, any globally induced physical fault on the processor chip destroys the internal SCFP state with high probability and leads to the execution of random instructions. Random code execution is a secure processor state, because it is hard to control and exploit for an attacker, and has a low probability of being meaningful. Nevertheless, timely detection of random execution is also supported.

SCFP is a highly flexible tool. We hence present two suitable sponge constructions as well as three different SCFP instances for different applications. First, Authentic-Encrypted Execution (AEE) provides all security features at cryptographic levels of security, *i.e.*, above 80 bits. Second, AEE-Light reduces memory overhead in trade for reduced software authenticity by using keyed permutations. Third, Infective Execution (IE) is a very lightweight CFI scheme to solely protect against code reuse and physical fault attacks on the control flow.

For demonstration, we integrated AEE-Light with a RISC-V microcontroller and evaluated a set of benchmarks on this chip by executing them both unprotected and encrypted with AEE-Light. It shows that the average overheads in code size and execution time of our AEE-Light instance are 19.8 % and 9.1 %, respectively, and thus practical for many IoT scenarios. In terms of hardware complexity, 35 % of the Central Processing Unit (CPU) core area has been devoted to SCFP support and power consumption is increased by 25 %. Note, however, that for a full chip with AEE-Light support, overhead numbers are much smaller due to the integrated peripherals and memory (e.g., caches).

Outline. This chapter is organized as follows. Section 3.1 describes the concept of SCFP and the application of authenticated encryption to the instruction stream. Section 3.2 gives two sponge modes suitable for SCFP and Section 3.3 presents different SCFP instances and their security properties. Section 3.4 discusses how SCFP was integrated into the RISC-V processor *Remus* and details the necessary extensions to the RISC-V Instruction-Set Architecture (ISA). Section 3.5 gives evaluation results and Section 3.6 concludes this chapter.

3.1 Overall Concept

Sponge-Based Control-Flow Protection (SCFP) is a novel security concept for IoT devices that is based on authenticated encryption from cryptographic sponges. In this section, we introduce the threat model we assume for IoT devices and present the architecture of SCFP. In particular, we describe how sponge-based authenticated encryption is applied to an instruction stream and discuss the adaptions required to support arbitrary code execution including control-flow transfers and interrupts.

3.1.1 Threat Model and Assumptions

This work considers IoT devices which are threatened by both software and physical attacks. In terms of software vulnerabilities, we assume a remote attacker who has arbitrary read and write access to the memory due to bugs in the software. Correspondingly, active physical attackers are assumed to have direct access to the device. This direct access can be used to dump and manipulate external memory, to probe and force signals on the Printed Circuit Board (PCB) (e.g., bus signals between chips), or to inject global faults into the system (e.g., clock glitches). On the other hand, micro probing and similar invasive techniques are considered out of scope in this work. Similarly, side-channel leakage of hard- and software implementations is not considered in this work.

Presumed targets for adversaries in this domain are to extract secret IP (e.g., firmware code), to bypass security checks (e.g., by skipping one or more instructions), or to achieve arbitrary code execution via code reuse or injection. In other words, adversaries try to compromise the confidentiality and/or authenticity of the code, either at rest or at runtime. Note, however, that Denial-of-Service

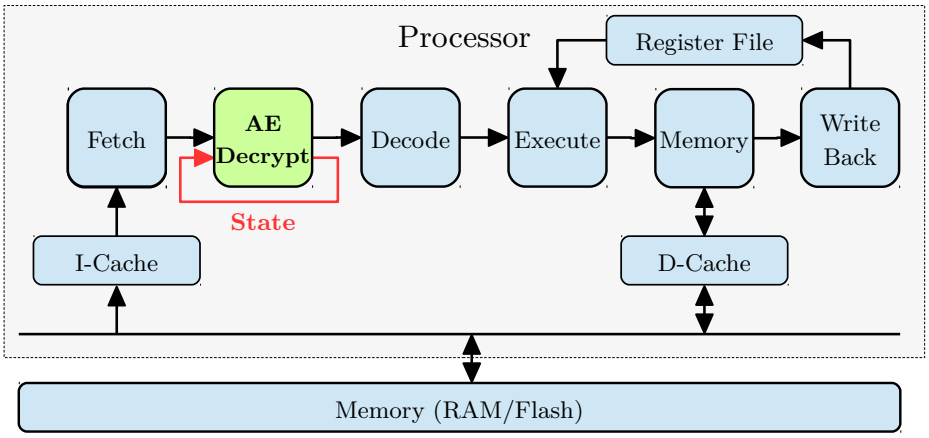


Figure 3.1: High-level system architecture of a classic RISC processor which has been extended for SCFP with a sponge-based AE decryption stage.

(DoS) as well as data-driven attacks are out of scope given that neither can be solved via a CFI scheme.

This work assumes that SCFP is deployed as the only countermeasure to the mentioned threats. Hence, if guarantees that exceed the capabilities of precisely enforced CFI (e.g., resistance against control-flow bending [Car+15]) are required, additional attack mitigation techniques (e.g., safe stack [Kuz+14]) have to be utilized. Further, note that the hardware interface of SCFP is implemented in such a way that there is no interface to access plaintext instructions, the sponge state or internal SCFP signals. All this information is inaccessible in software.

3.1.2 Architecture

The idea behind SCFP is to encrypt programs at compile time using a sponge-based Authenticated Encryption (AE) cipher. Decryption is then performed within the CPU, instruction by instruction, just in time for execution. At its heart, the sponge-based AE cipher uses an internal state z, which provides the foundation for the CFI protection in SCFP. This state accumulates information about all the processed instruction ciphertexts, which enforces that correct decryption is only possible iff all previous instructions have also been genuine. Conceptually, with every processed instruction ciphertext C, the plaintext instruction P as well as a new internal state are derived using a permutation f following $(P|z) = f(C|z)$. As a result, the correctness of the plaintext instructions that get executed by the CPU does not only depend on the fetched input (*i.e.*, ciphertext), but also on the history which has been accumulated within the internal state of the cryptographic primitive.

If either the state (e.g., through a CFI violation or clock glitch) or the ciphertext (e.g., through manipulation in memory) is erroneous, correct decryption is not possible anymore and pseudo-random instructions are produced as plaintext.

We consider the respective execution of random instructions a secure processor state for two reasons. First, the probability of random code which is generated by SCFP to be meaningful is extremely low, especially when attack gadgets of multiple instructions are required. Second, attackers neither have control over the random instructions being executed, nor can attackers directly observe what the plaintext instructions are during random execution. This effectively hampers any attacker attempts to execute harmful code. Besides, we will later show that SCFP supports the detection of random code execution to add error handling as desired.

From a processor architectural point of view, the ideal location within the processor pipeline for performing the decryption is between instruction fetching and decoding, as shown in Figure 3.1. The instructions are transferred from the fetch to the decode stage exactly in the execution sequence, which also matches the desired decryption sequence of SCFP. As the decode stage is the first to need plaintext instructions, performing decryption right in front of the decoder is in fact also the latest possible point for inserting SCFP and effectively minimizes the number of components with plaintext code access to the decode stage itself. All the other components, like peripherals, main memory, various caches, memory buses, and even the fetch unit, operate on encrypted code only.

Figure 3.2 depicts the instruction-data dependencies between the different pipeline stages for the processor from Figure 3.1. A traditional scalar processor with a pipelined architecture only has dependencies between the different stages (visualized horizontally) but not across multiple instructions (visualized vertically). The processor basically decodes each instruction completely isolated from other instructions. Dependencies between instructions are solely a result of data dependencies in the program (e.g., via the register file) which can lead to pipeline hazards and stalls. Extending the pipeline with an AE decryption unit breaks this isolation between instructions and introduces an additional dependency via the cipher state.

For scalar processors, it is additionally possible to feed the data independent decoder signals of each executed instruction back into the cipher. Such feedback extends authenticity protection up to the pipeline's execute stage and can, for example, be used as a link to fault countermeasures in the ALU. Note, however, that the SCFP approach is not limited to scalar processors. Superscalar microarchitectures can also be protected using SCFP with a coarser granularity, e.g., decrypting multiple instructions instead of individual instructions in one block.

3.1.3 Authenticated Encryption and Control Flow

Sponge-based authenticated encryption schemes use a single internal cipher state for both encryption and authentication. This common state leads to the nice property that the mapping between each encrypted and plain instruction depends on the actual values of all previously processed instructions. Hence, to be able to encrypt a program such that it can be executed on a processor that implements SCFP, the exact sequence of executed instructions needs to be known at compile

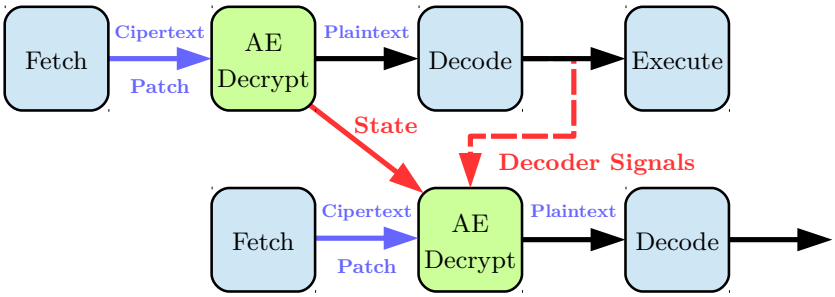


Figure 3.2: Data dependencies between two consecutive instructions within a processor pipeline when SCFP is implemented. The decoder signals can optionally be fed back.

time. However, exactly this property makes the combination of authenticated encryption with control flow challenging.

More concretely, at compile time, the exact instruction sequence can only be determined for a very limited number of programs. Basically, only programs that have a completely data independent control flow (e.g., no data dependent branches) can be trivially supported. Additionally, even genuine and intended code reuse (e.g., loop bodies or functions) is not easily possible anymore. This is due to the fact that after encryption, the ciphertext is fixed and correct decryption of an instruction is only possible given the correct unique cipher state (and thus execution history). Placing the sponge-based authenticated encryption scheme into the processor pipeline therefore provides a solid foundation for SCFP and thwarts code reuse by default.

The main idea to allow specific code reuse in SCFP and to make SCFP applicable to general programs is to deliberately introduce collisions into the internal state of the cryptographic primitive. These state collisions are conceptually a white listing of permitted control flow transfers and have to be introduced exactly at the required positions in a program. Note however that, in general, we want to have as little collisions as possible since they weaken the cryptographic primitive.

The simplest and most efficient way to generate the required state collisions is to inject additional metadata as correction terms into the cipher state at certain points during the execution of the program. We denote this process of deliberately adjusting the AE state as *patching* and the involved constants as *patch values*. Via patching, we effectively cancel out divergences in the cipher state which originate from taking different valid paths through the Control-Flow Graph (CFG). As the result, correct decryption of a program under SCFP is only possible as long as the execution adheres to the statically determined CFG.

It has to be noted that patching must be implemented as a differential update of the AE state. Otherwise, if patching was implemented by simple replacement, patching would destroy all the history which had been accumulated into the state. Besides, the patching process must be able to modify the full sponge state in order to create arbitrary collisions.

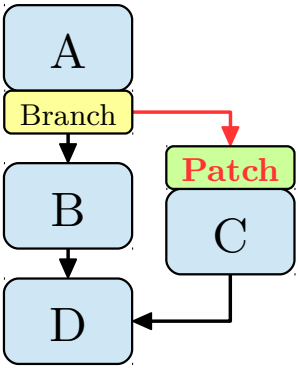


Figure 3.3: Simple example of patching the CFG of an if-then-else construct in SCFP.

3.1.4 Patch Handling, Placement and Calculation

The patch values in SCFP are conceptually very similar to the justifying signatures in the soft error and fault attack countermeasures based on Continuous Signature Monitoring (CSM) [WS90; WWM15]. Therefore, also similar implementation techniques can be used to find suitable patch locations as well as to determine the concrete values of the patch constants.

More concretely, the task of the patch values in SCFP is to introduce cipher state collisions at the merge points in the CFG of the program. Hence, all differences which originate from traversing the statically determined CFG along runtime data dependent paths have to be compensated. An example for patching a simple if-then-else construct is shown in Figure 3.3. There, a patch value is injected into the cipher state before the execution of Basic Block (BB) C (*i.e.*, on the red CFG edge) such that the state at the beginning of BB D is the same, regardless of whether the blocks A and B, or the blocks A and C (incl. the patch) have been executed.

The exact way how such a patch value is encoded into the program and how patches are processed during runtime strongly depends on the concrete implementation of SCFP and is highly ISA specific. However, an intuitive way to implement and think about cipher state patching is to consider the patch values as part of specialized control-flow instructions. Similar to immediate operands in standard instructions, the patch values are part of the instruction encoding and get fetched like regular code by the processor during execution.

From the toolchain perspective, implementing SCFP consists of two steps. In the first step, during compilation, patches have to be inserted at the correct positions into the program by emitting suitable instructions with patch support. In the second step, at link time or in a post processing phase, the program binary has to be encrypted and the correct patch values have to be inserted into the binary (*i.e.*, similar to relocations).

For a program which comprises only branches and direct calls, a functional solution for patch placement can be obtained by looking at the undirected CFG

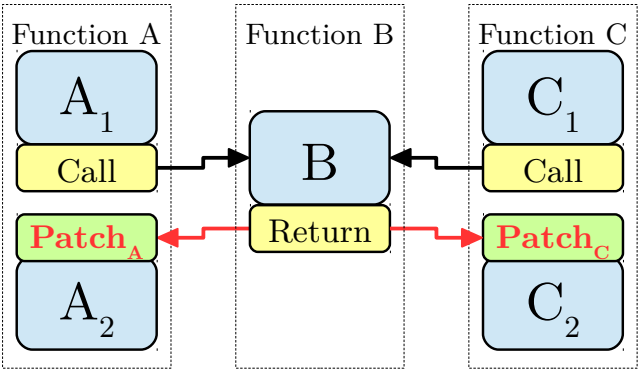


Figure 3.4: Example of a simple patching convention for direct function calls. Function B can be called from both, function A and C.

of the full program. Every cycle in this graph has to be broken by introducing a patch for the cipher state. Therefore, the minimum number of patches and possible positions can be obtained by comparing the CFG with its spanning tree. Taking the function call graph into account, this approach is also applicable to indirect and recursive function calls. Unfortunately, quite expensive whole program analysis has to be performed to acquire the mentioned graphs.

Nevertheless, also compilation in multiple translation units can be supported with SCFP when a well-defined patching convention is established around function calls. Similar to a regular calling convention, having a patching convention allows to correctly place patches in every function of the program in isolation. Within each function, it is then typically sufficient to always patch when a branch is taken as shown in Figure 3.3. Additionally, to cope with recursion, it has to be ensured that at least one patch is performed before the recursion is entered. Note, however, that the simplicity of the patching convention, compared to the graph based approach, comes at the cost of an increased number of patch values.

To illustrate the concept, in the following, an exemplary patching convention for direct and indirect function calls is presented.

Direct Calls

Every function which gets directly called from more than one call site within a program necessarily requires patching. In particular, at least $n - 1$ patches are required when n call sites exist. Interestingly, this situation is also similar to the direct branch example in Figure 3.3 where one patch is required since two paths in the CFG lead to BB D. However, placing patches at every call site except one again requires access to the full program during compilation. To relax this constraint, at the cost of one additional patch per function, patching can simply be performed on every call site as shown in Figure 3.4. In this example, $Patch_A$ has to be applied when the control flow returns from function B to function A. Returning from function B to C uses $Patch_C$, respectively.

Note that, in most cases, having one patch per direct call is sufficient regarding both functionality and security, because typical ISAs perform direct calls relative to the program counter. In this case, the program counter relative offset is part of the function call encoding and is different for each call site. This implies a different, internal SCFP state for each call site. As a result, besides the required state collisions at the call and return edges of the direct function call, there are no other, undesired collisions being introduced to the program.

In general, it does not matter whether the patch value is applied at the return operation or the call operation, as long as it is done consistently and aligned with the way branches are patched. Applying patches is therefore possible either after branches and on returns (as shown in Figure 3.3 and Figure 3.4), or before merge points of branches and during calls.

Indirect Calls

Similar to direct function calls, also indirect function calls require patching. However, determining the exact function which gets called at runtime by an indirect function call is not always possible at compile time. Moreover, often also multiple different functions get called from the same indirect call site during the runtime of a program (e.g., comparison callback of `qsort`). Therefore, the best one can do with static CFI such as SCFP is to determine a, possibly over approximated, set of potential call targets and to enforce that only calls to functions in this set are possible at runtime.

Our current approach to implement indirect function calls and returns with SCFP is shown in Figure 3.5. In total, two patch values have to be applied on every indirect control-flow transfer. The idea of this scheme is to use the first patch (e.g., $Patch_{A1}$) to reach a constant cryptographic intermediate state, which is then updated to the actual entry state of the called function using the second patch value (e.g., $Patch_{D1}$). The constant intermediate state can be freely chosen at encryption time and permits to restrict indirect calls to targets which were encrypted for the same intermediate state.

In summary, for the patching convention in Figure 3.5, two patch values are required for every indirect function call site as well as for every function which can be called indirectly. At runtime, in total four patches get applied for every indirect function call.

At the first glance, using four patch values for one indirect function call may seem excessive given that two patches would already suffice to build a functioning CFI scheme. However, using less patch values necessarily introduces undesired collisions into the SCFP state which weakens the confidentiality and authenticity properties of the scheme.

3.1.5 Initial State Derivation

In sponge-based AE ciphers with known permutation, the initial state is comparable to the key in regular encryption schemes. It is common to derive this initial state z_I from a secret key k and public nonce N by applying their permutation

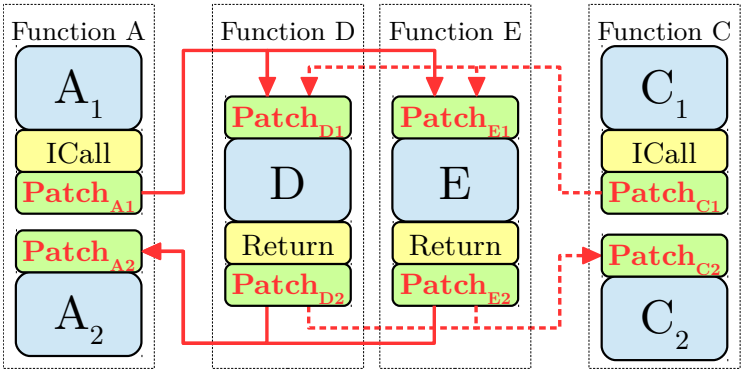


Figure 3.5: Example of a simple patching convention for indirect function calls. Functions A and C can call both, functions D and E at runtime.

f (e.g., $z_I = f(N|k)$). Conceptually, we recommend using a similar approach for deriving the initial state in SCFP. This ensures that, even when k is a device-specific fixed master key, every program for that device is still encrypted under a different initial state. Optionally, additional information like the start address or the program vendor can be used during the derivation of the initial state.

Note that binding initial states to the machine key also serves as software diversification. Namely, in case successful exploits against SCFP should be found in a program on a certain device, they cannot simply be transferred to other devices executing the same program.

3.1.6 Interrupt Handling

Unlike regular function calls, which are performed at precisely defined points during program execution, interrupts can occur at any point in time. It is therefore impossible to determine a unique differential update value for all the states which permit to call an interrupt handler. We cope with this problem in SCFP by treating interrupt handlers similar to the initial program entry point. Therefore, we derive a new AE cipher state to re-initialize SCFP when entering the interrupt handler. On the other hand, the SCFP state that is active before entering the interrupt handler is, similar to the old program counter, saved in an internal processor register. For the operating system, the SCFP state is therefore simply one additional register which has to be saved and restored during context switches. Note however that, to ensure that the confidentiality of the SCFP state is maintained at all times, the old state value which is stored in the processor register should be encrypted or similarly protected.

Implementing interrupt handling in this way effectively separates the protection of interrupt handlers from the regular code. This means that interrupts can be processed successfully even when a regular program executes pseudo random instructions due to an attack. On handler entry, this separation is desirable as it allows us to recover from errors in software as well as to perform scheduling of

programs via the operating system. On handler exit, on the other hand, we want to propagate errors occurring during the execution of the interrupt handler into the execution of the regular code.

We achieve this behavior by enforcing that the internal SCFP state has a predefined secret value when returning from the interrupt handler. Similar to the state derivation on interrupt entry, the secret handler exit state can, for example, again be computed from the key, the nonce, and the address of the interrupt handler. When returning to the regular code execution, the hardware can then simply combine the current state z, the expected exit state e, and the state from before the interrupt entry z_{entry} from the register (e.g., $z = z \oplus e \oplus z_{entry}$). By doing so, the entry value is only restored ($z = z_{entry}$) correctly if also the handler execution has been genuine ($z = e$).

3.1.7 Fast Error Recovery

As SCFP ensures security even without explicit fault checks, SCFP eliminates the existence of a single point of failure. Namely, the probability of random code execution in SCFP to be meaningful is extremely low. While this is one major benefit of SCFP, it may still be desirable to provide a timely way to perform error recovery after the processor started to execute a random instruction sequence. Interestingly, the execution of pseudo random instructions in the error case already provides one way to permit error recovery given that most processors are able to identify invalid instructions. The concrete detection probability follows a geometric distribution and can be computed when the ISA of the processor is known. More concretely, given the probability p_{inv} for a random instruction to be invalid, the expected detection latency l is computed as $l = 1/p_{inv}$. However, considering that modern ISAs are often quite dense, recovery latency can be comparably high.

Faster recovery can be achieved when additional redundancy bits are verified on the execution of every single instruction. Sponges permit to implement this additional integrity verification in an efficient and secure way by simply checking the desired amount of state bits. No additional permutation calls, but only a marginally bigger permutation is required. The strength, *i.e.*, the number of bits, for this verification can be freely chosen, but is typically rather weak for a single instruction. However, the continuous nature of this check compensates for this weakness quite fast. In general, the number of asserted bits allows to trade off between the code size overhead and the recovery latency.

3.2 Sponge Constructions for SCFP

SCFP relies on a scalable and strong sponge-based authenticated encryption cipher. This section introduces two eligible sponge-based constructions and presents arguments for their security as well as guidelines for parameter selection.

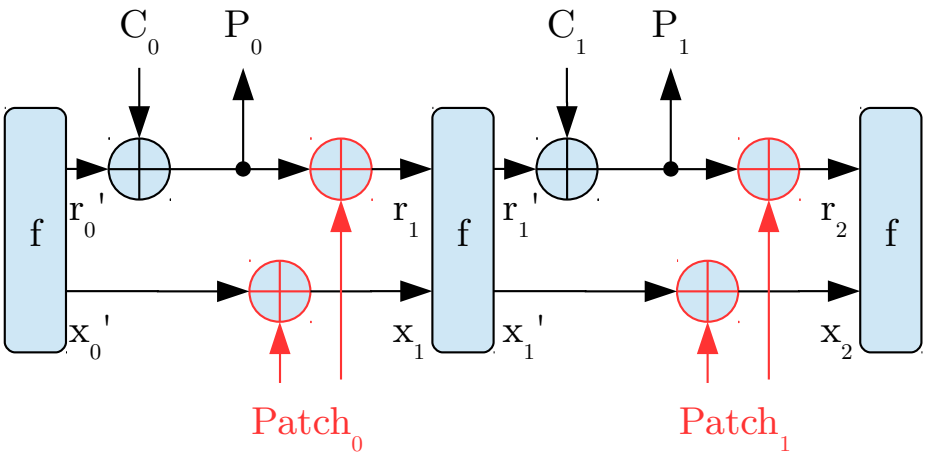


Figure 3.6: Decryption using a duplex construction similar to the one used in SpongeWrap.

3.2.1 Constructions

Cryptographic sponges have become quite popular since Keccak has been announced as the winner of the SHA-3 competition. However, sponges can also be used to build other cryptographic primitives. The Keccak designers themselves, for example, already proposed an AE mode called SpongeWrap [Ber+11a] early on and further pursued the idea with Keyak [Ber+16b] and Ketje [Ber+16a] in the CAESAR competition [Ber19]. The sheer number of sponge-based submissions [AJN16; And+16; Dob+16; Gli+15; Mor+15; SB15] to the competition underlines the potential of this research direction. Additionally, Ascon [Dob+16]—one of the sponge-based designs—has been selected for the final CAESAR portfolio in the lightweight category.

Considering the success and general properties of sponges, the following discusses two sponge-based constructions which have been adapted to support the patching of SCFP. This approach allows us to profit from the substantial amount of cryptanalysis performed on the various sponge constructions and the underlying permutations. In general, we therefore recommend well-analyzed permutations like Keccak-p. However, a more detailed discussion on suitable instantiations of SCFP, including permutations, can be found in Section 3.3.

SpongeWrap-like Decryption Mode

The first construction, shown in Figure 3.6, is based on the duplex construction, which has been introduced and proven to be secure in [Ber+11a]. This duplex construction is used in SpongeWrap for both encryption and decryption. When executing strictly sequential code, where no patching is required ($Patch_i = 0$), AE on the instruction stream is identical to SCFP. However, for generic code SCFP must also implement branching. Therefore, additional support for the

injection of patch values has to be added to the construction. Both the rate and the capacity of the sponge make up the previously described SCFP state z and must be modifiable by such a patch.

From the security point of view, these patch values can be considered as Associated Data (AD). AD means data that is authenticated, but not encrypted. It has been shown by Mennink et al. [MRV15] as well as Sasaki and Yasuda [SY15] that it is secure to absorb AD into the capacity of a keyed sponge. Considering that the construction in Figure 3.6 is a keyed full-state duplex sponge construction, it is therefore secure to inject the patch values into the capacity. Updating the rate with the patch is secure as well given that the rate is under the control of an attacker via the ciphertext in any case.

The SpongeWrap-like construction has two neat features. First, its implementation is comparably simple since encryption is identical to decryption. Second, the construction provides great flexibility as it permits to calculate and place patch values on arbitrary places in the CFG. However, there are also some drawbacks which have to be considered. For example, an attacker might be able to precisely control the first fault given that errors in the ciphertext directly propagate into the plaintext ($\Delta C_i = \Delta P_i$). In a known plaintext attack, this might even permit to inject one specific instruction before the plaintexts of subsequent instructions are randomized.

Note also that, if the control-flow merges at instruction i and patches are not applied directly before the merge point in the control-flow graph (*i.e.*, $Patch_{i-1} = 0$), all instructions directly preceding the merge point (P_{i-1}) have to be identical. This is due to the fact that, as soon as the instruction at the merge point is fixed (*i.e.*, P_i and C_i), also the plaintext of the predecessor P_{i-1} is determined by the dependency over the rate part of the sponge ($P_i = C_i \oplus f_r(P_{i-1}|x'_{i-1})$). However, this link can be broken by performing patches solely at merge points of the CFG instead of placing them freely.

APE-like Decryption Mode

The second construction is inspired by another AE-mode of operation called Authenticated Permutation-based Encryption (APE) [And+14]. The layout of the APE construction itself is similar to the duplex construction in SpongeWrap. However, APE is not inverse free, *i.e.*, the inverse permutation f^{-1} is needed for encryption when the permutation f is used for decryption. Moreover, the indices of the cipher- and plaintexts have been rearranged compared to SpongeWrap. Namely, in APE, the plaintext P_i, corresponding to a ciphertext C_i, is calculated as $P_i = C_{i+1} \oplus f_r(C_i|x_i)$.

The APE-inspired mode we propose in Figure 3.7 is calculated as $P_i = f_r(C_i|x_i)$ and modifies APE in two ways. First, P_i's dependency on C_{i+1} is removed. This solves the problem of the SpongeWrap-like mode where attackers can inject one specific instruction if they know the original value. Moreover, this modification makes our construction behave more like a block cipher than a stream cipher. Second, patching capabilities for the capacity are introduced to make the construction suitable for SCFP. Note that the APE-like construction

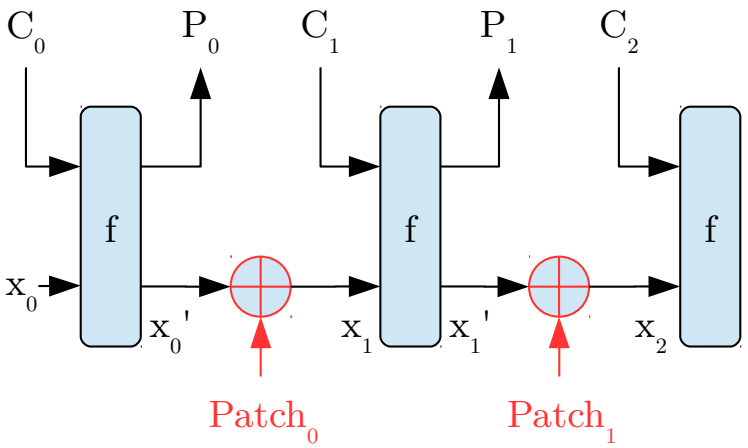


Figure 3.7: Decryption in an APE-like construction.

is superior to the SpongeWrap-like mode in this regard. It only needs patching of the sponge's capacity which corresponds to the SCFP state z. On the other hand, the sponge rate is not chained any more.

The main drawback of the APE-like construction is that it is less flexible, because the position of patches is fixed. Patches always have to be positioned at branching points in the control-flow graph. This is a result of the encryption that has to be performed in inverse direction to the decryption (*i.e.*, inverse to the execution sequence).

3.2.2 Parameter Selection

It has been shown that the duplex construction [Ber+11a] as well as the APE construction [And+14] are secure against generic attacks which do not exploit properties of the underlying permutation. The complexity of such attacks is lower bounded by $2^{x/2}$ and depends on the capacity size x. To provide s-bit security, x must thus be chosen as $x \geq 2 \cdot s$.

The size of the sponge rate depends on the actual implementation. The majority of the rate is needed for the decryption of the instructions. The instruction size i depends on the ISA and is typically 16 or 32 bits. However, additional bits may be used for fast error recovery. To enable fast error recovery of n bits without leaking parts of the plaintext nor reduction of the security, a rate of $r = i + n$ bits, and a permutation size of $b = i + n + x$ bits is needed.

The proofs in [And+15; Ber+11b] show that also smaller capacity sizes can result in cryptographic security. They can be used to reduce the permutation size while maintaining the security level, or to increase the security of a fixed permutation. However, a limit on the data complexity, which strongly depends on the implementation, is required to utilize these refined proofs. We therefore refrain from proposing parameters based on the proofs in [And+15; Ber+11b] and leave the exploitation to implementers knowing the respective system characteristics.

3.3 Instantiations

The flexibility of SCFP allows to tailor its protection level to the needs of the respective application by choosing a suitable permutation for the sponge-based AE scheme. In this section, we hence introduce three different instantiations of SCFP. First, AEE uses a large, unkeyed permutation to yield confidentiality and authenticity of the program binary as well as CFI to prevent fault, code reuse, and code injection attacks. Second, IE uses a small, unkeyed permutation to form a lightweight CFI scheme to prevent code reuse and fault attacks only. Third, the use of a small, keyed permutation in AEE-Light yields small overhead, CFI and IP protection in trade for weaker authenticity. We first discuss the properties of unkeyed permutations used in AEE and IE, and then proceed with keyed permutations utilized in AEE-Light.

3.3.1 Unkeyed Permutations

When instantiating SCFP with unkeyed permutations, the cryptographic security properties of SCFP are solely determined by the size of the sponge capacity x. Neglecting the proofs in [And+15; Ber+11b], a security level of s bits requires a sponge capacity of $2s$ bits. However, these are generic results without consideration of the actual application.

In particular, the cryptographic security level s is mainly determined by collisions in the cryptographic state. These can generically be exploited in Time-Memory Trade-Off (TMTO) attacks with birthday bound complexity $2^{x/2}$ and eventually allow state recovery and thus IP theft or forgery. However, to perform these TMTO attacks, the attacker must also be able to observe the output of the sponge, which is not the case for SCFP. Namely, the instructions decrypted by SCFP are internally processed by the processor and never directly revealed to the attacker. As a result, the complexity for state recovery for SCFP is 2^x in practice. In a similar way, the probability of arbitrary state collisions in a binary encrypted and authenticated with SCFP is in general determined by the birthday bound, *i.e.*, $2^{-x/2}$. However, attackers do not have access to the decrypted instructions and the internal state when using SCFP. Attackers are thus unable to observe and detect internal state collisions. Hence, meaningful exploitation of internal state collisions for SCFP is equivalent to state recovery and has complexity 2^x as well.

Software Attack Complexity. These considerations have a significant impact on the actual attack complexities for code injection and code reuse attacks when SCFP is in place, *i.e.*, the CFI properties of SCFP. Namely, attackers performing code injection or code reuse attacks require precise control over the executed instructions to succeed. For example, attackers can modify a single instruction with success probability 2^r, but will neither be able to observe whether they hit the right instruction, nor be able to modify the internal state such that all successive instructions remain the same. This means that the attacker must adapt all successive instructions too, because the processor will otherwise execute random instructions. However, precise manipulation of n instructions has even

higher complexity, namely $2^{n \cdot r}$. Alternatively, attackers can try to learn the internal state to correctly encrypt and inject their own program. However, this has complexity 2^x. A different example are modified jump targets in code reuse attacks. As attackers manipulate addresses to jump to well-defined instructions in the binary, the x-bit patch values must be adapted accordingly. However, finding a correct patch value has complexity 2^x too.

Fault Attack Complexity. The CFI properties of SCFP also increase the attack complexities for fault injection attacks that manipulate control flow or instructions prior to the decode stage. For example, simple instruction skips or repetition have a success probability of 2^{-x}. The same probability applies to arbitrary control-flow errors, e.g., caused by a randomly faulted program counter. On the other hand, performing a specific control-flow transfer via faults is as hard as forcing the x capacity bits and the program counter (e.g., 32 bits) to the desired value. However, this is non-trivial since the sponge state is secret and must be extracted or brute forced first. Furthermore, being able to control that many bits precisely is quite hard in practice.

Instead of altering the control flow, fault attackers can also try to manipulate code by injecting bit flips. For example, attackers can use clock or power glitches to inject random bit flips into code. However, it takes roughly 2^r tries to hit one specific instruction with random bit flips. Therefore, another approach is to use a small and limited number of precise bit flips in the fault attack instead. Yet, exploiting precise bit flips in the encoded value is as hard as utilizing a differential characteristic of the permutation. Only precise bit flips in the plain instruction can be exploited directly. However, regardless of whether random or precise bit flips are injected, bit flips in code modify the sponge state as well. This randomizes the sponge output of all subsequent instructions and therefore prevents further exploitation. Moreover, SCFP can also protect the plain instructions against fault attacks as well by feeding the decoder signals back into the sponge state.

Depending on the concrete security level s and choice of sponge parameters r and x, we identify two different types of SCFP instances using unkeyed permutations. First, AEE denotes instances with cryptographic security levels, *i.e.*, at least 80 bits, that offer CFI as well as confidentiality and authenticity of software IP. Second, IE denotes instances below cryptographic security levels to enforce CFI only.

Authentic-Encrypted Execution

AEE features cryptographic security levels for encrypting and authenticating code. This automatically defeats adversaries which wiretap the communication to the external memory chips without any need for further code encryption and/or authentication. Moreover, software attacks are made harder too. As other CFI schemes, AEE hampers return- and jump-oriented programming attacks. The strong encryption and authentication further mitigates both code injection and code disclosure attacks. AEE is therefore a replacement for established software attack countermeasures like DEP (*i.e.*, W^X), CFI, and R^X. In addition, by

enforcing CFI AEE also prevents fault attacks on the processor chip that aim at instruction or control-flow manipulation.

From a cryptographic perspective, AEE requires a permutation size of at least 192 bits to yield 80-bit security for a 32-bit instruction set. One suitable permutation to instantiate AEE hence is Keccak-p[200,12] [Ber+16b] with 200-bit state size and 12 rounds as used in Keyak. The exceeding 8 bits increase the capacity and thus the security level to 84 bits. However, as elaborated before, the specifics of AEE result in a complexity of 2^{168} for state recovery, control-flow hijacking, and fault attacks on control flow. Similarly, a single instruction can be successfully manipulated from software or using fault attacks with complexity 2^{32}, but the internal 168-bit state will cause the execution of random instructions afterwards.

Infective Execution

Contrary to AEE, IE uses a small permutation and thus, from a cryptographic point of view, cannot provide a strong level of security. In particular, IE behaves like a context-sensitive instruction-set randomization rather than authenticated encryption. IE thus fails to ensure confidentiality and authenticity of software IP. However, the parameterization of IE forms a practical CFI scheme that considerably complicates code reuse attacks as well as fault attacks on the processor chip itself. Yet, the concrete instantiation of IE is highly application specific.

