Samuel Weiser

Enclave Security and Address-based Side Channels

DOCTORAL THESIS

to achieve the university degree of
Doctor of Technical Sciences; Dr. techn.

submitted to

Graz University of Technology

Assessors

Prof. Stefan Mangard
Institute of Applied Information Processing and Communications
Graz University of Technology

Prof. Thomas Eisenbarth
Institute for IT Security
Universität zu Lübeck

Graz, June 2020

✳ ✳ ✳

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.

Date, Signature

✳ ✳ ✳

Prologue

Everyone has the right to life, liberty and security of person.
Universal Declaration of Human Rights, Article 3

Our life turned digital, and so did we. Not long ago, the globalized communication that we enjoy today on an everyday basis was the privilege of a few. Nowadays, artificial intelligence in the cloud, smartified handhelds, low-power Internet-of-Things gadgets, and self-maneuvering objects in the physical world are promising us unthinkable freedom in shaping our personal lives as well as society as a whole. Sadly, our collective excitement about the "new", the "better", the "more", the "instant", has overruled our sense of security and privacy. New features, better design, more resources and instant satisfaction make us happily pay for it with our own data. To phrase it differently: technology sells, security does not. But what is our digital society worth if it not only fails to protect our privacy but seeks to intrude our most intimate moments? "Study after study has show [*sic*] that human behavior changes when we know we're being watched. Under observation, we act less free, which means we effectively *are* less free." [Sno19].

Providing a solid basis for a privacy-friendly future is and should be core motivation of security research. Unfortunately, we live in a highly fragmented ecosystem where each device and every piece of code exposes its users to distinct security issues. Even worse, huge amounts of new code are being produced day by day; and tons of insecure legacy software have not yet reached their natural decay. Today, managing the sheer complexity is by far one of the most pressing issues. As Bruce Schneier phrased it: "Complexity is the worst enemy of security, and our systems are getting more complex all the time." [Sch15]. Research today literally shouts for more principled security technologies.

The Herculean task of securing our systems inevitably needs to be broken down into small manageable pieces. To this end, *enclaves* are a promising candidate. Enclaves are a recent technology built into our computers that can effectively shield our sensitive data from adversarial access, even remotely. Studying their security guarantees, precise capabilities, and limitations form an exciting and active field of research with enough substance to not only fill this thesis with hopefully valuable insight.

✳ ✳ ✳

Acknowledgements

We should really care about *what* we really care about.
A student

My first expression of gratitude is devoted to all the great people at university and in school who passed on both their knowledge and passion to me. Starting with my class teachers, Mrs. Rief and Mrs. Hackl in elementary and middle school, respectively, also my high school class teacher Mr. Friedl established an environment in which I could develop my strengths. They stand as representatives for many more in school who invested not only thought and time into us "kids" but helped us grow mature. Special thanks need to be given to my maths teacher, Mrs. Gabriel, and my electronics teacher, Mr. Töglhofer, who laid a solid ground for my entrance to Graz University of Technology. Many professors at university would deserve similar credit for their passion for teaching and the high quality of content mediated to us. As representatives I would like to thank Prof. Brenner for giving me a smooth start into the world of information and computer engineering, Prof. KC Posch for lively yet in-depth discussions on university and society as a whole, and my supervisor Prof. Mangard for igniting my fervor to work on IT security and providing an open playground for research. Being part of the IAIK institute fills me with deep gratitude, not least because of the encouraging and employee-friendly working environment but mainly because of the people, be it our ingenious system administrators, our supportive secretaries, friendly cleaning personnel and – of course – our professors who conduct this whole show through technical guidance as well as personal and group leadership, investing countless hours of thought and work – many of which are unseen!

Introducing my companions, it feels hard to divide between colleagues and friends. Mastering the university competition collaboratively, I owe to – without particular order – Christian, Julia, Tino, the two Stefans, Michael, Manuel, Richard, and Philipp. Guiding me in crossing the finish line of my master's studies and introducing me to the world of research – Mario, this was your fault. Bear the honor of a hero! Also, I owe big thanks to Anders Fogh, who initiated the idea of differential address trace analysis, and to Andi for pursuing this idea together with me. My Ph.D. would not have been possible without the help and advice of Raphael in bringing our results to the "research market". Countless discussions, not only transient speculations but for sure fruitful exchange of ideas are attributed to our ingenious "attack office". I very much appreciate sharing the "defense office" with David, authentic coffee breaks within and beyond our SESYS group, and the last Corona months with my great flatmate, Supermario.

I am thankful to many more fellows who became good friends, both within the university and outside. Special appreciation goes to Karli, whose friendship, joyful character, and straightforwardness companions me since high school, Reinhard for our thoughtful and deep conversations as well as inspiring bouldering trips, Kree for sharing life outside university, be it nature or game board evenings, and JP for continuing our uplifting "meetings" remotely, and Raphael for many late-night battles in who is the last man working. I want to especially thank Thomas for our early-day visions in school and our longstanding friendship since then, and to Lisa for our profound encounters and close amity, which evolved out of our shared pursuit towards a Ph.D. I feel very grateful to Chris for sharing his passion about and enlightening my way into student ministry, Tom for countless times of making music together and having dignified conversations, the Blooming team for dreaming big and influencing Graz for good, Berny for his invaluable encouragements, humor, and his beard, Josh for all the nuggets of wisdom he shared, Bernd for ingenious jam sessions and longstanding friendship, Ann-Sophia for enriching my research stay in Darmstadt and bringing joy in the little matters, Mathias for our friendly and profound coffee meetings, Tabsi for releasing joy in our student group in a unique way, Johannes for bringing together people from different backgrounds, Timna for creating a welcoming atmosphere much like a close family, Lukas for bringing consistency and fun into our group, and his inner beard, and Elsie for many encouraging hours spent together. Words are not enough to express my gratitude to Fabi for sharing life and becoming best friends over the last three years, sharpening and shaping each other's lives. I am more than thankful to Stefan for countless valuable reflections on personality, while establishing a decent apple strudel tradition, Lea for brightening the lives of so many around her, Anna for spreading important news to us, Jonatan for his energy in organizing events and loving people into our group, Iris for her deep passion and love for people around her, Oliver for cheering us up with his artistic vein, Matthias for many late-night considerations on the core values of life and faith we share, Benji for spending three exciting months of our lives together, and all the other precious people enriching my ways during the last years, whom I cannot mention by name. I am looking forward to many more valuable encounters with you that are yet to come.

To save the best for last, I want to sincerely thank my family for the supportive and healthy environment I grew up and live in: my dad for putting family first in so many decisions, my mom for waiving her job as teacher to all the more teach us kids true love, my brothers for bearing with me and sharing not only a great childhood together, and for my grandparents, uncles and aunts for supporting us in every aspect possible. I thank my parents, Siegfried, and many others for conveying to me the beauty of faith, and John Lennox and William Lane Craig for giving me unshakable reason to believe. Your reward will be eternal! Finally, I want to express sincerest thanks to *my* God for giving me the gift of life, the beauty of nature, the peace of heart, the joy of fellowship; for carrying me through illnesses, for manifesting His completeness in my insufficiency. Heavenly dad, your goodness in blessing me with all those people surpasses my understanding!

Abstract

Enclaves are a recent security technology for processors capable of safeguarding sensitive programs from malware and untrusted system operators alike. To understand the precise security properties of enclaves, we need more research. In this thesis, we study enclaves from two viewpoints. First, we take an outside perspective on the underlying hardware and effectively expand state-of-the-art enclave technology towards a broader range of application scenarios. Second, we inspect enclave's inner behavior with respect to side channels, showing novel attacks, and improving the automated search for unknown vulnerabilities.