For a 32-bit ISA, IE can, for example, be instantiated with 50-bit state size and the Keccak-p[50,12] [Ber+16b] permutation (*i.e.*, 12 rounds as in Keyak). Using two bits for fast error recovery gives a sponge rate $r = 34$ bits and a sponge capacity $x = 16$ bits, which also corresponds to the size of the patch values. From a cryptographic perspective, this IE instance yields merely 8-bit security. However, the probability for successful code injection and manipulation of control flow still is 2^{-16}.

The main drawback of IE is that an attacker with access to the encrypted binary can easily perform state recovery offline, in our example with complexity 2^{16}. State recovery eventually breaks the CFI property of IE for software attackers. Namely, a software attacker knowing the secret, internal IE state can compute correct ciphertexts and patch values, and inject these into the code from within software when performing code injection or reuse attacks. However, the complexity of physical fault injection on the processor chip itself is still high enough for the parameterization of IE. Nevertheless, to ensure CFI for software attackers as well, access to the encrypted binary must be limited. While this restricts the attacker compared to the original threat model in Section 3.1, access control can easily be enforced using two different mechanisms: (1) by using execute-only memory, software attackers lose online access to the encrypted binary, and (2) by storing the binary in on-chip memory, attackers with physical access cannot read the encrypted binary any more. As a result, IE is particularly interesting for tiny IoT devices without external memory and for smart cards. Note, however, that state recovery, code analysis, and wide-spread deployment of

attacks can easily be mitigated by using a different seed for IE on every device as this causes the internal states, patch values, ciphertexts, and positions of state collisions to change. Moreover, note that the probabilities for manipulating control flow stated above are enough to enforce CFI and are indeed in the range of entropy estimations of other techniques to prevent code reuse attacks, e.g., software diversification [Cle+17a].

3.3.2 Keyed Permutations

AEE enforces its security properties by using a sufficiently large permutation and thus capacity. However, a sponge capacity providing cryptographic security levels also implies larger AEE patch values and thus memory overhead. On the other hand, IE yields lower memory overhead by using a small permutation, but cannot sufficiently protect software IP and its authenticity. For this reason, an SCFP instance with low memory overhead, but with similar security properties as AEE, is desirable. One approach to tackle this problem are keyed permutations.

When using a keyed permutation, the security of SCFP does not only depend on the sponge capacity x, but also on the security level s_p of the permutation itself. As for AEE and IE, the authenticity when using keyed permutations is determined by the sponge capacity x, *i.e.*, the authenticity level is $x/2$ bits. However, the complexity of learning the plaintext of the encrypted binary is 2^{x+s_p} and thus also depends on the security guarantees of the permutation with respect to the permutation key.

Authentic-Encrypted Execution Light

We build on this observation and introduce AEE-Light to denote SCFP instances based on keyed permutations. AEE-Light offers the same security bounds as AEE with respect to authenticity and CFI. For example, control flow hijacking and fault attacks on control flow have complexity 2^x, whereas successful injection of a single instruction has complexity 2^r. On the other hand, successful recovery of the software IP or the internal state from the encrypted image has complexity 2^{x+s_p}. By using a permutation with sufficiently high security level s_p, the confidentiality of software IP is hence guaranteed and state recovery, code injection, and meaningful forgery are prevented. In particular, even if an attacker recovers the x-bit internal state, meaningful injection or forgery of more than one instruction still has complexity 2^{s_p} as the permutation key is unknown to the attacker.

For 32-bit instructions, a suitable choice for the keyed permutation is the 64-bit block cipher Prince [Bor+12a], which uses a 128-bit key to offer $s_p = 96$-bit security. This results in a sponge capacity $x = 32$ bits. State recovery using this AEE-Light instance has complexity 2^{128} and is thus infeasible. This effectively protects the software IP and prevents both code injection and analysis. Contrary to that, IE from before uses a similarly small permutation but cannot guarantee any of these features without further techniques to hide the encrypted binary. However, while the cryptographic level of authenticity guaranteed by this instance

of AEE-Light is only 16 bits, meaningful code reuse attacks and forgery are much harder. Namely, the expected complexity to find the correct patch value is 2^{32}, which is enough to enforce CFI and to prevent code reuse and physical fault attacks. Besides, the $s_p = 96$-bit security of the permutation further hardens any attempts to tamper with the software binary in a meaningful way. In particular, even though one single instruction can be manipulated with complexity 2^{32}, meaningful modification of multiple instructions is significantly harder since both the internal AEE-Light state and the permutation key are unknown to the attacker.

3.3.3 Discussion

Table 3.1 summarizes the exemplary instances of AEE, AEE-Light, and IE. In detail, Table 3.1 shows the respective attack complexities for Code Injection Attacks (CIA), Code Reuse Attacks (CRA), Extraction of Software IP (ESIP), and Fault Attack induced Instruction Skips (FAIS). AEE is the strongest variant with 168-bit security for all considered attacks. At the further end, IE is the smallest variant and offers merely 16-bit security for the mentioned attacks. However, this suffices to enforce CFI and prevent code reuse as well as fault attacks on control flow when the code binary remains hidden. As a trade-off between these two, AEE-Light uses keyed permutations to simultaneously attain small 32-bit patch values, *i.e.*, low memory overhead, and good security properties. In particular, AEE-Light provides 128-bit security in terms of IP recovery and code injection, whereas its security level with respect to code reuse and control-flow fault attacks is 32 bits and thus sufficiently high for CFI.

Table 3.1: Examples of SCFP instances for a 32-bit ISA and the respective attack complexities.

	Conf. [bit]		Attack Complexity [bit]				
Permutation	x	s_p	CIA[a]	CRA[b]	ESIP[a]	FAIS[b]	Type
Keccak-p[200,12]	168	—	168	168	168	168	AEE
Keccak-p[50,12]	16	—	16	16	16	16	IE
Prince	32	96	128	32	128	32	AEE-Light

[a]Requires the recovery of capacity and permutation key, *i.e.*, $x + s_p$ bits.
[b]Requires to find and inject the correct patch values, *i.e.*, x bits.

3.4 RISC-V Implementation

Building a processor with support for Sponge-Based Control-Flow Protection requires hardware as well as software modifications. In the following, we discuss the processor architecture of our CPU core *Remus*, detail our extensions to the RISC-V ISA, and introduce the used custom software toolchain.

3.4.1 Processor Architecture

Remus, our RISC-V core with SCFP extensions is based on an open-source implementation of RI5CY [ETH17b], the processor core from the Pulpino System-on-Chip (SoC) [ETH17a]. *Remus* implements the RV32I base ISA as well as the M-extension which provides multiplication and division instructions. In addition, major parts of the RISC-V privileged architecture draft Version 1.9.1 [Wat+16][1], including Memory-Management Unit (MMU) with Sv32 support, Translation-Lookaside Buffers (TLBs), and hardware Page-Table Walker (PTW), are supported by *Remus*. The RISC-V ISA [WA17] is particularly suited for this study as it keeps a reserved encoding space for custom extensions, which we have used to implement additional instructions to help with the SCFP implementation.

Figure 3.8 shows the 5-stage pipeline of the *Remus* core which features a dedicated SCFP decryption stage between the fetch and decode stages of the original RI5CY implementation. Our goal was to support one instruction per cycle operation with minimal impact on the overall performance. We have therefore selected an AEE-Light implementation with 64 bits (*i.e.*, 32 bits capacity and 32 bits rate) operated in an APE-like mode. To achieve the required throughput we used a fully unrolled implementation of the low-latency block cipher Prince [Bor+12a] as permutation.

Finally, to enable easier evaluation of SCFP, *Remus* has the capability to enable and disable the additional SCFP pipeline stage during context switches. It is therefore possible to execute standard RV32IM code in a 4-stage configuration, similar to RI5CY, as well as SCFP encrypted code in a 5-stage configuration on *Remus*. In combination with the preliminary privileged architecture support, this makes *Remus* an ideal platform for evaluating SCFP with different bare-metal and operating system workloads.

Remus was, furthermore, successfully integrated in an PULPissimo [ETH18] platform-based microcontroller called *Patronus*. *Patronus* features a rich set of peripherals including GPIO, timers, a uDMA controller as well as various communication interfaces like I^2C, I^2S, JTAG, SPI, and UART. The final design was taped out [Sch+18a] at ETH Zurich using UMC 65 nm technology and has been manufactured as an Application Specific Integrated Circuit (ASIC). Note that extending the RISC-V core with SCFP did not change the target frequency of 100 MHz.

[1] The latest version of the privileged architecture draft prior to the tape-out.

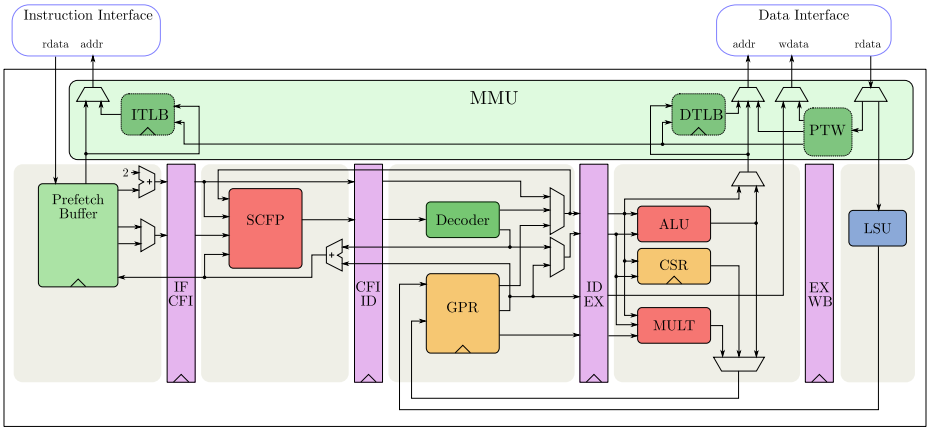


Figure 3.8: *Remus* core pipeline with dedicated SCFP decryption stage and MMU.

3.4.2 RISC-V Instruction Set Extensions to support SCFP

As detailed in Section 3.1.3, when SCFP is employed, patch values have to be injected into the cipher state to deal with arbitrary control-flow transfers. In our implementation we opt for augmenting the control-flow instructions in the RISC-V ISA with support for handling patch values. In a processor that exclusively supports SCFP encrypted code, it is possible to completely replace the original branch instructions with their extended counterparts.

However, while designing *Remus*, we wanted to retain full compatibility with the RISC-V ISA. Instead of replacing the original control-flow instructions we therefore added *patching* support by introducing new conditional branch (`BPEQ`, `BPNE`, ...) and jump-and-link instructions (`JALP` and `JALRP`). Furthermore, we have added a second variant of the `BPEQ` instruction called `BPDEQ`. This `BPDEQ` instruction provides an extension point for linking software redundancy with the SCFP state and can be used to protect data against fault attacks [SWM18].

To be able to reuse the fetch unit for loading the 32-bit patch values into the SCFP stage the patch values are located either directly following the respective instruction ($PC + 4$) or at the destination of the jump. The encoding for these new instructions is shown in Figure 3.9 and is similar to the original encodings. Note further that we deliberately decided against using 64-bit RISC-V instruction encodings to embed the patch values because the `BPxxx` instructions alone would already consume the entire 64-bit encoding space.

Listing 1 shows the functionality of the added `BPxxx` instructions. If the branch is taken, the patch value is injected into the SCFP state, otherwise the next 4 bytes are skipped as these hold the *patch* value. The `BPDEQ` instruction in addition also injects the first comparison operand into the SCFP state. As detailed in [SWM18], this second patching operation can be used to link data redundancy schemes, e.g., AN-codes, with SCFP.

The pseudo code in Listing 2 describes the semantic of the `JALRP` instruction.

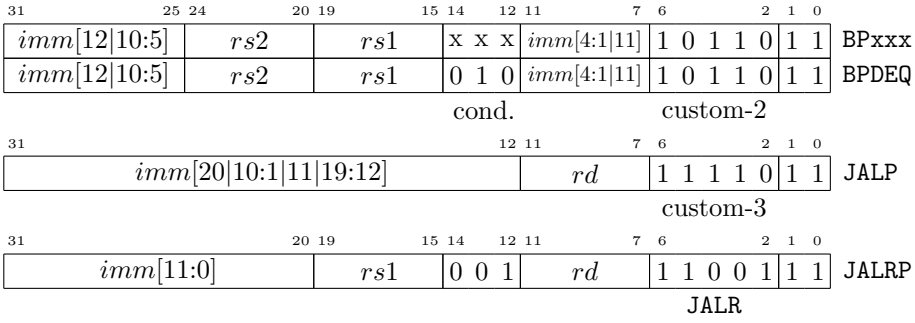


Figure 3.9: `BPEQ` (000), `BPNE` (001), `BPLT` (100), `BPGE` (101), `BPLTU` (110), `BPGEU` (111), and `BPDEQ` (010) implemented as 25-bit greenfield extension into the custom-2 major opcode. `JALP` implemented as 25-bit greenfield extension into the custom-3 major opcode and `JALRP` implemented as 22-bit brownfield extension into e `JALR` major opcode.

Similar to the regular `JALR` instruction, the target of the jump, *i.e.*, the next Program Counter (PC) value, is determined by Reg[$rs1$] plus imm offset and the address of the next instruction is saved in the destination register rd. However, additionally, two different types of patching are supported depending on the Least Significant Bit (LSB) of the $Target$ address. The first type, when the LSB is zero, simply applies one patch, *i.e.*, the $TargetPatch$, similar to the `BPxxx` instructions. The second type, on the other hand, applies two patches, *i.e.*, $SrcPatch$ and $TargetPatch$, with a sponge permutation in between. In any case, the LSB in the destination register is set. Note that we can use the LSB to decide on the patching methodology because it is guaranteed to be unused in RV32I code due to the 4-byte alignment.

Defining `JALRP` in this way permits to easily implement the proposed patching convention for function calls (see Figure 3.4 and Figure 3.5) without requiring separate functions, branches based on the call type, or call wrappers. Namely, direct function calls, performed with `JAL` or `JALR`, patch once when they return with `JALRP`. On the other hand, indirect function calls, performed with `JALRP` and unaligned target address, patch twice on call and twice on return with `JALRP`.

Listing 1 Pseudo code for the `BPxxx` instructions.

Note: SPC denotes the SCFP state, PC the program counter
Note: $PatchValue$ is located at $PC + 4$

```
1: if opcode = BPDEQ then
2:     SPC ← SPC ⊕ Reg[rs1]                          // apply patch
3: if Reg[rs1] {=,≠,<,≥} Reg[rs2] then
4:     SPC ← SPC ⊕ PatchValue                        // apply patch
5:     PC ← PC + signExtend(imm)                     // perform branch
6: else
7:     PC ← PC + 8                                   // fall-through but skip patch
```

3.4.3 Extensions of the RISC-V Privileged Architecture

An additional challenge in our SCFP implementation is to support context switches as well as interrupt handling. We have added several Control and Status Registers (CSRs) to control the behavior of the SCFP decryption unit for each supported privilege level (*i.e.*, machine, supervisor, user). For example, CSRs that configure which 128-bit Prince key is used have been implemented (*i.e.*, `xKEY0`, `xKEY0H`, `xKEY1`, `xKEY1H`, where `x` is either `M`, `S`, or `U` for the different processor modes). On trap entry and exit, depending on the entered privilege mode, the correct key is transferred from these CSRs into the SCFP unit.

Furthermore, similar to the `xEPC` registers, `xSPONGE` registers have been added that capture the sponge state when an exception or interrupt occurs. Depending on the trap cause, either the state before or after decrypting the currently executed instruction is saved. More concretely, traps due to executing `ECALL` capture the state after decrypting the `ECALL` instruction which permits to continue with the next instruction after trap handling. In all other cases, the current instruction has to be replayed and therefore the state before the decryption is stored.

Note that the entry state of a trap handler is independent of the regular execution state. It gets derived, similar to the `JALRP` instruction, by permuting the zero state and the trap handler address followed by applying a $TargetPatch$ that is located at the trap handler entry point. This approach permits to preempt and resume code at arbitrary points. Furthermore, this also enables us to detect and recover from the random execution state that is entered when the processor is under attack (see Section 3.1.7).

3.4.4 Software Toolchain

Generating binaries for our custom processor in AEE-Light mode requires a custom toolchain. The initial version of this toolchain, as used for evaluation in the EuroS&P paper [Wer+18] and for the reported results in Section 3.5, was rather simple. We employed the standard RISC-V GNU toolchain which we only extended with assembling support for our custom instructions. Additionally, a

Listing 2 Pseudo code for the `JALRP` instruction.

Note: SPC denotes the SCFP state, PC the program counter
Note: $SrcPatch$ is located at $PC + 4$
Note: $TargetPatch$ is located at $AlignedTarget$

```
1: Target ← Reg[rs1] + signExtend(imm)
2: AlignedTarget ← Target & ∼3                    // determine target
3: if Target & 1 then
4:     SPC ← SPC ⊕ SrcPatch                       // apply patch
5:     SPC ← permute(SPC, AlignedTarget)
6:     Reg[rd] ← PC + 9                           // set link reg.
7: else
8:     Reg[rd] ← PC + 5                           // set link reg.
9: SPC ← SPC ⊕ TargetPatch                        // apply patch
10: PC ← AlignedTarget + 4                        // perform jump
```

post-processing tool has been developed which consumes the final elf binary in order to perform the encryption of the code and to fill in the required patch values.

Since the C compiler has not been extended with SCFP support, this early toolchain can natively only handle assembler code which contains placeholders for the patch values and uses our protected control-flow instructions. However, due to the way we designed our instruction-set integration, it is quite easy to support the protection of C programs via simple textual replacement of instructions on the assembly level.

More concretely, when compiling C code for our processor, we first compiled the C code to assembly, where we replace all ordinary control-flow instructions through the protected counterparts and embed `NOP` instructions as placeholders for the patch values. The resulting assembly files are then assembled and linked. Finally, the resulting elf file is processed using our post-processing tool which emits the encrypted binary. While not particularly intuitive to use, this simple flow already suffices to demonstrate the practicality of SCFP as well as its strong performance.

However, more sophisticated toolchain support has been developed in the meantime. Our current tooling directly integrates the emission of SCFP instructions and placeholders into the RISC-V MC backend of LLVM. Building a C program for our SCFP-enabled *Remus* processor is, subsequently, as simple as recompiling and linking the binary using clang. As before, generation of the patch values and the actual encryption is performed via our post-processing tool. Note, however, that the compiler still not really aware of SCFPs cost model (e.g., loops get more expensive due to the patching). Improve the performance even further should, therefore, be possible by tweaking the heuristics of the compiler.

3.5 Evaluation

While SCFP protects software and its execution from a large set of attacks, SCFP also has an impact on chip area, power, and performance. This section evaluates the cost of adding AEE-Light support to the *Remus* CPU as well as the *Patronus* chip in terms of chip area overhead and power consumption. Additionally, the performance impact of AEE-Light is quantified by analyzing binary size increase and execution time overhead on our RISC-V processor. Finally, the actual error detection latency of our *Remus* CPU is evaluated. Our results demonstrate the practicality of SCFP with a size overhead of 19.8 % and a performance overhead of 9.1 % on average.

3.5.1 Area

The *Patronus* chip, including our *Remus* CPU, has been implemented in the UMC65LL process with 8 metal layers and occupies a total area of 6.75 mm^2 including I/O buffers. The *Remus* core contributes 0.120 mm^2 (about 80 kGE)[2],

[2] 1 GE conforms to the area of a 2-input NAND gate with driving strength 1.

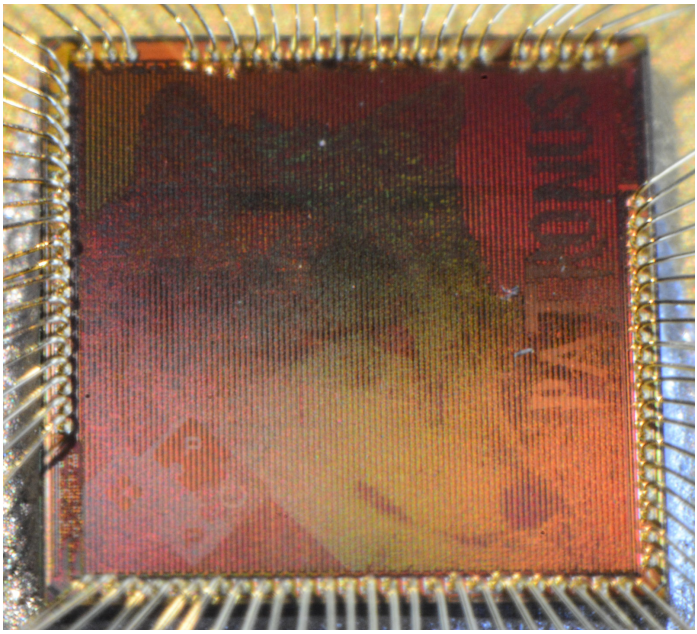


Figure 3.10: Die shot [Sch+18a] of the *Patronus* chip after bonding.

which is only a small fraction of the total SoC area. The majority of the chip area is occupied by a total 640 kB of memory which is sufficiently large to run a port of the SeL4 operating system. The *Patronus* chip includes several additional blocks unrelated to the work described in this paper. The timing of the system is dominated by the access speed of the large RAM macros, and therefore a relatively conservative target clock frequency of 100 MHz for worst case corners has been used in the design. A chip photograph of the manufactured *Patronus* chip can be seen in Figure 3.10.

A detailed analysis of the active circuit area of *Remus* for different clock constraints is presented in Table 3.2. It can be seen that the SCFP stage alone contributes already between 19 and 32 % of *Remus*' area, most of which is due to the fully unrolled Prince implementation which requires between 15 and 29 % of the core. Within a complete system, this overhead is much smaller as typically the entire core occupies only a small part of the system. For example, in *Patronus*, *Remus* occupies less than 5 % of the total area.

3.5.2 Code Size and Runtime

We used our software toolchain to instrument, compile and encrypt a set of C benchmarks to evaluate our implementation of AEE-Light. Several benchmarks from the PULPino repository [ETH17a] were used: AES in CBC mode (`aes_cbc`), a 2-dimensional matrix convolution (`conv2d`), 100 runs of `dhrystone`, a finite response filter (`fir`), a fast Fourier transform (`fft`), and an implementation of the

inflection point method (`ipm`). Moreover, we used two implementation variants of the elliptic curve point multiplication (SECP192R1) that were internally available at our department. Both `ecc` and `ecc_opt` are pure C implementations targeted at microcontrollers. However, while `ecc` uses a generic implementation of the underlying multi-precision integer arithmetic, the multi-precision integer arithmetic in `ecc_opt` uses completely unrolled loops and only works for the specific elliptic curve. We compiled all programs at optimization level `-O3`. Since the manufactured ASIC was not yet available when writing the original paper [Wer+18], all runtime values have been determined using cycle accurate HDL simulation. Unfortunately, only a rather small number of short test programs could be evaluated this way since cycle accurate HDL simulation is rather slow.

Table 3.3 shows the results of our evaluation of code size and execution time. In particular, Table 3.3 compares the unprotected, standard executables of our benchmark programs with the executables protected and encrypted via our instance of AEE-Light. Both program versions have been executed on our modified processor which features either a four stage (*i.e.*, baseline) or a five stage (*i.e.*, AEE-Light) pipeline.

For our set of benchmarks, it shows that the overhead in code size due to the inserted patch values ranges between 14.8 % and 25.6 % and averages to 19.8 %. This overhead is mainly affected by the number of branches and function calls in the binary. On the other hand, the runtime overhead ranges between 3.8 % and 14.9 % and averages to 9.1 %. This runtime overhead is significantly lower than the code size overhead and mainly depends on the number of branches and function calls that are effectively taken during runtime. This becomes especially visible for the two implementations of elliptic curves. Namely, as the inner loops are unrolled in `ecc_opt`, the number of executed branches drops massively from 170 k taken branches to 20 k and hence the runtime overhead for `ecc_opt` is much lower than for `ecc`. On the other hand, `ecc` and `ecc_opt` yet have very similar code size and code overhead.

In terms of related work, only SOFIA [Cle+16; Cle+17b] comes with similar security properties as SCFP. The most similar Prince-based SOFIA instance features an average overhead of 141 % in terms of clock cycles and 203 % in code size. However, this instance uses 64-bit tags and thus offers stronger authenticity than the evaluate AEE-Light variant with 32-bit capacity. Still, regarding runtime, even though it is hard to estimate, we are quite confident that AEE-Light is faster than SOFIA even at equal security level. In terms of code size overhead, on the other hand, we can assert with certainty that AEE-Light would be smaller. After all the numer of patch values does not change and their size is directly determined by the capacity size.

3.5.3 Power

As part of the tape-out, also detailed post-layout simulations of the manufactured netlist under typical conditions at 50 MHz where performed. The gathered power values reflect the consumed power during the main computation part of the benchmark and are summarized in Table 3.4. These simulation runs predate the

Table 3.2: *Remus* post-synthesis area breakdown (kGE) for different clock constraints, using worst-case libraries (1.08V/125°C).

	Area (in kGE)				
Frq. (in MHz)	33	50	75	100	150
IF-stage	3.3	3.6	3.4	4.0	5.5
SCFP-stage	10.9	10.9	13.4	25.3	26.0
Prince	8.9	8.8	11.4	23.0	23.3
ID-stage	15.3	15.4	16.0	16.7	17.6
EX-stage	14.4	14.5	15.5	16.2	16.7
WB-stage	1.8	1.8	1.9	2.4	3.2
MMU	3.4	3.5	3.7	3.8	3.9
iTLB	1.4	1.5	1.5	1.5	1.6
dTLB	1.4	1.5	1.5	1.6	1.6
CSR	8.9	8.9	9.9	10.3	10.3
Remus Total	58.3	58.8	64.2	79.0	83.6

Table 3.3: Evaluation results of AEE-Light in HDL simulation.

	Code Size (`text` + `data`)		Runtime	
	Baseline [kB]	Overhead [%]	Baseline [kCycles]	Overhead [%]
`aes_cbc`	10.0	14.8	43.4	9.5
`conv2d`	4.6	25.6	5.4	4.8
`dhrystone`	7.5	20.1	50.6	14.4
`ecc`	9.3	21.0	4282	9.2
`ecc_opt`	9.6	20.1	3032	3.8
`fir`	5.5	20.9	24.0	9.5
`fft`	7.1	16.8	45.6	7.0
`ipm`	8.8	19.4	4.5	14.9
Average		19.8		9.1

performance simulations for the EuroS&P publication [Wer+18] and were much more extensive (*i.e.*, more benchmarks from the PULPino repository [ETH17a]). However, although runtime performance was basically identical, we opted to perform new performance simulations to ensure consistency after a libc upgrade in our toolchain.

Taking only *Remus* into account, power consumption with SCFP increases between 21 and 32 % (average around 25 %). For the full *Patronus* chip, the power overhead of enabling SCFP is with 7 to 17 % noticeably smaller. As with the area, the system level overhead is therefore much smaller and varies depending on the executed software as well as the overall system composition.

The power consumption obtained through post-layout simulations have shown good agreement with actual measurement results in this technology, and allow us, unlike measurement on the chip which only has a single core power supply, to identify the contribution of individual blocks at a much finer level. However, to verify that the simulated results indeed are sensible, also actual measurements of the *Patronus* ASIC have been performed. For example, the measured results for the fir benchmark, that induced the highest overhead in simulation, is shown in Table 3.5. Across different frequencies, the measured overhead varies between 17.8 % and 21.3 %, slightly above the simulated result. On the other hand, linearly approximating the absolute values at 50 MHz based on the measurements at 40 MHz and 60 MHz yields with 11.44 mW and 13.78 mW a slightly lower power consumption for *Patronus* than in simulation. Still, overall the measured values are in line with the simulation results and make us confident that our evaluation is correct.

3.5.4 Fast Error Recovery Latency

Failed attempts to tamper with SCFP result in an invalid decryption unit state which thwarts any controlled further exploitation since the processor starts executing a pseudo random instruction sequence. As mentioned in Section 3.4.3, *Remus* can recover from this state via its trap handling. However, how many instructions are executed until a trap is triggered can only be answered probabilistically and depends, besides other factors, on the density of the implemented instruction-set architecture.

For *Remus*, the implemented RISC-V RV32IM instruction set is quite sparse meaning that more than 80 % of the available 32-bit instruction encodings are invalid. Subsequently, the expected detection latency is less than 1.3 instructions. We practically verified this expectation on *Patronus* by deliberately destroying the SCFP state via software and by measuring the number of executed instructions via the performance counters. Interestingly, in this experiment, around 91.5 % of the tampering attempts triggered a trap while or before the first instruction has been executed. More than 99.2 % of the attempts where detected within two executed instructions and all of our tries triggered a trap within four instructions. The reason why our experiments perform even better than estimated is that also other trap causes like, for example, memory access faults can cause traps and therefore contribute when recovering from the random execution state.

Table 3.4: Estimated power consumption for *Remus* and *Patronus* with and without SCFP at 50MHz clock frequency.

Benchmark	Power (in mW)			
	Remus		*Patronus*	
	basel.	with SCFP	basel.	with SCFP
aes_cbc	7.45	9.25 (24.1 %)	13.59	15.36 (13.0 %)
bubblesort	6.18	8.16 (32.0 %)	13.21	15.05 (13.9 %)
conv2d	7.40	8.97 (21.2 %)	14.48	15.48 (6.9 %)
fdctfst	7.56	9.35 (23.7 %)	13.89	15.55 (12.0 %)
fft	7.31	9.13 (24.9 %)	13.93	15.42 (10.7 %)
fir	6.98	9.20 (31.8 %)	13.45	15.71 (16.8 %)
keccak	7.71	9.57 (24.1 %)	14.52	16.38 (12.8 %)
matrixAdd	7.18	9.06 (26.2 %)	13.54	15.50 (14.5 %)
matrixMul16_dotp	7.56	9.23 (22.1 %)	13.89	15.75 (13.4 %)
matrixMul8_dotp	7.36	9.13 (24.0 %)	13.94	15.68 (12.5 %)
sha	7.88	9.80 (24.4 %)	14.04	15.85 (12.9 %)
stencil	6.71	8.53 (27.1 %)	13.58	15.25 (12.3 %)
Average	7.27	9.12 (25.4 %)	13.84	15.58 (12.6 %)

Table 3.5: Measured power consumption for *Patronus* with and without SCFP for the fir benchmark.

Frequency	Power (in mW)		
	basel.	with SCFP	overhead
20	5.25	6.16	17.8 %
40	9.44	11.28	19.5 %
60	13.44	16.28	21.1 %
80	16.42	19.92	21.3 %
100	20.13	24.10	19.7 %

3.6 Conclusion

Modern devices are exposed to a wide range of attacks, such as code injection, code reuse attacks, and fault attacks. While there are suitable countermeasures for each of these attacks, nowadays' IoT devices hardly implement any protection mechanism. On the other hand, it requires several of the existing countermeasures to mitigate all of the mentioned attacks. However, a combination of different countermeasures is hard to analyze and may result in overheads that are too large for IoT devices.

To overcome this limitation, this chapter introduced Sponge-Based Control-Flow Protection (SCFP). SCFP uses sponge-based authenticated encryption to encrypt and authenticate software with instruction-level granularity. During runtime, a hardware extension continuously decrypts instructions at the latest possible point before the processor's decode stage. As a result, SCFP effectively protects confidentiality and authenticity of the software IP, and provides fine-grained CFI to prevent code injection, code reuse, and fault attacks on the control-flow. The CFI enforced by SCFP is compatible with interrupts and standard operating systems. To emphasize the flexibility of SCFP, we further introduced three different instances of SCFP for different application purposes. While AEE provides all security features at cryptographic levels of security, AEE-Light reduces the level of software authenticity in trade for smaller memory overhead. In addition, IE is a very lightweight CFI scheme without any guarantees w.r.t. software authenticity and confidentiality.

Finally, we demonstrated the practicality of SCFP by extending a RISC-V processor core with an instance of AEE-Light and evaluating a set of benchmarks. Our evaluations indicate that AEE-Light is suitable for many IoT scenarios with low code size and runtime overheads of 19.8 % and 9.1 % on average, respectively. The area and power overhead, determined from the manufactured ASIC, are in a similar range and support this conclusion.

Interestingly, techniques like SCFP not only harden a processor against various kinds of attacks but also introduce new opportunities for building additional security features on top of them. In the following chapter, we explore one particular extension idea and build a novel remote attestation scheme on top of SCFP.

4

Remote Attestation and Licensing via Secure Code Execution

The authenticity of devices and software is a central problem that any Internet-of-Things (IoT) based service provider is challenged with. After all, counterfeited gadgets as well as tampered software can harm a service's reputation and/or cause financial losses. Remote attestation is the proven tool to solve this authentication problem that helps in establishing trust between communication partners.

Typically, remote attestation techniques provide evidence that a specific software image has been loaded into the memory of a certain device. For this purpose, the embedded device computes an authentic hash over the software image before it is being used and sends this digest in a challenge-response protocol to the remote verifier. However, such static attestation schemes [Eld+12; Noo+13] do not include information about the actual execution of the software on the embedded device. Static remote attestation hence inheres a Time-Of-Check Time-Of-Use (TOCTOU) vulnerability as it fails to detect runtime attacks on both hardware, e.g., fault injection and memory manipulation, and software, e.g., code-injection and code-reuse attacks.

To detect runtime attacks and prevent TOCTOU vulnerabilities, various runtime attestation schemes have been proposed. Initially, these schemes focused on preventing software attacks during runtime [Abe+16; Des+17; Sun+18] only. More recent works [Zei+17], however, take specific physical attacks into account as well, such as changing software in memory during the execution. Runtime attestation schemes typically perform so-called *path attestation* and generate their attestation report by computing an authentic hash of all executed instructions and their respective addresses in memory. While this fingerprints the specific execution, the remote verifier requires information about the concrete execution

path, such as taken branches and return addresses, to be able to verify the attestation report. Sending and processing this metadata within path attestation eventually results in significant communication and verification overheads.

Contribution. In this chapter, we tackle this efficiency and security issues of contemporary remote attestation schemes and present a new concept based hardware-supported Control-Flow Integrity (CFI) schemes like SOFIA [Cle+17b] and Sponge-Based Control-Flow Protection (SCFP) (see Chapter 3). In the following we denote such schemes—based on authenticating and executing encrypted code—as Secure Code Execution (SCE) techniques.

In more detail, we present a concept for realizing both static and runtime attestation on embedded devices in the presence of software- as well as hardware-based attacks by making use of SCE techniques. In our concept, we introduce the novel *graph attestation*, which leverages properties that are inherent with SCE techniques in order to attest the execution of software at minimal cost. Namely, SCE schemes feature a secret device state that is unique for each particular instruction within a program. As this state reflects the sequence of all previously executed instructions, graph attestation at runtime makes use of this device state to determine the authenticity of execution. Using this approach, graph attestation is capable of proving that the execution reached particular checkpoints within a program.

Our remote attestation concept works at arbitrary granularity, which allows to trade off between performance and detection accuracy. For example, graph attestation may be used to attest a single checkpoint at the end of a program, but our concept also allows to build a path attestation scheme by attesting every executed instruction. While the first variant gives evidence that the program executed validly until the end, the second attests the concrete execution path. Hereby, our concept is compatible with previous techniques to encode the execution path as it is transferred to the verifier for checking the attestation report. However, our combination of remote attestation and SCE also is a measure to remotely enforce the execution of specific code parts and to obtain a proof of its execution. Moreover, our attestation concept supports the attestation of runtime user data to tackle data-oriented attacks.

We further present an approach to realize online licensing for IoT devices based on remote attestation and SCE. In this approach, an IoT device uses remote attestation to prove to a verifier that its execution is authentic as it reaches a certain checkpoint within a program. The verifier then responds by sending a token to the IoT device for updating the state within SCE which allows to proceed its execution. This licensing technique can, for example, be used to unlock software features online.

To demonstrate the practicality of our techniques, we present a prototype implementation of our remote attestation concept based on SCFP. This implementation uses the Keccak permutation in a sponge-based Message Authentication Code (MAC) to generate the attestation report. The software implementation is evaluated on an SCFP-enabled 32-bit RISC-V processor in terms of runtime,

memory efficiency, and size of the Keccak permutation. This evaluation reveals that the 800-bit Keccak permutation is the most suitable on the utilized platform with an average attestation performance of 78 cycles-per-byte (cpb), an attestation call overhead of 300 cycles, and a code size of 3540 bytes for the attestation part.

Outline. This chapter is organized as follows. Section 4.1 gives an overview on the state of the art of remote attestation and SCE. Section 4.2 presents our concept for realizing remote attestation based on secure code execution and in particular introduces our novel graph attestation and licensing approaches. The implementation is part of Section 4.3, which is evaluated in Section 4.4. Section 4.5 finally concludes this work.

4.1 Background

Remote attestation techniques allow to verify the authenticity of a program or its execution on a remote device. On the other hand, SCE techniques are designed to prevent malicious code from being executed on a device at all.

As this work aims to close the gap between these two mechanisms, this section gives an overview on existing approaches to remote attestation as well as SCE. We first focus on remote attestation and discuss current techniques to attest program code and its execution paths. We then present state-of-the-art concepts for SCE.

4.1.1 Remote Attestation

Remote attestation techniques are usually constructed as a challenge-response protocol, where the verifier sends a challenge to a prover, who then returns an attestation report to the verifier. Using this attestation report, the verifier can determine whether the software run by the prover is authentic. However, the hard- or software component generating the attestation report at the prover needs to be trusted, *i.e.*, to be part of a Trusted Computing Base (TCB). Commonly, this TCB is formed by a Trusted Platform Module (TPM) or isolation techniques such as ARM TrustZone.

Static Attestation

The easiest approach to remote attestation is static code attestation, such as in [Eld+12; Noo+13]. Hereby, the prover generates the attestation report before the attested code is being executed by computing a MAC or hash over both the input challenge and a measurement of the program code. While this approach proves to the verifier that the correct code has initially been loaded to the memory, it does not prevent an attacker from changing the program binary after the MAC has been computed, e.g., by injecting code from within software or physically manipulating the memory. State-of-the-art technologies, such as Intel

SGX [Gue16], hence additionally use access control and memory authentication to counteract these kinds of TOCTOU attacks on static attestation. However, all static attestation schemes are still inherently vulnerable to fault-injection and code-reuse attacks, such as Return-Oriented Programming (ROP) [Sha07] and Jump-Oriented Programming (JOP) [Ble+11]. These TOCTOU attacks can, on the other hand, be prevented by attesting the actual program execution.

Path Attestation

The common approach to attest the actual program execution is path attestation. Contrary to static attestation, path attestation prevents TOCTOU attacks by taking into account the actually executed program path on the Control-Flow Graph (CFG). In particular, the prover generates the attestation report by computing a MAC over the input challenge and the measurement hash of the sequence of executed Basic Blocks (BBs) on the program's CFG. Such a BB is a linear sequence of instructions with a single entry and a single exit point, *i.e.*, without jumps, and a program's CFG describes transitions between a program's BBs. In this respect, BBs are characterized by their start and end addresses. Hence, a variant to implement path attestation [Abe+16; Des+17] is to hash source and destination address of each Control-Flow Transfer (CFT).

While using BB address information to attest the executed control-flow path protects against software attackers, it fails when attackers have physical access to the attested device. In particular, physical access allows attackers to circumvent such attestation scheme [Zei+17] by replacing the attested program with another one having the same layout of the CFG, but different instructions. As a result, Zeitouni et al. [Zei+17] additionally include the instruction encodings in their measurement hash to protect against physical attackers.

However, path attestation includes information about the actually executed program path, meaning that the measurement hash depends on the program's data inputs. The attestation report must hence include metadata about the taken Control-Flow Path (CFP) in order to allow verification at the verifier. Unfortunately, metadata about the CFP rises with the number of branches taken during the execution, which leads to significant communication overheads and computational effort to check the attestation report at the verifier. Consequently, several proposals [Abe+16; Des+17; Sun+18] exist to minimize the metadata to be transferred to the verifier, e.g., in case of loops.

4.1.2 Secure Code Execution

Path attestation allows to retrospectively find out whether a remote device executed the wrong code, but yet cannot prevent a device from executing malicious code. On the other hand, SCE techniques [Cle+17b; Wer+18] only execute code if its authenticity has been verified beforehand.

For this purpose, SCE schemes typically use a modified processor core that fetches encrypted instructions from the memory and decrypts them at the latest point possible before they are fed into the processor's decoder. In addition,

the encrypted instruction stream can be augmented with redundancy such as authentication tags that the processor can use to explicitly check the instructions' authenticity. In this way, any attacker not in possession of the correct encryption and authentication key is unable to create valid code for the target processor. Instead, malicious code created by attackers either leads to the execution of a pseudo-random instruction sequence [Wer+18] or is detected by the processor's authenticity check [Cle+17b].

SCE schemes additionally enforce that instructions are executed in their intended order. For this purpose, SCE schemes use a stateful encryption and/or authentication primitive where the state reflects the execution order of instructions. In particular, the state in a SCE scheme takes into account the concrete location of an instruction in the program binary as well as information about its predecessor instructions. However, such a stateful technique requires an unique state for each instruction that must be available independent of which execution path in the CFG led to the respective instruction. For this reason, secure code execution schemes adjust the internal state at valid merge points in the CFG to obtain a unique state value for each instruction in the program. As a result, the encryption/authentication primitive is bound to the CFG of the program and enforces that the execution adheres to its intended CFG.

Note, however, that SCE schemes provide only limited protection against data-oriented attacks such as manipulation of function pointers or return addresses. In this respect, code analysis aids to restrict allowable address targets at specific call sites and shadow stacks can offer protection of return addresses. SCE-based concepts, as in Section 4.2, naturally inherit many of the SCE security properties.

4.2 Remote Attestation Concept

As pointed out in Section 4.1, state-of-the-art remote attestation schemes suffer from TOCTOU vulnerabilities or inhere high overheads. On the other hand, SCE schemes fail to prove to a remote verifier which program is actually executed on a device.

In this section, we overcome these shortcomings by bridging the gap between remote attestation and SCE techniques. In particular, we introduce a holistic concept for remote attestation, which does not only allow to implement conventional static and path attestation, but also presents the novel *graph attestation*. This graph attestation leverages SCE techniques to prove the authenticity of execution at low cost. Besides, this section uses our attestation concepts to further present an innovative method for implementing online licensing checks.

4.2.1 Threat Model and Trusted Computing Base

This work deals with the attestation of software running on IoT devices, which face both software and physical attacks. Software attacks are assumed to remotely exploit software bugs that enable arbitrary read and write accesses to the memory as well as code injection and code-reuse attacks. Physical attacks, on the other

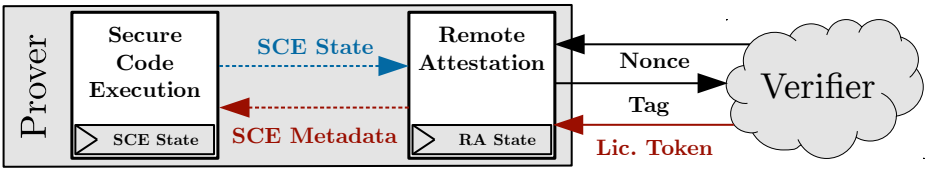


Figure 4.1: Concept for remote attestation based on secure code execution. Blue and red paths were added to support graph attestation and licensing, respectively. Dashed paths are confidential.

hand, become feasible as attackers have direct physical access to the IoT device. Physical attacks are very powerful and range from reading and manipulating data stored in external memory to probing and forcing Printed Circuit Board (PCB) buses. Moreover, physical attackers may also inject global faults into a microchip, e.g., by using clock glitches, in order to exploit faulty computations. However, we do not consider attackers performing invasive attacks such as probing of the signals on a chip. Similarly, we regard side-channel attacks on hard- and software implementations to be out of scope of this work.

In terms of the Trusted Computing Base (TCB), only the runtime components that implement the SCE scheme and the implementation of the cryptographic primitives for the remote attestation are part of the TCB. On the other hand, the application that gets attested, as well as the protected functionality that can be unlocked via the licensing extension, is *outside* of our TCB and can be arbitrarily exploitable.

In the most minimal case, as discussed in Section 4.3, the TCB comprises the SCE hardware, a working MMU/MPU with on-chip memory, and a small amount of code—executed in the most privileged processor mode—that implements the cryptographic operations. However, if needed, large parts of this cryptographic software can also be implemented in hardware instead. Note also that all the key material that is either stored (e.g., decryption key for the SCE scheme, attestation device key), received (e.g., SCE metadata for the licensing feature), or computed at runtime (e.g., internal SCE and attestation states) has to be kept confidential and should only be accessible to the TCB components.

4.2.2 Overview

Our holistic attestation concept in Figure 4.1 combines ordinary remote attestation with SCE. As usual, remote attestation receives a challenge nonce from the verifier in order to attest data by computing and returning a tag. However, to further attest the execution of code on the prover device, remote attestation generates its report by including SCE states, which reflect the validity of all previously executed instructions. To additionally realize an efficient licensing technique, our remote attestation component is extended to receive licensing tokens from the verifier and to transform them into SCE metadata, which allows to unlock the execution within the SCE scheme.

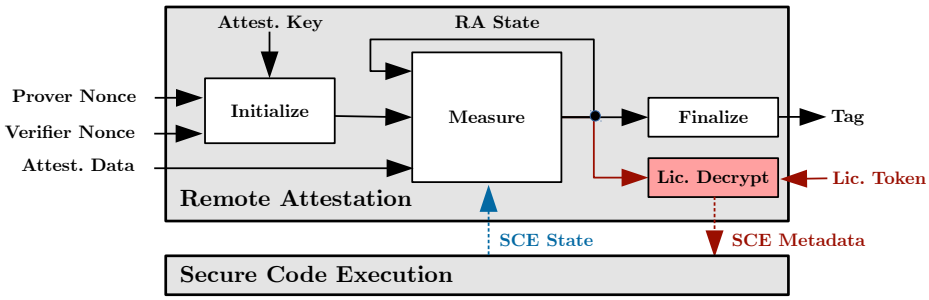


Figure 4.2: Dataflow of remote attestation for a prover. The white components form a MAC and the block highlighted in red is a cipher.

Internally, remote attestation builds upon a MAC as shown in Figure 4.2. In this MAC construction, first the internal attestation state is *initialized* using a secret attestation key as well as some randomness. Both the prover and the verifier should contribute randomness during initialization, either explicit via nonces as depicted in Figure 4.2, or implicit as part of attestation-key agreement or derivation. In the second step, the *measurement* phase, the data being attested is iteratively injected into the attestation state. Hereby, the data being attested may comprise internal SCE states as well as any kind of data that is stored inside memory, such as the encrypted binary or data constants. Each measurement iteratively updates the attestation state given the previous state and the injected data. Similar to hash-based schemes, this permits to map an arbitrarily long input stream to a fixed size and results in *Measure* being a compression function.

In our concept, we use the internal attestation state in two different ways. The first use case is the calculation of authentication tags via a *finalization* function. These tags are part of our attestation reports and are sent to the verifier, along with metadata about the initialization and the measurements performed, for verification. Since it is not necessary to invert the finalization operation, we recommend instantiating *Finalize* with a one-way function to ensure that the tag cannot leak the internal state to adversaries.

The second use case of the internal attestation state is the *decryption* of metadata values from licensing tokens. The verifier can provide such licensing tokens to the prover to unlock specific functionalities. Without this licensing token, the underlying SCE scheme will be unable to run the respective code any further. Unlike the tag computation, *decryption* has to be invertible, because the verifier calculates the licensing token from the required metadata value and the corresponding attestation state. From a cryptographic point of view, licensing tokens can hence be considered to be ciphertext and the metadata values to be plaintext. In this respect, our construction in Figure 4.2 is similar to an Authenticated Encryption (AE) cipher where the attested data is considered to be associated data. However, in our concept, explicit authentication of the tokens and the metadata is not a requirement. Namely, authentication is already done implicitly given that erroneous values result in a modified SCE state, which

subsequently prevents further program execution. In addition, measurements, (intermediate) tag generation, and token decryption can be interleaved arbitrarily in our use case.

Note that to validate an authentication tag and to generate licensing tokens, the verifier needs to have access to the attested data, the measured SCE states, and the attestation key. When the verifier generated the executed program itself, these requirements can be easily fulfilled. Otherwise, the respective information has to be provided by the program creator and/or the prover.

Moreover, note that the dataflow visualized in Figure 4.2 bears high resemblance with a sponge construction. Our prototype in Section 4.3 hence builds upon such a construction and showcases its suitability for realizing our concept.

4.2.3 Attestation Modes

In the following, we discuss how the various attestation modes can be implemented using our attestation concept.

Static Attestation

Static attestation schemes typically calculate a hash or a MAC over a challenge nonce and a specific chunk of memory. The resulting digest or tag is then sent to the verifier and compared with a locally computed value. If the check succeeds, the verifier is convinced that the prover possesses the attested code and/or data is genuine.

Considering that the required MAC functionality is part of our concept, implementing static attestation in this way is easily possible. Keys and challenges can be injected into the attestation state either via the *initialize* step or via the first *measurement* steps. Afterwards, the memory area that should be attested is iteratively measured and a tag is generated.

Security. Francillon et al. [Fra+14] already mentioned that the attestation function in a static remote attestation scheme is conceptually very similar to the computation of a MAC. However, as they detail, certain properties need to be fulfilled in order to achieve security. Namely, attestation keys have to be exclusively accessible by the attestation implementation and are not allowed to leak. Furthermore, the attestation process should not be arbitrarily interruptible and must be invoked in a well-defined, controlled manner. Finally, the attestation routines should be immutable. Note that all of these properties aim to protect the implementation of the attestation routines itself against attacks. Our threat model captures these properties by placing the implementation in the TCB. However, for static attestation the immutability property is also a requirement for the attested memory in order to prevent TOCTOU attacks. Nevertheless, all these properties are considered to be implementation requirements that have to be argued on a case-by-case basis as soon as the concrete instantiation is fixed.

Regarding cryptographic security, the strength of static attestation using our concept is solely determined by the strength of the MAC. It is therefore advisable to instantiate the MAC, that is formed by the *Initialize*, *Measure*, and *Finalize*

functions in our concept, with well-analyzed building blocks. Our proof of concept implementation presented in Section 4.3, for example, uses an instantiation of the well-studied Keccak permutation that has, in a different parameterization, been standardized as part of the SHA-3 hash function.

Graph Attestation

Graph attestation is a novel type of attestation for code and can be considered a hybrid between static attestation and path attestation. In more detail, in the first step, a key and a challenge are injected into the attestation state, either via the *initialize* function or via a *measurement* call. In the second step, the SCE state, e.g., from SCFP, is injected into the attestation state using the *measurement* function. Finally, in the last step, a tag is generated and sent to the verifier for validation.

The successful validation of the SCE state attests that, up to the measurement point, all executed instructions have been authentic and in the desired order. This is equivalent to ensuring that the executed path adhered to the CFG of the program, hence the name. Interestingly, even a simple one-shot graph attestation at the end of the program already provides very valuable information about the execution of a program to a verifier. Namely, due to the use of a verifier-selected challenge, and as long as all internal states and keys are properly protected by the prover hardware, successful graph attestation affirms that the attested code was actually executed on request of the verifier. Graph attestation can hence be used to build proof-of-work systems for arbitrary (meaningful) code.

Graph attestation leverages the fact that SCE techniques like SCFP and SOFIA require a unique mapping between the CFG and the SCE execution state. SCE schemes establish this mapping by actively compensating the state differences that result from different valid execution paths within the program. Invalid execution paths as well as previously detected errors, on the other hand, are not affected by this transformation. As a result, there is no need to record the exact execution path and to transmit it to the verifier for validation.

Assuming that SCE is already deployed, one-shot graph attestation therefore only induces little additional overhead. Namely, the attestation state must be initialized during startup and only one measurement and the finalization have to be computed at the end of the program. On the other hand, no additional overhead during the execution of the program is induced.

Security. The security of the composition proposed for graph attestation in Figure 4.1 naturally relies on both the security of SCE and remote attestation. However, such composition must also avoid any side effects that may result from establishing a link between these two components. In particular, this means that revealing the internal SCE state to remote attestation must not break the security of SCE and remote attestation. Fortunately, the security of remote attestation is independent of the type of data being attested, *i.e.*, the secret SCE state. On the other hand, the security of SCE is lost in case internal SCE states leak. This effectively imposes an additional requirement on the implementation. Namely, besides the tag output, the implementation of remote attestation and

the respective link to SCE must not reveal anything about the secret SCE states being processed.

From another perspective, graph attestation slightly relaxes the needed protection for code compared to static attestation. In more detail, the used SCE scheme already actively counteracts TOCTOU attacks by preventing the execution of tampered code. For this reason, no additional arrangements have to be taken to ensure the immutability of the attested code. The proper protection of the TCB, comprising the involved keys and the attestation implementation, is however still necessary.

Regarding cryptographic properties, the security of graph attestation is determined by the security of both the attestation MAC and the SCE scheme. Breaking either of the two building blocks defeats the overall attestation and may even render the other component useless. Assuming that the SCE scheme has the greater impact on performance, we anticipate that typically the SCE scheme will dictate the level of security for most instantiations. On the other hand, the MAC instantiation can usually be strong since only little amounts of data have to be processed. Still, having the possibility to adapt the overhead and security of the MAC in our attestation concept to the individual needs of the application is highly valuable.

Path Attestation

Using graph attestation as a foundation, path attestation can also be easily implemented in our attestation concept. In more detail, while the overall initialization and finalization approach is the same as in graph attestation, path attestation is achieved by simply performing multiple measurements of the SCE state during the runtime of the program.

Performing attestation of the SCE state instead of the actually executed instructions provides again a huge advantage over related work. Namely, the position of the measurement points can be chosen arbitrarily. Any parameterization between measuring the SCE state at every instruction and measuring the SCE state only at the end of the program is possible. Between the single measurement points, graph attestation is in use.

Note that to validate a tag in the path attestation concept, the verifier needs to know the exact positions in the code where and in which order measurements have been performed during the execution. In this respect, all generic techniques that have been proposed in related work to capture this information can similarly be applied to our concept.

Security. As our path attestation approach is a generalized version of graph attestation, its security properties and requirements are very similar. Namely, the SCE scheme implicitly provides code immutability and the implementation of remote attestation must not reveal anything about the secret SCE states other than the attestation tag. This requirement is typically fulfilled by remote attestation using a MAC to iteratively absorb multiple SCE state values.

The cryptographic security of the path attestation approach is as well determined by the security of both the MAC and the SCE scheme. Therefore, the

same basic instantiation considerations hold true. Note however that, due to the increased number of measurements, performance of the MAC is getting more important.

Interestingly, path attestation does not only rely on the security of the underlying SCE scheme, but, in terms of attestation capabilities, also improves it. For example, Data-Oriented Programming (DOP) attacks are not prevented by solely enforcing CFI using the SCE scheme, whereas path attestation is capable of detecting such DOP attacks. Furthermore, internal state collisions within the SCE scheme are much harder to exploit when path attestation is performed.

Hybrid Approaches

The MAC-based attestation approach within our concept allows to arbitrarily mix and match the different attestation variants. For example, it is possible to use static attestation to measure code and constants after startup and combine it with graph attestation at selected positions within the software. Similarly, path attestation can be intertwined with attestation of static data as it is used during runtime. For example, values from processed lookup tables and other intermediate data like loop counters can be injected into the attestation state as desired.

Interestingly, exposing the measurement functionality to the attested code itself, e.g., via a syscall interface or as a special instruction, allows to delegate the decision on what to measure completely to the software. This eventually adds a high degree of flexibility to path attestation, because, contrary to previous approaches, it is not any more required to perform measurements with fixed granularity, e.g., every basic block or control-flow transfer. Even further, having a software that itself defines the measurement points used for remote attestation can also lead to very compact representations for the metadata to be sent to the verifier for checking the attestation report. Our prototype implementation in Section 4.3 will give a concrete example of how to realize attestation with measurement points defined in software.

4.2.4 Licensing Extension

Remote attestation, as presented in the previous subsections, enables a prover to convince a verifier that a certain program is executed and that it behaves correctly. Our licensing extension augments these attestation concepts with the capability to unlock the execution of protected code fragments.

The general idea behind our licensing extension is to embed special, protected code snippets, *i.e.*, some functions, into the program of the prover. Without additional information from the verifier, these functions cannot be executed by the prover. It is therefore up to the verifier to provide this additional information to the prover in the form of an encrypted licensing token, typically after successful remote attestation. Subsequently, the prover is able to perform the call to the protected function.

Note that the proposed licensing tokens are specific to the executed program, the targeted function, and the current remote attestation state, which in turn depends on the used keys, the selected randomness, and the previously attested code and data. Each token is therefore a highly specialized piece of data that enables exactly one call to one specific protected method. As a result, a multitude of use cases can be realized with our licensing extension.

One of the most prominent use cases is Intellectual Property (IP) protection. Licensing tokens can be used to unlock software functionality, which in turn can enable hardware features, based on preceding attestation results. Another potential use case is, for example, to enforce interactive attestation. In the previous subsection, we described a hybrid attestation approach where a program is statically measured before the actual execution is attested via graph attestation at the end of the program. Guarding the transition between static attestation and graph attestation via a protected call, for example, ensures that only programs which pass the initial verification can be executed at all.

On the technical side, implementing this licensing approach relies on the observation that SCE schemes have to embed some kind of metadata into the program to deal with code reuse and for error detection. SOFIA, for example, interleaves MAC words with the code and introduces multiplexer blocks to enable code reuse. Similarly, SCFP intertwines patch constants that are applied on control-flow transfers. Without access to the required metadata, executing the SCE-protected code is simply not possible.

Our licensing extension utilizes this observation and protects code snippets from execution by simply removing the metadata that is needed to call the protected code. The code itself, given that it is encrypted anyway, can still be deployed as part of the program. The licensing tokens, which are provided by the verifier to the prover, contain exactly this missing information and subsequently enable the prover to execute the protected code. Note, however, that the licensing tokens are actually encrypted versions of the metadata, where the encryption key/tweak depends on the current attestation state. Therefore, even though the metadata itself is constant, a new licensing token is required as soon as the attestation state changes.

Security. The licensing extension builds upon the attestation state from our remote attestation concept. As a result, secure instantiations of both the MAC and the SCE scheme are a prerequisite for having a sane overall concept also in this use case. However, while pure attestation requires the SCE states to be protected from being revealed as they are passed to and processed within the attestation MAC, our licensing concept introduces another sensitive link between attestation and SCE to facilitate feature unlocking. This link, which is also shown in Figure 4.1, transfers sensitive SCE metadata from the attestation part to SCE and must be kept secret. Otherwise, the licensing feature will break. However, note that neither SCE itself nor attestation loses security if SCE metadata falls into the hands of attackers, because for SCE without licensing enabled, this metadata is anyway a part of the public binary to enable balancing of different CFPs.

Besides the SCE metadata link, the cipher used for decrypting the licensing tokens is another component that interacts with the attestation state. This cipher must thus neither facilitate attacks on the attestation scheme nor leak the SCE metadata. Both the cipher decrypting licensing tokens and the SCE metadata link are hence part of the TCB and need to be properly protected by the implementation.

Note that the way we use licensing tokens further stresses the need for fresh randomness during initialization of the attestation scheme. Namely, it is imperative that the implementation of the attestation scheme on the prover side injects fresh randomness during initialization in order to prevent replay attacks of the licensing tokens.

4.3 Implementation

Section 4.2 presented a concept to extend SCE techniques to remote attestation with comprehensive capabilities to trade off between performance and attestation granularity. To demonstrate the practicality of our concept, this section presents an implementation of the hybrid remote attestation approach using SCFP as the SCE scheme. We first detail the cryptographic instance we used for remote attestation and then describe our soft- and hardware implementation.

4.3.1 Instance

We realized the MAC in our attestation concept, comprising initialization, measurement and finalization functionality, using a sponge-based MAC [Ber+12b; MRV15] as shown in Figure 4.3. After initialization with a secret key K and the nonce n, this MAC continuously absorbs the data D_i to be attested into its secret state, and then outputs an attestation tag T. In this sponge-based MAC, the data absorption rate can be chosen to be as large as the permutation size b without loss of security. During tag generation, however, security is bounded by the size of the secret capacity c. In general, to achieve a certain security level κ, the capacity c must be chosen such that it fulfills $c \geq 2\kappa$ [Ber+08].

Our instances of the attestation MAC target the 128-bit security level. We hence use a 128-bit key K and a 128-bit nonce n and squeeze a 128-bit tag T. As permutation f_r, we utilize Keccak-p[b,r] with state sizes $b \in \{400, 800, 1600\}$ bits, which is also well suited to offer 128-bit security. However, depending on the concrete application, the frequency of data attestation steps, and the underlying architecture, a different state size b yields the best efficiency. Besides the state size, the number of permutation rounds is relevant for security as well. As in the CAESAR submission Keyak [Ber+16b], we use $r = 12$ rounds for Keccak-p[b,r] in all state sizes b.

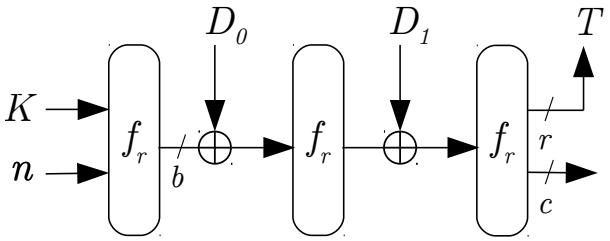


Figure 4.3: Sponge-based MAC used in our prototype.

4.3.2 Hardware

The hardware for testing our attestation concept is the *Patronus* microcontroller System-on-Chip (SoC) design (see Section 3.4). The integrated *Remus* processor features a four-stage, in-order implementation of the RISC-V RV32IM Instruction-Set Architecture (ISA) [WA17]. Most notably, this core provides support for SCFP by integrating the SCFP decryption functionality into an additional pipeline stage in between the processor's fetch and decode stage and by adding a set of new instructions to enable modification of the SCFP state in case of branches.

The SCFP instance implemented in *Remus* is AEE-Light. This lightweight SCFP instance uses the 64-bit Prince [Bor+12a] block cipher as a permutation. For the 32-bit RISC-V ISA implemented by the processor core, this results in a secret SCFP capacity of 32 bits per instruction. Combined with the 96-bit security from the keyed permutation, this is sufficient to resist common code-injection, code-reuse, and fault attacks.

4.3.3 Software

We implemented our instance of remote attestation in a C library that can be used by both the prover and the verifier. This library offers functionality to initialize the attestation state, to absorb attested data into the state, and to finalize attestation by squeezing a tag. To implement the respective sponge-based attestation MAC, the library uses simple, readable C implementations of Keccak without any architecture-dependent optimizations and which are based on the implementation of Keccak-f[1600] by Saarinen [Saa16]. The library addresses different attestation types by offering a set of measurement functions. For static attestation, a measurement routine feeds the attestation MAC with data stored in a specified memory region. For graph and path attestation, another measurement function allows to inject the 32-bit secret capacity from the SCFP state into the attestation MAC. In addition, it allows to absorb user-defined data into the MAC, enabling the attestation of runtime data like program inputs and loop variables.

Note that, in the general case this particular instantiation, due to the comparably small secret SCFP state, does only provide probabilistic assurance that the

correct software has been executed in one-shot graph attestation mode. However, cryptographic certainty levels can easily be reached with just a few measurements in a path or hybrid attestation mode. Moreover, in environments where, for example, the verifier knows that only a limited number of programs have been deployed for the prover, even one-shot graph attestation is sufficient to reliably attest the software.

Prover

The prover runs the software to be attested and uses our attestation library for generating the attestation report. The prover software is compiled for our RISC-V architecture and post-processed to adapt and encrypt the binary according to SCFP. This allows entangling attestation with SCFP. In particular, after initializing the attestation state with a secret key K and a nonce n comprising the challenge from the verifier and the randomness from the prover, the prover software absorbs the SCFP state and user data into the attestation state as desired.

The prover software is split among two different privilege levels. The functionality to be attested runs in user mode and is untrusted, whereas the attestation library runs in machine mode as part of the TCB. As a result, the prover software needs to switch to machine mode to perform attestation operations. For this purpose, the prover software uses a supervisor call to trigger an interrupt handler in machine mode. Depending on the syscall type, the interrupt handler reads the provided syscall arguments from the Central Processing Unit (CPU) registers and passes them to the respective function in the library. For path- and graph-attestation calls, the interrupt handler further reads protected Control and Status Registers (CSRs) to also provide the SCFP state and the instruction address of the respective measurement call to the library. Finally, the interrupt handler prints the instruction address and sends it as metadata to the verifier to enable verification of the prover's execution path.

Verifier

The verifier obtains an attestation tag and metadata from the prover. This metadata includes the prover's execution trace consisting of the instruction addresses where the prover absorbed the SCFP state as well as runtime data that has been absorbed. Besides this execution trace, the verifier is given a list of instruction addresses and their respective SCFP decryption state for the prover binary. This list is generated beforehand during compilation of the prover software. Using this list, the verifier software looks up the valid SCFP states belonging to the instruction addresses in the prover's execution trace and computes the correct attestation tag. In this way, the verifier can simply compare the computed and the received attestation tag to check whether the prover's computation has been correct. Furthermore, comparisons between the program and the measurement points can be performed to check if the executed path is valid.

4.3.4 Security

According to Francillon et al. [Fra+14], an implementation of remote attestation has to fulfill a set of properties to be secure. In the following, we argue on the security of our SCFP-based implementation of remote attestation with respect to these properties.

- **Exclusive Access** and **No Leaks:** These two properties relate to the implementation of the attestation library and are fulfilled by several, different security mechanisms. In particular, both the secret key material used by the attestation library and the attestation state are stored in secure on-chip memory to protect it from attackers with physical access to off-chip memory and PCB buses. To prevent malicious user software from accessing key-related data, our attestation library is run exclusively in machine mode. The Memory-Management Unit (MMU) hereby enforces that no code other than the attestation library can access attestation keys and data in machine mode. The attestation library itself does not leak any information about the attestation key other than the attestation tag, assuming that the software implementation is correct. In addition, SCFP protects the execution of the attestation routines in that library and ensures its CFI.

- **Immutability:** This property refers to the attestation routines in our TCB and TOCTOU vulnerabilities in the user software being attested.

  For the *user software*, immutability is only relevant when performing static attestation. In our implementation, we again rely on the combination of on-chip memory and the access control features provided by the MMU to achieve the desired immutability of code and constants. On the other hand, in the context of graph and path attestation of code, our prototype is immune to TOCTOU attacks anyway since code execution is protected by the SCE scheme. In more detail, our implementation uses SCFP to ensure authentic execution of the attested software by decrypting the software directly within the processor's pipeline. Hereby, any malicious modifications to the attested software result in a modified SCFP state which prevents subsequent program execution.

  Concerning the *attestation routines*, again a set of different mechanisms ensures their validity. While the attestation routines reside in on-chip memory to hamper modifications by physical attackers, MMU access control further adds protection against malicious accesses from within user software. Moreover, the attestation routines are as well run using SCFP to maintain their authentic execution.

- **Uninterruptibility** and **Controlled Invocation:** These two properties relate to the implementation of the attestation library. Our implementation adheres to these security properties, because SCFP ensures that the attestation routines can only be run in their intended manner. Namely, SCFP enforces code execution to start at well-defined entry points and to stay on valid CFPs. In addition, invocation of the attestation routines

implies changing from user to machine mode via a well-defined syscall interface, which taken by itself already hampers the malicious execution of code fragments within the attestation library.

In addition to the aforementioned properties, Section 4.2 pointed out that, in order to maintain SCE security, implementations of our concepts must further ensure the confidentiality of the SCE states as they are passed to and processed within the attestation routines. This section's implementation adheres to this additional requirement. Similar to the protection of attestation keys, MMU-based access control for the machine-mode attestation code, on-chip memory for processing SCFP secrets, and the application of SCFP to the attestation routines themselves aid to prevent the leakage of a user application's SCFP state. Moreover, CPU-internal SCFP state registers are only accessible by code running in machine mode, *i.e.*, the attestation routines. As a result, SCFP states are considered to remain confidential when the attestation routines' implementation in the TCB is assumed to be correct.

4.3.5 Implementation Aspects

Our remote attestation concept is highly flexible and offers a wide design space for implementations. However, there are also several challenges such as key management and support for multiple processes. In the following, we discuss these implementation aspects in more detail.

Implementation Challenges

While this section's implementation is confined to the attestation of bare-metal embedded applications, a more challenging target are operating systems running multiple processes. In this respect, SCE-enabled hardware typically provides secure mechanisms to read and write SCE states that allow an operating system to schedule multiple processes and threads. Our attestation concept can hence build upon these capabilities by individually attesting multiple SCE-protected threads and combining the results in a system-wide attestation report, which besides a tag should contain sufficient metadata about all attested threads. On the other hand, and as highlighted in [Sun+18], it will in many cases be sufficient to protect and attest specific, critical operations on a target platform rather than a complete system. Further note that more complex platforms featuring multiple cores and out-of-order execution do not prevent the deployment of our attestation concepts as long as SCE schemes are supported.

Quite noteworthy, implementations of our attestation and licensing techniques do not require any hardware modifications to SCE-enabled platforms. As our implementation shows, attestation can easily be implemented in software when the trusted attestation routines are given access to the SCE state in hardware. Hereby, note that a proper implementation of SCE must anyway provide access to the SCE state to allow a trusted hypervisor or operating system schedule multiple SCE-protected processes. In the same way, our online licensing technique

may be implemented purely in software by applying the SCE metadata update directly to the SCE state register and resuming program execution. Nevertheless, additional hardware components can help to significantly speed up the attestation routines, which seems particularly interesting for realizations of path attestation. In such case, an attestation hardware component may be triggered by a dedicated attest instruction or may compute the attestation MAC with BB granularity completely in parallel to SCE. As for SCE, attestation states from multiple scheduled processes can in this case be managed by a secure hypervisor or operating system.

Key Management

For the sake of simplicity, this work so far focused on the interaction of SCE with remote attestation when each of these components possess and share their own symmetric key with the program vendor and the verifier, respectively. A practical application, however, also requires a strategy to deploy these keys to a target device. A common approach to tackle this problem is the use of public-key cryptography. Hereby, each target device possesses its own pair of public and private key. The respective private key is kept secretly within the device itself and the public key is made publicly available to facilitate encryption of symmetric key material for SCE and remote attestation. Upon startup, the target device then loads and decrypts the respective key material by using its private key. In the following, the target device uses the decrypted symmetric keys to run deployed SCE-protected software and to attest its state to a remote verifier. The benefit of such key management approach is that, completely independent of a remote verifier, everyone is able to deploy SCE-encrypted software to a specific device in the field. This further emphasizes the relevance of remote attestation on SCE-enabled devices: Even when program execution is protected by SCE, remote parties do not know which program is running on the target device. Remote attestation hence complements SCE techniques by allowing remote parties to ascertain that a specific program is running on the device they communicate with.

4.4 Evaluation

The overhead of our attestation concept strongly depends on the particular type of remote attestation being used, *i.e.*, static, graph, path, or hybrid attestation. To quantify the runtime and code size overhead of our prototype, we hence perform a two-step approach. First, we perform micro benchmarks to characterize the performance of our attestation library. Second, we use these numbers to estimate the cost of arbitrary types of remote attestation based on our library. We then verify these estimates by also performing macro benchmarks.

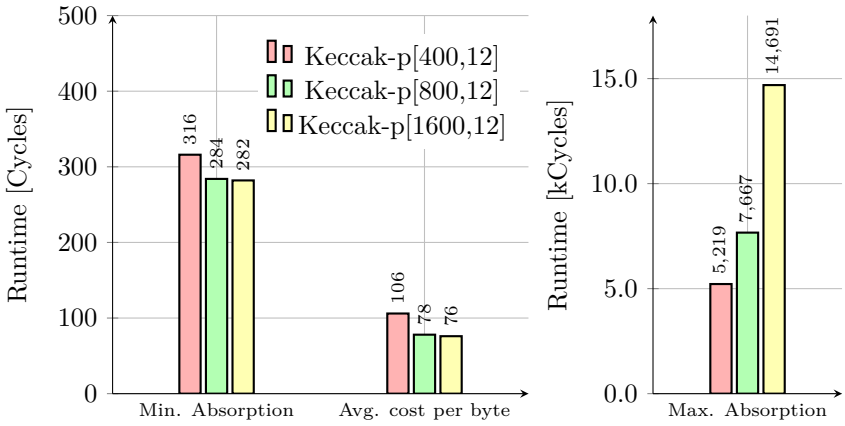


Figure 4.4: Runtime performance.