For the first part of this thesis, we ask ourselves three questions: (i) How can enclaves securely interact with their (physical) environment? (ii) How can enclaves be realized on tiny resource-constrained devices? (iii) How can we prevent enclaves from running wild? For (i) we give theoretical results and show how a trusted hypervisor can provide secure enclave interaction in a generic way. Connecting enclaves with hypervisors yields subtle but fatal attack vectors, which we address by using a Trusted Platform Module. For (ii) we design and prototype enclaves on the open RISC-V architecture. Our system dubbed TIMBER-V makes use of a specially tagged memory to provide tighter integration and higher flexibility than comparable schemes. For (iii) we encapsulate enclaves themselves within a sandbox called SGXJail to contain potential misbehavior.

In the second part, we study side-channel attacks on enclaves with a focus on cryptographic software, again with three contributions. As shown by others, enclaves face stronger page-based side-channel attacks than previous systems. (i) We demonstrate that these attacks also directly affect the generation of cryptographic key material. To that end, we successfully attack RSA key generation in OpenSSL and provide patches to close the vulnerability. (ii) To automate side-channel analysis, we develop Differential Address Trace Analysis (DATA). DATA covers not only classical cache attacks but also fine granular single-instruction or single-byte leakage, which was believed impractical to exploit until very recently. DATA helped discover previously unknown leakage in OpenSSL. (iii) For a systematic study, we adapt DATA to detect leakage of secret nonces in DSA-like cryptosystems. Our analysis reveals known and several unknown vulnerabilities in all essential DSA computation steps of OpenSSL and others, many of which allow full key recovery. Our reports helped fix many of the issues.

✳ ✳ ✳

Table of Contents

1 Introduction

Trust takes years to build, seconds to break, and forever to repair.
Abraham Lincoln

Software vulnerabilities are a prevalent issue for the security of computing devices. More than 17.000 software vulnerabilities were reported within the last year[1] [NIS], 57% of which are ranked with high or critical severity. As distributed, networked computing becomes omnipresent in the Internet of Things (IoT), the risk of damage even increases, allowing remote exploits on a large scale. In the recent past, we have been witnessing attacks on million devices, like cameras and routers [NJC16], cars [MV15], cardiac devices [Lar17], and light bulbs [Ron+17], to name a few. Also, due to long service life, the security of such devices becomes decisive, making exploitation only a matter of time. A significant reason for this threat is code complexity, which makes traditional secure design paradigms like software verification or testing reach its limits. These security issues attenuate future use cases in the cloud or the IoT dealing with sensitive corporate or personal data since a compromise could have immediate monetary, legal, or privacy consequences [Hel17].

Rather than trying to achieve full system security, a promising line of research shifted the focus on securing sensitive code only and executing it in architecturally isolated containers, often referred to as enclaves. An enclave is protected against all non-enclave code by the hardware. Thus, the security of an enclave solely relies on the computing hardware and the enclave code itself. At the same time, the whole operating system can be safely considered untrusted from an enclave's perspective.

[1] This refers to the period between 01/01/2019 and 31/12/2019.

Intel SGX [Int16a] made enclaves available to the public via its x86 mainline CPUs. SGX targets high-performance cloud computing, where the cloud provider currently needs to be trusted entirely, as well as Digital Rights Management (DRM). Research has devised various scenarios of SGX, ranging from generic containers [BPH15; Arn+16; Shi+17b; TPV17] to specific application scenarios [Sch+15; PVC18; Lin+16; Bre+17]. Being relatively new, the security of enclaves is not as well understood as traditional concepts. In the following, we discuss open challenges and present our contributions.

1.1 Challenges

Despite the ability of enclaves to significantly improve application security, various new challenges arise. In this thesis, we address the following challenges:

1. **Secure I/O**. While Intel SGX securely shields a piece of code inside an enclave, it does not provide means for enclaves to communicate with an end user securely. This lack of secure I/O renders many potential application scenarios impractical, as sensitive user input like passwords or credit card information could be eaves-dropped before it arrives in the enclave. Also, presenting enclave output securely to the end user faces similar issues.
2. **Small and Open Systems**. Second, enclave technology is not yet as open as required for supporting a diverse ecosystem. While Intel SGX is the culmination of a long history of research on enclaves, its proprietary nature prohibits more in-depth security analysis. Attack scenarios range from speculation-based vulnerabilities [Bul+18] to deliberate backdoors [Dom18]. Open enclave systems, e.g., for the RISC-V [Ris] processor architecture, are necessary for transparent security analysis. Enclaves have been brought to larger RISC-V processors [CLD16] as well as to embedded systems [Eld+12; Noo+13; Göt+15; Noo+17; Koe+14; Bra+15; Arm17]. However, prior to our research, no enclave system existed for small RISC-V microcontrollers useful for deeply embedded IoT applications.
3. **Enclave Malware**. There is a flip side to enclave technology, as one can misuse them for malicious activity. Similar to rootkits [Kin+06; MY07], enclaves can effectively hide malware [Rut13; DF14; CD16]. Currently, no reasonable defense against enclave malware has been proposed or implemented.
4. **Side Channels**. Enclave technology does not solve the issue of software side-channel attacks [Sch+17; Bra+17b]. Even worse, Intel SGX aggravates this threat via novel deterministic side channels [XCP15]. While the research community is exploring these new attack techniques, it is unclear how popular cryptographic libraries such as OpenSSL and SGX-SSL [Int19] are affected by those attacks. Finally, searching for side-channel vulnerabilities in production software such as OpenSSL is not only tedious but also error-prone. This is mainly due to a lack of proper tool support side-channel analysts can use to assist their investigations. Existing side-channel analysis tools suffer from imprecision, accuracy, and performance issues.

1.2 Contributions and Outline

This thesis is split into two parts, where each contribution is represented by one chapter. We preface each part with an introductory chapter covering background information and related work. We list our publications as well as further collaborations in the end.

Part I discusses enclaves from an architectural perspective (challenge 1–3):

In Chapter 3, we develop the first concept for secure and generic user interaction with Intel SGX enclaves called SGXIO. To do so, we augment SGX with a small and trusted hypervisor. The key contribution is a proper binding of enclaves with the trusted hypervisor by utilizing a Trusted Platform Module (TPM). This work was published at CODASPY'16 [WW17a].

In Chapter 4, we, for the first time, bring enclaves to small RISC-V microcontrollers. To address the issue of limited memory, we devise novel tag-based isolation, which allows dynamic enclave memory management. Our scheme further enables novel use cases, such as interleaving of trusted and untrusted data on the same application stack or heap, thus reducing overall memory fragmentation. This work was published at NDSS'19 [Wei+19b].

In Chapter 5, we show how to prevent potential enclave malware from attacking the system. Enclave malware might attack its host application due to a lack of bi-directional isolation between enclaves and applications [SWG19]. We propose enclave sandboxing as a generic defense against enclave malware and show how enclave sandboxing can be efficiently instantiated both in software and in hardware. This work was published at RAID'19 [Wei+19a].

Part II discusses enclaves concerning side-channel vulnerabilities (challenge 4):

In Chapter 7, we demonstrate novel side-channel attacks on cryptographic key generation. The use of SGX exposes key-generation algorithms to side-channel attacks, which had been overlooked so far. To highlight this issue, we demonstrate the first single-trace attack on RSA key generation in SGX-SSL [Int19] and recover the secret key in 100% of the cases. Our proposed countermeasures helped fix the vulnerability. This work was published at ASIACCS'18 [WSB18a].

In Chapter 8, we develop Differential Address Trace Analysis (DATA). Rather than addressing specific attacks only, DATA is capable of uncovering arbitrary software side-channel vulnerabilities that rely on address information, including cache attacks [Per05], controlled-channel attacks [XCP15], DRAM-based attacks [Pes+16], and branch shadowing attacks [Lee+17b]. We implement DATA in a fully automated evaluation tool and use it to discover vulnerabilities in OpenSSL and PyCrypto. This work was published at USENIX'18 [Wei+18a].