4.4.1 Library Characterization

As discussed in Section 4.3, our attestation library supports the Keccak permutation in three different sizes, *i.e.*, 400, 800, and 1600 bits. We hence characterize our library by performing a micro benchmark for each of these configurations. This micro benchmark invokes measurement syscalls, as used for graph and path attestation, in a loop and hereby measures the syscall execution time t_{absorb}.

In more detail, the benchmark invokes the measurement syscall with 64 bits of data to be absorbed into the MAC. This syscall then either yields a high execution time, when the permutation needs to be computed, or a low execution time, when the data can be directly xor-ed into the Keccak state. The minimum execution time for a measurement syscall, *i.e.*, `min` (t_{absorb}), then gives a rough estimate for the overhead of trapping to machine mode, extracting the SCFP state, and `xor`-ing 64 bits of data to the attestation state. Similarly, the maximum execution time, *i.e.*, `max` (t_{absorb}), provides an approximation of the additional cost for computing one permutation. Finally, by measuring the attestation of a larger chunk of memory and by taking into account the number of bytes that can be absorbed with one permutation, the expected net runtime cost for the permutation in terms of cycles-per-byte (cpb) has been determined.

The runtime results gathered from this naive evaluation of our attestation library are depicted in Figure 4.4. The visualization shows that the minimum cost for absorbing data into the attestation MAC is roughly 300 cycles for all configurations. However, the actual performance of the tested Keccak permutations differs notably. While Keccak-p[400,12] consumes 106 cpb on average for transforming its state, Keccak-p[800,12] and Keccak-p[1600,12] consume less than 80 cpb. The reason for this discrepancy in performance is the amount of data that can be processed with each permutation call and the performance of the respective permutation. We captured this figure by measuring the maximum

absorption runtime, which is dominated by the execution time of the permutation. Even though, in this measurement, computing the Keccak-p[400,12] permutation is the fastest with 5219 cycles, it can still not make up for the significantly smaller state compared to the larger instantiations. Keccak-p[1600,12], on the other hand, achieves a similar net runtime performance as Keccak-p[800,12], because its state is twice the size and its permutation requires approximately twice the time of the smaller configuration.

Note, however, that these results, while sufficient for our prototype, are still far from optimized. In particular, the Keccak-p[800,12] configuration should in theory be slightly faster than the Keccak-p[400,12] configuration given that the same number of rounds is computed and considering that the 32-bit lane size best suits the given processor architecture. However, less then optimal code is currently emitted by the customized, SCFP-enhanced compiler, which explains the current results.

4.4.2 Runtime Overhead Estimation and Validation

Based on our library characterization, the following formulas estimate the runtime cost for initialization t_{init}, finalization t_{final}, arbitrary measurement operations $t_{measure}$, and the whole attestation t_{total}.

$$\begin{aligned} t_{init} &= \texttt{max}\,(t_{absorb}) \\ t_{final} &= \texttt{max}\,(t_{absorb}) \\ t_{measure} &= n_{measurements} \cdot \texttt{min}\,(t_{absorb}) + n_{bytes} \cdot cpb \\ t_{total} &= t_{init} + t_{measure} + t_{final} \\ &= 2 \cdot \texttt{max}\,(t_{absorb}) + n_{bytes} \cdot cpb \\ &\quad + n_{measurements} \cdot \texttt{min}\,(t_{absorb}) \end{aligned}$$

In more detail, initialization and finalization each consist of a context switch and exactly one permutation call and are hence approximated by the maximum measured absorption time. On the other hand, the overhead of the measurement operation depends on the concrete attestation type and linearly scales with the number of absorbed bytes n_{bytes} and the number of measurement calls $n_{measurements}$. Hereby, context switches induce an overhead in each measurement call that is approximated with the minimum absorption time. In addition, each absorbed byte requires cpb time in the permutation on average. Finally,

Table 4.1: Runtime overhead when attesting coremark.

	Measured [cycles]	Estimated [cycles]
Static Attestation	4,615,903	4,629,993
Graph Attestation	13,547	16,242
Path Attestation	198,065	197,848

the runtime cost for the overall attestation operation t_{total} is the sum of the individual components. Inserting the numbers acquired during library characterization yields a formula for t_{total} that estimates the overhead of our prototype for all the discussed attestation modes including the hybrid approach. Using Keccak-p[800,12] within our library, e.g., yields the following overhead formula:

$$t_{total} = 15334 + 284 \cdot n_{measurements} + 78 \cdot n_{bytes}$$

Note that the algorithmic complexity for the different remote attestation approaches can directly be deduced from the derived formula. Namely, for straight-forward static attestation, the overhead scales with the number of attested bytes, *i.e.*, $O(n_{bytes})$, given that the number of measurement calls is typically fixed to one. Similarly, one-shot graph attestation only performs a single measurement of the fixed-size SCE state and thus has the complexity $O(1)$. Finally, traditional path attestation attests a fixed amount of bytes per measurement call, *i.e.*, $n_{bytes} \propto n_{measurement}$, and hence its overhead scales with the number of measurement calls and has complexity $O(n_{measurements})$.

Validation. To countercheck that our overhead estimation provides sensible results, we implemented different remote attestation approaches for coremark[1] and compare the measured overhead with the expected value. In more detail, for 20 iterations of the program, we measure exactly how much runtime overhead is introduced by computing attestation tags for static, one-shot graph, and path attestation, respectively. For static attestation, we measure the SCE encrypted code segment of the application, *i.e.*, 59156 bytes, before executing the program. For graph attestation, the SCE state at the end of the benchmark including the total execution time of the SCE loop are measured. Finally, for path attestation, additionally the SCE state as well as the result of the `core_bench_list`, `core_bench_state`, and `core_bench_matrix` functions is measured. Thus, already 201 measurements of 8 bytes each are performed when 20 benchmark iterations are calculated. Note that, even though reporting actual values is not permitted when the benchmark code is modified, neither static nor graph attestation negatively impacts the determined coremark score. Both modes do not require to perform additional computations within the benchmarked region. Path attestation, on the other hand, decreases the score given that additional measurements are intertwined with the benchmark.

The determined runtime overhead values, for the Keccak-p[800,12] variant of our library, are summarized in Table 4.1. It can be seen that even our simple runtime overhead estimation approach already provides quite reliable results on the benchmarked hardware. Considering that only the total number of measurements and the total number of measured bytes has to be known, the overhead of arbitrary hybrid schemes can therefore be estimated using the formula. Alternatively, when the tolerable overhead is constrained by a maximum value, inverting the formula and deriving the required performance characteristics for the attestation implementation is also possible.

[1] https://github.com/eembc/coremark

4.4.3 Memory

In addition to runtime, we determined the code size of our attestation routines and the amount of data stored in memory. The respective results are visualized in Table 4.2. It shows that the size of data held in memory is dominated by the size of the attestation state. In terms of code size, the implementations for the different state sizes differ considerably. For example, the implementation using a 1600-bit Keccak state is with 5472 bytes the largest due to extensive loop unrolling, more complex rotation operations, and additional memory accesses. The implementations using the 400- and 800-bit Keccak permutations, on the other hand, require around 3500 bytes for code, which is only 64 % of the code size for Keccak-p[1600,12].

4.4.4 Further Remarks

As shown in our estimation formula, the attestation runtime overhead does not only depend on the performance characteristics of the implementation, but also on the selected attestation methodology and the attested application.

We therefore deliberately refrain from directly comparing the performance and overhead of our prototype with related work. On the one hand, in terms of graph attestation, there is yet no related work which features similar capabilities. On the other hand, the software that has been attested by related work is either not available or hardly meaningful. The commonly referenced open syringe pump Arduino code[2], for example, has to be heavily modified to be runnable on custom hardware which defeats the purpose of being comparable. Moreover, it only consists of a few nested loops with function calls and can easily be replaced by any benchmark program. Finally, for path attestation in the context of physical attacks, all the related work we are aware of solely relies on hardware implementations. Considering that our prototype only implements the SCE scheme in hardware and the remaining attestation functionality in hardly optimized software, no sensible results are expected from such a comparison.

[2] https://github.com/naroom/OpenSyringePump/blob/master/syringePump/syringePump.ino

Table 4.2: Memory requirements.

	Code Size [bytes]	Data Size [bytes]
Keccak-p[400,12]	3,464	64
Keccak-p[800,12]	3,540	112
Keccak-p[1600,12]	5,472	216

4.5 Conclusion

Remote attestation provides a tool to prove the authenticity of an embedded device to a remote user. However, current remote attestation schemes have significant drawbacks. While static attestation schemes are vulnerable to TOCTOU attacks, path attestation is able to detect runtime attacks, but requires the transmission of the whole execution path for verification.

To overcome these issues, this work presented a novel approach to remote attestation for embedded devices in the presence of both software and hardware-based attacks. In this concept, we leveraged the properties of SCE techniques to introduce graph attestation. Graph attestation is a highly efficient variant to attest the execution of an embedded device. In particular, graph attestation makes use of the instruction- and address-dependent secret that is used for instruction stream decryption within SCE to prove that the execution has been valid until a certain checkpoint. This approach is highly flexible and allows to trade off between performance and detection accuracy. Namely, a single checkpoint at the end of a program is sufficient to attest the validity of the program execution up to that point. On the other hand, placing checkpoints at arbitrary positions in the program, possibly even at every single instruction, is feasible as well and permits to implement full path attestation on top of graph attestation.

We further presented an approach to perform online license checks in embedded devices by extending our remote attestation concept based on SCE. In particular, our licensing extension provides a mechanism to remote parties that allows to unlock the execution of protected code snippets at the embedded device by using special licensing tokens. Namely, only if the embedded device executed correctly and remote attestation yields a positive result, the device is able to decrypt these tokens and to inject them into the SCE scheme to enable execution of the protected code. Otherwise, correct decryption and application of the licensing token is impossible for the embedded device.

Finally, we implemented our remote attestation concept using a sponge-based MAC and the Keccak permutation with different state sizes. We evaluated our implementations on an 32-bit RISC-V processor supporting SCFP-encoded software regarding runtime and code size. This evaluation showed that on this platform the 800-bit Keccak permutation is the most suitable for attestation as it yields a code sized as small as 3540 bytes and a throughput of efficient 78 cpb when performing measurements.

Part II

Counteracting Physical Attacks on the Memory System

Efficiently and securely interacting with memory is, after the actual computation, the second most important task of a processor. Unfortunately, contemporary processors also fail in this domain as soon as physical characteristics of the device can be monitored (e.g., side-channel attacks) or direct access to the memory is granted (e.g., probing of PCB signals).

This second part of the thesis, hence, focuses on the protection of arbitrary data in the memory subsystem by presenting a hardware framework for memory encryption and a side-channel hardened cache architecture. The content of Part II is largely based on two publications [Wer+17; Wer+19b]. The enumeration below maps the individual chapters to the respective papers, clarifies my contributions, and acknowledges the work of my collaborators.

- Chapter 5 is primarily based on the following publication that was presented at FPL 2017 in Ghent (Belgium):

  Mario Werner, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard. "Transparent memory encryption and authentication." In: *Field Programmable Logic and Applications – FPL*. 2017, pp. 1–6. DOI: 10.23919/FPL.2017.8056797

  I am the main author of this paper, wrote the majority of the text, built the proof of concept, designed and implemented large parts of the presented HDL framework, and performed all benchmarks. Thomas Unterluggauer contributed to the text, the design, and heavily to the implementation of the final framework. Robert Schilling and David Schaffenrath implemented a prior memory encryption hardware module. While the initial implementation had to be dropped due to design issues, valuable insights on the problem domain were gained. Stefan Mangard provided the original idea on investigating memory encryption on Zynq FPGAs.

- Chapter 6 is primarily based on the following publication that was presented at the USENIX Security Symposium 2019 in Santa Clara (California, USA):

  Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization." In: *USENIX Security Symposium*. 2019, pp. 675–692. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/werner

  Again, I am the main author of this paper, contributed the initial idea, and lead the design of ScatterCache in the numerous design discussions. Furthermore, I wrote a major part of the text, built the Yocto-based software stack, and performed final evaluations with gem5. Thomas Unterluggauer contributed to the text, performed the initial gem5 simulations, and helped greatly in shaping the concept as well as the analytical analysis. Lukas Giner implemented the custom cache simulator, performed the SPEC evaluations, and delivered numerical analysis results including the respective paper text. Michael Schwarz, Daniel Gruss, and Stefan Mangard worked on the text and provided invaluable guidance on the concept in terms of feasibility and security implications.

5 Transparent Memory Encryption and Authentication

Techniques like disk/firmware encryption [Wil15], and to a certain degree secure boot [San15], are established concepts for protecting software Intellectual Property (IP) as well as sensitive data in Non-Volatile Memory (NVM). However, basically every modern device additionally relies on substantial amounts of unprotected Random-Access Memory (RAM) to process the increasing amounts of data. Subsequently, attackers with physical access to the RAM are able to read from and/or tamper with sensitive data.

There already exist several encryption and authentication techniques to protect data in RAM. The concurrently developed SCM [Cle+17c] approach, for example, provides authentication for read-only memory—usable for code. Also, CBC-ESSIV [Fru05], XEX [Rog04], XTS [Sto08], and the counter mode [DK06; Rog+07; SOD05; Yan+06] have been proposed for RAM encryption. While these encryption modes ensure confidentiality, none of them provides authenticity. Even worse, certain modes like counter mode encryption even lose confidentiality in case of active attacks such as spoofing, splicing and replay attacks [Elb+09]. In *spoofing attacks*, an attacker simply replaces an existing memory block with arbitrary data, in *splicing attacks*, the data at address A is replaced with the data at address B, and in *replay attacks*, the data at a given address is replaced with an older version of the data at the same address. To protect against these active attacks, various tree-based RAM authentication techniques, e.g., Tamper Evident Counter (TEC) trees [Elb+07], exist.

While previous work [DK06; Rog+07; SOD05; Yan+06] continually improved RAM encryption techniques, virtually all lack practical implementations and do only simulations to estimate the performance. On the other hand, recent imple-

mentations of RAM encryption and authentication as in, e.g., Intel's SGX [Gue16], AMD [KPW16], remain closed source. Yet, there is a strong need for freely available implementations given the threat of physical attacks and the trend towards custom hardware featuring embedded RISC-V System-on-Chips (SoCs).

Contribution. In this chapter, we present MEMSEC, a modular open-source framework[1] for transparent RAM encryption and authentication which is configurable for different ciphers, cryptographic modes, and block sizes. The building blocks of our framework are written in VHDL and, while being developed for Field Programmable Gate Arrays (FPGAs), are also suitable for Application Specific Integrated Circuit (ASIC) designs. We evaluate our framework to give the first comprehensive comparison of performance results of practical implementations of RAM encryption and authentication in various cryptographic configurations. For evaluation, we use the Xilinx Zynq platform and let the ARM Central Processing Unit (CPU) access the memory via our transparent memory encryption module in the FPGA. At 50 MHz, our implementations of different cryptographic modes using Prince [Bor+12a; Bor+12b] and AES give a performance upper and lower bound of 187 and 35 MB/s read bandwidth, respectively. We further show that the Authenticated Encryption (AE) cipher Ascon [Dob+16] gives very practical results for RAM encryption and authentication when replay attacks are not concerned. For applications further threatened by replay attacks, we provide an Ascon-based implementation of the TEC tree, reaching up to 47 MB/s read bandwidth.

Outline. This chapter is organized as follows. Section 5.1 describes the challenges and introduces the concepts of our memory encryption framework. Authentication trees, including the implementation using our framework, are discussed in Section 5.2. Finally, the evaluation and conclusion are content of Section 5.3 and Section 5.4.

5.1 RAM Encryption Framework

RAM is in general a very fast and heavily used system resource. Transparently encrypting it, by placing an encryption pipeline between CPU and memory controller (outlined in Figure 5.1), is therefore a challenging task. This section discusses the various challenges involved and gives details on the functionality and design rationales behind our framework. Furthermore, the application of the framework to achieve transparent memory encryption for the AXI4 bus is discussed.

5.1.1 Challenges

In modern FPGAs, RAM is typically exposed to the programmable logic via memory controllers which feature standard bus interfaces (e.g., AXI4, Avalon, ...).

[1] https://github.com/IAIK/memsec

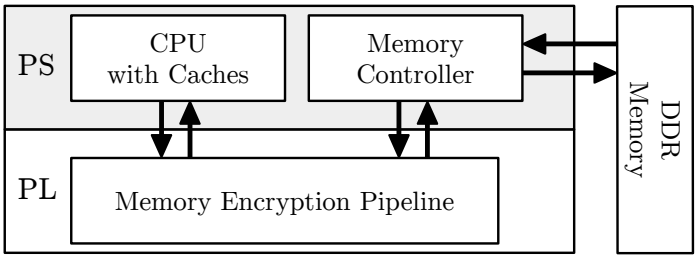


Figure 5.1: Zynq platform with memory encryption module.

Using such an interface, reading from or writing to memory can be performed by simply issuing the respective bus request. Even though in practice most of the memory requests have a well defined format (e.g., processor cache lines), there are in general no restrictions regarding alignment and request size. On the other hand, cryptographic primitives always have alignment and block size requirements which have to be matched. These diverging constraints make the transparent encryption of RAM quite challenging. Additionally, some ciphers and modes of operation additionally require metadata (e.g., counters, nonces, tags) to operate correctly. Processing this metadata at the correct time is essential to achieve good performance and complicates the issue of data alignment even further.

Finally, many optimizations and peculiarities of the used bus architecture itself have to be considered. A common performance tweak to speed up cache line fills is, for example, the use of wrapping burst. Such burst are problematic given that the requested data order does not match the order of the data in memory. Other peculiarities, which have to be considered for memory encryption, are for example write strobes, narrow transfers, and even complete interface width mismatches. In summary, every possible request which can be issued via the bus interface also has to be supported with transparent memory encryption in place.

5.1.2 Framework and Application to AXI4

Even though each individual challenge is minor, the overall resulting complexity is quite high. To cope with this complexity, a divide and conquer approach is used in our framework. The result is a comprehensive collection of modular building blocks which individually implement very limited functionality. However, arbitrary memory encryption units, with support for any cipher and encryption mode, can be built by arranging the individual modules in a pipeline structure.

Key to this flexibility are fully synchronized, unidirectional stream interfaces to interconnect the building blocks. On the majority of blocks, these stream interfaces receive and forward metadata (e.g., addresses, lengths, flags, ...) as well as a configurable amount of memory data (*i.e.*, depending on the external interface widths). The synchronization ensures that neither timing issues nor congestion cause data to be lost. Furthermore, registers and FIFOs can be placed

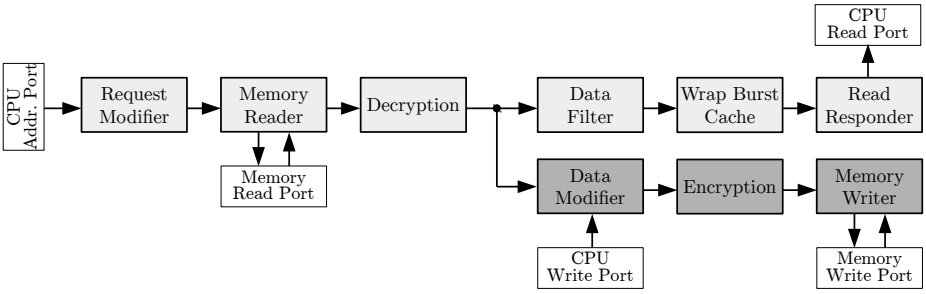


Figure 5.2: Simple AXI4 memory encryption pipeline which processes write requests using a RMW approach.

at arbitrary positions to cut combinatorial paths and to decouple the individual modules for better performance.

Transparent memory encryption for the AXI4 bus can, for example, be realized using a pipeline as shown in Figure 5.2. The depicted pipeline provides one slave and one master interfaces (see boxes with white background). The boxes which are shaded in light gray are used for reading from encrypted memory. Blocks which are shaded in dark gray are dedicated to writing to encrypted memory.

The slave interface (denoted as CPU) receives unencrypted requests that are serviced like without memory encryption. The master interface (denoted as Memory) on the other hand is used to actually store the encrypted data to the physical memory. The pipeline in Figure 5.2 is able to deal with all the previously discussed challenges and supports the use of arbitrary block-based cryptographic primitives or modes. Alignment and block size mismatches are addressed by artificially widening every request during request modification. For memory writes, this leads to the need for a Read-Modify-Write (RMW) approach when writing small data fragments. Interestingly, a RMW approach is required for AXI4 in any case to properly support write strobes. Therefore, all memory writes in this example pipeline are performed as RMW, which even permits to reuse the request modification, the memory reader, and the decryption for writes. The following types of building block categories are provided by the framework:

Bus Interface

Depending on the FPGA vendor, different types of bus interfaces are used to interact with memory. To outsource this dependency, interface converters are needed to establish the connection between the external bus interfaces and the framework-internal stream interface. The bus interface plays a major role in the framework on both the unencrypted slave and the encrypted master side. On the slave side, converters are involved in the translation of the initial request, the decoding of written data, the encoding of read data, and the error reporting. Similarly, on the master side, support for performing memory reads and writes (e.g., request issuing, data encoding, response processing, ...) is required.

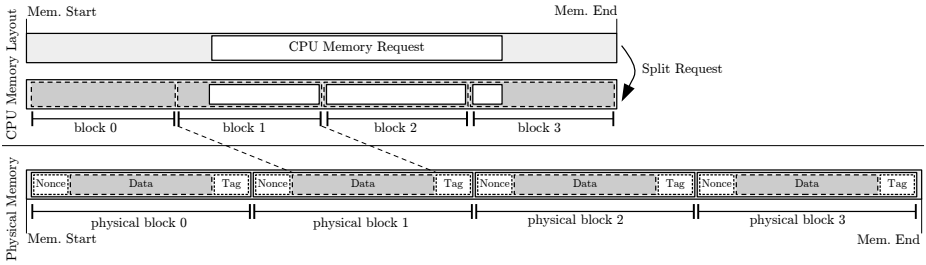


Figure 5.3: Request modification for a nonce based encryption and authentication scheme like Ascon [Dob+16]. CPU memory requests are split into chunks with additional alignment to incorporate metadata for the AE scheme.

Request Modification

Probably the most important part of the memory encryption pipeline is request modification. In this step, requests from the slave interface are translated to the requests on the master interface. This translation takes the memory alignment and block size requirements of the employed cipher mode into account and widens the requests accordingly. Additionally, also metadata is considered in cases where the encryption scheme is not length preserving and even additional requests can be injected into the pipeline when needed.

An example for a translation, suitable for a nonce-based authenticated encryption scheme, is shown in Figure 5.3. In the first step, the actually received CPU request is split based on the data block size of the cryptographic primitive. This splitting determines which logical blocks are affected by the request and have to be fetched. In the second step, taking into account the logical blocks and the amount of required metadata, it is then possible to determine the actual physical memory request. Note that during request modification only the size of the metadata is important. The actual semantic and positioning of the metadata within the physical block on the other hand is not.

En-/Decryption

Encryption and decryption blocks contain the actual ciphers which can typically be further decomposed into a cryptographic primitive and a suitable mode of operation. The cipher blocks only have to support encryption/decryption of memory requests with alignment and block size appropriate for the respective primitive, which greatly reduces implementation complexity. Furthermore, the actual layout of each block (e.g., what bytes are metadata) can be freely defined by the cipher blocks. However, to keep latency as low as possible it is advised to interleave the metadata with the ciphertext. By doing so, the metadata arrives at the cipher exactly in the moment it is actually needed. Figure 5.3 shows such an interleaving for a nonce based authenticated encryption scheme like Ascon. In this example, the nonce, used for initialization, is placed at the beginning and the tag, used for verification, is placed at the end of each block.

Data Stream Modification

Operating with the data which passes through the pipeline is another important part of the framework. Therefore, various building blocks which transform the data stream are provided. This includes support for injecting new data beats into the stream, for dropping existing data beats, for zero initializing whole requests, for filtering data based on the address, and for replacing individual bytes by taking into account address and write strobe information. Furthermore, the support for reordering individual data beats, which is needed to process wrapping bursts efficiently, can be assigned to this category of building blocks.

Miscellaneous

In addition to the main building blocks, also a comprehensive selection of supporting building blocks is provided by the framework. These blocks provide common functionality to the main blocks and are further handy for newly developed components. Examples for such supporting blocks are synchronization primitives for handshake signals, register stages with synchronization, and serialization as well as deserialization blocks for data rate conversions.

5.1.3 Optimizations

Performance optimization is in general a tough challenge given that detailed knowledge about the usage profile is required. However, some simple tweaks can also be performed by exploiting knowledge about the used hardware. For example, a CPU cache with AXI4 interface typically refills cache lines by using wrapping bursts to decrease latency. The framework's `WrapBurstCache` permits to implement such bursts efficiently (*i.e.*, single memory read) by reordering data beats within the pipeline.

Another important property of the framework regarding optimization is that each building block is highly configurable via VHDL generics. This not only is a necessity to support various ciphers, but also permits to perfectly adopt a memory encryption pipeline to the expected workload. Aligning the cipher block size (excl. metadata) with the expected request size (e.g., cache line size), for example, typically maximizes the performance.

Finally, requesting data from and writing data to the memory controller has been optimized. In particular, separate bus interface blocks are used for issuing requests as well as for sending and receiving data. This separation permits to place each operation at the earliest possible point into the pipeline which reduces latency. Furthermore, even multiple sequential requests (e.g., several reads) can be scheduled before any data block on the first request has been processed.

5.2 Authentication Trees

In this section we extend our pipeline in Figure 5.2 to implement authentication trees that provide replay protection.

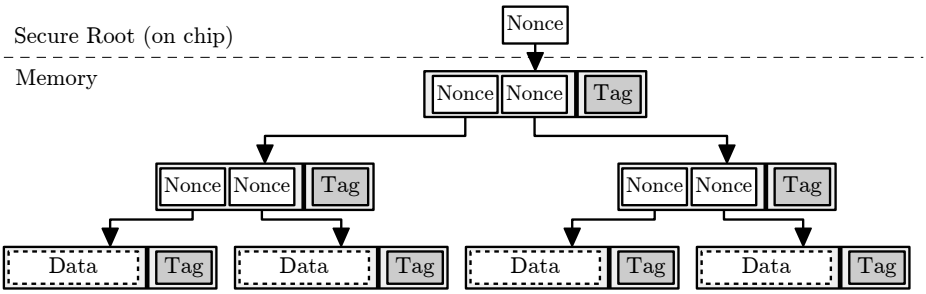


Figure 5.4: Binary TEC tree.

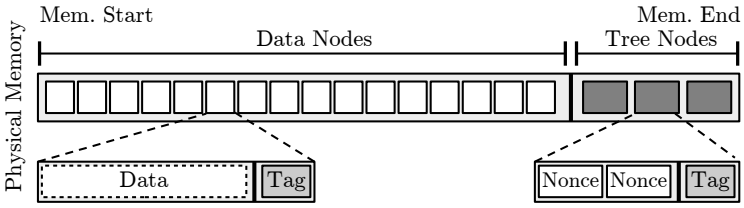


Figure 5.5: Physical memory layout of the nodes in a binary TEC tree.

5.2.1 Requirements

The pipeline depicted in Figure 5.2 facilitates the implementation of various variants of RAM encryption and authentication that provide RAM confidentiality and protection against active RAM spoofing and splicing attacks. However, many applications also require protection against replay attacks, where an active attacker replaces parts of the memory with valid ciphertexts (and tags) observed at a previous point in time. Such feature can be obtained from using an AE scheme like Ascon in an extended version of the pipeline in Figure 5.2. Namely, this pipeline must store all nonces securely on the FPGA such that they cannot be modified by attackers. In this way, nonces cannot be replayed and any malicious modification is detected. However, since the amount of available secure storage is typically limited, RAM authentication with replay protection usually relies on authentication trees. By storing every block in RAM within an authentication tree, only the tree root must be stored in a trusted environment. Since the tree root reflects the current state of the tree and is authentic, any tampering in RAM can be detected.

5.2.2 Functionality

The authentication tree used in this work is a variant of the TEC [Elb+07] tree as depicted in Figure 5.4. Hereby, the data in RAM is split into blocks and the blocks are authenticated and encrypted using the AE cipher Ascon. The nonces required for AE are recursively stored in a tree where all nodes are authenticated and encrypted as well. The root nonce is stored on the trusted FPGA chip. The

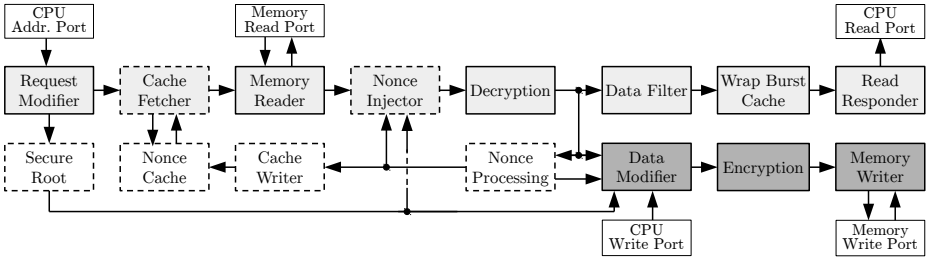


Figure 5.6: Memory encryption and authentication pipeline.

tree nodes themselves are stored after the data nodes and are located at the end of the RAM as visualized in Figure 5.5.

To support TEC trees, the pipeline in Figure 5.2 is extended as shown in Figure 5.6. Its basic data flow is thus identical. Light and dark shades denote modules required for read and write operations, respectively. Modules required to support authentication trees are depicted with dashed edges. For both read and write accesses, the pipeline traverses the tree in a single pass from the root to the respective data leaf node by decrypting (and updating) the nonce of the next lower level and verifying (and computing) the respective node's tag. Hereby, the `RequestModifier` injects all tree node requests additionally required to decrypt and authenticate the data requested by the CPU. Besides, authentication errors of all nodes accessed in the read part are accumulated to form the CPU response.

However, for uninitialized memory tag verification will typically fail. Hence, zero-valued nonces are used to tag uninitialized memory. This works as follows. The root nonce is initialized zero upon startup. On read accesses, verification errors are suppressed whenever a zero-valued nonce is encountered on the path from the root to the accessed leaf. On write accesses, the plaintexts of all nodes are initialized zero on the path from the root to the leaf from the point on when a zero nonce is encountered, and the actual write path is written as desired. In this way, uninitialized subtrees are automatically assigned a zero nonce.

5.2.3 Optimizations

Authentication trees as in Figure 5.4 have significant memory and performance overheads due to the processing of the additional tree nodes. An important parameter to reduce this overhead is the tree arity since it has direct influence on the tree height. The implementation thus allows the configuration of different tree arities to find the setup with the lowest overhead.

Furthermore, on the implementation side, the design in Figure 5.6 uses a `NonceCache` to optimize read performance. The `NonceCache` is directly mapped using the least significant RAM address bits and can be configured for different sizes. However, the `NonceCache` does not improve write performance. An optimization that improves both read and write accesses are multiple tree roots in the `SecureRoot`. While multiple roots increase the demand for secure on-chip

storage, they effectively reduce the tree height. Our implementation can be configured for an arbitrary number of on-chip roots.

5.3 Evaluation and Discussion

The proposed framework has been evaluated using a ZedBoard featuring a Xilinx Zynq XC7Z020 SoC and 512 MB DDR3 RAM. This SoC provides a dual core ARM Cortex-A9 Processing System (PS) and a Xilinx Artix-7 Programmable Logic (PL) which are connected using AXI interfaces. Figure 5.1 shows how the encryption pipelines from the previous sections are placed into the PL to perform transparent memory encryption. To ease comparison, all designs have been evaluated at 50 MHz FPGA frequency, provide 256 MB of protected memory to the ARM processors in the PS, and use the 32-bit GP0 interface to the CPU as well as the 64-bit HP0 interface to the memory. Note that operating the 32-bit interface at 50 MHz limits the maximum achievable bandwidth between processor and memory to 200 MB/s. As benchmarks, tinymembench[2] 0.3 and lmbench 3.0-a9 [MS96] have been used on top of the Xilinx Linux kernel[3] 4.4 (git tag 2016.2). Note that not only the benchmarks, but also the operating system itself has been executed within the transparently encrypted memory.

Cipher Modes and Configurations

The performance of the design in Figure 5.2 has been evaluated with the Ascon AE cipher (64-bit nonce and tag) as well as the block ciphers Prince and AES in ECB, CBC-ESSIV, and XTS mode. Additionally, the design in Figure 5.6, which provides full memory encryption and authentication using an 8-ary TEC tree with 1024 roots, has been measured using Ascon as the cryptographic primitive. Depending on the cipher, different implementation strategies (Prince = fully pipelined, Ascon + AES = round based) have been used.

However, we stress that these cryptographic primitives have only be chosen to evaluate how the performance of the encryption pipeline is affected by the cipher/mode. Namely, from the security point of view, we pronounce against using AES and Prince in ECB mode. Furthermore, using Prince in CBC and XTS mode is also not recommended given that the cipher does not offer related-key security. In particular, both CBC and XTS use Prince in a related-key setting where the whitening key is either eliminated or tweaked.

All configurations have been evaluated in their most promising parameterization. In most cases, this corresponds to aligning the block size of the cryptographic mode with the processor's last-level cache line size (32 bytes). The only exception is the Ascon-based TEC tree, which is configured with a data block size of 64 bytes.

[2] https://github.com/ssvb/tinymembench
[3] https://github.com/Xilinx/linux-xlnx.git

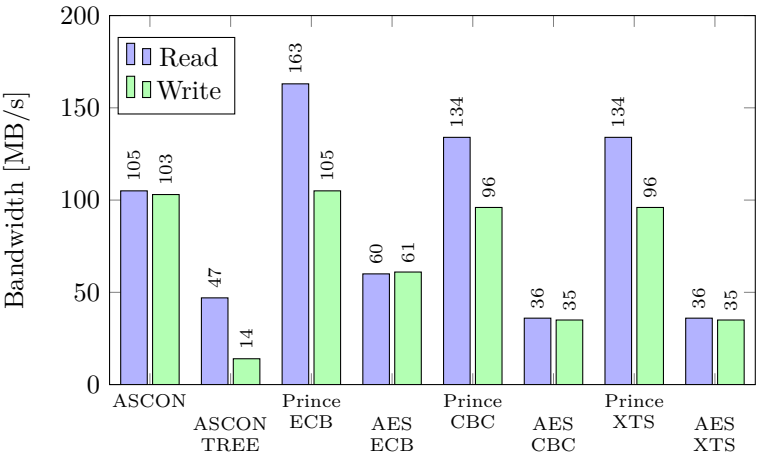


Figure 5.7: Memory bandwidth determined with tinymembench (`NEON read prefetched` (64 bytes `step`), `NEON fill`).

Bandwidth and Parameter selection

Figure 5.7 depicts the memory bandwidth of the various ciphers and modes of operation. The results for Prince clearly dominate the comparison, reaching between 82 % and 67 % of the maximum possible read bandwidth. This is due to the fully pipelined implementation which features only two cycles latency. The performance achieved with Prince ECB is in fact even comparable to using the pipeline without any cipher and for rate conversion only.

Regarding write bandwidth, all modes are capped at around 105 MB/s although the non-tree modes are supposed to have identical read and write performance. As it turns out, the reason for the observed write bandwidth limit is not the encryption pipeline itself, but the sequential way write requests are issued from the CPU cache in our setup. To achieve full write bandwidth, multiple parallel write requests would be needed.

Compared to Prince, the bandwidth results for the round-based AES implementation show the other side of the spectrum for the ECB, CBC, and XTS modes. Note that the use of multiple AES cores in parallel would be possible to increase the bandwidth of the ECB and XTS mode. CBC on the other hand would not benefit from additional cipher hardware at all given its algorithmic dependencies.

Ascon covers the middle ground regarding bandwidth, but additionally provides spoofing and splicing protection. Interestingly, also the replay-protected Ascon TEC-tree performs comparable to the AES modes regarding read bandwidth. The write bandwidth of the tree on the other hand is worse. However, even this number is comprehensible considering that, in the evaluated parameterization, writing between 1 and 32 bytes of memory actually requires to read, decrypt, encrypt, and write 360 bytes. Decreasing the size of the protected memory as

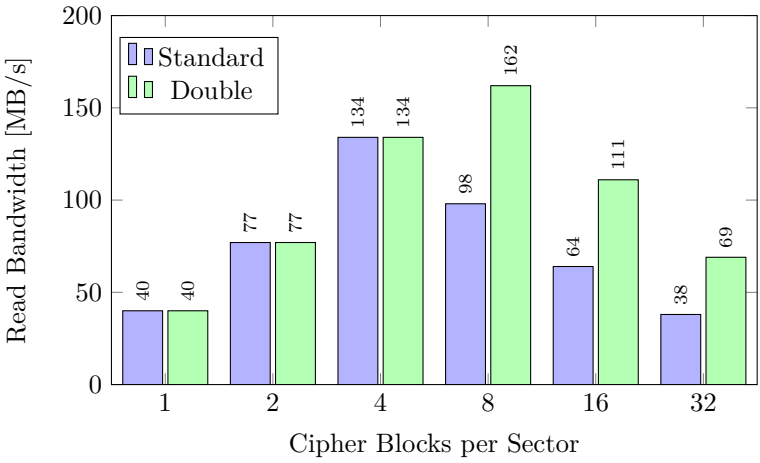


Figure 5.8: Memory read bandwidth determined with tinymembench of Prince CBC with different block sizes and cache controller configurations.

well as increasing the cache line size of the processor are ways to improve write performance for the tree.

At least for reads, the effect of bigger cache line sizes can be evaluated by enabling the double line fill feature of the cache controller. Due to the bigger requests, read bandwidth is typically increased. With double line fill enabled, Prince-ECB even reaches up to 94 % of the possible bandwidth (*i.e.*, 187 MB/s). However, the correct parameterization of the pipeline is important as shown in Figure 5.8. Unfortunately, the double line fill feature cannot replace a cache with doubled line size since only read requests are widened. Namely, write requests still have standard size and scale like standard read requests. Operating the encryption pipeline with double line fill enabled and parameters that increase read performance thus typically reduces write performance.

Latency

Compared to cache accesses, RAM accesses are slow and adding transparent RAM encryption further exacerbates this situation. However, as shown in Figure 5.9, the actual impact on the latency strongly depends on the used cryptographic primitive. The 265 ns from the Prince ECB implementation again can be considered as an estimate for the latency cost of our memory encryption framework. However, taking the real memory latency of the hardware ($\sim$ 80 ns) into account, the actual overhead of the FPGA design is around 185 ns. At our evaluation frequency (50 MHz), this corresponds to a minimum overhead of merely 9 cycles. The Ascon-based TEC tree on the other hand has the highest latency of all evaluated designs. Yet, it has to be put into perspective that the tree mode has to decrypt much more data (4 tree nodes + 1 data node).

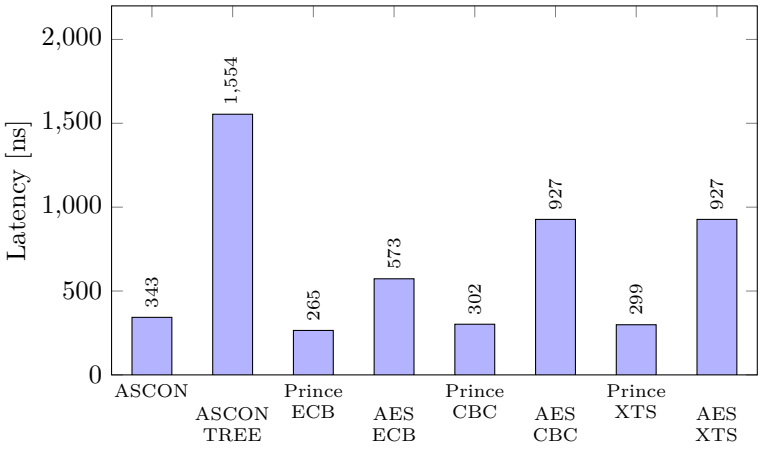


Figure 5.9: Memory read latency determined with lmbench (`lat_mem_rd 8M`).

Frequency

The maximum clock frequency is also an important property given that higher frequencies increase bandwidth and decrease latency. While we evaluated all designs at the same frequency of 50 MHz, all of them can be clocked higher. Namely, enabling optimizations in the Electronic Design Automation (EDA) tool (Vivado) already increases the maximum frequency of the designs to values between and 63 MHz and 75 MHz (up to +50 %). Nevertheless, even then, the critical path is mainly determined by routing delay. This is due to the fact that the used Artix 7 FPGA (speed grade 1) is an entry model. The next stronger Zynq XC7Z030 speed grade 1 model with Kintex 7 FPGA, for example, can already operate the slowest design (Ascon tree) at 93 MHz (= +86 %). Using an XC7Z030 with speed grade 3 even yields a maximum frequency of 126 MHz (= +152 %) for the tree design.

FPGA Utilization

Figure 5.10 visualizes the consumed hardware resources on our target SoC FPGA in terms of flip flops and lookup tables. The XC7Z020 features a total of 53,200 lookup tables of which between 8.9 % (Prince ECB) and 19.2 % (Ascon tree) are occupied by our designs. Similarly, between 2.3 % and 4.4 % of the 106,400 available flip flops are used. The use of the 36 kbit block RAMs is also negligible (4.5 blocks of the available 140) given that they are solely used in the Ascon tree design for the tree roots and as simple nonce cache. Considering that the used FPGA is more or less an entry-level device, more than enough resources remain available for other use cases.

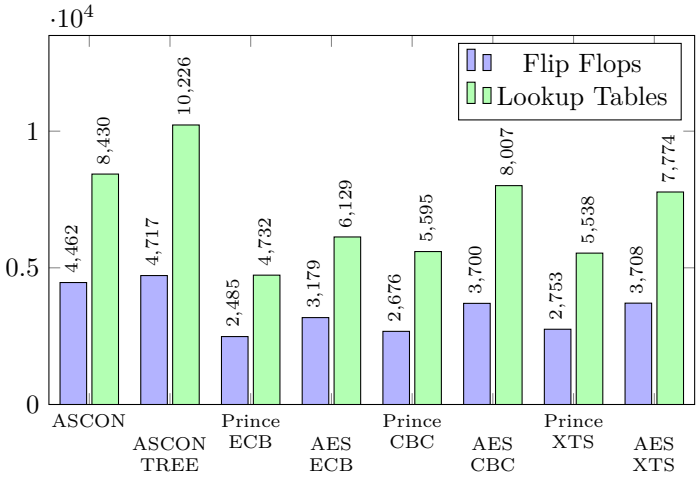


Figure 5.10: FPGA Utilization of the used Xilinx Zynq XC7Z020 SoC.

5.4 Conclusion

In this chapter, we presented an open-source framework of modular building blocks to implement RAM encryption solutions. A simple, fully synchronized stream interface is used to connect the individual blocks and permits to easily replace specific components as needed. As the result, realizing arbitrary encryption pipelines is as simple as connecting the needed blocks according to the data flow graph of the design. The evaluation, using various cipher primitives and modes, shows that our framework is very flexible and can easily support differing block sizes and memory alignment constraints. An example that showcases this flexibility is that also novel cryptographic modes, like the side-channel protected MEAS [UWM19] scheme, can be implemented successfully with our framework. The fact that the winning team of MITRE's eCTF security competition[4] in 2019 employed our open-source framework in their design further underlines its usefulness. Finally, the results demonstrate that retrofitting memory encryption to Zynq SoCs is feasible and that Ascon (with and without tree) is a decent choice for memory encryption, when authenticity is desired in addition to confidentiality.

[4] https://mitrecyberacademy.org/competitions/embedded/

6

ScatterCache: Thwarting Cache Attacks via Cache Set Randomization

Caches are core components of today's computing architectures. They bridge the performance gap between Central Processing Unit (CPU) cores and a computer's main memory. However, in the past two decades, caches have turned out to be the origin of a wide range of security threats [Ber05; Bul+18; GSM15; Koc+19; Koc96; Lip+18b; Liu+15; OST06; YF14]. In particular, the intrinsic timing behavior of caches that speeds up computing systems allows for cache side-channel attacks (cache attacks), which are able to recover secret information.

Historically, research on cache attacks primarily focused on cryptographic algorithms [Ber05; Liu+15; OST06; YF14]. More recently, however, cache attacks like Prime+Probe [Liu+15; Mau+17; OST06; Per05; Sch+17] and Flush+Reload [GSM15; YF14] have also been used to attack address-space-layout randomization [Gra+17; Gru+16a; JLK16], keystroke processing and inter-keystroke timing [Gru+16b; GSM15; Sch+18c], and general-purpose computations [Zha+14]. For shared caches on modern multi-core CPUs, Prime+Probe and Flush+Reload even work across cores executing code from different security domains, e.g., processes or virtual machines.

The most simple cache attacks, however, are covert channels [Mau+15a; Mau+17; WXW12]. In contrast to a regular side-channel attack, in a covert channel, the "victim" is colluding and actively trying to transmit data to the attacker, e.g., running in a different security domain. For instance, Meltdown [Lip+18b], Spectre [Koc+19], and Foreshadow [Bul+18] use cache covert channels to transfer secrets from the transient execution domain to an attacker. These recent examples highlight the importance of finding practical approaches to thwart cache attacks.

To cope with cache attacks, there has been much research on ways to identify information leaks in a software's memory access pattern, such as static code [DK17; Doy+13; KMO12; MWK17] and dynamic program analysis [Ira+17; Wei+18; Xia+17; ZHS16]. However, mitigating these leaks both generically and efficiently is difficult. While there are techniques to design software without address-based information leaks, such as unifying control flow [Cop+09] and bitsliced implementations of cryptography [Kön08; KS09; RSD06], their general application to arbitrary software remains difficult. Hence, protecting against cache attacks puts a significant burden on software developers aiming to protect secrets in the view of microarchitectural details that vary a lot across different Instruction-Set Architecture (ISA) implementations.

A different direction to counteract cache attacks is to design more resilient cache architectures. Typically, these architectures modify the cache organization in order to minimize interference between different processes, either by breaking the trivial link between memory address and cache index [Gal+19; Qur18; Tri+18; WL07; WL08] or by providing exclusive access to cache partitions for critical code [Pag05; Raj+09; WL07]. While cache partitioning completely prevents cache interference, its rather static allocation suffers from scalability and performance issues. On the other hand, randomized cache (re-)placement [WL07; WL08] makes mappings of memory addresses to cache indices random and unpredictable. Yet, managing these cache mappings in lookup tables inheres extensive changes to the cache architecture and cost. Finally, the introduction of a keyed function [Qur18; Tri+18] to pseudorandomly map the accessed memory location to the cache-set index can counteract Prime+Probe attacks. However, these solutions either suffer from a low number of cache sets, weakly chosen functions, or cache interference for shared memory and thus require to change the key frequently at the cost of performance.

Hence, there is a strong need for a practical and effective solution to thwart both cache attacks and cache covert channels. In particular, this solution should (1) make cache attacks sufficiently hard, (2) require as little software support as possible, (3) embed flexibly into existing cache architectures, (4) be efficiently implementable in hardware, and (5) retain or even enhance cache performance.

Contribution. In this chapter, we present ScatterCache, which achieves all these goals. ScatterCache is a novel and highly flexible cache design that prevents cache attacks such as Evict+Reload and Prime+Probe and severely limits cache covert channel capacities by increasing the number of cache sets beyond the number of physically available addresses with competitive performance and implementation cost. Hereby, ScatterCache closes the gap between previous secure cache designs and today's cache architectures by introducing a minimal set of cache modifications to provide strong security guarantees.

Most prominently, ScatterCache eliminates the fixed cache-set congruences that are the cornerstone of Prime+Probe attacks. For this purpose, ScatterCache builds upon two ideas. First, ScatterCache uses a keyed mapping function to translate memory addresses and the active security domain, e.g., process, to cache set indices. Second, similar to skewed associative caches [Sez93],

the mapping function in ScatterCache computes a different index for each cache way. As a result, the number of different cache sets increases exponentially with the number of ways. While ScatterCache makes finding fully identical cache sets statistically impossible on state-of-the-art architectures, the complexity for exploiting inevitable partial cache-set collisions also rises heavily. The reason is in part that the mapping of memory addresses to cache sets in ScatterCache is different for each security domain. Attacks on ScatterCache are by construction probabilistic and require that targeted memory accesses can be observed many times for both, the actual attack as well as the needed profiling.

Additionally, ScatterCache effectively prevents Flush+Reload-based cache attacks, e.g., on shared libraries, as well. The inclusion of security domains in ScatterCache and its mapping function preserves shared memory in Random-Access Memory (RAM), but prevents any cache lines to be shared across security boundaries. Yet, ScatterCache supports shared memory for inter-process communication via dedicated separate security domains. To achieve highest flexibility, managing the security domains of ScatterCache is done by software, e.g., the operating system. However, ScatterCache is fully backwards compatible and already increases the effort of cache attacks even without any software support. Nevertheless, the runtime performance of software on ScatterCache is highly competitive and, on certain workloads, even outperforms cache designs implemented in commodity CPUs.

ScatterCache constitutes a comparably simple extension to cache and processor architectures with minimal hardware cost: ScatterCache essentially only adds additional index derivation logic, *i.e.*, a lightweight cryptographic primitive, and an index decoder for each scattered cache way. Moreover, to enable efficient lookups and writebacks, ScatterCache stores the index bits from the physical address in addition to the tag bits, which adds $< 5\,\%$ storage overhead per cache line. Finally, ScatterCache consumes one bit per page-table entry ($\approx 1.5\,\%$ storage overhead per page-table entry) for the kernel to communicate with the user space.

Outline. The remainder of this chapter is structured as follows: In Section 6.1, we provide background information on caches and cache attacks. In Section 6.2, we describe the design and concept of ScatterCache. In Section 6.3, we analyze the security of ScatterCache against cache attacks. In Section 6.4, we provide a performance evaluation. We conclude in Section 6.5.

6.1 Background

In this section, we provide background on caches, cache side-channel attacks, and resilient cache architectures.

6.1.1 Caches

Modern computers have a memory hierarchy consisting of many layers, each following the principle of locality, storing data that is expected to be used in the

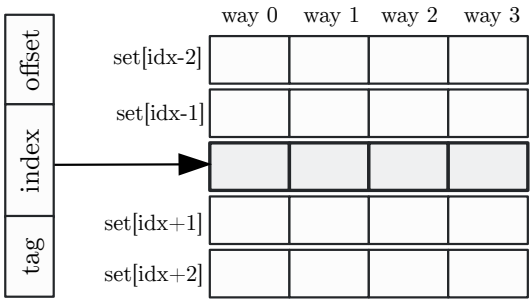


Figure 6.1: Indexing cache sets in a 4-way set-associative cache.

future, e.g., based on what has been accessed in the past. Modern processors have a hierarchy of caches that keep instructions and data likely to be used in the future near the core to avoid the latency of accesses to the comparably slow main memory—usually Dynamic Random-Access Memory (DRAM). This hierarchy typically consists of 2 to 4 layers, where the lowest layer is the smallest and fastest, typically only a few kilobytes. The last-level cache is the largest cache, typically in the range of several megabytes. On most processors, the last-level cache is shared among all cores. The last-level cache is often inclusive, *i.e.*, any cache line in a lower level cache must also be present in the last-level cache.

Caches are typically organized into *cache sets* that are composed of multiple *cache lines* or *cache ways*. The cache set is determined by computing the cache index from address bits. Figure 6.1 illustrates the indexing of a 4-way set-associative cache. As the cache is small and the memory large, many memory locations map to the same cache set (*i.e.*, the addresses are *congruent*). The replacement policy (e.g., pseudo-LRU, random) decides which way is replaced by a newly requested cache line. Any process can observe whether data is cached or not by observing the memory access latency which is the basis for cache side-channel attacks.

6.1.2 Cache Side-Channel Attacks

Cache side-channel attacks have been studied for over the past two decades, initially with a focus on cryptographic algorithms [Ber05; Koc96; OST06; Pag02; Per05; Tsu+03]. Today, a set of powerful attack techniques enable attacks in realistic cross-core scenarios. Based on the access latency, an attacker can deduce whether or not a cache line is in the cache, leaking two opposite kinds of information. (1) By continuously removing (*i.e.*, evicting or flushing) a cache line from the cache and measuring the access latency, an attacker can determine whether this cache line has been accessed by another process. (2) By continuously filling a part of the cache with attacker-accessible data, the attacker can measure the contention of the corresponding part, by checking whether the attacker-accessible data remained in the cache. Contention-based attacks work on different layers:

The Entire Cache or Cache Slices. An attacker can measure contention of the entire cache or a cache slice. Maurice et al. [Mau+15a] proposed a covert channel where the sender evicts the entire cache to leak information across cores and the victim observes the cache contention. A similar attack could be mounted on a cache slice if the cache slice function is known [Mau+15b]. The granularity is extremely coarse, but with statistical attacks can leak meaningful information [Sch+18d].

Cache Sets. An attacker can also measure the contention of a cache set. For this, additional knowledge may be required, such as the mapping from virtual addresses to physical addresses, as well as the functions mapping physical addresses to cache slices and cache sets. The attacker continuously fills a cache set with a set of congruent memory locations. Filling a cache set is also called cache-set eviction, as it evicts any previously contained cache lines. Only if some other process accessed a congruent memory location, memory locations are evicted from a cache set. The attacker can measure this for instance by measuring runtime variations in a so-called Evict+Time attack [OST06]. The Evict+Time technique has mostly been applied in attacks on cryptographic implementations [HWH13; Lip+16; OST06; SP13]. Instead of the runtime, the attacker can also directly check how many of the memory locations are still cached. This attack is called Prime+Probe [OST06]. Many Prime+Probe attacks on private L1 caches have been demonstrated [ABG10; BH09; OST06; Per05; Zha+12]. More recently, Prime+Probe attacks on last-level caches have also been demonstrated in various generic use cases [AES15; Liu+15; Mau+17; Ore+15; Ris+09; Zha+11].

Cache Lines. At a cache line granularity, the attacker can measure whether a memory location is cached or not. As already indicated above, here the logic is inverted. Now the attacker continuously evicts (or flushes) a cache line from the cache. Later on, the attacker can measure the latency and deduce whether another process has loaded the cache line into the cache. This technique is called Flush+Reload [GBK11; YF14]. Flush+Reload has been studied in a long list of different attacks [AES15; Ape+14; Ape+15; GSM15; IES16; Inc+16; Lip+16; YF14; Zha+14; ZXZ16]. Variations of Flush+Reload are Flush+Flush [Gru+16b] and Evict+Reload [GSM15; Lip+16].

Cache Covert Channels

Cache covert channels are one of the simplest forms of cache attacks. Instead of an attacker process attacking a victim process, both processes collude to covertly communicate using the cache as transmission channel. Thus, in this scenario, the colluding processes are referred to as sender and receiver, as the communication is mostly unidirectional. A cache covert channel allows bypassing all architectural restrictions regarding data exchange between processes.

Various cache attacks, such as Prime+Probe [Liu+15; Mau+17; WXW15; Xu+11] and Flush+Reload [Gru+16b], can be used to build cache covert channels. They achieve transmission rates of up to 496 kB/s [Gru+16b]. Besides native attacks, covert channels have also been shown to work within virtualized

environments, across virtual machines [Liu+15; Mau+17; Xu+11]. Even in these restricted environments, cache-based covert channels achieve transmission rates of up to 45 kB/s [Mau+17].

6.1.3 Resilient Cache Architectures

The threat of cache-based attacks sparked several novel cache architectures designed to be resilient against these attacks. While fixed cache partitions [Pag05] lack flexibility, randomized cache allocation appears to be more promising. The following briefly discusses previous designs for a randomized cache.

RPCache [WL07] and NewCache [WL08] completely disrupt the meaningful observability of interference by performing random (re-)placement of lines in the cache. However, managing the cache mappings efficiently either requires full associativity or content addressable memory. While optimized addressing logic can lead to efficient implementations, these designs differ significantly from conventional architectures.

Time-Secure Caches [Tri+18] is based on standard set-associative caches that are indexed with a keyed function that takes cache line address and Process IDentifier (PID) as an input. While this design destroys the obvious cache congruences between processes to minimize cache interference, a comparably weak indexing function is used. Eventually, re-keying needs to be done quite frequently, which amounts to flushing the cache and thus reduces practical performance. ScatterCache can be seen as a generalization of this approach with higher entropy in the indexing of cache lines.

CEASER [Qur18] as well uses standard set-associative caches with keyed indexing, which, however, does not include the PID. Hence, inter-process cache interference is predictable based on in-process cache collisions. As a result, CEASER strongly relies on continuous re-keying of its index derivation to limit the time available for conducting an attack. However, as found by the author [Qur19], rekeying alone is not sufficient to make CEASER secure while maintaining its efficiency. Therefore, a new skewed variant of CEASER named **CEASER-S [Qur19]** has been proposed. Interestingly, the independently and concurrently developed CEASER-S design (with n_{ways} partitions) is basically identical to our ScatterCache design. Still, unlike in our design which relies on established primitives, CEASER(-S) uses a custom low latency-block cipher for index derivation. Unfortunately, this cipher fails to deliver the expected properties [Bod+20] and needs to be replaced.

HybCache [DFS19], another concurrently developed cache architecture, distinguishes between isolated and non-isolated execution domains. For non-isolated workloads, contemporary caching behavior is maintained. Isolated workloads, on the other hand, are restricted to a fully-associative subcache—comprising n isolated cache ways—with random replacement. This approach apparently limits interference to the cache occupancy channel. However, requiring the fully-associative lookup, especially in large Last-Level Caches (LLCs), may still be too expensive in terms of area and/or power consumption.

6.2 ScatterCache

As Section 6.1 showed, caches are a serious security concern in contemporary computing systems. In this section, we hence present ScatterCache—a novel cache architecture that counteracts cache-based side-channel attacks by skewed pseudorandom cache indexing. After discussing the main idea behind ScatterCache, we discuss its building blocks and system integration in more detail. ScatterCache's security implications are, subsequently, analyzed in Section 6.3.

6.2.1 Targeted Properties

Even though contemporary transparent cache architectures are certainly flawed from the security point of view, they still feature desirable properties. In particular, for regular computations, basically no software support is required for cache maintenance. Also, even in the case of multitasking and -processing, no dedicated cache resource allocation and scheduling is needed. Finally, by selecting the cache size and the number of associative ways, chip vendors can trade hardware complexity and costs against performance as desired.

ScatterCache's design strives to preserve these features while adding the following three security properties:

- Between software defined security domains (e.g., different processes or users on the same machine, different virtual machines, ...), even for exactly the same physical addresses, cache lines should only be shared if cross-context coherency is required (*i.e.*, writable shared memory).
- Finding and exploiting addresses that are congruent in the cache should be as hard as possible (*i.e.*, we want to "break" the direct link between the accessed physical address and the resulting cache set index for adversaries).
- Controlling and measuring complete cache sets should be hard in order to prevent eviction-based attacks.

Finally, to ease the adoption and to utilize the vast knowledge on building efficient caches, the ScatterCache hardware should be as similar to current cache architectures as possible.

6.2.2 Idea

Two main ideas influenced the design of ScatterCache to reach the desired security properties. First, addresses should be translated to cache sets using a keyed, security-domain aware mapping. Second, which exact n_{ways} cache lines form a cache set in a n_{ways}-way associative cache should not be fixed, but depend on the currently used key and security domain too. ScatterCache combines both mappings in a single operation that associates each address, depending on the key and security domain, with a set of up to n_{ways} cache lines. In other words, in a generic ScatterCache, any possible combination of up to n_{ways} cache lines can form a cache set.

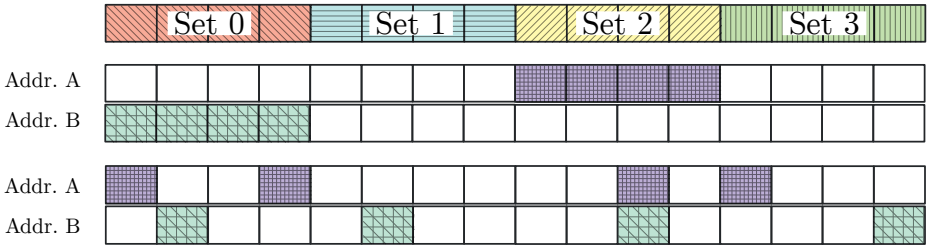


Figure 6.2: Flattened visualization of mapping addresses to cache sets in a 4-way set-associative cache with 16 cache lines. *Top:* Standard cache where index bits select the cache set. *Middle:* Pseudorandom mapping from addresses to cache sets. The mapping from cache lines to sets is still static. *Bottom:* Pseudorandom mapping from addresses to a set of cache lines that dynamically form the cache set in ScatterCache.

Figure 6.2 visualizes the idea and shows how it differs from related work. Traditional caches as well as alternative designs which pseudorandomly map addresses to cache sets statically allocate cache lines to cache sets. Hence, as soon as a cache set is selected based on (possibly encrypted) index bits, always the same n_{ways} cache lines are used. This means that all addresses mapping to the same cache set are congruent and enables Prime+Probe-style attacks.

In ScatterCache, on the other hand, the cache set for a particular access is a pseudorandom selection of arbitrary n_{ways} cache lines from all available lines. As a result, there is a much higher number of different cache sets and finding addresses with identical cache sets becomes highly unlikely. Instead, as shown at the bottom of Figure 6.2, at best, partially overlapping cache sets can be found (cf. Section 6.3.3), which makes exploitation tremendously hard in practice.

A straightforward concept for ScatterCache is shown in Figure 6.3. Here, the Index Derivation Function (IDF) combines the mapping operations in a single cryptographic primitive. In a set-associative ScatterCache with set size n_{ways}, for each input address, the IDF outputs n_{ways} indices to form the cache set for the respective access. How exactly the mapping is performed in ScatterCache is solely determined by the used key, the Security Domain IDentifier (SDID), and the IDF. Note that, as will be discussed in Section 6.2.3, hash-based as well as permutation-based IDFs can be used in this context.

Theoretically, a key alone is sufficient to implement the overall idea. However, separating concerns via the SDID leads to a more robust and harder-to-misuse concept. The key is managed entirely in hardware, is typically longer, and gets switched less often than the SDID. On the other hand, the SDID is managed solely by the software and, depending on the implemented policy, has to be updated quite frequently. Importantly, as we show in Section 6.3, ScatterCache alone already provides significantly improved security in Prime+Probe-style attack settings even without software support (*i.e.*, SDID is not used).

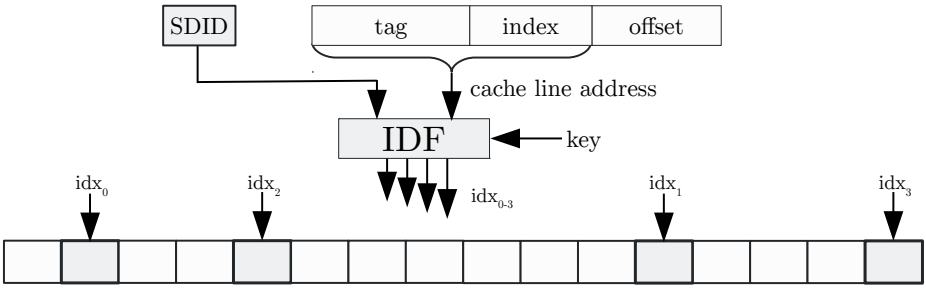


Figure 6.3: Idea: For an n_{ways} associative cache, n_{ways} indices into the cache memory are derived using a cryptographic IDF. This IDF effectively randomizes the mapping from addresses to cache sets as well as the composition of the cache set itself.

6.2.3 ScatterCache Design

In the actual design we propose for ScatterCache, the indices (*i.e.*, IDF output) do not address into one huge joint cache array. Instead, as shown in Figure 6.4, each index addresses a separate memory, *i.e.*, an independent cache way.

On the one hand, this change is counter-intuitive as it decreases the number of possible cache sets from $\binom{n_{ways} \cdot 2^{b_{indices}} + n_{ways} - 1}{n_{ways}}$ to $2^{b_{indices} \cdot n_{ways}}$. However, this reduction in possibilities is acceptable. For cache configurations with up to 4 cache ways, the gap between both approaches is only a few bits. For higher associativity, the exponential growth ensures that sufficiently many cache sets exist.

On the other hand, the advantages gained from switching to this design far outweigh the costs. Namely, for the original idea, no restrictions on the generated indices exist. Therefore, a massive n_{ways}-fold multi-port memory would be required to be able to lookup a n_{ways}-way cache-set in parallel. The design shown in Figure 6.4 does not suffer from this problem and permits to instantiate ScatterCache using n_{ways} instances of simpler/smaller memory. Furthermore, this design guarantees that even in case the single index outputs of the IDF collide, the generated cache always consists of exactly n_{ways} many cache lines. This effectively precludes the introduction of systematic biases for potentially "weak" address-key-SDID combinations that map to fewer than n_{ways} cache lines.

In terms of cache-replacement policy, ScatterCache uses simple random replacement to ensure that no systematic bias is introduced when writing to the cache and to simplify the security analysis. Furthermore, and as we will show in Section 6.4, the performance of ScatterCache with random replacement is competitive to regular set associative caches with the same replacement policy. Therefore, evaluation of alternative replacement policies has been postponed. Independent of the replacement policy, it has to be noted that, for some IDFs, additional tag bits have to be stored in ScatterCache. In particular, in case of a non invertible IDF, the original index bits need to be stored to facilitate write back of dirty cache lines and to ensure correct cache lookups. However,

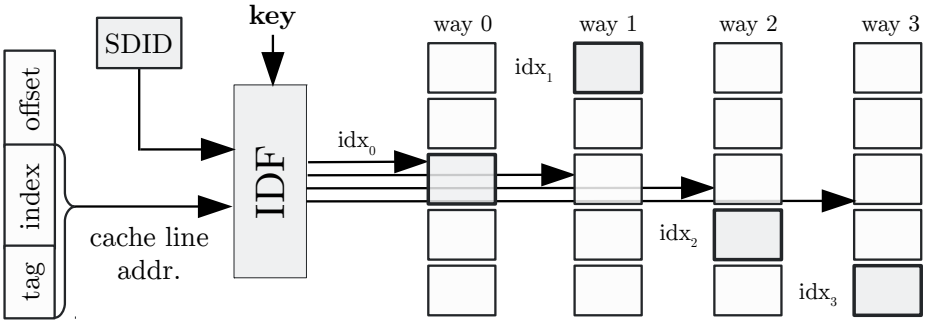


Figure 6.4: 4-way set-associative ScatterCache where each index addresses exclusively one cache way.

compared to the amount of data that is already stored for each cache line, the overhead of adding these few bits should not be problematic ($< 5\,\%$ overhead).

In summary, the overall hardware design of ScatterCache closely resembles a traditional set-associative architecture. The only differences to contemporary fixed-set designs is the more complex IDF and the amount of required logic which permits to address each way individually. However, both changes are well understood. As we detail in the following section, lightweight (*i.e.*, low area and latency) cryptographic primitives are suitable building blocks for the IDF. Similarly, duplication of addressing logic is already common practice in current processors. Modern Intel architectures, for example, already partition their LLC into multiple smaller cache slices with individual addressing logic.

Suitable Index Derivation Functions

Choosing a suitable IDF is essential for both security and performance. In terms of security, the IDF has to be an unpredictable (but still deterministic) mapping from physical addresses to indices. Following Kerckhoffs's principle, even for attackers which know every detail except the key, three properties are expected from the IDF: (1) Given perfect control over the public inputs of the function (*i.e.*, the physical address and SDID) constructing colliding outputs (*i.e.*, the indices) should be hard. (2) Given colliding outputs, determining the inputs or constructing further collisions should be hard. (3) Recovering the key should be infeasible given input and output for the function.

Existing Building Blocks: Cryptographic primitives like (tweakable) block ciphers, Message Authentication Codes (MACs), and hash functions are designed to provide these kind of security properties (e.g., indistinguishability of encryptions, existential unforgeability, pre-image and collision resistance). Furthermore, design and implementation of cryptographic primitives with tight performance constraints is already a well-established field of research which we want to take advantage of. For example, with Prince [Bor+12a], a low-latency block cipher, and QARMA [Ava17], a family of low-latency tweakable block ciphers, exist and can be used as building blocks for the IDF. Such tweakable block ciphers are a

flexible extension to ordinary block ciphers, which, in addition to a secret key, also use a public, application-specific tweak to en-/decrypt messages. Similarly, sponge-based MAC, hash and cipher designs are a suitable basis for IDFs. These sponge modes of operation are built entirely upon permutations, e.g., Keccak-p, which can often be implemented with low latency [Arr+18; Ber+12a]. Using such cryptographic primitives, we define the following two variants of building IDFs:

Hashing Variant (SCv1): The idea of SCv1 is to combine all IDF inputs using a single cryptographic primitive with pseudo random output. MACs (e.g., hash-based) are examples for such functions and permit to determine the output indices by simply selecting the appropriate number of disjunct bits from the calculated tag. However, also other cryptographic primitives can be used for instantiating this IDF variant.

It is, for example possible to slice the indices from the ciphertext of a regular block cipher encryption which uses the concatenation of cache line address and the SDID as the plaintext. Similarly, tweakable block ciphers allow to use the SDID as a tweak instead of connecting it to the plaintext. Interestingly, finding cryptographic primitives for SCv1 IDFs is comparably simple given that the block sizes do not have to match perfectly and the output can be truncated as needed.

However, there are also disadvantages when selecting the indices pseudo randomly, like in the case of SCv1. In particular, when many accesses with high spatial locality are performed, index collisions get more likely. This is due to the fact that collisions in SCv1 output have birthday-bound complexity. Subsequently, performance can degrade when executing many different accesses with high spatial locality. Fortunately, this effect weakens with increasing way numbers, *i.e.*, an increase in associativity decreases the probability that all index outputs of the IDF collide.

In summary, SCv1 translates the address without distinguishing between index and tag bits. Given a fixed key and SDID, the indices are simply pseudo random numbers that are derived using a single cryptographic primitive.

Permutation Variant (SCv2): The idea behind the permutation variant of the IDF is to distinguish the index from the tag bits in the cache line address during calculation of the indices. Specifically, instead of generating pseudo random indices from the cache line address, tag dependent permutations of the input index are calculated.

The reason for preferring a permutation over pseudo random index generation is to counteract the effect of birthday-bound index collisions, as present in SCv1. Using a tag dependent permutation of the input index mitigates this problem by design since permutations are bijections that, for a specific tag, cannot yield colliding mappings.

Like in the hashing variant, a tweakable block cipher can be used to compute the permutation. Here, the concatenation of the tag bits, the SDID and the way index constitutes the tweak while the address' index bits are used as the plaintext. The resulting ciphertext corresponds to the output index for the respective way. Note that the block size of the cipher has to be equal to the size of the index. Additionally, in order to generate all indices in parallel, one instance of the

tweakable block cipher is needed per cache way. However, as the block size is comparably small, each cipher instance is also smaller than an implementation of the hashing IDF (SCv1).

Independently of the selected IDF variant, we leave the decision on the actually used primitive to the discretion of the hardware designers that implement ScatterCache. They are the only ones who can make a profound decision given that they know the exact instantiation parameters (e.g., SDID/key/index/tag bit widths, number of cache ways) as well as the allocatable area, performance, and power budget in their respective product. However, we are certain that, even with the already existing and well-studied cryptographic primitives, ScatterCache implementations are feasible for common computing platforms, ranging from Internet-of-Things (IoT) devices to desktop computers and servers.

Note further that we expect that, due to the limited observability of the IDF output, weakened (*i.e.*, round reduced) variants of general-purpose primitives are sufficient to achieve the desired security level. This is because adversaries can only learn very little information about the function output by observing cache collisions (*i.e.*, no actual values). Subsequently, many more traces have to be observed for mounting an attack. Cryptographers can take advantage of this increase in data complexity to either design fully custom primitives [Qur18] or to decrease the overhead of existing designs.

Key Management and Re-Keying

The key in our ScatterCache design plays a central role in the security of the entire approach. Even when the SDIDs are known, it prevents attackers from systematically constructing eviction sets for specific physical addresses and thwarts the calculation of addresses from collision information. Keeping the key confidential is therefore of highest importance.

We ensure this confidentiality in our design by mandating that the key of is fully managed by hardware. There must not be any way to configure or retrieve this key in software. This approach prevents various kinds of software-based attacks and is only possible due to the separation of key and SDID.

The hardware for key management is comparably simple as well. Each time the system is powered up, a new random key is generated and used by the IDF. The simplicity of changing the key during operation strongly depends on the configuration of the cache. For example, in a write-through cache, changing the key is possible at any time without causing data inconsistency. In such a scenario, a timer or performance-counter-based re-keying scheme is easily implementable. Note, however, that the interval between key changes should not be too small as each key change corresponds to a full cache flush.