In Chapter 9, we refine DATA towards detecting nonce leakage vulnerabilities in (EC)DSA-like cryptosystems. We systematically analyze the whole lifetime of nonces in OpenSSL, LibreSSL, and BoringSSL. We report various side-channel vulnerabilities across all essential computation steps, most of which stem from leaky Bignumber implementations. Our work helped fix many of the issues and was accepted for publication at USENIX'20 [Wei+20].

To make our results reproducible and help other researchers advance upon it, we open-sourced various code developed for this thesis.

✳ ✳ ✳

Part I

Enclave Security

✳ ✳ ✳

2

From Bugs to Enclaves. A Primer

> Whoever is careless with the truth in small matters cannot be trusted with important matters.
>
> *Albert Einstein*

Everything in a computer system is built on trust: trust in individual components, trust in their composition, and trust in the humans interacting with the system. A minor inadvertence inside a single core component, a wrong assumption about the user, and all trust might collapse like a house of cards. Ideally, the construction of components into a complete system is such that trust can be reduced to the bare minimum necessary, that is, to the piece of code and hardware responsible for managing critical tasks.

In this chapter, we first introduce the reader to two broad classes of attacks, which help understand the ideas and concepts presented later on. Afterward, we discuss the history of isolation technologies and finally introduce enclaves, the core subject for this part of the thesis.

2.1 Attack Scenarios

The landscape of attack scenarios is almost as diverse as the number of protection mechanisms. For this work, we cluster them into attacks on the inner behavior of a program and attacks on a program's environment in a larger system. While the first group of attacks exploit software vulnerabilities *within* a single program, the second group of attacks are performed *across* different programs by tampering with their execution environment. Often, a single software vulnerability in one program serve as an entry point into the whole system. Security architectures

shall uphold security guarantees of the system – even in case a program is compromised. Our actual contribution lies in the latter – security architectures. Although many security architectures involve trusted code components that need to be free of vulnerabilities, we focus on the design of security architectures rather than their correct implementation. Nevertheless, in the following, we outline both scenarios. We first discuss software vulnerabilities by touching on their threat models, describing prominent attacks, and giving countermeasures. Next, we discuss security architectures by elaborating on their responsibilities and the underlying threat model, introducing the trusted computing base, and highlighting important research goals to put this thesis into context.

2.1.1 Bug Alert! On Software Vulnerabilities

Software security focuses on analyzing the security of a given program with respect to its intended behavior. In this setting, one considers the whole operating environment – including the operating system and hardware – as secure. However, an attacker might play with the program's legitimate communication interface to trick the program into misbehaving. Typically, one precludes malicious intent of the programmer, such as backdoors coded into the program. Instead, one assumes that an honest but imperfect programmer made some inadvertent programming bugs. If those bugs affect the security of the system, they are called vulnerabilities. An attacker might exploit such a vulnerability by providing specially crafted input to the program.

Early attacks managed to directly inject or modify code at runtime, e.g., via a buffer overflow. However, this is not possible anymore with current systems. Instead, an attacker nowadays typically tries to manipulate control data and, thus, change the control flow of an application. By overwriting a code pointer, an attacker can divert the control flow to existing code snippets, resulting in so-called control-flow hijacking attacks. This gives an attacker the capability to perform touring-complete computation. From a functional perspective, an attacker has managed to transform the normal state machine of a program into a more expressive *weird machine* [Bra+11]. In its strongest form, a control-flow hijacking attack allows arbitrary code execution at the attacker's discretion. One of the most generic and powerful control-flow hijacking attacks is return-oriented programming (ROP) [Sha07], which overwrites return addresses to create arbitrary attack payloads. Similar attacks exist for overwriting function pointers [Che+10; Ble+11; Lan+15; CW14; Gök+14; Sch+15] or signal handlers [BB14].

Research on vulnerability defenses covers a wide range of strategies. The ultimate goal is to prove a program free of bugs, *i.e.*, verify its functionality against a given model. Typically, one matches software against a formal model, specified in some abstract language. Writing such a verification code can easily outweigh the effort for coding the program itself. For example, the development effort for the seL4 microkernel took 2.2 person-years, whereas the correctness proof took 20.5 person-years [Kle+14]. Hence, one tends to invest such costs only in high assurance systems [Kle+18]. Moreover, many program verification

techniques suffer from complexity issues such as state explosion [Val96]. Software testing reduces the state space at the expense of looser guarantees [MSB11].

Rather than analyzing precise functionality, a different line of research focuses on invariants that are inherent to most programs. For example, memory safety tackles the semantic gap between a programming language and the compiled binary from the perspective of objects in memory. The goal is to maintain invariants such as spatial memory safety, temporal memory safety, and type safety across the compilation step. These safety properties eradicate buffer overflow attacks, dangling pointers, or type confusion attacks. Rust already integrates memory safety in the language design [Rus]. Unfortunately, a large body of legacy code remains unprotected, and one resorts to weaker protection schemes, such as address-space layout randomization (ASLR) [PaX03], stack canaries [Cow98; PaX15], and shadow stacks [CH01]. While stronger control-flow integrity (CFI) [Aba+05; Kuz+14] can eradicate control-flow attacks, they leave data-only attacks [Car+15; Isp+18] unaddressed. A good overview of the struggle towards memory safety is given in [Sze+14].

2.1.2 Intrusion Alert! On Security Architectures

Having dug into the security of a single program, we now look at a complete system of multiple interacting components. In particular, we are interested in cases where not all components are behaving well. Such misbehavior might be due to the exploitation of software vulnerabilities. Alternatively, the user might unwittingly have installed malicious software. In many cases, the execution of potentially untrusted code is even unavoidable (cf. JavaScript execution in browsers) or part of the business model (cf. cloud computing). No matter how an attacker might infiltrate a system, security architectures shall maintain the secure operation of the other components in this case.

In light of powerful control-flow hijacking attacks, one typically assumes that an attacker has *arbitrary code execution*. That is, the attacker can execute arbitrary CPU instructions on the victim system an unlimited number of times, and interact with the victim system to exfiltrate gained information, or inject new instructions. It is the goal of the attacker to illegitimately access sensitive data, e.g., by elevating privileges and bypassing protection mechanisms. Note that the availability of the system is not necessarily the goal of security architectures, and the attacker might successfully mount denial-of-service attacks. Instead, security architectures shall protect the secrecy (or confidentiality) and integrity (or authenticity) of sensitive data even in case of intrusion. Protection involves proper isolation of different security domains and safeguarding their interaction.

Each security architecture builds upon a core set of components that are crucial for security and, hence, need to be trusted. Those components are also referred to as the trusted computing base (TCB). Any vulnerability inside the TCB is considered fatal for overall security. The TCB typically includes CPU hardware but also software components such as the operating system and user programs that are entrusted access to sensitive data.

Research on security architectures is two-fold. First, one studies the precise security guarantees an architecture offers, together with the underlying mechanics. While some systems focus on the isolation of data only, others also attest or authenticate code operating on it. An important question is also how a program can securely interact with the user, should the system be compromised. Security architectures range from trusted software running on commodity hardware down to specialized hardware extensions. Second, one needs to study the security of the TCB itself. Trusting the TCB is not necessarily based on security reasoning but an inevitable obligation to anyone using the system. Research shall provide convincing arguments that the TCB is also secure and, thus, trustworthy. This links back to our previous discussion on how to prevent (software) vulnerabilities. A mature security architecture shall provide its services at the smallest possible complexity. To phrase it differently: an important goal in designing security architectures is to minimize their TCB. A minimal TCB decreases the likelihood of bugs in the TCB and paves the way for full verification.

As mentioned before, our focus is not on vulnerability analysis of the TCB but its complexity reduction. Plus, we are interested in studying existing architectures in terms of their security properties.

2.2 A History of Isolation Technology

A fundamental building block for securing systems is isolation. The goal is to prevent compromised code from harming the rest of a system, which requires partitioning of memory into security domains. Isolation can be realized on various layers. Figure 2.1 depicts the most widely used schemes. In traditional process isolation, each application process belongs to a different security domain and cannot access the memory of other domains. Also, the operating system forms its own security domain, which is isolated from the rest. To protect against untrusted operating systems, Arm TrustZone runs so-called trustlets on top of a small, trusted operating system inside a *secure world*. One can bootstrap trust in software via secure boot, or with the help of a Trusted Platform Module (TPM) via trusted boot. Intel SGX integrates those concepts inside the CPU, allowing enclaves to run securely on a compromised operating system. In the following, we explain those concepts and their evolution in more detail.

2.2.1 Process Isolation

Process isolation is a fundamental security concept that combines hardware and software techniques to isolate the memory of processes from each other. Resource-constrained devices use physical memory protection for that purpose, while large systems isolate processes in separate virtual address spaces. Isolation is usually enforced by the operating system taking advantage of the processor's privilege levels.

Physical Memory Protection. Highly embedded systems, such as the Arm Cortex M series [Armb], operate on a single flat address space called physical mem-

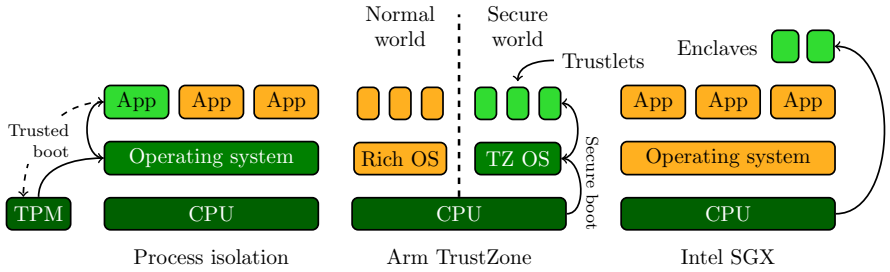

Figure 2.1: Process isolation is built on top of the operating system, which could be verified via a TPM and trusted boot. Arm TrustZone provides a so-called secure world for executing trustlets atop of a TrustZone operating system. Trust is typically established via secure boot. Intel SGX runs enclaves within an untrusted application, and trust is directly bootstrapped by the CPU. The TCB is colored green, including trusted components (dark green) and secured code (light green). Potentially malicious code is orange.

ory. Typically, part of this address space is used for read-only memory (ROM), random-access memory (RAM), and memory-mapped input/output devices denoted as MMIO. In such systems, memory isolation is achieved via a so-called Memory Protection Unit (MPU), a hardware module that safeguards access to physical memory. For the RISC-V architecture, MPU's are even called physical memory protection (PMP). An MPU sanitizes the address bus for malicious accesses, *i.e.*, accesses that do not belong to the current security domain. In case malicious access is detected, the MPU issues an exception that interrupts normal program execution and hands over control to the operating system. An MPU offers a fixed number of slots, each one holding a single contiguous memory range (e.g., the Arm Cortex M0+ offers eight MPU slots [Armb]). Only if a memory location belongs to the memory range of at least one MPU slot, access is allowed. In order to schedule multiple applications, the operating system is responsible for loading and unloading MPU slots upon each context switch. Due to the fragmentation of memory, an application might comprise a higher number of memory ranges than there are available MPU slots. In this case, the operating system can lazily swap MPU slots whenever the application accesses memory that is currently not loaded in the MPU.

In general, an MPU slot holds the start and end address of a memory range, but more optimized designs are possible by requiring memory ranges to be aligned. To increase security, MPUs can hold additional metadata for distinguishing read-only, writable, and executable memory regions, typically denoted as `rwx` bits.

Memory Segmentation. Having a single flat address space for all applications comes with its drawbacks. For example, the more applications are executed, the more tedious memory partitioning becomes. In particular, all memory addresses need to be known a priori, that is, at compile/link time, impeding dynamic memory management. For that reason, memory segmentation was invented. A memory segment is represented by a base address and a size, similar to an

MPU slot. Applications can reference any memory relative to a memory segment rather than via an absolute address. Thus, the application becomes position independent, and the operating system can decide at runtime at which memory location to load a particular application. Similar to an MPU, segmentation can also be used as an isolation technique by making an application's memory segments immutable and disjoint to other applications.

Virtual Memory. For mid and high-class application processors [Arma; Int16a; AMD20b], memory is virtualized, solving not only the fragmentation issue but also offering a strong primitive for isolation. For this, each application is given its own virtual view on the full memory range. E.g., on systems with 32-bit addresses, this covers $2^{32} = 4$GB of memory. This virtual address space is created atop of a 1:n address translation mechanism which transforms virtual addresses to physical ones. To simplify translation, both virtual and physical memory is split into chunks of pages (e.g., 4KB). A translation table maps virtual pages to physical pages. These translations are denoted as Page Table Entries (PTE). Each PTE is indexed by the virtual page address, and contains the corresponding physical page address (e.g., all but the lower 12 address bits for 4KB pages). Moreover, a PTE holds metadata such as permission bits (`rwx`) and access history bits. In order to increase efficiency, modern systems use a hierarchy of translation tables with up to five layers [Int16a].

As address translations are an expensive operation, a dedicated module called Memory Management Unit (MMU) does them in hardware. Moreover, successful translations are cached in the so-called translation lookaside buffer (TLB). On every context switch, the operating system not only switches translation tables, but it also needs to invalidate the corresponding TLB entries.

Privilege Levels. In all the above scenarios, the operating system is responsible for partitioning and, thus, isolating memory between applications. Obviously, the operating system needs to protect its own memory as well. It does so by the same means of physical or virtual memory protection used to protect applications. However, isolation alone does not suffice to defend against malicious applications bypassing it (e.g., by reconfiguring the MPU or MMU). Privilege levels are needed in hardware to constrain sensitive operations to the legit operating system. For example, the privileged kernel mode has access to the MPU or MMU. In contrast, the unprivileged user mode provides a constrained environment for executing potentially malicious application code. That is, specific CPU instructions are disallowed in user mode. Modern systems have more than two privilege levels. For example, x86 features four so-called protection rings indexed from 3 to 0, a separate virtualization layer informally indexed with -1, and a system management mode indexed with -2 for low-level hardware control. RISC-V provides a user mode for applications, a supervisor mode for the operating system, and a machine mode for emulating missing hardware features.

2.2.2 Bootstrapping Trust

Process isolation fundamentally relies on the correct operation of the operating system. If an attacker manages to compromise the operating system, e.g., by corrupting the boot images, all security guarantees are lost. Secure boot prevents compromised components from being loaded. An attacker might also trick a remote user into communicating with a compromised device controlled by the attacker instead of the legit device. Here, we need a trusted boot mechanism proving the device configuration to the outside world.

Secure Boot. A simple yet powerful method for bootstrapping trust is secure boot. Here, the first piece of software executed at power on - the so-called first-stage bootloader - will perform checks on the code of the next stage before actually invoking it. Only if the next stage matches an expected signature, it will be handed over control. Signatures are typically based on cryptographic hashes over the code. To increase flexibility, the hash itself is usually cryptographically signed, and the signature key is embedded in the code of the respective boot stage. By iteratively checking the boot process, trust can eventually be extended to a full operating system. Secure boot can be purely implemented in software, assuming the first stage bootloader can be trusted as is. To do so, one typically places the first stage bootloader in read-only memory. Secure boot is used for bootstrapping the Arm TrustZone [Arm09] and UEFI [UEF19]. On Android, secure boot is called *verified boot* [Edg15].