On the other hand, in a cache with write-back policy, the key has to be kept constant as long as dirty cache lines reside in the cache. Therefore, before the key can be changed in this scenario without data loss, all modified cache lines have to be written back to memory first. The x86 ISA, for example, features the `WBINVD` instruction that can be used for that purpose.

If desired, also more complex rekeying schemes, like way-wise or cache-wide

dynamic remapping [Qur18], can be implemented. However, it is unclear if adding the additional hardware complexity is worthwhile. Even without changing the key, mounting cache attacks against ScatterCache is much harder than on traditional caches (see Section 6.3). Subsequently, performing an occasional cache flush to update the key can be the better choice.

Integration into Existing Cache Architectures

ScatterCache is a generic approach for building processor caches that are hard to exploit in cache-based side channel attacks. When hardening a system against cache attacks, independent of ScatterCache, we recommend to restrict flush instructions to privileged software. These instruction are only rarely used in benign user space code and restricting them prevents the applicability of the whole class of flush-based attacks from user space. Fortunately, recent ARM architectures already support this restriction.

Next, ScatterCaches can be deployed into the system to protect against eviction based attacks. While not inherently limited to, ScatterCaches are most likely to be deployed as LLCs in modern processor architectures. Due to their large size and the fact that they are typically shared across multiple processor cores, LLCs are simply the most prominent cache attack target and require the most protection. Compared to that, lower cache levels that typically are only accessible by a single processor core, hold far less data and are much harder to attack on current architectures. Still, usage of (unkeyed) skewed [Sez93] lower level caches is an interesting option that has to be considered in this context.

Another promising aspect of employing a ScatterCache as LLC is that this permits to hide large parts of the IDF latency. For example, using a fully unrolled and pipelined IDF implementation, calculation of the required ScatterCache indices can already be started, or even performed entirely, in parallel to the lower level cache lookups. While unneeded results can easily be discarded, this ensures that the required indices for the LLC lookup are available as soon as possible.

Low latency primitives like QARMA, which is also used in recent ARM processors for pointer authentication, are promising building blocks in this regard. The minimal latency Avanzi [Ava17] reported for one of the QARMA-64 variants is only 2.2 ns. Considering that this number is even lower than the time it takes to check the L1 and L2 caches on recent processors (e.g., 3 ns on a 4 GHz Intel Kabylake [7cpb], 9 ns on an ARM Cortex-A57 in an AMD Opteron A1170 [7cpa]), implementing IDFs without notable latency seems feasible.

6.2.4 Processor Interaction and Software

Even without dedicated software support, ScatterCache increases the complexity of cache-based attacks. However, to make full use of ScatterCache, software assistance and some processor extensions are required.

Security Domains. The ScatterCache hardware permits to isolate different security domains from each other via the SDID input to the IDF. Unfortunately, depending on the use case, the definition on what is a security

domain can largely differ. For example, a security domain can be a chunk of the address space (e.g., SGX enclaves), a whole process (e.g., TrustZone application), a group of processes in a common container (e.g., Docker, LXC), or even a full virtual machine (e.g., cloud scenario). Considering that it is next to impossible to define a generic policy in hardware that can capture all these possibilities, we delegate the distinction to software that knows about the desired isolation properties, e.g., the Operating System (OS).

ScatterCache Interface. Depending on the targeted processor architecture, different design spaces can be explored before deciding how the current SDID gets defined and what channels are used to communicate the identifier to the ScatterCache. However, at least for modern Intel and ARM processors, binding the currently used SDID to the virtual memory management via user defined bits in each Page Table Entry (PTE) is a promising approach. In more detail, one or more bits can be embedded into each PTE that select from a list, via one level of indirection, which SDID should be used when accessing the respective page.

Both ARM and Intel processors already support a similar mechanism to describe memory attributes of a memory mapping. The x86 architecture defines so-called Page Attribute Tables (PATs) to define how a memory mapping can be cached. Similarly, the ARM architecture defines Memory Attribute Indirection Registers (MAIRs) for the same purpose. Both PAT and MAIR define a list of 8 memory attributes which are applied by the Memory-Management Unit (MMU). The MMU interprets a combination of 3 bits defined in the PTE as index into the appropriate list, and applies the corresponding memory attribute. Adding the SDID to these attribute lists permits to use up to 8 different security domains within a single process. The absolute number of security domains, on the other hand, is only limited by the used IDF and them number of bits that represent the SDID.

Such indirection has a huge advantage over encoding data directly in a PTE. The OS can change a single entry within the list to affect all memory mappings using the corresponding entry. Thus, such a mechanism is beneficial for ScatterCache, where the OS wants to change the SDID for all mappings of a specific process.

Backwards Compatibility. Ensuring backwards compatibility is a key factor for gradual deployment of ScatterCache. By encoding the SDID via a separate list indexed by PTE bits, all processes, as well as the OS, use the same SDID, *i.e.*, the SDID stored as first element of the list (assuming all corresponding PTE bits are '0' by default). Thus, if the OS is not aware of the ScatterCache, all processes—including the OS—use the same SDID. From a software perspective, functionally, ScatterCache behaves the same as currently deployed caches. Only if the OS specifies SDIDs in the list, and sets the corresponding PTE bits to use a certain index, ScatterCache provides its strong security properties.

Implementation Example. In terms of capabilities, having a single bit in each PTE, for example, is already sufficient to implement security domains with process granularity and to maintain a dedicated domain for the OS. In this case,

$SDID_0$ can always be used for the OS ID while $SDID_1$ has to be updated as part of the context switch and is always used for the scheduled user space process. Furthermore, by reusing the SDID of the OS, also shared memory between user space processes can easily be implemented without security impact.

Interestingly, ScatterCache fully preserves the capability of the OS to share read-only pages (*i.e.*, libraries) also across security domains as no cache lines will be shared. In contrast, real shared memory has to always be accessed via the same SDID in all processes to ensure data consistency. In general, with ScatterCache, as long as the respective cache lines have not been flushed to RAM, data always needs to be accessed with the same SDID the data has been written with to ensure correctness. This is also true for the OS, which has to ensure that no dirty cache lines reside in the cache, e.g., when a page gets assigned to a new security domain.

A case which has to be explicitly considered by the OS is copying data from user space to kernel space and vice versa. The OS can access the user space via the direct-physical map or via the page tables of the process. Thus, the OS has to select the correct SDID for the PTE used when copying data. Similarly, if the OS sets up page tables, it has to use the same SDID as the MMU uses for resolving page tables.

6.3 Security Evaluation

ScatterCache is a novel cache design to efficiently thwart cache-based side-channel attacks. In the following, we investigate the security of ScatterCache using both theoretical analysis and empirically via simulation. In particular, this section presents our initial complexity results for building the eviction sets from the original publication [Wer+19b] with small updates based on the generalized and improved strategy by Purnal and Verbauwhede [PV19]. Moreover, necessary changes to the standard Prime+Probe technique—to make it viable on the ScatterCache architecture—are discussed.

6.3.1 Applicability of Cache Attacks

While certain types of cache attacks, such as Flush+Flush, Flush+Reload and Evict+Reload, require a particular cache line to be shared, attacks such as Prime+Probe have less stringent constraints and only rely on the cache being a shared resource. As sharing a cache line is the result of shared memory, we analyze the applicability of cache attacks on ScatterCache with regard to whether the underlying memory is shared between attacker and victim or not.

Shared, read-only memory. Read-only memory is frequently shared among different processes, e.g., in case of shared code libraries. ScatterCache prevents cache attacks involving shared read-only memory by introducing security domains. In particular, ScatterCache maintains a separate copy of shared read-only memory in cache for each security domain, *i.e.*, the cache lines belonging to the same shared memory region are not being shared in cache across

security domains anymore. As a result, reloading data into or flushing data out of the cache does not provide any information on another security domain's accesses to the respective shared memory region. Note, however, that the cache itself is shared, leaving attacks such as Prime+Probe still feasible.

Shared, writable memory. Exchanging data between processes requires shared, writable memory. To ensure cache coherency, writing shared memory regions must always use the same cache line and hence the same security domain for that particular memory region—even for different processes. While attacks on these shared memory regions involving `flush` instructions can easily be mitigated by making these instructions privileged, Evict+Reload remains feasible. Still, ScatterCache significantly hampers the construction of targeted eviction sets by skewing, *i.e.*, individually addressing, the cache ways. Moreover, its susceptibility to Evict+Reload attacks is constrained to the processes sharing the respective memory region. Nevertheless, ScatterCache requires writable shared memory to be used only as an interface for data transfer rather than sensitive computations. In addition, Prime+Probe attacks are still possible.

Unshared memory. Unshared memory regions never share the same cache line, hence making attacks such as Flush+Flush, Flush+Reload and Evict+Reload infeasible. However, as the cache component itself is shared, cache attacks such as Prime+Probe remain possible.

As our analysis shows, ScatterCache prevents a wide range of cache attacks that exploit the sharing of cache lines across security boundaries. While Prime+Probe attacks cannot be entirely prevented as long as the cache itself is shared, ScatterCache vastly increases their complexity in all aspects. The pseudorandom cache-set composition in ScatterCache prevents attackers from learning concrete cache sets from memory addresses and vice versa. Even if attackers are able to profile information about the mapping of memory addresses to cache-sets in their own security domain, it does not allow them infer the mapping of cache-sets to memory addresses in other security domains. To gain information about memory being accessed in another security domain, an attacker needs to profile the mapping of the attacker's address space to cache lines that are being used by the victim when accessing the memory locations of interest. The effectiveness of Prime+Probe attacks thus heavily relies on the complexity of such a profiling phase. We elaborate on the complexity of building eviction sets in Section 6.3.3.

6.3.2 Other Microarchitectural Attacks

Many other microarchitectural attacks are not fully mitigated but hindered by ScatterCache. For instance, Meltdown [Lip+18b] and Spectre [Koc+19] attacks cannot use the cache efficiently anymore but must resort to other covert channels. Also, DRAM row buffer attacks and Rowhammer attacks are negatively affected as they require to bypass the cache and reach DRAM. While these attacks are already becoming more difficult due to closed row policies in modern processors [Gru+18], we propose to make flush instructions privileged, removing the most widely used cache bypass. Cache eviction gets much more difficult

with ScatterCache and additionally, spurious cache misses will open DRAM rows during eviction. These spurious DRAM row accesses make the row hit side channel impractical and introduce a significant amount of noise on the row conflict side channel. Hence, while these attacks are not directly in the scope of this paper, ScatterCache arguably has a negative effect on them.

6.3.3 Complexity of Building Eviction Sets

Cache skewing significantly increases the number of different cache sets available in cache. However, many of these sets will overlap partially, *i.e.*, in $1 \leq i < n_{ways}$ ways. The complexity of building eviction sets for Evict+Reload and Prime+Probe in ScatterCache thus depends on the overlap of cache sets.

Full Cache-Set Collisions

There are $2^{b_{indices} \cdot n_{ways}}$ different possibilities to form one specific cache set in a ScatterCache. For a given target address, this results in a probability of $2^{-b_{indices} \cdot n_{ways}}$ of finding another address that maps exactly to the same cache lines in its assigned cache set. Even state-of-the-art systems commonly do not have sufficient number of physical addresses available to find such a full cache-set collision. A 4-way cache with $b_{indices} = 12$ index bits, for example, yields already 2^{48} different cache sets. Mounting an attack based on full cache-set collision can, hence, be considered impractical in real-world scenarios.

Partial Cache-Set Collisions

While full cache-set collisions are impractical, partial collisions of cache sets frequently occur in skewed caches such as ScatterCache. If the cache sets of two addresses overlap, two cache sets will most likely have a single cache line in common. For this reason, we analyze the complexity of eviction for single-way collisions in more detail.

Randomized Single-Set Eviction. Without knowledge of the concrete mapping from memory addresses to cache sets, the trivial approach of eviction is to access arbitrary memory locations, which will result in accesses to pseudorandom cache sets in ScatterCache. To elaborate on the performance of this approach, we consider a cache with $n_{lines} = 2^{b_{indices}}$ cache lines per way and investigate the eviction probability for a single cache way, which contains a specific cache line to be evicted. Given that ScatterCache uses a random (re-)placement policy, the probabilities of each cache way are independent, meaning that each way has the same probability of being chosen. Subsequently, the attack complexity on the full ScatterCache increases linearly with the number of cache ways, *i.e.*, the attack gets harder.

The probability of an arbitrary memory accesses to a certain cache way hitting a specific cache line is $p = n_{lines}^{-1}$. Performing $n_{accesses}$ independent accesses to this cache way increases the odds of eviction to a certain confidence level α.

$$\alpha = 1 - (1 - n_{lines}^{-1})^{n_{accesses}}$$

Equivalently, to reach a certain confidence α in evicting the specific cache line, attackers have to perform

$$\mathbb{E}(n_{accesses}) = \frac{\log(1-\alpha)}{\log(1-n_{lines}^{-1})}$$

independent accesses to this cache way, which amounts to their attack complexity. Hence, to evict a certain cache set from an 8-way ScatterCache with 2^{11} lines per way with $\alpha = 99\,\%$ confidence, the estimated attack complexity using this approach is $n_{accesses} \cdot n_{ways} \approx 2^{16}$ independent accesses.

Randomized Multi-Set Eviction. Interestingly, eviction of multiple cache sets using arbitrary memory accesses has similar complexity. In this regard, the *coupon collector's problem* gives us a tool to estimate the number of accesses an attacker has to perform to a specific cache way to evict a certain percentage of cache lines in the respective way. In more detail, the coupon collector's problem provides the expected number of accesses $n_{accesses}$ required to a specific cache way such that n_{hit} out of all n_{lines} cache lines in the respective way are hit.

$$\mathbb{E}(n_{accesses}) = n_{lines} \cdot (H_{n_{lines}} - H_{n_{lines}-h_{hit}})$$

Hereby, H_n denotes the n-th Harmonic number, which can be approximated using the natural logarithm. This approximation allows to determine the number of cache lines n_{hit} that are expected to be hit in a certain cache way when $n_{accesses}$ random accesses to the specific way are performed.

$$\mathbb{E}(n_{hit}) = n_{lines} \cdot (1 - e^{-\frac{n_{accesses}}{n_{lines}}}) \tag{6.1}$$

Using n_{hit}, we can estimate the number of independent accesses to be performed to a specific cache way such that a portion β of the respective cache way is evicted.

$$\mathbb{E}(n_{accesses}) = -n_{lines} \cdot \ln(1-\beta)$$

For the same 8-way ScatterCache with 2^{11} lines per way as before, we therefore require roughly 2^{16} independent accesses to evict $\beta = 99\,\%$ of the cache.

Profiled Eviction for Prime+Probe. As shown, relying on random eviction to perform cache-based attacks involves significant effort and yields an over-approximation of the eviction set. Moreover, while random eviction is suitable for attacks such as Evict+Reload, in Prime+Probe settings random eviction fails to provide information related to the concrete memory location that is being used by a victim. To overcome these issues, attackers may profile a system to construct eviction sets for specific memory addresses of the victim, *i.e.*, they try to find a set of addresses that map to cache sets that partially overlap with the cache set corresponding to the victim address. Eventually, such sets could be used to speed up eviction and to detect accesses to specific memory locations. In the following, we analyze the complexity of finding these eviction sets and present our original approach. In more detail, we perform analysis w.r.t.

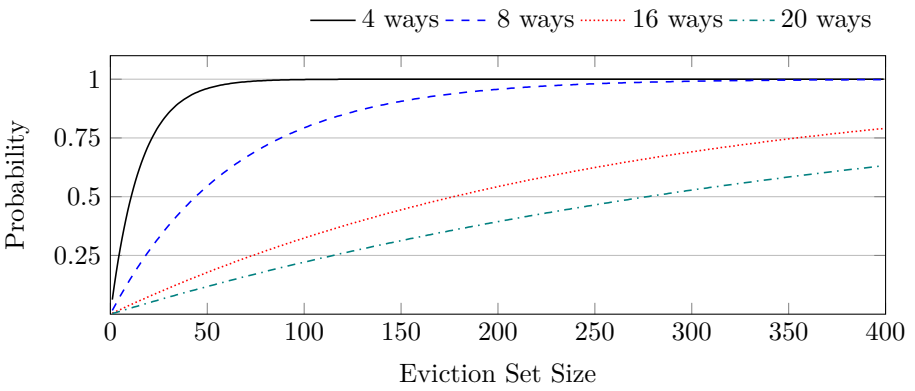


Figure 6.5: Eviction probability depending on the size of the eviction set and the number of ways.

eviction addresses whose cache sets overlap with the cache set of a victim address in a single cache way only.

To construct a suitable eviction set for PRIME+PROBE, the attacker needs to provoke the victim process to perform the access of interest. In particular, the attacker tests a candidate address for cache-set collisions by accessing it (prime), waiting for the victim to access the memory location of interest, and then measuring the time when accessing the candidate address again (probe). In such a profiling procedure, after the first attempt, we have to assume that the cache line belonging to the victim access already resides in the cache. As a result, attackers need to evict a victim's cache line in their prime step. Hereby, hitting the right cache way and index have probability ${n_{ways}}^{-1}$ and $2^{-b_{indices}}$, respectively. To be able to detect a collision during the probe step, the victim access must then fall into the same cache way as the candidate address, which has a chance of ${n_{ways}}^{-1}$. In total, the expected number of memory accesses required to construct an eviction set of t colliding addresses hence is

$$\mathbb{E}(n_{accesses}) = {n_{ways}}^2 \cdot 2^{b_{indices}} \cdot t.$$

The number of memory addresses t needs to be chosen according to the desired eviction probability for the victim address with the given set. When the eviction set consists of addresses that collide in the cache with the victim in exactly one way each, the probability of evicting the victim with an eviction set of size t is

$$p(\text{Eviction}) = 1 - \left(1 - \frac{1}{n_{ways}}\right)^{\frac{t}{n_{ways}}}.$$

Figure 6.5 depicts this probability for the size of the eviction set and different numbers of cache ways. For an 8-way SCATTERCACHE with 2^{11} cache lines per way, roughly 275 addresses with single-way cache collisions are needed to evict the respective cache set with 99 % probability. Constructing this eviction set using the

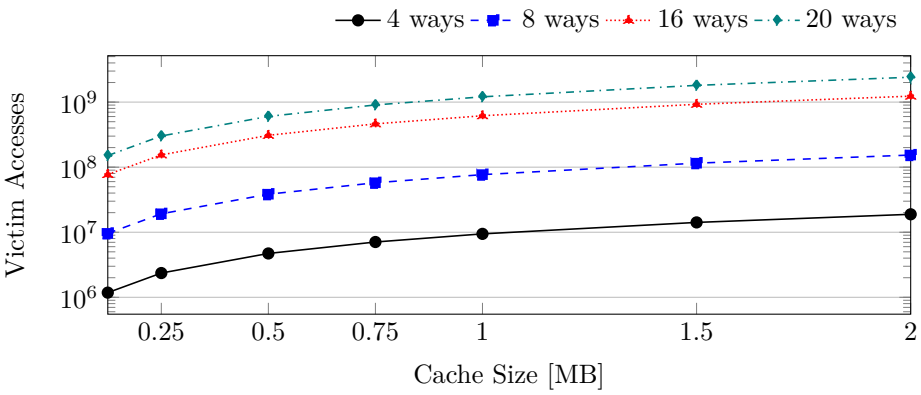


Figure 6.6: Number of required accesses to the target address to construct a set large enough to achieve 99 % eviction rate when no shared memory is available (cache line size: 32 bytes).

presented approach requires profiling of approximately $8^2 \cdot 2^{11} \cdot 275 \approx 2^{25}$ (33.5 million) victim accesses. Figure 6.6 shows the respective number of PRIME+PROBE experiments needed to generate sets with 99 % eviction probability for different cache configurations. We were able to empirically confirm these numbers within a noise-free standalone simulation of SCATTERCACHE. For comparison, to generate an eviction set on a commodity cache, e.g., recent Intel processors, for a specific victim memory access, an attacker needs fewer than 103 observations of that access in a completely noise-free attacker-controlled scenario.

After publication, Purnal and Verbauwhede [PV19] generalized and considerably improved our approach for finding eviction sets. When searching for colliding addresses, their improved profiling strategy probes k' different addresses in parallel which considerably reduces the number of required victim accesses. This parallel search is possible due to the addition of a *pruning step* that guarantees that all k' addresses reside within the cache before the victim access is triggered. The improved approach, therefore, reduces the number of victim accesses by trading them against additional attacker accesses in the pruning step. In the best attack parameterization, for the previously discussed cache configuration, as little as 825 controlled victim accesses are required for building the eviction set.

Profiled Eviction for Evict+Reload. For shared memory, such as in EVICT+RELOAD, the construction of eviction sets, however, becomes easier, as shared memory allows the attacker to simply access the victim address. Hence, to build a suitable eviction set, the attacker first primes the victim address, then accesses a candidate address, and finally probes the victim address. In case a specific candidate address collides with the victim address in the cache way the victim access falls into, the attacker can observe this collision with probability $p = {n_{ways}}^{-1}$. As a result, the expected number of memory accesses required to

build an eviction set of t colliding addresses for Evict+Reload is

$$\mathbb{E}(n_{accesses}) = n_{ways} \cdot 2^{b_{indices}} \cdot t.$$

Constructing an Evict+Reload eviction set of 275 addresses (*i.e.*, 99 % eviction probability) requires profiling with roughly $8 \cdot 2^{11} \cdot 275 = 2^{22}$ memory addresses for an 8-way ScatterCache with 2^{11} lines per way. Note, however, that Evict+Reload only applies to writable shared memory as used for Inter-Process Communication (IPC), whereas ScatterCache effectively prevents Evict+Reload on shared read-only memory by using different cache-set compositions in each security domain. Moreover, eviction sets for both Prime+Probe and Evict+Reload must be freshly created whenever the key or the SDID changes.

6.3.4 Complexity of Prime+Probe

As demonstrated, ScatterCache strongly increases the complexity of building the necessary sets of addresses for Prime+Probe. However, the actual attacks utilizing these sets are also made more complex by ScatterCache.

In this section, we make the strong assumption that an attacker has successfully profiled the victim process such that they have found addresses which collide with the victim's target addresses in exactly 1 way each, have no collisions with each other outside of these and are sorted into subsets corresponding to the cache line they collide in.

Where in normal Prime+Probe an attacker can infer victim accesses (or a lack thereof) with near certainty after only 1 sequence of priming and probing, ScatterCache degrades this into a probabilistic process. At best, one Prime+Probe operation on a target address can detect an access with a probability of ${n_{ways}}^{-1}$. This is complicated further by the fact that any one set of addresses is essentially single-use, as the addresses will be cached in a non-colliding cache line with a probability of $1 - {n_{ways}}^{-1}$ after only 1 access, where they cannot be used to detect victim accesses anymore until they themselves are evicted again.

Given the profiled address sets, we can construct general probabilistic variants of the Prime+Probe attack. While other methods are possible, we believe the 2 described in the following represent lower bounds for either victim accesses or memory requirement.

Variant 1: Single collision with eviction. We partition our set of addresses, such that one Prime+Probe set consists of n_{ways} addresses, where each collides with a different way of the target address. To detect an access to the target, we prime with one set, cause a target access, measure the primed set and then evict the target address. We repeat this process until the desired detection probability is reached. This probability is given by $p(n_{accesses}) = 1 - (1 - {n_{ways}}^{-1})^{n_{accesses}}$. The eviction of the target address can be achieved by either evicting the entire cache or using preconstructed eviction sets (see Section 6.3.3). After the use of an eviction set, a different priming set is necessary, as the eviction sets only

target the victim address. After a full cache flush, all sets can be reused. The amount of colliding addresses we need to find during profiling depends on how often a full cache flush is performed. This method requires the least amount of accesses to the target, at the cost of either execution time (full cache flushes) or memory and profiling time (constructing many eviction sets).

Variant 2: Single collision without eviction. Using the same method but without the eviction step, the detection probability can be recursively calculated as

$$p(n_{acc.}) = p(n_{acc.} - 1) + (1 - p(n_{acc.} - 1))(\frac{2 \cdot n_{ways} - 1}{{n_{ways}}^3})$$

with $p(1) = {n_{ways}}^{-1}$. This variant provides decreasing benefits for additional accesses. The reason for this is that the probability that the last step evicted the target address influences the probability to detect an access in the current step. While this approach requires many more target accesses, it has the advantage of a shorter profiling phase.

These two methods require different amounts of memory, profiling time and accesses to the target, but they can also be combined to tailor the attack to the target. Which is most useful depends on the attack scenario, but it is clear that both come with considerable drawbacks when compared to Prime+Probe in current caches. For example, achieving a 99 % detection probability in a 2 MB Cache with 8 ways requires 35 target accesses and 9870 profiled addresses in 308 MB of memory for variant 1 if we use an eviction set for every probe step. Variant 2 would require 152 target accesses and 1216 addresses in 38 MB of memory. In contrast, regular Prime+Probe requires 1 target access and 8 addresses while providing 100 % accuracy (in this ideal scenario). Detecting non-repeating events is made essentially impossible; to measure any access with confidence requires either the knowledge that the victim process repeats the same access pattern for long periods of time or control of the victim in a way that allows for repeated measurements. In addition to the large memory requirements, variant 1 also heavily degrades the temporal resolution of a classical Prime+Probe attack because of the necessary eviction steps. This makes trace-based attacks like attacks on square-and-multiply in RSA [YF14] much less practical. Variant 2 does not suffer from this drawback, but requires one Prime+Probe set for each time step, for as many high-resolution samples as one trace needs to contain. This can quickly lead to an explosion in required memory when thousands of samples are needed.

6.3.5 Challenges with Real-World Attacks

We failed at mounting a real-world attack (*i.e.*, with even the slightest amounts of noise) on ScatterCache. Generally, for a Prime+Probe attack we need to (1) generate an eviction set (cf. Section 6.3.3), and (2) use the eviction set to monitor a victim memory access. If we assume step 1 to be solved, we can mount a cache attack (*i.e.*, step 2) with a complexity increases by a factor of 152 (cf. Section 6.3.4). For some real-world attacks this would not be a problem,

in particular if a small fast algorithm is attacked, e.g., AES with T-tables. Gülmezoglu et al. [Gül+15] recovered the full AES key from an AES T-tables implementation with only 30 000 encryptions in a fully synchronized setting (that can be implemented with Prime+Probe as well [Gru+16b]), taking 15 seconds, *i.e.*, 500 µs per encryption. The same attack on ScatterCache takes $4.56 \cdot 10^6$ encryptions, *i.e.*, 38 minutes assuming the same execution times, which is clearly viable.

However, the real challenge is solving step 1, which we did not manage for any real-world example when writing the paper. In particular, even if AES would only perform a single attacker-chosen memory access (instead of 160 to the T-tables alone, plus additional code and data accesses), which would be ideal for the attacker in the profiling during step 1, using our initial profiling approach we need to observe 33.5 million encryptions. In addition to the runtime reported by Gülmezoglu et al. [Gül+15] we also need a full cache flush after each attack round (*i.e.*, each encryption). For a 2 MB cache, we need to iterate over a 6 MB array to have a high probability of covering all cache lines. The time for an L3-cache access is e.g., for Kaby Lake 9.5 ns [7cpb]. The absolute minimum number of cache misses here is 65536 (=4 MB), but in practice it will be much higher. A cache miss takes around 50 ns, hence, the full cache eviction will take at least 3.6 ms. Consequently, using the original profiling approach, with 33.5 million tests required to generate the eviction set and a runtime of 4.1 ms per test, the total runtime to generate the eviction set is 38 hours.

Using the improved profiling approach by Purnal and Verbauwhede [PV19], on the other hand, time for the generation of the eviction set becomes more than feasible and drops to less than 5 seconds. Still, these numbers only considers the theoretical setting of a completely noise-free and idle system. The process doing AES computations must not be restarted during the profiling and the attack phase. The operating system must not replace any physical pages and, most importantly, our hypothetical AES implementation only performs a single memory access. With a second memory access, these two memory accesses can already not be distinguished anymore with the generated eviction set, because the eviction set is generated for an invocation of the entire victim computation, not for an address. In any realistic setting with only the slightest amount of activity (noise) on the system, the required time for mounting an attack easily explodes.

6.3.6 Noise Sampling

The previous analysis considered a completely noise-free scenario, where the attacker performs Prime+Probe on a single memory access executed by the victim. However, in a real system, an attacker will typically not be able to perform an attack on single memory accesses, but face different kinds of noise. Namely, on real systems cache attacks will suffer from both systematic and random noise, which reduces the effectiveness of profiling and the actual attack.

Systematic noise is introduced, for example, by the victim as it executes longer code sequences in between the attacker's prime and probe steps. The

victim's code execution intrinsically performs additional memory accesses to fetch code and data that the attacker will observe in the cache deterministically. In ScatterCache, the mappings of memory addresses to cache lines is unknown. Hence, without additional knowledge, the attacker is unable to distinguish the cache collision belonging to the target memory access from collisions due to systematic noise. Instead, the attacker can only observe and learn both simultaneously. As a result, larger eviction sets need to be constructed to yield the same confidence level for eviction. Specifically, the size of an eviction set must increase proportionally to the number of systematic noise accesses to achieve the same properties. While this significantly increases an attackers profiling effort, they may be able to use clustering techniques to prune the eviction set prior to performing an actual attack.

Random noise, on the other hand, stems from arbitrary processes accessing the cache simultaneously or as they are scheduled in between. Random noise hence causes random cache collisions to be detected by an attacker during both profiling and an actual attack, *i.e.*, produces false positives. While attackers cannot distinguish between such random noise and systematic accesses in a single observation, these random noise accesses can be filtered out statistically be repeating the same experiment multiple times. Yet, it increases an attackers effort significantly. For instance, when building eviction sets an attacker can try to observe the same cache collision multiple times for a specific candidate address to be certain about its cache collision with the victim.

Random noise distributes in ScatterCache according to Equation 6.1 and hence quickly occupies large parts of the cache. As a result, there is a high chance of sampling random noise when checking a candidate address during the construction of eviction sets. Also when probing addresses of an eviction set in an actual attack, random noise is likely to be sampled as attacks on ScatterCache demand for large eviction sets. As our analysis shows, for a single cache way the distribution of cache line indices corresponding to the memory accesses of profiled eviction sets (cf. Section 6.3.3) adheres to Figure 6.7. Clearly, due to profiling there is a high chance of roughly $1/n_{ways}$ to access the index that collides with the victim address. However, with $p = (n_{ways} - 1)/n_{ways}$ the index adheres to an uniformly random selection from all possible indices and hence provides a large surface for sampling random noise. Consequently, for a cache with $n_{lines} = 2^{b_{indices}}$ lines per way and n_{noise} lines being occupied by noise in each way, the probability of sampling random noise when probing an eviction set address is

$$p(\text{Noise}) \approx \frac{n_{ways} - 1}{n_{ways}} \frac{n_{noise}}{n_{lines}}.$$

Figure 6.8 visualizes this effect and in particular the percentage of noisy samples encountered in an eviction set for different cache configurations and noise levels. While higher random noise clearly increases an attackers effort, the actual noise level strongly depends on the system configuration and load.

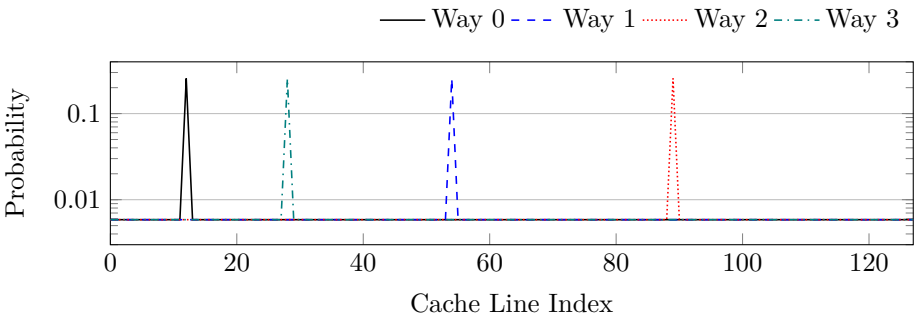


Figure 6.7: Example distribution of cache indices of addresses in profiled eviction sets ($n_{ways} = 4$, $b_{indices} = 7$).

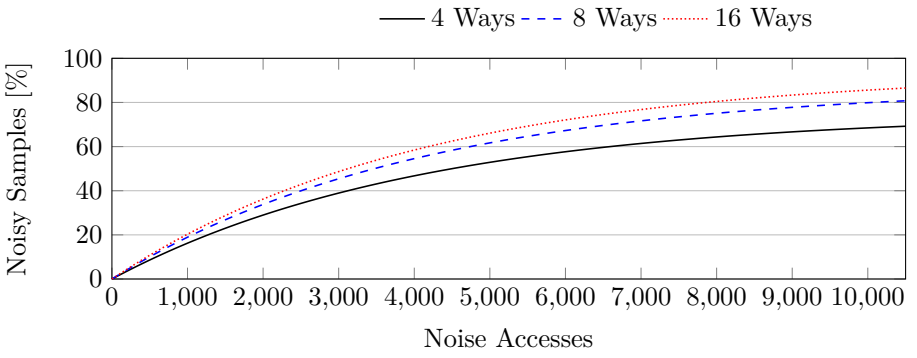


Figure 6.8: Expected percentage of noisy samples in an eviction set for a cache consisting of 2^{12} cache lines.

6.3.7 Further Remarks

In the previous analysis, the SDIDs of both attacker and victim were assumed to be constant throughout all experiments for statistical analysis to be applicable. Additionally, systematic and random noise introduced during both profiling and attack further increase the complexity of actual attacks, rendering attacks on most real-world systems impractical.

Also note that the security analysis in this section focuses on SCv1. In a noise-free scenario, SCv2 may allow to construct eviction sets slightly more efficiently since its IDF is a permutation. This means that, once a collision in a certain cache way is found, there will not be any other colliding address for that cache way in the same index range, *i.e.*, for the same address tag. Considering the expected time to find the single collision in a given index range, this could give an attacker a benefit of up to a factor of two in constructing eviction sets. However, in practice multiple cache ways are profiled simultaneously, which results in a high chance of finding a collision in any of the cache ways independent of the address index bits, *i.e.*, the n_{ways} indices for a certain memory address will very

likely be scattered over the whole index range. Independent of that, the presence of noise significantly hampers taking advantage of the permuting property of SCv2.

6.4 Performance Evaluation

ScatterCache significantly increases the effort of attackers to perform cache-based attacks. However, a countermeasure must not degrade performance to be practical as well. This section hence analyzes the performance of ScatterCache using the gem5 full system simulator and GAP [BAP15], MiBench [Gut+01], lmbench [MS96], and the C version of scimark2 [1] as micro benchmarks. Additionally, to closer investigate the impact of ScatterCache on larger workloads, a custom cache simulator is used for SPEC CPU 2017 benchmarks. Our evaluations indicate that, in terms of performance, ScatterCache behaves basically identical to traditional set-associative caches with the same random replacement policy.

6.4.1 gem5 Setup

We performed our cache evaluation using the gem5 full system simulator [Bin+11] in 32-bit ARM mode. In particular, we used the CPU model TimingSimpleCPU together with a cache architecture such as commonly used in ARM Cortex-A9 CPUs: the cache line size was chosen to be 32 bytes, the 4-way L1 data and instruction caches are each sized 32 kB, and the 8-way L2 cache is 512 kB large. We adapted the gem5 simulator such as to support ScatterCache for the L2 cache. This allows to evaluate the impact of six different cache organizations. Besides ScatterCache in both variants (1) SCv1 and (2) SCv2 and standard set-associative caches with (3) LRU, (4) BIP, and (5) random replacement, we also evaluated (6) skewed associative caches [Sez93] with random replacement as we expect them to have similar performance characteristics as SCv1 and SCv2.

On the software side, we used the Poky Linux distribution from Yocto 2.5 (Sumo) with kernel version 4.14.67 after applying patches to run within gem5. We then evaluated the performance of our micro benchmarks running on top of Linux. In particular, we analyzed the cache statistics provided by gem5 after booting Linux and running the respective benchmark. Using this approach, we reliably measure the cache performance and execution time for each single application, *i.e.*, without concurrent processes. Since only the L2-cache architecture (*i.e.*, replacement policy, skewed vs. fixed sets) changed between the individual simulation runs, execution performance is simply direct proportional to the resulting cache hit rate. To enable easier comparison between the individual benchmarks as well as with related work we therefore mainly report L2-cache hit results.

ScatterCache IDF Instantiations. Both ScatterCache variants have been instantiated using the low-latency tweakable block cipher QARMA-64 [Ava17].