Trusted Boot. According to Casper et al. [CP11], "Trusted Boot refers to the ability to have confidence or trust in the security of a system startup, beginning with the initial configuration boot of the system." Secure boot alone does not provide a mechanism to verify remotely whether a device was initialized securely. Hence, trusted boot combines secure boot with a concept called authenticated boot [Arb05]. On Windows, authenticated boot is called measured boot [Sat18].

Authenticated boot generates an "[...] accurate record of the way that the platform booted" [Tru12]. This record might involve relevant software and the hardware configuration. In any case, it shall reveal whether security mechanisms are present and initialized correctly. Similar to secure boot, authenticated boot scans each successive boot stage before running it. The resulting signatures are called measurements. However, rather than aborting in case of a signature mismatch, the obtained signatures are accumulated into one signature, e.g., via a chain of hashes. Verification of signatures is deferred to the future, where an external verifier checks signatures in a process called attestation.

Trusted Platform Module. Authenticated boot requires secure storage for signatures (*i.e.*, measurements). The Trusted Platform Module (TPM) [TCG14] is a dedicated security co-processor built into a tamper-proof hardware chip that offers secure storage alongside various cryptographic services. It allows software to accumulate signatures, that is, extend measurements, but not revert them. One can also bind TPM key material to a particular measurement such that keys are only unlocked if the boot process was performed correctly. This is used for full disk encryption, such as BitLocker, amongst others.

Bootstrapping trust with a TPM is prone to cold-boot attacks [Hal+08] and cuckoo attacks [Par08], assuming an attacker intercepts the communication interface between TPM and the CPU. Also, a major disadvantage of trusted boot is that the trusted computing base might become significantly large. E.g., when using the TPM from a user application, the whole operating system needs to be measured and trusted as well (cf. Figure 2.1). Even worse, if a single component in the trust chain is compromised, there is no way to re-establish trust other than a complete reboot of the system. Since trust is derived from a single static component, namely the first-stage bootloader, it is also entitled *static root of trust*.

Dynamic Root of Trust. Intel Trusted Execution Technology (TXT) [Int15] is a CPU extension that tackles the inflexibility issues of a static root of trust. TXT allows freezing execution of whatever software might be running, and bootstrap into a fresh environment, effectively giving a dynamic root of trust. Inside this environment, one can load a small security kernel and perform security-critical tasks. Once finished, TXT switches back to the original context and resumes normal execution. The open-source tboot project builds upon Intel TXT to provide trusted boot for an operating system.[1]

2.2.3 Untrusted Operating Systems

We have seen that operating systems can be securely bootstrapped, and provide process isolation towards user applications. However, the amount of code that needs to be trusted (the TCB) comprises the whole operating system. Unfortunately, most widely used operating systems rely on a monolithic design, where a single bug can suffice to undermine process isolation completely. Microkernels, on the other hand, take the opposite approach by reducing kernel complexity.

Monolithic Kernels. Monolithic kernels perform all kernel-related tasks in a single security domain. This includes various forms of resource management as well as device drivers. Since monolithic kernels tend to have a large code base (e.g., the Linux kernel 5.6-rc4 comprises more than 18.5mio lines of code), the chance for unknown bugs is huge. For example, a total of 2300 vulnerabilities were reported for the Linux kernel within the last two decades, 246 of which are labeled as dangerous code execution bugs.[2] Finding such bugs in an automated way is a highly active line of research [Gen+18; Jeo+19; Yav19; LPW19b; Zha+19; Son+19; WLY18; LPW19a; Sri+19].

To reduce the attack surface, Linux randomizes the kernel layout via KASLR [Edg13]. Furthermore, it leverages supervisor mode access/execution prevention (SMAP and SMEP) to avoid misinterpretation of user memory as kernel memory. Also, other hardware features can be used to confine kernel operation further [GEL16; Pom+19; Gra+19].

[1] `http://sourceforge.net/projects/tboot`. (Accessed 05/03/2020)

[2] `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html` (Accessed 02/03/2020)

Microkernels. Unlike monolithic kernels, a microkernel attempts to reduce kernel complexity and, thus, the attack surface, to a bare minimum.[3] Typically, drivers are run in user mode, and resource management is handed over to userspace via so-called capabilities. A prominent example is the seL4 microkernel. The developers of seL4 reduced kernel code size to 8700 lines of C code and conducted formal verification to prove its correctness [Kle+09]. Designing a fully-featured, secure OS on top of a microkernel is a challenge on its own [Jac+15]. Nevertheless, Google is currently developing a capability-based operating system called Fuchsia atop of the Zirkon microkernel as a long-term replacement for Android.[4]

2.2.4 Towards Enclaves

Operating systems are arguably a weak link for the security of the overall system. We rely on them mainly because we have to. Removing the operating system from the trusted computing base (TCB) sounds appealing but impractical. To give an analogy in the physical world: one could barely remove the motor of a car and still expect it to move. Nevertheless, researchers asked the question to what extent one can distrust the operating system and still guarantee security. As a result, enclaves were developed. Enclaves are small containers that shield a piece of unprivileged code from the outside. The "outside" covers *all* code running outside of the enclave, including the operating system.

Enclaves evolved during a long journey of research and industry efforts, of which a partial, yet informative overview is given in [Mae+18]. Various terminologies related to enclaves exist such as trusted, secure, isolated, or shielded execution. In this thesis, we will use the term *enclave* as follows: an enclave is an *unprivileged* execution environment that allows one to securely execute a piece of code on a compromised system. We assume that the operating system and other software is untrusted and under control by an attacker. That is, the TCB only covers enclaves and the hardware (cf. Intel SGX in Figure 2.1). In contrast, the more general term *trusted execution environment (TEE)* may be used not only for enclaves but also for *privileged* execution environments that are shielded from a compromised operating system. In these privileged environments the TCB also covers a trusted operating system kernel (cf. Arm TrustZone in Figure 2.1).

Enclaves (as well as TEEs) are built upon three security primitives: memory isolation for protecting their code and data, entry point protection to prevent control-flow hijacking attacks, and attestation for bootstrapping trust. Furthermore, enclaves are unprivileged. The operating system can decide on their management.

Enclaves evolved from various techniques for secure code execution. Early designs put critical computation in a separate security co-processor [SW99]. The co-processor features cryptographic operations similar to a TPM but can also run a full general-purpose software stack, authenticated with a secure boot mechanism,

[3] For a list of popular microkernels see `http://www.microkernel.info/`
[4] `https://fuchsia.dev` (Accessed 02/03/2020)

and an attestation scheme. XOM explores execute-only memory [Lie+00], based on previous work on transparent memory encryption et al. [GLQ98; GLQ99]. AEGIS is a hardware-software co-design based on memory encryption and an optional security kernel [Suh+03].

Arm TrustZone. Arm developed TrustZone in 2008 [Arm09], which since then, became a de-facto industry standard. As shown in Figure 2.1, TrustZone provides a secure world for running secure software alongside an untrusted rich operating system, running in the normal world. This world split is orthogonal to the processor's privilege levels and effectively creates a secure virtual CPU. The secure world can access all system resources, while the non-secure world can only access non-secure memory regions. Moreover, the secure world can be extended to security-critical hardware peripherals. TrustZone demands a security kernel (e.g., a microkernel) which is responsible for running so-called trustlets, small user programs that shall be shielded against software attacks. The trusted computing base (TCB) covers all secure world code plus the hardware.

Other Isolation Techniques. Flicker bootstraps secure applications with the help of Intel TXT and the TPM [McC+08] and a small privileged bootstrapping code. However, during the execution of the secure application, Intel TXT suspends all other applications. Hypervisors have been used to separate secure applications from an untrusted operating system [TLL06; Che+08; YS08; CL10; McC+10; Chh+11; Hof+13]. Virtual ghost isolates sensitive memory via compiler transformations on the untrusted kernel [CDA14]. SICE uses system management RAM on x86 for creating a secure environment [ANZ11]. CARMA [Vas+12] and Oasis [Owu+13] create a secure environment purely inside CPU caches.

Enclave Architectures. IBM SecureBlue++ [WB11; BW12] is an extension to the Power architecture, which reduces the trusted computing base to the enclave (denoted as secure executable) and hardware only. It is compatible with existing software, protects enclaves also in the file system via encryption, and provides enclaves with shared memory and multithreading. Intel SGX [McK+13] brings enclaves to x86 off-the-shelf CPUs. SGX embeds enclaves in ordinary application processes and comes with an enclave-to-enclave and a remote attestation scheme. SGX outsources memory mapping to the untrusted operating system and also encrypts all enclave memory in DRAM. For this, enclave memory has to be allocated inside a pre-defined portion of DRAM, and the memory mapping is verified against an SGX shadow structure. We discuss SGX in detail in Section 2.3. Unlike SGX, Iso-X [Evt+14] allows using any memory for enclaves, however, without memory encryption. Also, Iso-X does not prevent the operating system from remapping enclave memory, which could lead to runtime attacks. Sanctum [CLD16] brings the SGX concept to the RISC-V architecture and uses a memory coloring scheme that is more resistant to side-channel attacks.

In order to isolate whole virtual machines rather than enclaves only, AMD added full memory encryption in their CPUs called Secure Encrypted Virtualization (SEV) [KPW16]. To address attacks tampering with encrypted memory [Mor+18], AMD recently integrated a memory integrity scheme for SEV, called Secure Nested Paging (SEV-SNP) [AMD20a].

Enclaves were also ported to small embedded systems via dedicated hardware support [Eld+12; Noo+13; Göt+15; Noo+17; Koe+14; Bra+15; Arm17]. We will discuss them in Section 4.9. MultiZone and Keystone both provide a TrustZone-like software stack for RISC-V MPU-based systems [Hex; Lee+20].

2.3 Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions (SGX) are an x86 instruction-set extension that has been rolled out with the Skylake microarchitecture. SGX first introduced the term *enclave* for running trusted code isolated from the remaining system. Enclaves opened a wide range of new application scenarios, such as trusted cloud computing [BPH14; Sch+15; Gje+17], protection of web browser input fields [Esk+19; Dha+20a] copyrighted material [BL16] or cryptocurrencies [Lia+17], secure networking [MAP18], and many more. Also, programs that were not intended for enclave execution can benefit from enclave-based container approaches. Haven shifts a whole Windows 8 library into an SGX enclave [BPH14], and SCONE [Arn+16], Panoply [Shi+17b], and Graphene-SGX [TPV17] provide such an abstraction for Linux.

In the following, we focus on SGX features and discuss attacks on SGX. For more details, we refer to the available literature [McK+13; Ana+13; Hoe+13; XSL16; Cha+17a; Fer+17; Ana+15; CD16; Int16a; Int16b; Int17b].

2.3.1 Basic SGX Features

The central concept of SGX is a hardware-isolated enclave container in which sensitive parts of an application are placed. Unlike TrustZone, an enclave directly resides in the address space of an untrusted user application (cf. Figure 2.1). Only the enclave itself can access its memory while the hardware prevents any other access to it. SGX does not rely on any privileged software (trusted kernel, hypervisor, etc.) to isolate enclaves, thus reducing the TCB to only the CPU and enclaves themselves.[5] To protect against physical attackers, SGX encrypts and integrity-protects all enclave memory on the fly when written to DRAM using a dedicated hardware encryption module.

Enclave Interaction. Figure 2.2 shows the process of invoking an enclave. The enclave defines secure functions denoted as ECALLs, which the application can call with the `EENTER` instruction. Call gates (CG) restrict ECALLs to valid entry points. This prevents simple control-flow attacks from the application. To further reduce the TCB, syscalls are disallowed from within enclaves. Instead, enclaves can request OS services such as syscalls via OCALLs. In order to leave the enclave, it can issue the `EEXIT` instruction.

Intel assists enclave developers with a so-called Enclave Definition Language (EDL). The EDL file is used to specify the ECALL/OCALL interface of

[5] SGX relies on CPU microcode, which is considered part of the hardware. For remote attestation, SGX interacts with the Intel Management Engine (ME) [Rua14b] as a trusted component.

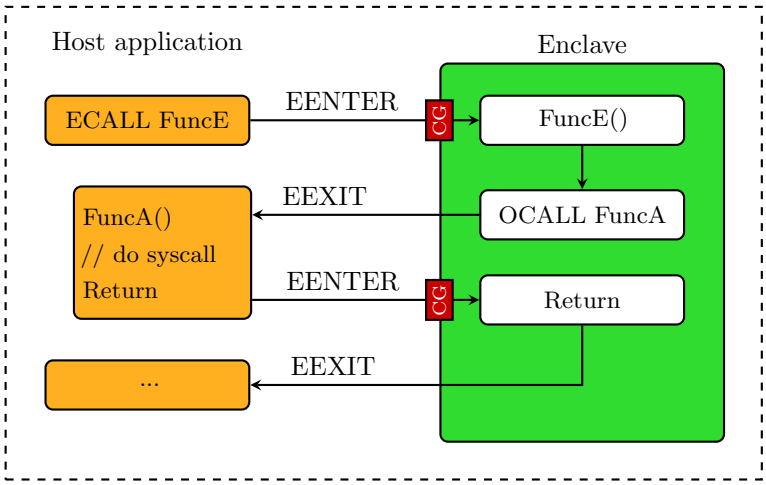

Figure 2.2: SGX enclaves are tightly integrated in a host application. The application can invoke the enclave via ECALLs while the enclave can perform OCALLs. Enclaves can only be entered via the `EENTER` instruction at dedicated call gates (CG) and can only be left via `EEXIT`.

a particular enclave. As such, it contains function signatures of the enclave's ECALLs and OCALLs, augmented with additional security attributes (e.g., `in`, `out`). Intel provides developers with an SDK [Int16b] that automatically generates glue code from the EDL file with appropriate parameter validation and buffer copying inside the enclave. The glue code is also responsible for maintaining CPU state across `EENTER` and `EEXIT` by saving and restoring CPU registers appropriately.

Shielding against the OS. In SGX, the operating system is entirely distrusted. Nevertheless, it is responsible for managing enclaves. The CPU safeguards all such enclave management operations and prevents the enclave from executing in case of operating system misbehavior. This affects three areas: enclave launch, memory mapping, and interruption.

Enclave Launch. To launch an enclave, SGX provides new initialization instructions to the operating system. `ECREATE` initializes a fresh, empty enclave. `EADD` lets the operating system load memory pages into the enclave. `EEXTEND` can be used to measure those enclave pages in a chained cryptographic hash log, stored in a register called `MRENCLAVE`. This process of hashing is comparable to a TPM [TCG14]. Not only the enclave page content is measured, but also the exact sequence of SGX initialization instructions. When completing initialization via `EINIT`, the CPU verifies the value of `MRENCLAVE` against a vendor-signed version and aborts on a mismatch. Only if the operating system has loaded the enclave in the intended way, this verification will succeed. Hence, `MRENCLAVE` vouches for the integrity of the enclave startup; it serves as load-time attestation.