[1] https://math.nist.gov/scimark2/

In particular, in the SCv1 variant, the index bits for the individual cache ways have been sliced from the ciphertext of encrypting the cache line address under the secret key and SDID. On the other hand, due to the lack of an off-the-shelf tweakable block cipher with the correct block size, a stream cipher construction was used in the SCv2 variant. Namely, the index is computed as the XOR between the original index bits and the ciphertext of the original tag encrypted using QARMA-64. Note, however, that, although this construction for SCv2 is a proper permutation and entirely sufficient for evaluating the performance of SCv2, we do not recommend the construction as pads are being reused for addresses having the same tag bits.

While the majority of the following results are latency agnostic LLC hit rates, all following results are reported for the zero cycle latency case. For QARMA-64 with 5 rounds, Application Specific Integrated Circuit (ASIC) implementation results with as little as 2.2 ns latency have been reported [Ava17]. We are therefore confident that, if desired, hiding the latency of the IDF by computing it in parallel to the lower level cache lookup is feasible.

However, we still also conducted simulations with latency overheads between 1 and 5 cycles by increasing the `tag_latency` of the cache in gem5. The acquired results show that, even for IDFs which introduce 5 cycles of latency, less than 2 % performance penalty are encountered on the GAP benchmark suite. These numbers are also in line with Qureshi's results reported for CEASER [Qur18].

6.4.2 Hardware Overhead Discussion

ScatterCache is designed to be as similar to modern cache architectures as possible in terms of hardware. Still, area and power overheads have to be expected due to the introduction of the IDF and the additional addressing logic. Unfortunately, while probably easy for large processor and System-on-Chip (SoC) vendors, determining reliable overhead numbers for these two metrics is a difficult task for academia that requires an actual ASIC implementation of the cache. To the best of our knowledge, even in the quite active RISC-V community, no open and properly working LLC designs are available that can be used as foundation. Furthermore, for merely simulating such a design with a reasonably large cache, commercial Electronic Design Automation (EDA) tools, access to state-of-the-art technology libraries, and large memory macros with power models are required. As the result, secure cache designs typically fail to deliver hardware implementation results (see Table 6 in [DXS19]).

Because of these problems, similar to related work, we can also not provide concrete numbers for the area and power overhead. However, due to the way we designed ScatterCache and the use of lightweight cryptographic primitives, we can assert that the hardware overhead is reasonable. For example, the 8-way SCv1 ScatterCache with 512 kB that is simulated in the following section, uses two parallel instances of QARMA-64 with 5 rounds as IDF. One fully unrolled instance has a size of 22.6 kGE [Ava17][2] resulting in an IDF size

[2] 1 GE conforms to the area of a 2-input NAND gate with driving strength 1.

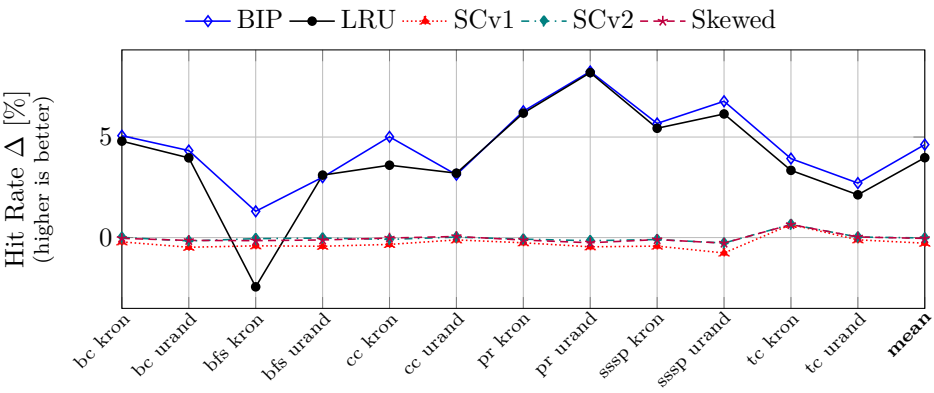


Figure 6.9: Cache hit rate, simulated with gem5, for the synthetic workloads in the GAP benchmark suite with random replacement policy as baseline.

of less then 50 kGE even in case additional pipeline registers are added. The added latency of such an IDF is the same as the latency of the used primitive which has been reported as 2.2 ns. However, this latency can (partially or fully) be hidden by computing the IDF in parallel to the lower level cache lookup. Interestingly, with similar size, also a sponge-based SCv1 IDF (e.g., 12 rounds of Keccak[200] [Ber+12a]) can be instantiated. Finally, there is always the option to develop custom IDF primitives [Qur18] that demand even less resources.

For comparison, in the BROOM chip [Cel+19], the RAM macros in the 1 MB L2 cache already consume roughly 50 % of the 4.86 mm^2 chip area. Assuming an utilization of 75 % and a raw gate density of merely 3 MGE/mm^2 [Eur19] for the used 28 nm TSMC process, these 2.43 mm^2 already correspond to 5.5 MGE. Subsequently, even strong IDFs are orders of magnitude smaller than the size of a modern LLC.

In terms of overhead for the individual addressing of the cache ways, information is more sparse. Spjuth et al. [SKH05] observed a 17 % energy consumption overhead for a 2-way skewed cache. They also report that skewed caches can be built with lower associativity and still reach similar performance as traditional fixed set-associative caches. Furthermore, modern Intel architectures already feature multiple addressing circuits in their LLC as they partition it into multiple smaller caches (*i.e.*, cache slices).

6.4.3 gem5 Results and Discussion

Figure 6.9 visualizes the cache hit rate of our L2 cache when executing programs from the GAP benchmark suite. To ease visualization, the results are plotted in percentage points (pp), *i.e.*, the differences between percentage numbers, using the fixed set-associative cache with random replacement policy as baseline. All six algorithms (*i.e.*, `bc`, `bfs`, `cc`, `pr`, `sssp`, `tc`) have been evaluated. Moreover, as trace sets, both synthetically generated `kron` (`-g16 -k16`) and `urand` (`-u16`

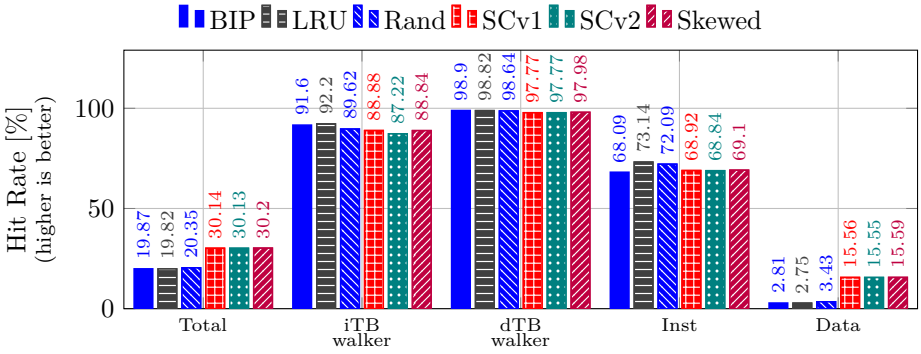


Figure 6.10: Cache hit rate, simulated with gem5, for scimark2.

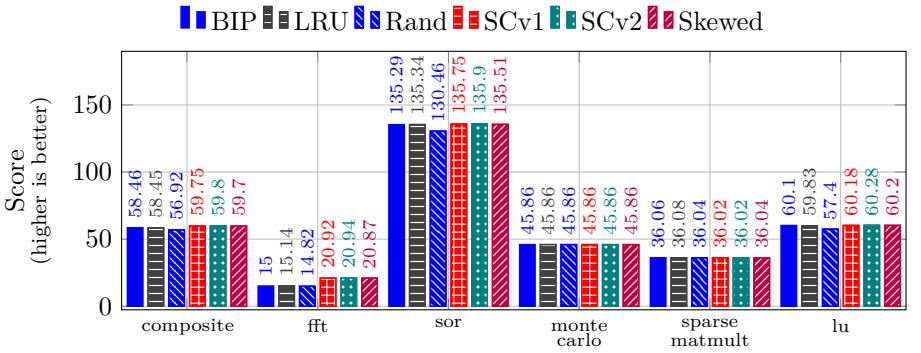


Figure 6.11: Scimark2 score simulated with gem5.

`-k16`) sets have been used. As can be seen in the graph, the BIP and LRU replacement policies outperform random replacement on average by 4.6 pp and 4 pp respectively. Interestingly, however, all random replacement based schemes, including the skewed variants, perform basically identical.

The next benchmark, we visualized in Figure 6.10, is scimark2 (`-large 0.5`). This benchmark shows an interesting advantage of the skewed cache architectures over the fixed-set architectures, independent of the replacement policy, of approximately 10 pp for the total hit rate. This difference is mainly caused by the 5x difference in hit rate for data accesses. Comparing the achieved benchmark scores in Figure 6.11 further reveals that the `fft` test within scimark2 is the reason for the observed discrepancy in cache performance.

To investigate this effect in more detail, we measured the memory read latency using using `lat_mem_rd 8M 32` from lmbench in all cache configurations. The respective results in Figure 6.12 feature two general steps in the read latency at 32 kB (L1-cache size) and at 512 kB (L2-cache size). Notably, configurations with random replacement policy feature a smoother transition at the second step,

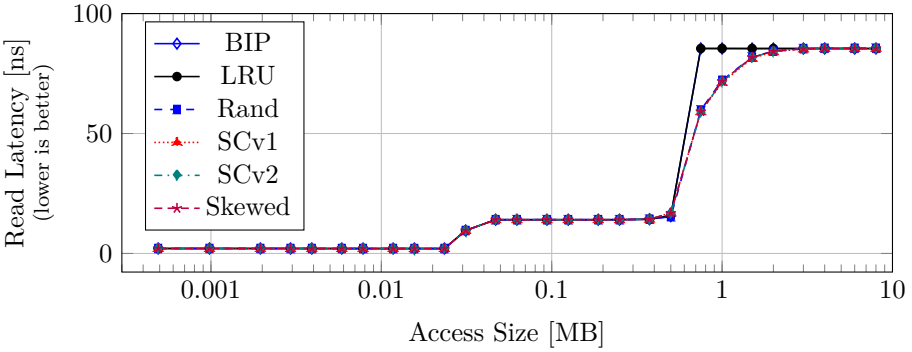


Figure 6.12: Memory read latency, simulated with gem5, with 32 byte stride (*i.e.*, one access per cache line).

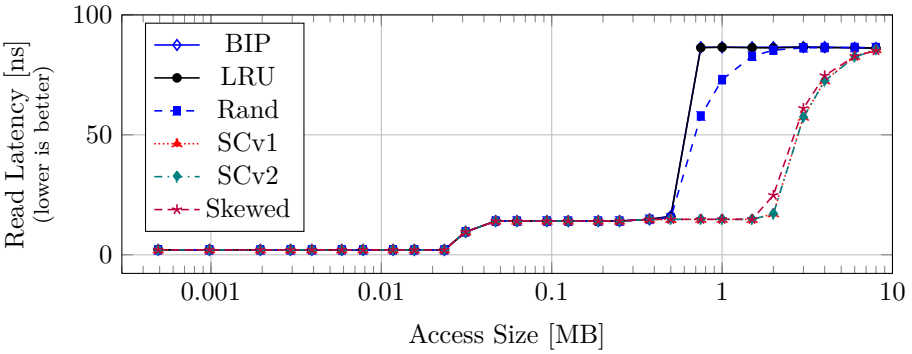


Figure 6.13: Memory read latency, simulated with gem5, with 128 byte stride (*i.e.*, one access in every fourth cache line).

i.e., when accesses start to hit main memory instead of the L2 cache.

Even more intersting results, as shown in Figure 6.13, have been acquired by increasing the stride size to four times the cache line size. Skewed caches like ScatterCache break the strong alignment of addresses and cache set indices. As a consequence, a sparse, but strongly aligned memory access pattern such as in `lat_mem_rd`, which in a standard set-associative caches only uses every 4^{th} cache index, gives high cache hit rates and low read latencies for larger memory ranges due to less cache conflicts. This effect becomes visible in Figure 6.13 as shift of the second step from 512 kB to 2 MB for the skewed cache variants.

Finally, as last benchmark, MiBench has been evaluated in small and large configuration. The individual results are visualized in Figure 6.14 and Figure 6.15 respectively. On average, the achieved performance results in MiBench are very similar to the results from the GAP benchmark suite. Again, caches with BIP and LRU replacement policy outperform the configurations with random replacement policy by a few percent. However, in some individual benchmarks (e.g., qsort in

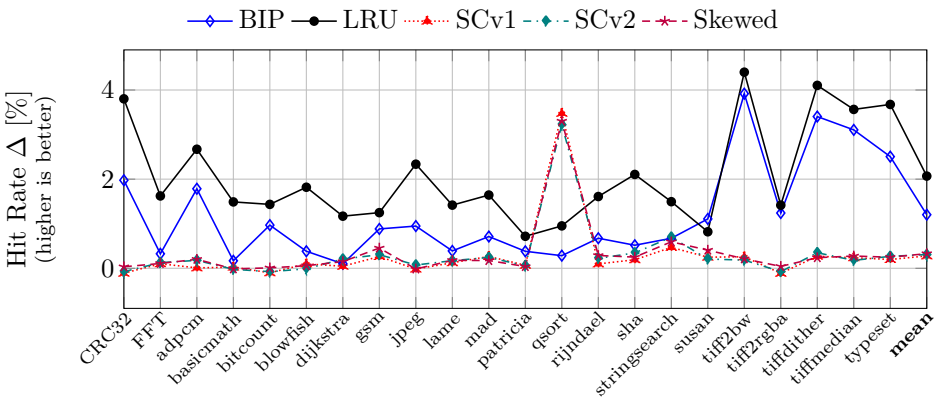


Figure 6.14: Cache hit rate, simulated with gem5, for MiBench in small configuration compared to random replacement.

small, `jpeg` in large), skewed cache architectures like ScatterCache outmatch the fixed set appraoches.

In summary, our evaluations with gem5 in full system simulation mode show that the performance of ScatterCache, in terms of hit rate, is basically identical to contemporary fixed set-associative caches with random replacement policy. Considering that we employ the same replacement strategy, this is an absolutely satifying result by itself. Moreover, no tests indicated any notable performance degradation and in some tests ScatterCache even outperformed BIP and LRU replacement policies.

6.4.4 Cache Simulation and SPEC Results

Lastly, we evaluated the performance of ScatterCache using the SPEC CPU 2017 [Sta] benchmark with both the "SPECspeed 2017 Integer" and "SPECspeed 2017 Floating Point" suites. We performed all benchmarks in these suites with the exception of `gcc`, `wrf` and `cam4`, as these failed to compile on our system. Because these benchmarks are too large to be run in full system simulation, we created a software cache simulator, capable of simulating different cache models and replacement policies. Even so, the benchmarks proved to be too large to run in full, so we opted to run segments of 250 million instructions from each, following the methodology of Qureshi et al. [Qur+07]. We made an effort to select parts of the benchmarks that are representative of their respective core workloads. To be able to run the benchmarks with our simulator, we recorded a trace of all instruction addresses and memory accesses with the Intel PIN Tool [Int]. We then replayed this access stream for different cache configurations. The simulator implements the set-associative replacement policies Pseudo-LRU (Tree-PLRU), LRU (ideal), BIP as described in [Qur+07], and random replacement, as well as the two ScatterCache variants. The number of ways per set, total cache

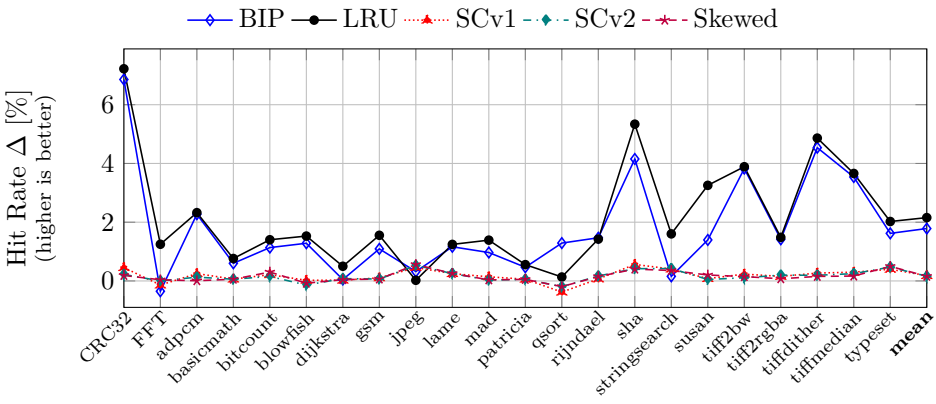


Figure 6.15: Cache hit rate, simulated with gem5, for MiBench in large configuration compared to random replacement.

size, number of slices, and cache line size are fully configurable. Additionally, the simulator supports multiple levels of inclusive caches, as well as a cache that is split for data and instructions. All simulations were run on an inclusive two level cache, where the L1 was separated into instruction and data caches, both of which use LRU replacement. Figure 6.16 shows results for the cache configuration, as described in Section 6.4.1, as the difference in percentage points for last-level hit rates when compared to random replacement. While we can see large differences in individual tests, the mean shows that both versions of ScatterCache perform at least as well as random replacement and very similar to LRU. Using the same cache configuration but with 64 B cache lines, we actually observe a mean advantage of 0.23 ± 0.76 pp of ScatterCache over random replacement, where LRU sees a marginally worse result of -0.21 ± 1.02 pp. On a larger configuration with 64 B cache lines, 32 kB 8-way L1 and 2 MB 16-way LLC, the results show a slim improvement of 0.035 ± 0.10 pp for ScatterCache and 0.37 ± 1.14 pp for LRU over random replacement.

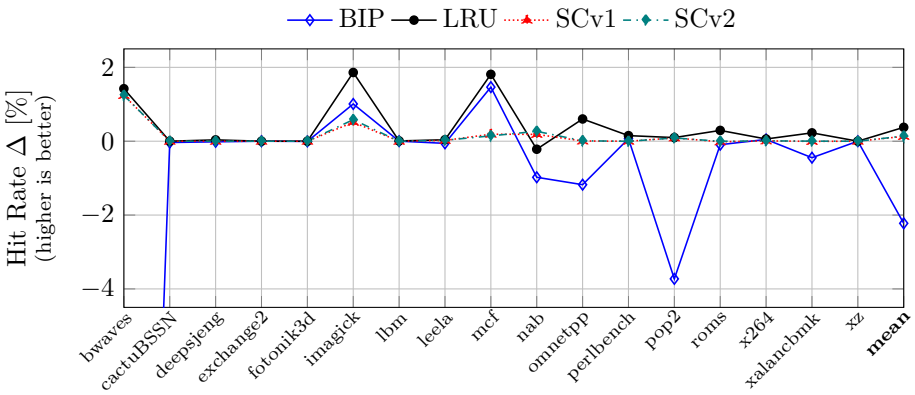


Figure 6.16: Average cache hit rate for SPEC CPU 2017 benchmarks compared to random replacement over 10 runs.

6.5 Conclusion

In this chapter, we presented ScatterCache, a novel cache architecture designed to make eviction-based cache attacks unpractical by eliminating fixed cache-set congruences. Internally, ScatterCache combines the individual cache-way addressing from skewed set-associative caches with a lightweight keyed mapping function. Additionally, usage of an optional software-controlled tweak, *i.e.*, the SDID, is supported that enables ScatterCache-aware operating system to control if data, while shared memory in RAM, should also be shared in the cache. The resulting design is, in terms of hardware, still very similar to traditional caches which eases possible migration efforts. Moreover, also runtime performance of the software on ScatterCache-enabled processors is not curtailed.

In terms of security, attacks on ScatterCache are by construction probabilistic and require that targeted memory accesses can be observed many times for both, the actual attack and the needed profiling. Furthermore, novel security policies can be implemented using the added SDID parameter. ScatterCache is, therefore, definitely an improvement over regular set-associative caches. However, given that we currently only have comparably simple models to analyze the design and the resulting security implications, quantifying the security gain against real-world attacks is still an unsolved problem that requires further research.

7 Conclusion

In this thesis, we worked towards fixing the shortcomings of current architectures in the context of physical attacks. To make progress towards the ambitious goal of building more secure systems, we followed the approach of augmenting contemporary system architectures with minimal hardware extensions. These hardware extensions were then used as security-anchor for building further countermeasures efficiently in software. Following this methodology, we presented several novel techniques that can be deployed in general-purpose processor-cores and their memory subsystems, respectively.

In terms of code protection, two hardware-supported CFI schemes were developed that harden CPUs against software and physical attacks. Both techniques have been implemented in real processor designs, come with appropriate toolchain support, and were tested in simulation and/or on actual FPGA/ASIC hardware.

The first scheme, as discussed in Chapter 2, is based on GPSA and extends a stock processor core (*i.e.*, no added instructions) with a none-invasive signature monitor that is capable of computing signatures over the executed instructions. Checking full computed signatures against compile-time derived expected values is possible at arbitrary programmer defined positions. Additionally, CSM can be used to check a few bits with every executed instruction. Various signature functions have been evaluated and the best, in terms of fault attack complexity, has been implemented in a processor with ARMv7-M instruction set (*i.e.*, similar to an ARM Cortex-M3). The achieved runtime and size overhead—$< 75\,\%$ and $< 120\,\%$ with 4-bit CSM—is very reasonable for a software-heavy prototype.

Compared to our GPSA implementation, the second scheme—named SCFP—is more invasive in exchange for stronger security properties (e.g., added confidentiality) and better performance. As presented in Chapter 3, the idea of SCFP is to place a stateful sponge-based cryptographic primitive between a CPU's

fetch and decode stage that continuously decrypts and authenticates instructions. Programs for such a CPU, hence, have to be encrypted by taking into account the sequence in which individual instructions have to be executed. During execution of such a program, any tampering with instructions or their sequence leads to a pseudo random instruction stream that can hardly be predicted or controlled by an attacker. SCFP yields fine-grained control-flow integrity and thus prevents, for instance, code reuse, code injection, as well as fault attacks on the code and the control flow. For evaluation, we built a RISC-V core with SCFP support named *Remus*, which was even fabricated as part of the *Patronus* chip. On this chip, code size and execution time overheads of 19.8 % and 9.1 %, respectively, indicate that SCFP is quite efficient and even meets the requirements of typical embedded devices.

Following our methodology of building upon minimal hardware extensions, utilizing SCE techniques like SCFP, a new remote attestation concept featuring graph-based attestation has been introduced in Chapter 4. Given SCE capable hardware, this new concept can be implemented purely in software and additionally provides online licensing capabilities. Furthermore, commonly used static and path-based remote attestation, or any hybrid scheme that combines these approaches, can be implemented with our attestation scheme too. Our proof of concept implementation for the *Patronus* chip, even though it was not highly optimized, showcases this flexibility.

In the second part of this thesis, two techniques for protecting data in the memory subsystem of a processor have been investigated. The first contribution in the memory subsystem domain, as discussed in Chapter 5, is our *open-source* framework for building transparent memory encryption and authentication pipelines. The building blocks of our framework are written in VHDL and, while being developed for FPGAs, are also suitable for ASIC designs. In addition to the building blocks, we provide example encryption/authentication pipelines that feature an AXI4 Interface and showcase the use of various ciphers (e.g., Prince, AES, Ascon) in different modes of operation (e.g., ECB, CBC, XTS, TEC tree). The evaluation of all configurations has been performed on a Xilinx Zynq-7020 SoC FPGA where the full memory traffic of the ARM processors running Linux gets transparently encrypted/authenticated. Our results show that the data processing of our encryption pipeline is highly efficient and utilizes up to 94 % of the achievable read bandwidth.

Finally, ScatterCache—a novel cache design that hardens CPU caches against timing attacks—has been presented in Chapter 6 of this thesis. To ease adoption, we aimed for building a cache that is as similar as possible to traditional caches and, at the same time, considerably harder to attack than commodity designs. In particular, in ScatterCache we retrofit skewed set-associative caches with a lightweight cryptographic primitive as keyed mapping function. This approach breaks the fixed cache-set congruences that are the cornerstone of Prime+Probe attacks and even enables ScatterCache-aware software to selectively control if memory shared in RAM should also be shared in cache. Attacks on ScatterCache are by construction probabilistic and require that

targeted memory accesses can be observed many times for both, the actual attack as well as the needed profiling. In terms of performance, our evaluation using the gem5 full system simulator as well as custom cache simulations show that performance is on par with traditional fixed-set designs with random replacement policy. Hence, ScatterCache is a viable drop-in replacement for traditional cache designs that considerably complicates practical attacks and that features an extension point on which further policies can be implemented in software.

Outlook

The approaches and techniques developed in this thesis notably advanced the field of protecting general-purpose processors against physical attacks. However, although our countermeasures are a solid foundation, we certainly did not manage to address all problems in this domain. The mapping of our techniques to the affected attack classes in Figure 1.1 already visualizes a few potential areas that can be explored in future work. In the following, we discuss some of these opportunities in addition to selected topics that complement this thesis.

Proactive Protection of Data and Addresses. In terms of processor core extensions, one of the most important topics for followup work is the protection of on-chip data and addresses. GPSA and SCFP exclusively protect code but the actual data—including the corresponding address information—within the processor, on busses, and in caches, is not yet protected by any of our hardware-supported countermeasures.

The current approach to deal with this deficiency is to utilize software-based redundancy approaches, *i.e.*, data encoding, re-computation, round trips. However, all these approaches have serious drawbacks when they are applied to general-purpose code. To complement our CFI techniques, future work on finding suitable hardware-supported data/address protection schemes, or generic software based-approaches, is needed.

Evaluation of newly added Side Channels. Another interesting avenue for further research is the evaluation of our techniques regarding newly introduced side-channels. In particular, most of our designs use cryptographic primitives as building blocks to achieve their respective goals. However, except for the RAM encryption, analysis of the leakage for these building blocks, is still an open research problem.

It would, for example, be interesting if power or EM side-channel leakage from the permutation within SCFP can effectively be exploited to recover the capacity of the sponge. Moreover, investigating how the choice of the cryptographic mode (e.g., keyed vs regular permutation, APE vs SpongeWrap) affects side-channel leakage would provide important insights regarding the best parameterization.

Development of Special-Purpose Cryptographic Primitives. Finding suitable new cryptographic primitives for implementing our approaches is another

challenge that should be tackled in future work. Currently, lightweight primitives like Ascon, Prince, and QARMA are used in most of our designs. However, more specialized primitives could greatly improve our techniques in terms of applicability, security, latency, and hardware costs.

Having access to a fast block cipher with configurable block size (e.g., between 64 and 128 bits) would, for example, permit instantiating AEE-Light in stronger configurations and with faster error detection on dense ISAs. ScatterCache would also profit from such a primitive (e.g., with 5 and 15 bits) and enable us, for example, to actually build SCv2.

Quantifying the Security of Caches against Real Attacks. Another very interesting opportunity for future work is to quantify the security gain of secure cache-architectures like ScatterCache against real-world attacks. Neither our current analytic models nor the available simulation-based approaches can reliably estimate the actual strength of a cache-architecture.

The used models are still comparably minimalist and consider only isolated and simplified settings. Full system simulation, on the other hand, is simply to slow to gather sufficient amounts of usable data. Moreover, building actual hardware is also not that simple and requires considerable amounts of resources. Considering the sheer number of recently proposed secure cache-architecture designs, finding an approach to systematically solve this challenge would be an important and valuable contribution.

Evaluating our Concepts on Larger Processors. As detailed in Chapter 1, physical attacks are not only a threat to IoT devices but can also be mounted on server and cloud hardware. Moreover, features like remote attestation are especially important in cloud scenarios since the hardware is under the physical control of an untrusted third party.

It would, therefore, be interesting to investigate if it is sensible to use techniques like SCFP on larger processors—possibly as part of a TEE like SGX. We expect, for example, that some current properties (e.g., decryption of individual instructions) have to be sacrificed to reach the required performance target. However, most likely an actual hardware design is needed to determine how far our techniques can be pushed and what compromises are needed.

Author's Publications

[Gro+16] Hannes Gross, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and **Mario Werner**. "Concealing Secrets in Embedded Processors Designs." In: *Smart Card Research and Advanced Applications – CARDIS*. 2016, pp. 89–104. DOI: 10.1007/978-3-319-54669-8_6.

[Kal+20] Daniel Kales, Sebastian Ramacher, Christian Rechberger, Roman Walch, and **Mario Werner**. "Efficient FPGA Implementations of LowMC and Picnic." In: *The Cryptographers' Track at the RSA Conference – CT-RSA*. 2020. URL: https://eprint.iacr.org/2019/1368.

[Sch+18b] Robert Schilling, **Mario Werner**, Pascal Nasahl, and Stefan Mangard. "Pointing in the Right Direction - Securing Memory Accesses in a Faulty World." In: *Annual Computer Security Applications Conference – ACSAC*. 2018, pp. 595–604. DOI: 10.1145/3274694.3274728.

[SWM18] Robert Schilling, **Mario Werner**, and Stefan Mangard. "Securing conditional branches in the presence of fault attacks." In: *Design, Automation & Test in Europe – DATE*. 2018, pp. 1586–1591. DOI: 10.23919/DATE.2018.8342268.

[UWM17a] Thomas Unterluggauer, **Mario Werner**, and Stefan Mangard. "Securing Memory Encryption and Authentication Against Side-Channel Attacks Using Unprotected Primitives." In: *Conference on Computer and Communications Security – CCS*. 2017, pp. 690–702. DOI: 10.1145/3052973.3052985.

[UWM17b] Thomas Unterluggauer, **Mario Werner**, and Stefan Mangard. "Side-channel plaintext-recovery attacks on leakage-resilient encryption." In: *Design, Automation & Test in Europe – DATE*. 2017, pp. 1318–1323. DOI: 10.23919/DATE.2017.7927197.

[UWM19] Thomas Unterluggauer, **Mario Werner**, and Stefan Mangard. "MEAS: memory encryption and authentication secure against side-channel attacks." In: *J. Cryptographic Engineering* 9 (2019), pp. 137–158. DOI: 10.1007/s13389-018-0180-2.

[Wei+19a] Samuel Weiser, **Mario Werner**, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V." In: *Network and Distributed System Security Symposium – NDSS*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/timber-v-tag-isolated-memory-bringing-fine-grained-enclaves-to-risc-v/.

[Wer+17] **Mario Werner**, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard. "Transparent memory encryption and authentication." In: *Field Programmable Logic and Applications – FPL*. 2017, pp. 1–6. DOI: 10.23919/FPL.2017.8056797.

[Wer+18] **Mario Werner**, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. "Sponge-Based Control-Flow Protection for IoT Devices." In: *European Symposium on Security and Privacy – EuroS&P*. **Best Paper Award**. 2018, pp. 214–226. DOI: 10.1109/EuroSP.2018.00023.

[Wer+19a] **Mario Werner**, Robert Schilling, Thomas Unterluggauer, and Stefan Mangard. "Protecting RISC-V Processors against Physical Attacks." In: *Design, Automation & Test in Europe – DATE*. 2019, pp. 1136–1141. DOI: 10.23919/DATE.2019.8714811.

[Wer+19b] **Mario Werner**, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization." In: *USENIX Security Symposium*. 2019, pp. 675–692. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/werner.

[WUW13] Erich Wenger, Thomas Unterluggauer, and **Mario Werner**. "8/16/32 Shades of Elliptic Curve Cryptography on Embedded Processors." In: *Progress in Cryptology – INDOCRYPT*. 2013, pp. 244–261. DOI: 10.1007/978-3-319-03515-4_16.

[WW11] Erich Wenger and **Mario Werner**. "Evaluating 16-Bit Processors for Elliptic Curve Cryptography." In: *Smart Card Research and Advanced Applications – CARDIS*. 2011, pp. 166–181. DOI: 10.1007/978-3-642-27257-8_11.

[WW17] Samuel Weiser and **Mario Werner**. "SGXIO: Generic Trusted I/O Path for Intel SGX." In: *Conference on Data and Application Security and Privacy – CODASPY*. 2017, pp. 261–268. DOI: 10.1145/3029806.3029822.

[WWM15] **Mario Werner**, Erich Wenger, and Stefan Mangard. "Protecting the Control Flow of Embedded Processors against Fault Attacks." In: *Smart Card Research and Advanced Applications – CARDIS*. 2015, pp. 161–176. DOI: 10.1007/978-3-319-31271-2_10.

Bibliography

[7cpa] 7-cpu. *ARM Cortex-A57*. URL: https://www.7-cpu.com/cpu/Cortex-A57.html (visited on 12/13/2019).

[7cpb] 7-cpu. *Intel Skylake*. URL: https://www.7-cpu.com/cpu/Skylake.html (visited on 12/13/2019).

[Aba+09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications." In: *ACM Trans. Inf. Syst. Secur.* 13 (2009), 4:1–4:40. DOI: 10.1145/1609956.1609960.

[Abe+16] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. "C-FLAT: Control-Flow Attestation for Embedded Systems Software." In: *Conference on Computer and Communications Security – CCS*. 2016, pp. 743–754. DOI: 10.1145/2976749.2978358.

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. "New Results on Instruction Cache Attacks." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2010, pp. 110–124. DOI: 10.1007/978-3-642-15031-9_8.

[AES15] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES." In: *IEEE Symposium on Security and Privacy – S&P*. 2015, pp. 591–604. DOI: 10.1109/SP.2015.42.

[Agr+02] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. "The EM Side-Channel(s)." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2002, pp. 29–45. DOI: 10.1007/3-540-36400-5_4.

[AJN16] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. *NORX v3*. Sept. 15, 2016. URL: https://norx.io/ (visited on 12/11/2019).

[AK97] Ross J. Anderson and Markus G. Kuhn. "Low Cost Attacks on Tamper Resistant Devices." In: *Security Protocols Workshop – SPW*. 1997, pp. 125–136. DOI: 10.1007/BFb0028165.

[And+14] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. "APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography." In: *Fast Software Encryption – FSE*. 2014, pp. 168–186. DOI: 10.1007/978-3-662-46706-0_9.

[And+15] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. "Security of Keyed Sponge Constructions Using a Modular Proof Approach." In: *Fast Software Encryption – FSE*. 2015, pp. 364–384. DOI: 10.1007/978-3-662-48116-5_18.

[And+16] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Florian Mendel, Bart Mennink, Nicky Mouha, Qingju Wang, and Kan Yasuda. *PRIMATEs v1.1*. July 13, 2016. URL: http://web.archive.org/web/20181228123124/http://primates.ae/ (visited on 12/11/2019).

[Ann16] Anna-senpai. *Source code for the Mirai botnet*. Sept. 30, 2016. URL: https://github.com/jgamblin/Mirai-Source-Code (visited on 01/07/2020).

[Ape+14] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a Minute! A fast, Cross-VM Attack on AES." In: *Recent Advances in Intrusion Detection – RAID*. 2014, pp. 299–319. DOI: 10.1007/978-3-319-11379-1_15.

[Ape+15] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Lucky 13 Strikes Back." In: *Conference on Computer and Communications Security – CCS*. 2015, pp. 85–96. DOI: 10.1145/2714576.2714625.

[ARM09] ARM. *ARM Security Technology Building a Secure System using TrustZone Technology*. ARM. 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[Aro+06] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors." In: *IEEE Trans. VLSI Syst.* 14 (2006), pp. 1295–1308. DOI: 10.1109/TVLSI.2006.887799.

[Arr+18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. "Rhythmic Keccak: SCA Security and Low Latency in HW." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018), pp. 269–290. DOI: 10.13154/tches.v2018.i1.269-290.

[Aum+02] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2002, pp. 260–275. DOI: 10.1007/3-540-36400-5_20.

[Ava17] Roberto Avanzi. "The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes." In: *IACR Trans. Symmetric Cryptol.* (2017), pp. 4–44. DOI: 10.13154/tosc.v2017.i1.4-44.

[BAP15] Scott Beamer, Krste Asanovic, and David A. Patterson. "The GAP Benchmark Suite." In: *arXiv abs/1508.03619* (2015). URL: http://arxiv.org/abs/1508.03619.

[Bar+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks." In: *ePrint 2004/100* (2004). URL: http://eprint.iacr.org/2004/100.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)." In: *Advances in Cryptology – EUROCRYPT*. 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4.

[Ber+08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "On the Indifferentiability of the Sponge Construction." In: *Advances in Cryptology – EUROCRYPT*. 2008, pp. 181–197. DOI: 10.1007/978-3-540-78967-3_11.

[Ber+11a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the sponge: single-pass authenticated encryption and other applications." In: *ePrint 2011/499* (2011). URL: http://eprint.iacr.org/2011/499.

[Ber+11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "On the security of the keyed sponge construction." In: *SKEW* (2011). URL: https://keccak.team/files/SpongeKeyed.pdf.

[Ber+12a] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Keccak implementation overview*. May 29, 2012. URL: http://keccak.noekeon.org/Keccak-implementation-3.2.pdf (visited on 12/13/2019).

[Ber+12b] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. "Permutation-based encryption, authentication and authenticated encryption." In: *Workshop Records of DIAC 2012*. 2012. URL: https://keccak.team/files/KeccakDIAC2012.pdf.

[Ber+16a] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: KETJE v2*. Sept. 15, 2016. URL: https://keccak.team/ketje.html (visited on 12/11/2019).

[Ber+16b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: KEYAK v2*. Sept. 15, 2016. URL: https://keccak.team/keyak.html (visited on 12/11/2019).

[Ber05] Daniel Julius Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. University of Illinois at Chicago, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[Ber19] Daniel Julius Bernstein. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. Feb. 20, 2019. URL: https://competitions.cr.yp.to/caesar-submissions.html (visited on 12/11/2019).

[BH09] Billy Bob Brumley and Risto M. Hakala. "Cache-Timing Template Attacks." In: *Advances in Cryptology – ASIACRYPT*. 2009, pp. 667–684. DOI: 10.1007/978-3-642-10366-7_39.

[Bin+11] Nathan L. Binkert et al. "The gem5 simulator." In: *SIGARCH Computer Architecture News* 39 (2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[Bio+18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX." In: *USENIX Security Symposium*. 2018, pp. 1213–1227. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

[Ble+11] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack." In: *Conference on Computer and Communications Security – CCS*. 2011, pp. 30–40. DOI: 10.1145/1966913.1966919.

[Bod+20] Rahul Bodduna, Vinod Ganesan, Patanjali SLPSK, Kamakoti Veezhinathan, and Chester Rebeiro. "Brutus: Refuting the Security Claims of the Cache Timing Randomization Countermeasure Proposed in CEASER." In: *IEEE Computer Architecture Letters* 19 (2020), pp. 9–12. DOI: 10.1109/LCA.2020.2964212.

[Bor+12a] Julia Borghoff et al. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract." In: *Advances in Cryptology – ASIACRYPT*. 2012, pp. 208–225. DOI: 10.1007/978-3-642-34961-4_14.

[Bor+12b] Julia Borghoff et al. "PRINCE - A Low-latency Block Cipher for Pervasive Computing Applications (Full version)." In: *ePrint 2012/529* (2012). URL: http://eprint.iacr.org/2012/529.

[Bul+18] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium*. 2018, pp. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[Car+15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity." In: *USENIX Security Symposium*. 2015, pp. 161–176. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[CC11] Liang Cai and Hao Chen. "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion." In: *USENIX Security Symposium*. 2011. URL: https://www.usenix.org/conference/hotsec11/touchlogger-inferring-keystrokes-touch-screen-smartphone-motion.

[Cel+19] Christopher Celio, Pi-Feng Chiu, Krste Asanovic, Borivoje Nikolic, and David A. Patterson. "BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS." In: *IEEE Micro* 39 (2019), pp. 52–60. DOI: 10.1109/MM.2019.2897782.

[CIS16] Yahoo CISO. *An Important Message About Yahoo User Security*. Sept. 22, 2016. URL: https://yahoo.tumblr.com/post/150781911849/an-important-message-about-yahoo-user-security (visited on 01/07/2020).

[Cle+16] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen De Bosschere, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. "SOFIA: Software and control flow integrity architecture." In: *Design, Automation & Test in Europe – DATE*. 2016, pp. 1172–1177. URL: http://ieeexplore.ieee.org/document/7459489/.

[Cle+17a] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. "Protecting Bare-Metal Embedded Systems with Privilege Overlays." In: *IEEE Symposium on Security and Privacy – S&P*. 2017, pp. 289–303. DOI: 10.1109/SP.2017.37.

[Cle+17b] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. "SOFIA: Software and control flow integrity architecture." In: *Computers & Security* 68 (2017), pp. 16–35. DOI: 10.1016/j.cose.2017.03.013.

[Cle+17c] Ruan de Clercq, Ronald De Keulenaer, Pieter Maene, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. "SCM: Secure Code Memory Architecture." In: *Conference on Computer and Communications Security – CCS*. 2017, pp. 771–776. DOI: 10.1145/3052973.3053044.

[Coj+19] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks." In: *IEEE Symposium on Security and Privacy – S&P*. 2019, pp. 55–71. DOI: 10.1109/SP.2019.00089.

[Cop+09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors." In: *IEEE Symposium on Security and Privacy – S&P*. 2009, pp. 45–60. DOI: 10.1109/SP.2009.19.

[Cor11] Sony Corporation. *Sony Online Entertainment announces theft of data from its systems*. May 3, 2011. URL: https://www.sony.net/SonyInfo/News/Press/201105/11-0503E/index.html (visited on 01/07/2020).

[Cor13] MITRE Corporation. *CVE-2014-0160, also known as Heartbleed*. Dec. 3, 2013. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (visited on 01/07/2020).

[Cor14] MITRE Corporation. *CVE-2014-6271, also known as Shellshock*. Sept. 9, 2014. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271 (visited on 01/07/2020).

[Cor16] MITRE Corporation. *CVE-2016-5195, also known as Dirty COW*. May 31, 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195 (visited on 01/07/2020).

[Cow98] Crispan Cowan. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In: *USENIX Security Symposium*. 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[Dav+14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection." In: *USENIX Security Symposium*. 2014, pp. 401–416. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[Des+17] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware." In: *Design Automation Conference – DAC*. 2017, 24:1–24:6. DOI: 10.1145/3061639.3062276.

[Det] CVE Details. *Vulnerabilities per Android Version*. URL: http://www.cvedetails.com/version-list/1224/19997/1/Google-Android.html (visited on 01/07/2020).

[Dey+14] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. "AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable." In: *Network and Distributed System Security Symposium – NDSS*. 2014. URL: https://www.ndss-symposium.org/ndss2014/accelprint-imperfections-accelerometers-make-smartphones-trackable.

[DFS19] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments." In: *arXiv abs/1909.09599* (2019). URL: http://arxiv.org/abs/1909.09599.

[DK06] Guillaume Duc and Ronan Keryell. "CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection." In: *Annual Computer Security Applications Conference – ACSAC*. 2006, pp. 483–492. DOI: 10.1109/ACSAC.2006.21.

[DK17] Goran Doychev and Boris Köpf. "Rigorous analysis of software countermeasures against cache attacks." In: *Programming Language Design and Implementation – PLDI*. 2017, pp. 406–421. DOI: 10.1145/3062341.3062388.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *ASCON v1.2*. Sept. 15, 2016. URL: https://ascon.iaik.tugraz.at/ (visited on 12/11/2019).

[Doy+13] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. "CacheAudit: A Tool for the Static Analysis of Cache Side Channels." In: *USENIX Security Symposium*. 2013, pp. 431–446. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev.

[DXS19] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. "Analysis of Secure Caches and Timing-Based Side-Channel Attacks." In: *ePrint 2019/167* (2019). URL: https://eprint.iacr.org/2019/167.

[Elb+07] Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. "TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2007, pp. 289–302. DOI: 10.1007/978-3-540-74735-2_20.

[Elb+09] Reouven Elbaz, David Champagne, Catherine H. Gebotys, Ruby B. Lee, Nachiketh R. Potlapally, and Lionel Torres. "Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines." In: *Trans. Computational Science* 4 (2009), pp. 1–22. DOI: 10.1007/978-3-642-01004-0_1.

[Eld+12] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." In: *Network and Distributed System Security Symposium – NDSS*. 2012. URL: https://www.ndss-symposium.org/ndss2012/smart-secure-and-minimal-architecture-establishing-dynamic-root-trust.

[ETH17a] ETH Zurich. *PULPino Source Repository*. 2017. URL: https://github.com/pulp-platform/pulpino (visited on 12/11/2019).

[ETH17b] ETH Zurich. *RI5CY Source Repository*. 2017. URL: https://github.com/pulp-platform/riscv (visited on 12/11/2019).

[ETH18] ETH Zurich. *PULPissimo Source Repository*. 2018. URL: https://github.com/pulp-platform/pulpissimo (visited on 12/11/2019).

[Eur19] Europractice Web Archive. *TSMC Standard cell libraries*. Apr. 1, 2019. URL: http://web.archive.org/web/20190401134822/http://www.europractice-ic.com/libraries_TSMC.php (visited on 12/13/2019).

[Fac] Facebook. *Bug Bounty Program*. URL: https://www.facebook.com/whitehat (visited on 01/07/2020).

[fai14] fail0verflow. *Console Hacking 2013: Omake*. Jan. 2, 2014. URL: https://fail0verflow.com/blog/2014/console-hacking-2013-omake/ (visited on 01/07/2020).

[fai15] fail0verflow. *Console Hacking 2015: Liner Notes*. Dec. 30, 2015. URL: https://fail0verflow.com/blog/2015/console-hacking-2015-liner-notes/ (visited on 01/07/2020).

[FPC09] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. "Defending Embedded Systems Against Control Flow Attacks." In: *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*. ACM, 2009, pp. 19–26. DOI: 10.1145/1655077.1655083.

[Fra+14] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. "A minimalist approach to Remote Attestation." In: *Design, Automation & Test in Europe – DATE*. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.257.

[fre] free60project. *The Xbox 360 reset glitch hack*. URL: https://free60project.github.io/wiki/Reset_Glitch_Hack.html (visited on 01/07/2020).

[Fru05] Clemens Fruhwirth. *New methods in hard disk encryption*. Tech. rep. Vienna University of Technology, 2005. URL: http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf.

[Gal+19] Mark Gallagher et al. "Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS*. 2019, pp. 469–484. DOI: 10.1145/3297858.3304037.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games - Bringing Access-Based Cache Attacks on AES to Practice." In: *IEEE Symposium on Security and Privacy – S&P*. 2011, pp. 490–505. DOI: 10.1109/SP.2011.22.

[Gli+15] Danilo Gligoroski, Hristina Mihajloska, Simona Samardjiska, Håkon Jacobsen, Mohamed El-Hadedy, Rune Erlend Jensen, and Daniel Otte. *π–Cipher v2.01*. Oct. 12, 2015. URL: http://pi-cipher.org/ (visited on 12/11/2019).

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. 2016, pp. 300–321. DOI: 10.1007/978-3-319-40667-1_15.

[Gök+14] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. "Out of Control: Overcoming Control-Flow Integrity." In: *IEEE Symposium on Security and Privacy – S&P*. 2014, pp. 575–589. DOI: 10.1109/SP.2014.43.

[Gol+06] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. ISBN: 978-0-387-26060-0. DOI: 10.1007/0-387-32937-4.

[Gooa] Google. *Distribution of Android platform versions*. URL: https://developer.android.com/about/dashboards/index.html#Platform (visited on 01/07/2020).

[Goob] Google. *Vulnerability Reward Program*. URL: https://www.google.com/about/appsecurity/reward-program/ (visited on 01/07/2020).

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)." In: *Cryptographic Hardware and Embedded Systems – CHES*. 1999, pp. 158–172. DOI: 10.1007/3-540-48059-5_15.

[Gra+17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *Network and Distributed System Security Symposium – NDSS*. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/aslrcache-practical-cache-attacks-mmu/.

[Gru+16a] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *Conference on Computer and Communications Security – CCS*. 2016, pp. 368–379. DOI: 10.1145/2976749.2978356.

[Gru+16b] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. 2016, pp. 279–299. DOI: 10.1007/978-3-319-40667-1_14.

[Gru+18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses." In: *IEEE Symposium on Security and Privacy – S&P*. 2018, pp. 245–261. DOI: 10.1109/SP.2018.00031.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015, pp. 897–912. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[Gue16] Shay Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors." In: *ePrint 2016/204* (2016). URL: http://eprint.iacr.org/2016/204.

[Gül+15] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES." In: *Constructive Side-Channel Analysis and Secure Design – COSADE*. 2015, pp. 111–126. DOI: 10.1007/978-3-319-21476-4_8.

[Gut+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. "MiBench: A free, commercially representative embedded benchmark suite." In: *Workshop on Workload Characterization – WWC*. 2001. DOI: 10.1109/WWC.2001.990739.

[Hal+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. "Lest We Remember: Cold Boot Attacks on Encryption Keys." In: *USENIX Security Symposium*. 2008, pp. 45–60. URL: http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf.

[Han+12] Jun Han, Emmanuel Owusu, Le T. Nguyen, Adrian Perrig, and Joy Zhang. "ACComplice: Location inference using accelerometers on smartphones." In: *International Conference on Communication Systems and Networks – COMSNETS*. 2012, pp. 1–9. DOI: 10.1109/COMSNETS.2012.6151305.

[Hoe+13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. "Using innovative instructions to create trustworthy software solutions." In: *International Symposium on Computer Architecture – ISCA*. 2013, p. 11. DOI: 10.1145/2487726.2488370.

[Hu+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks." In: *IEEE*

Symposium on Security and Privacy – S&P. 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *IEEE Symposium on Security and Privacy – S&P*. 2013, pp. 191–205. DOI: 10.1109/SP.2013.23.

[IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross Processor Cache Attacks." In: *Conference on Computer and Communications Security – CCS*. 2016, pp. 353–364. DOI: 10.1145/2897845.2897867.

[Inc+16] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2016, pp. 368–388. DOI: 10.1007/978-3-662-53140-2_18.

[Inc16] Dyn Inc. *Update Regarding DDoS Event Against Dyn Managed DNS on October 21, 2016*. Oct. 29, 2016. URL: https://www.dynstatus.com/incidents/5r9mppc1kb77 (visited on 01/07/2020).

[Int] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool (visited on 12/13/2019).

[Ira+17] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. "Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries." In: *arXiv abs/1709.01552* (2017). URL: http://arxiv.org/abs/1709.01552.

[Jan+17] Yeongjin Jang, Jae-Hyuk Lee, Sangho Lee, and Taesoo Kim. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack." In: *Workshop on System Software for Trusted Execution – SysTEX@SOSP*. 2017, 5:1–5:6. DOI: 10.1145/3152701.3152709.

[JLK16] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX." In: *Conference on Computer and Communications Security – CCS*. 2016, pp. 380–392. DOI: 10.1145/2976749.2978321.

[JT12] Marc Joye and Michael Tunstall, eds. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012. ISBN: 978-3-642-29655-0. DOI: 10.1007/978-3-642-29656-7.

[Ken+19] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. "V0LTpwn: Attacking x86 Processor Integrity from Software." In: *arXiv abs/1912.04870* (2019). URL: http://arxiv.org/abs/1912.04870.

[KH14] Thomas Korak and Michael Hoefler. “On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms.” In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2014, pp. 8–17. DOI: 10.1109/FDTC.2014.11.

[Kim+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.” In: *International Symposium on Computer Architecture – ISCA*. 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis.” In: *Advances in Cryptology – CRYPTO*. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[KMO12] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. “Automatic Quantification of Cache Side-Channels.” In: *Computer Aided Verification – CAV*. 2012, pp. 564–580. DOI: 10.1007/978-3-642-31424-7_40.

[KNQ15] Dae-Hyun Kim, Prashant J. Nair, and Moinuddin K. Qureshi. “Architectural Support for Mitigating Row Hammering in DRAM Memories.” In: *Computer Architecture Letters* 14 (2015), pp. 9–12. DOI: 10.1109/LCA.2014.2332177.

[Koc+19] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution.” In: *IEEE Symposium on Security and Privacy – S&P*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[Koc96] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.” In: *Advances in Cryptology – CRYPTO*. 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[Kön08] Robert Könighofer. “A Fast and Cache-Timing Resistant Implementation of the AES.” In: *Topics in Cryptology – CT-RSA*. 2008, pp. 187–202. DOI: 10.1007/978-3-540-79263-5_12.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*. Apr. 21, 2016. URL: http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf (visited on 12/13/2019).

[KS09] Emilia Käsper and Peter Schwabe. “Faster and Timing-Attack Resistant AES-GCM.” In: *Cryptographic Hardware and Embedded Systems – CHES*. 2009, pp. 1–17. DOI: 10.1007/978-3-642-04138-9_1.

[KSV13] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. “Hardware Designer’s Guide to Fault Attacks.” In: *IEEE Trans. VLSI Syst.* 21 (2013), pp. 2295–2306. DOI: 10.1109/TVLSI.2012.2231707.

[Kuz+14] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity." In: *USENIX Symposium on Operating Systems Design and Implementation – OSDI*. 2014, pp. 147–163. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

[Lee+17] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves." In: *USENIX Security Symposium*. 2017, pp. 523–539. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. "Software Countermeasures for Control Flow Integrity of Smart Card C Codes." In: *European Symposium on Research in Computer Security – ESORICS*. 2014, pp. 200–218. DOI: 10.1007/978-3-319-11212-1_12.

[Lip+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. 2016, pp. 549–564. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[Lip+18a] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. "Nethammer: Inducing Rowhammer Faults through Network Requests." In: *arXiv abs/1805.04956* (2018). URL: http://arxiv.org/abs/1805.04956.

[Lip+18b] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium*. 2018, pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[Liu+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *IEEE Symposium on Security and Privacy – S&P*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[Lou+19] Xiaoxuan Lou, Fan Zhang, Zheng Leong Chua, Zhenkai Liang, Yueqiang Cheng, and Yajin Zhou. "Understanding Rowhammer Attacks through the Lens of a Unified Reference Framework." In: *arXiv abs/1901.03538* (2019). URL: http://arxiv.org/abs/1901.03538.

[Mae+18] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid Verbauwhede. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation." In: *IEEE Trans. Computers* 67 (2018), pp. 361–374. DOI: 10.1109/TC.2017.2647955.

[Mau+15a] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. 2015, pp. 46–64. DOI: 10.1007/978-3-319-20550-2_3.

[Mau+15b] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters." In: *Recent Advances in Intrusion Detection – RAID*. 2015, pp. 48–65. DOI: 10.1007/978-3-319-26362-5_3.

[Mau+17] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *Network and Distributed System Security Symposium – NDSS*. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/hello-other-side-ssh-over-robust-cache-covert-channels-cloud/.

[McK+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative instructions and software model for isolated execution." In: *International Symposium on Computer Architecture – ISCA*. 2013, p. 10. DOI: 10.1145/2487726.2488368.

[Mic] Microsoft. *Bug Bounty Program*. URL: https://www.microsoft.com/en-us/msrc/bounty (visited on 01/07/2020).

[MMS01] David May, Henk L. Muller, and Nigel P. Smart. "Random Register Renaming to Foil DPA." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2001, pp. 28–38. DOI: 10.1007/3-540-44709-1_4.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9.

[Mor+15] Pawel Morawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wojcik. *ICEPOLE v2*. Aug. 24, 2015. URL: https://competitions.cr.yp.to/round2/icepolev2.pdf (visited on 12/11/2019).

[MRV15] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. "Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption." In: *Advances in Cryptology – ASIACRYPT*. 2015, pp. 465–489. DOI: 10.1007/978-3-662-48800-3_19.

[MS96] Larry W. McVoy and Carl Staelin. "lmbench: Portable Tools for Performance Analysis." In: *USENIX Annual Technical Conference*. 1996, pp. 279–294.

[Mur+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX." In: *IEEE Symposium on Security and Privacy – S&P*. 2020. URL: https://www.plundervolt.com/doc/plundervolt.pdf.

[MWK17] Heiko Mantel, Alexandra Weber, and Boris Köpf. "A Systematic Study of Cache Side Channels Across AES Implementations." In: *Engineering Secure Software and Systems – ESSoS*. 2017, pp. 213–230. DOI: 10.1007/978-3-319-62105-0_14.

[Nag+09] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. "SoftBound: highly compatible and complete spatial memory safety for c." In: *Programming Language Design and Implementation – PLDI*. 2009, pp. 245–258. DOI: 10.1145/1542476.1542504.

[Nag+10] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. "CETS: compiler enforced temporal safety for C." In: *International Symposium on Memory Management – ISMM*. 2010, pp. 31–40. DOI: 10.1145/1806651.1806657.

[Nam82] Masood Namjoo. "Techniques for Concurrent Testing of VLSI Processor Operation." In: *International Test Conference – ITC*. 1982, pp. 461–468.

[Ner01] Nergal. *The advanced return-into-lib(c) exploits: PaX case study*. Dec. 28, 2001. URL: http://phrack.org/issues/58/4.html (visited on 01/07/2020).

[Noo+13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base." In: *USENIX Security Symposium*. 2013, pp. 479–494. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman.

[Off18] United States Government Accountability Office. *Actions Taken by Equifax and Federal Agencies in Response to the 2017 Breach*. Aug. 2018. URL: https://www.warren.senate.gov/imo/media/doc/2018.09.06%20GAO%20Equifax%20report.pdf (visited on 01/28/2020).

[Ole+17] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches." In: *arXiv abs/1702.00719* (2017). URL: http://arxiv.org/abs/1702.00719.

[Ore+15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *Conference on Computer and Communications Security – CCS*. 2015, pp. 1406–1418. DOI: 10.1145/2810103.2813708.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES." In: *Topics in Cryptology – CT-RSA*. 2006, pp. 1–20. DOI: 10.1007/11605805_1.

[Pag02] Dan Page. "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel." In: *ePrint 2002/169* (2002). URL: http://eprint.iacr.org/2002/169.

[Pag05] Dan Page. "Partitioned Cache Architecture as a Side-Channel Defence Mechanism." In: *ePrint 2005/280* (2005). URL: http://eprint.iacr.org/2005/280.

[PaX01] PaX Team. *PaX Address Space Layout Randomization (ASLR)*. 2001. URL: http://pax.grsecurity.net/docs/aslr.txt (visited on 12/11/2019).

[Per05] Colin Percival. "Cache missing for fun and profit." In: *BSDCan*. 2005. URL: https://papers.freebsd.org/2005/cperciva-cache_missing.files/cperciva-cache_missing-paper.pdf.

[Pic14] Sony Pictures. *Letter to Employees*. Nov. 24, 2014. URL: http://web.archive.org/web/20150921225520/http://www.sonypictures.com/corp/notification/current/Non-US_and_Non-Canada_11515.pdf (visited on 01/07/2020).

[PV19] Antoon Purnal and Ingrid Verbauwhede. "Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache." In: *arXiv abs/1908.03383* (2019). URL: http://arxiv.org/abs/1908.03383.

[Qiu+19a] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies." In: *Conference on Computer and Communications Security – CCS*. 2019, pp. 195–209. DOI: 10.1145/3319535.3354201.

[Qiu+19b] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults." In: *Asian Hardware Oriented Security and Trust Symposium – AsianHOST*. 2019. URL: https://voltjockey.com/flies/paper/2.pdf.

[Qur+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. "Adaptive insertion policies for high performance caching." In: *International Symposium on Computer Architecture – ISCA*. 2007, pp. 381–391. DOI: 10.1145/1250662.1250709.

[Qur18] Moinuddin K. Qureshi. "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping." In: *IEEE/ACM International Symposium on Microarchitecture – MICRO*. 2018, pp. 775–787. DOI: 10.1109/MICRO.2018.00068.

[Qur19] Moinuddin K. Qureshi. "New attacks and defense for encrypted-address cache." In: *International Symposium on Computer Architecture – ISCA*. 2019, pp. 360–371. DOI: 10.1145/3307650.3322246.

[Raj+09] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. "Resource management for isolation enhanced cloud services." In: *Cloud Computing Security Workshop – CCSW*. 2009, pp. 77–84. DOI: 10.1145/1655008.1655019.

[RCS02] Francisco Rodríguez, José Carlos Campelo, and Juan José Serrano. "A Watchdog Processor Architecture with Minimal Performance Overhead." In: *Computer Safety, Reliability and Security – SAFECOMP*. 2002, pp. 261–272. DOI: 10.1007/3-540-45732-1_26.

[Ris+09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds." In: *Conference on Computer and Communications Security – CCS*. 2009, pp. 199–212. DOI: 10.1145/1653662.1653687.

[Rog+07] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly." In: *IEEE/ACM International Symposium on Microarchitecture – MICRO*. 2007, pp. 183–196. DOI: 10.1109/MICRO.2007.16.

[Rog04] Phillip Rogaway. "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC." In: *Advances in Cryptology – ASIACRYPT*. 2004, pp. 16–31. DOI: 10.1007/978-3-540-30539-2_2.

[Roo19] Marriott News Room. *Marriott Provides Update on Starwood Database Security Incident*. Jan. 4, 2019. URL: https://news.marriott.com/2019/01/marriott-provides-update-on-starwood-database-security-incident/ (visited on 01/28/2020).

[RS05] Francisco Rodríguez and Juan José Serrano. "Control Flow Error Checking with ISIS." In: *Embedded Software and Systems – ICESS*. 2005, pp. 659–670. DOI: 10.1007/11599555_63.

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. "Bitslice Implementation of AES." In: *Cryptology and Network Security – CANS*. 2006, pp. 203–212. DOI: 10.1007/11935070_14.

[SA02] Sergei P. Skorobogatov and Ross J. Anderson. "Optical Fault Induction Attacks." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2002, pp. 2–12. DOI: 10.1007/3-540-36400-5_2.

[Saa16] Markku-Juhani O. Saarinen. *TinySHA3 Source Repository*. 2016. URL: https://github.com/mjosaarinen/tiny_sha3 (visited on 12/11/2019).

[Sam+02] David Samyde, Sergei P. Skorobogatov, Ross J. Anderson, and Jean-Jacques Quisquater. "On a New Way to Read Data from Memory." In: *Security in Storage Workshop – SISW*. 2002, pp. 65–69. DOI: 10.1109/SISW.2002.1183512.

[San15] Lester Sanders. *XAPP1175: Secure Boot of Zynq-7000 All Programmable SoC*. v2.0. Xilinx Inc. Apr. 3, 2015. URL: https://www.xilinx.com/support/documentation/application_notes/xapp1175_zynq_secure_boot.pdf.

[SB15] Markku-Juhani O. Saarinen and Billy B. Brumley. *STRIBOBr2: "WHIRLBOB"*. Aug. 28, 2015. URL: https://competitions.cr.yp.to/round2/stribobr2.pdf (visited on 12/11/2019).

[Sch+10] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. "ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software." In: *Computer Safety, Reliability and Security – SAFECOMP*. 2010, pp. 169–182. DOI: 10.1007/978-3-642-15651-9_13.

[Sch+15] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *IEEE Symposium on Security and Privacy – S&P*. 2015, pp. 745–762. DOI: 10.1109/SP.2015.51.

[Sch+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. 2017, pp. 3–24. DOI: 10.1007/978-3-319-60876-1_1.

[Sch+18a] David Schaffenrath, Markus Wegmann, Antonio Pullini, Davide Schiavone, Beat Muheim, Stefan Mangard, and Mario Werner. *The IIS Chip Gallery: Patronus*. Apr. 3, 2018. URL: http://asic.ethz.ch/2016/Patronus.html (visited on 02/07/2020).

[Sch+18c] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In: *Network and Distributed System Security Symposium – NDSS*. 2018. URL: http://arxiv.org/abs/1706.06381.

[Sch+18d] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *arXiv abs/1807.10535* (2018). URL: http://arxiv.org/abs/1807.10535.

[SD15] M. Seaborn and T. Dullien. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. Mar. 9, 2015. URL: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (visited on 01/30/2020).

[Sel+15] Bodo Selmke, Stefan Brummer, Johann Heyszl, and Georg Sigl. "Precise Laser Fault Injections into 90 nm and 45 nm SRAM-cells." In: *Smart Card Research and Advanced Applications – CARDIS*. 2015, pp. 193–205. DOI: 10.1007/978-3-319-31271-2_12.

[Sez93] André Seznec. "A Case for Two-Way Skewed-Associative Caches." In: *International Symposium on Computer Architecture – ISCA*. 1993, pp. 169–178. DOI: 10.1145/165123.165152.

[Sha+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the effectiveness of address-space randomization." In: *Conference on Computer and Communications Security – CCS*. 2004, pp. 298–307. DOI: 10.1145/1030083.1030124.

[Sha07] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *Conference on Computer and Communications Security – CCS*. 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[Sil12] Vicente Silveira. *An Update on LinkedIn Member Passwords Compromised*. June 6, 2012. URL: https://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised (visited on 01/07/2020).

[SKH05] Mathias Spjuth, Martin Karlsson, and Erik Hagersten. "Skewed caches from a low-power perspective." In: *Computing Frontiers – CF*. 2005, pp. 152–160. DOI: 10.1145/1062261.1062289.

[Sko02] Sergei Skorobogatov. *Low temperature data remanence in static RAM*. Tech. rep. University of Cambridge, 2002. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf.

[Sko16] Sergei Skorobogatov. "The bumpy road towards iPhone 5c NAND mirroring." In: *arXiv abs/1609.04327* (2016). URL: http://arxiv.org/abs/1609.04327.

[SOD05] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. "AEGIS: A single-chip secure processor." In: *Inf. Sec. Techn. Report* 10 (2005), pp. 63–73. DOI: 10.1016/j.istr.2005.05.002.

[Sol97] Solar Designer. *Getting around non-executable stack (and fix)*. Aug. 10, 1997. URL: http://seclists.org/bugtraq/1997/Aug/63 (visited on 01/07/2020).

[SP13] Raphael Spreitzer and Thomas Plos. "Cache-Access Pattern Attack on Disaligned AES T-Tables." In: *Constructive Side-Channel Analysis and Secure Design – COSADE*. 2013, pp. 200–214. DOI: 10.1007/978-3-642-40026-1_13.

[Spr+18] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. "Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices." In: *IEEE Communications Surveys and Tutorials* 20 (2018), pp. 465–488. DOI: 10.1109/COMST.2017.2779824.

[Spr14] Raphael Spreitzer. "PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices." In: *Conference on Computer and Communications Security – CCS*. 2014, pp. 51–62. DOI: 10.1145/2666620.2666622.

[Sta] Standard Performance Evaluation Corporation. *SPEC CPU 2017*. URL: https://www.spec.org/cpu2017/ (visited on 12/13/2019).

[Sta10] François-Xavier Standaert. "Introduction to Side-Channel Attacks." In: *Secure Integrated Circuits and Systems*. 2010. DOI: 10.1007/978-0-387-71829-3_2.

[Sto08] SIS-WG - Security in Storage Working Group. *IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. IEEE Std 1619-2007*. 2008.

[Sun+18] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. "OEI: Operation Execution Integrity for Embedded Devices." In: *arXiv abs/1802.03462* (2018). URL: http://arxiv.org/abs/1802.03462.

[SWG19] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*. 2019, pp. 177–196. DOI: 10.1007/978-3-030-22038-9_9.

[SY15] Yu Sasaki and Kan Yasuda. "How to Incorporate Associated Data in Sponge-Based Authenticated Encryption." In: *Topics in Cryptology – CT-RSA*. 2015, pp. 353–370. DOI: 10.1007/978-3-319-16715-2_19.

[Sze+13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory." In: *IEEE Symposium on Security and Privacy – S&P*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[Tat+18] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses." In: *USENIX Annual Technical Conference*. 2018, pp. 213–226. URL: https://www.usenix.org/conference/atc18/presentation/tatar.

[Tri+18] David Trilla, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. "Cache side-channel attacks and time-predictability in high-performance critical real-time systems." In: *Design Automation Conference – DAC*. 2018, 98:1–98:6. DOI: 10.1145/3195970.3196003.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management." In: *USENIX Security Symposium*. 2017, pp. 1057–1074. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[Tsu+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. "Cryptanalysis of DES Implemented on Computers with Cache." In: *Cryptographic Hardware and Embedded Systems – CHES*. 2003, pp. 62–76. DOI: 10.1007/978-3-540-45238-6_6.

[WA17] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Tech. rep. EECS Department, University of California, Berkeley, May 7, 2017. URL: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[Wat+16] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1*. Tech. rep. EECS Department, University of California, Berkeley, Nov. 4, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.pdf.

[Wei+18] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries." In: *USENIX Security Symposium*. 2018, pp. 603–620. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser.

[Wei+19b] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. "JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms." In: *arXiv abs/1912.11523* (2019). URL: http://arxiv.org/abs/1912.11523.

[Wei+94] Mark Weiser, Brent B. Welch, Alan J. Demers, and Scott Shenker. "Scheduling for Reduced CPU Energy." In: *USENIX Symposium on Operating Systems Design and Implementation – OSDI*. 1994, pp. 13–23. URL: http://dl.acm.org/citation.cfm?id=1267640.

[Wik] Wikipedia. *iOS jailbreaking*. URL: https://en.wikipedia.org/wiki/IOS_jailbreaking#History_of_tools (visited on 01/07/2020).

[Wil15] Kyle Wilkinson. *XAPP1239: Using Encryption to Secure a 7 Series FPGA Bitstream*. v1.0. Xilinx Inc. Apr. 15, 2015. URL: https://www.xilinx.com/support/documentation/application_notes/xapp1239-fpga-bitstream-encryption.pdf.

[WL07] Zhenghong Wang and Ruby B. Lee. "New cache designs for thwarting software cache-based side channel attacks." In: *International Symposium on Computer Architecture – ISCA*. 2007, pp. 494–505. DOI: 10.1145/1250662.1250723.

[WL08] Zhenghong Wang and Ruby B. Lee. "A novel cache architecture with enhanced performance and security." In: *IEEE/ACM International Symposium on Microarchitecture – MICRO*. 2008, pp. 83–93. DOI: 10.1109/MICRO.2008.4771781.

[WS88] Kent D. Wilken and John Paul Shen. "Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors." In: *International Test Conference – ITC*. 1988, pp. 914–925. DOI: 10.1109/TEST.1988.207880.

[WS90] Kent D. Wilken and John Paul Shen. "Continuous signature monitoring: low-cost concurrent detection of processor control errors." In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 9 (1990), pp. 629–641. DOI: 10.1109/43.55193.

[WWM11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. "Practical Optical Fault Injection on Secure Microcontrollers." In: *Fault Diagnosis and Tolerance in Cryptography – FDTC*. 2011, pp. 91–99. DOI: 10.1109/FDTC.2011.12.

[WXW12] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud." In: *USENIX Security Symposium*. 2012, pp. 159–173. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu.

[WXW15] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud." In: *IEEE/ACM Trans. Netw.* 23 (2015), pp. 603–615. DOI: 10.1109/TNET.2014.2304439.

[Xia+17] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. "STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves." In: *Conference on Computer and Communications Security – CCS*. 2017, pp. 859–874. DOI: 10.1145/3133956.3134016.

[Xu+11] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh R. Joshi, Matti A. Hiltunen, and Richard D. Schlichting. "An exploration of L2 cache covert channels in virtualized environments." In: *Cloud Computing Security Workshop – CCSW*. 2011, pp. 29–40. DOI: 10.1145/2046660.2046670.

[Yan+06] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. "Improving Cost, Performance, and Security of Memory Encryption and Authentication." In: *International Symposium on Computer Architecture – ISCA*. 2006, pp. 179–190. DOI: 10.1109/ISCA.2006.22.

[YF14] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014, pp. 719–732. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[Zei+17] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. "ATRIUM: Runtime attestation resilient under memory attacks." In: *Conference on Computer-Aided Design – ICCAD*. 2017, pp. 384–391. DOI: 10.1109/ICCAD.2017.8203803.

[Zha+11] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis." In: *IEEE Symposium on Security and Privacy – S&P*. 2011, pp. 313–328. DOI: 10.1109/SP.2011.31.

[Zha+12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys." In: *Conference on Computer and Communications Security – CCS*. 2012, pp. 305–316. DOI: 10.1145/2382196.2382230.

[Zha+14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds." In: *Conference on Computer and Communications Security – CCS*. 2014, pp. 990–1003. DOI: 10.1145/2660267.2660356.

[ZHS16] Andreas Zankl, Johann Heyszl, and Georg Sigl. "Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software." In: *Smart Card Research and Advanced Applications – CARDIS*. 2016, pp. 228–244. DOI: 10.1007/978-3-319-54669-8_14.

[ZS13] Mingwei Zhang and R. Sekar. "Control Flow Integrity for COTS Binaries." In: *USENIX Security Symposium*. 2013, pp. 337–352. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.

[ZXZ16] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. "Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices." In: *Conference on Computer and Communications Security – CCS*. 2016, pp. 858–870. DOI: 10.1145/2976749.2978360.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.