Memory Mapping. In SGX, memory management is solely done by the operating system, as opposed to TrustZone. SGX does not prevent an operating system from mapping enclave pages in an illegitimate way. However, the CPU will refuse to run enclaves with an incorrect mapping. The corresponding checks are implemented in the virtual-to-physical address translation mechanism. A so-called Enclave Page Cache Map (EPCM) keeps a copy of the intended page mappings specified during `EADD`. This includes the virtual address, the permission bits, and the enclave this page belongs to. The EPCM can also be seen as a table of inverse PTEs, mapping physical enclave pages back to virtual ones. If any of the EPCM checks fail, an exception is triggered. Checks only need to be done once, as successful translations are cached in the TLB. However, the TLB needs to be flushed when entering an enclave with `EENTER`, causing a noticeable slowdown [WBA17].

Interruption. For normal ECALLs and OCALLs, the enclave software preserves the CPU register state. However, an enclave might get interrupted, e.g., via an external interrupt or an exception. This is called Asynchronous Exit Event (AEX). Upon an AEX, the CPU suspends enclave execution and stores its CPU state inside a special State-Save Area (SSA) in the enclave. Moreover, it will clear registers to avoid information leakage and hand over control to the operating system. To resume from interruption, SGX provides an `ERESUME` instruction that restores the CPU state from the SSA.

An enclave might want to handle some AEXs, such as floating point exceptions, itself. To do so, the application can re-enter an interrupted enclave via `EENTER`. Once the AEX is resolved and the enclave issued `EEXIT`, the OS can resume the enclave via `ERESUME`. Also, nested AEX are possible. For this, the SSA is organized as stack, and enclaves can be re-entered until the SSA is full.

2.3.2 Advanced SGX Features

Enclaves fundamentally build upon attestation and sealing, which we discuss in the following. Moreover, we will cover enclave debugging, licensing, and an SGX extension called flexible launch control.

Attestation. In order to bootstrap trust, SGX provides two attestation mechanisms [Ana+13], namely local and remote attestation.

Local attestation enables enclaves to verify each other. It is based on a signed report structure that contains `MRENCLAVE`. The report structure is signed by a CPU-specific key, and local attestation only succeeds between enclaves running on the same physical CPU. The report structure can hold additional user data. One can use this user data to authentically exchange information between enclaves and agree on an encryption key, for example. The SGX SDK uses local attestation to exchange a Diffie-Hellman key between two enclaves.

Remote attestation can be used by a remote party to check if the attested enclave is indeed running on a genuine Intel CPU. It allows the initial provisioning of keys and secrets. Provisioning is required since enclave code is public (in contrast to encrypted SecureBlue++ enclaves), and secrets cannot be embedded directly in the code. To perform remote attestation, Intel ships a so-called quoting

enclave with its SDK. After performing local attestation to the target enclave, the quoting enclave will prove its authenticity to an Intel Attestation Service (IAS). By verifying a remote attestation report against the IAS, developers can ensure that the attested enclave runs on a genuine Intel CPU.

Sealing. SGX also allows an enclave to obtain a sealing key, which is cryptographically bound to the local CPU. The enclave can use the sealing key to encrypt arbitrary data for offline storage and, thus, preserve its state among multiple system reboots. The Intel SDK comes with a protected filesystem (PFS) that automates the process of writing and reading encrypted SGX files, with integrity checks [Sel16].

SGX permits making the sealing key dependent on `MRENCLAVE` [Ana+13]. Hence, the same sealing key can only be queried from exactly the same enclave, running on the same, genuine Intel CPU. Furthermore, the sealing key can be derived from the enclave developer's public key. Thus, an enclave vendor can use the same sealing key among different enclaves.

Debugging. SGX distinguishes between debug and production enclaves. Debug enclaves can be accessed from the operating system via the `EDBGRD` and `EDBGWR` instructions, while production enclaves cannot. Moreover, debug enclaves can opt-in to ordinary x86 breakpoint handling and performance monitoring [Int16a]. This supports the enclave development process. In a production setting, however, enclaves have to run in production mode to protect against an untrusted OS. During the initialization of an enclave, one needs to specify a debug mode flag. The choice of this flag yields different `MRENCLAVE` values, making a debug enclave distinguishable from a production enclave.

Enclave Licensing. SGX has a controversially discussed peculiarity, called launch enclave [Bee15]. The launch enclave is a gatekeeper, controlling all other enclave launches. During enclave initialization with `EINIT`, the CPU verifies a so-called `EINITTOKEN`, which contains several enclave attributes to enforce, including the debug mode flag. The `EINITTOKEN` has to be signed by a special launch key only accessible to the launch enclave. The launch enclave is issued by Intel, giving it full control over which enclaves can be executed. The SGX SDK is shipped with a launch enclave issuing `EINITTOKEN`s for debug enclaves only [Int16b]. In order to run an enclave in production mode, one needs to obtain a proper license from Intel [JZD16].

Flexible Launch Control. Reacting to customer feedback, Intel introduced flexible launch control [Joh18; Sca+18]. Flexible launch control not only allows software vendors to use their own remote attestation infrastructure but also to take control over the launch enclave mechanism.

2.3.3 Attacks on Enclaves

Due to their strong security guarantees, enclave architectures are subject to new attack vectors. Side-channel attacks are of particular interest, but we defer a detailed discussion to Part II. In the following, we discuss various enclave vulnerabilities, replay attacks, and enclave malware.

Enclave vulnerabilities. Similar to application code, enclaves might be subject to memory safety vulnerabilities [Lee+17a; Bio+18], addressed by related defense mechanisms [Kuv+17]. However, synchronization issues become more pressing, since enclaves can be forcefully interrupted via induced AEX [Wei+16; Swa17]. They were generalized as COIN attacks [Kha+20], which relates to concurrency, order, input, and nesting issues. Also, any communication from an enclave with the untrusted operating system is problematic and might allow Iago attacks [PG08; CS13]. In an Iago attack, the operating system forges syscall return arguments to compromise the enclave. Proper checking of return values is especially relevant to library operating systems that allow hosting unmodified binaries inside an enclave container [BPH14; Arn+16; TPV17; Tia+17; Shi+17b].

Replay Attacks. In a replay attack, a malicious operating system downgrades the sealed SGX data blob to a previous, valid version. In order to counteract replay attacks on Intel SGX, sealing can be combined with persistent storage of the Intel Management Engine, a dedicated security co-processor present on recent Intel chipsets [Rua14b; Gue16]. Alternatively, one can achieve distributed rollback protection among multiple enclaves [Bra+17a; Mat+17].

Enclave Malware. SGX assumes that all non-enclave code (*i.e.*, operating system and host application) is untrusted. SGX provides no means to protect applications from misbehaving enclaves, giving rise to enclave malware [Rut13; DF14; CD16]. By design, an enclave can access the entire virtual address space of the host application. By using this mechanism, an enclave can share data with the host application (e.g., function parameters for ECALLs and OCALLs). However, it also creates an asymmetry in access permissions that an enclave can use to corrupt the host application [SWG19]. We address this issue in Chapter 5.

✳ ✳ ✳

3

SGXIO: Enclaves using I/O

Prayer is simply a two-way conversation between you and God.
Billy Graham

Secure computation is a core feature of enclaves; secure communication is not. Although the interaction between enclaves can be secured utilizing attestation and cryptographic channels, conversations with the outside world need other security means. Enclaves need to be able to discern input of an actual outside person from adversarial content, e.g., induced by the operating system. Also, enclaves might want to deliver sensitive output only to legitimate persons. Securing this enclave I/O is an ongoing field of research.

To achieve secure I/O, one requires *trusted paths* between enclaves and I/O devices. Stock SGX only works with proprietary trusted paths like Intel Protected Audio Video Path (PAVP). However, proprietary trusted paths are hard to analyze regarding security. Moreover, they are not generic and address specific devices and scenarios only. Unfortunately, SGX lacks support for generic trusted paths that work with any I/O device.

In this chapter, we present SGXIO, which at the time of publication to our knowledge has been the first generic trusted path architecture for Intel SGX. SGXIO runs user applications securely on top of an untrusted operating system while providing trusted paths to generic I/O devices. User applications benefit from SGX protection, while trusted paths are established via a small and trusted hypervisor. To that end, we identify and solve several challenges in linking the security domains of SGX and the trusted hypervisor, based on a Trusted Platform Module (TPM).

SGXIO allows a remote party to attest not only enclave code but also the whole trusted path setup. Also, SGXIO enables human end users to verify trusted paths without requiring additional hardware. SGXIO improves upon existing generic trusted paths for pre-SGX systems with a more intuitive programming model. We furthermore give a novel zero-overhead, non-interactive key transport scheme for establishing a 128-bit symmetric key between two local SGX enclaves. Our scheme is more efficient than Diffie-Hellman local attestation implemented by the Intel SGX SDK. Finally, SGXIO can tweak insecure debug enclaves to behave like secure production enclaves.

Traditional SGX targets high-performance cloud computing, where the cloud provider is entirely distrusted, as well as Digital Rights Management (DRM). SGXIO surpasses those use cases and makes SGX technology usable for protecting user-centric, local applications against compromised systems. It works on modern commodity notebooks and is compatible with unmodified operating systems. Hence, SGXIO is particularly promising for the broad x86 community to which SGX is readily available.

Contributions. We summarize our contributions as follows:

- We present the concept of SGXIO, the first generic trusted path architecture for Intel SGX.
- We highlight novel challenges in linking the security domains of SGX and the trusted hypervisor, and give solutions based on the Trusted Platform Module (TPM).
- We give a novel fast non-interactive key transport scheme that is more efficient than local attestation implemented by the Intel SGX SDK.
- We show how SGXIO can tweak debug enclaves to behave like production enclaves.

This chapter is based on the publication [WW17a] and the extended author manuscript [WW17b], both of which I am the main author. These papers, in turn, expand ideas from my master thesis [Wei16]. The rest of this chapter is structured as follows: Section 3.1 gives related work on trusted paths. Section 3.2 discusses the threat model and challenges. Section 3.3 presents our SGXIO architecture, while Section 3.4 gives a thorough security analysis. Section 3.5 gives our novel key transport scheme for enclaves. Section 3.6 shows how SGXIO can apply the debug enclave tweak. It is followed by further considerations in Section 3.7 and a summary in Section 3.8.

3.1 Trusted Paths

A trusted path is a secure communication channel between a system's trusted computing base (TCB) and a user [Dep85]. The user shall be able to initiate trusted path communication and to distinguish a trusted path from any other untrusted communication channel. In the following, we discuss trusted path solutions that work on standard I/O devices and systems that require modifications

to the I/O device itself (e.g., via a USB bridge). We continue with recent research results that were published after SGXIO and conclude with trusted paths on Android and the Arm TrustZone.

Standard I/O Devices. Garfinkel et al. [Gar+03] proposed Terra, which isolates different distrusting applications in separate virtual machines (VM), and establishes trusted paths between users and a VM. However, a VM itself might operate a large, untrusted operating system. To remove trust from device drivers, Garfinkel et al. advocate for more hardware support to achieve secure I/O. Nitpicker [FH05] addresses the often large TCB necessary for rendering secure output. Ta-Min et al. [TLL06] proposes Proxos, which isolates the normal operating system from a private one via virtualization. They support a trusted path towards a protected X-server. Borders et al. [BP07] propose a trusted input proxy in the virtual machine monitor for relaying user input to a server.

Filyanov et al. [Fil+11] discuss a pure unidirectional trusted path for secure user transactions using Flicker. Zhou et al. [Zho+12] build the first comprehensive, generic trusted path for x86 systems. Their isolation is based on TrustVisor [McC+10] and protects a trusted path all the way from the application level down to the device level. They consider PCI device misconfiguration, DMA attacks as well as interrupt spoofing attacks. However, pure hypervisor-based designs come at a price. They strictly separate the untrusted stack from the trusted one. Hence, the hypervisor is in charge of managing all secure applications and all associated resources itself. This includes secure process and memory management with scheduling, verified launch, and attestation. Also, communication between both security domains might be non-trivial due to synchronization issues or potentially mismatching Application Binary Interfaces (ABI). In contrast, SGXIO uses the comparably simple programming model of SGX enclaves, in which the untrusted operating system remains in charge of managing secure enclaves. Moreover, SGX provides verified launch and attestation out of the box.

In [Zho14] and [ZYG14], Zhou et al. discuss a trusted path to a USB device. Their so-called wimpy kernel runs alongside the virtualized operating system. Yu et al. [YGZ15] apply the trusted path concept to GPU separation.

Modified I/O Devices. Another way of establishing a trusted path is via dedicated I/O devices that support cryptographic channels to tunnel through untrusted software. However, this idea does not generalize to other I/O devices.

Bumpy [MPR09], which is based on [MPR06], proposes encrypted communication between input devices and a Flicker-protected system that, in turn, communicates with a web server. They achieve a trusted input path with a USB interposition device. Intel's Protected Audio Video Path (PAVP), as well as its successor, Intel Insider [Rua14b; Knu11], provides a proprietary trusted output path. Both rely on Intel's Management Engine (ME) to establish a cryptographic channel to the GPU. Ruan [Rua14b] describes a protected transaction display via the Intel ME, which makes use of PAVP to securely obtain a one-time PIN from the user. Unfortunately, ME-related code is proprietary, which limits potential open-source use cases and hinders transparent security assessment. Hoekstra et al. outline the integration of PAVP in SGX applications to achieve secure video

conferencing and one-time password generation [Hoe+13]. They defer trusted input paths on SGX as future work.

Recent Work. Research continued after publication of SGXIO. Zhou [Jan17] discusses SGX remote attestation towards a USB dongle called SGX-USB. Kirchengast [Kir19] discusses relay attacks against SGX-USB in light of a compromised platform, and show how to combine SGX enclaves with a MACSec-protected network. Bastion-SGX [Pet+18] establishes a trusted path between an SGX enclave and a trusted Bluetooth device.

ProximiTEE [Dha+20b] provides a trusted path from an enclave to a USB dongle. In a trust-on-first-use scheme, the user boots a small ProximiTEE kernel which initializes an enclave by creating and sealing encryption keys and deploying them to the USB dongle. The dongle has an integrated LCD and buttons to allow users to choose the enclave to open a trusted path to. Furthermore, ProximiTEE comes with a distance bounding scheme to detect whether the dongle is indeed communicating with the correct enclave and that no relay attacks are ongoing.

Fidelius [Esk+19] establishes a trusted path between a user and a web enclave running in the browser. The web enclave protects HTML input form fields which are tagged as secure. To establish trusted paths from the enclave to the keyboard and the screen, a USB and an HDMI proxy device are used. Screen overlays protect sensitive input forms and also present the current trusted path to the user (*i.e.*, which domain and which input fields are active). Separate LEDs on the proxy devices indicate an established trusted path channel. Trusted path channels have a constant, encrypted data stream to avoid side-channel leakage.

ProtectIOn [Dha+20a] protects HTML forms via an IOHub device rather than a web enclave. It is based on IntegriKey [Dha+17] and also uses the same hardware as ProximiTEE [Dha+20b], although without SGX support. The IOHub device intercepts all keyboard, mouse, and HDMI signals. It receives cryptographically protected input forms from the web server via a QR code, which it renders as screen overlay. If the user enters the overlay region, IOHub freezes and greys out unprotected screen content and hides all further input signals from the host. Instead, IOHub encrypts user input before sending it back to the host. IOHub needs to emulate the mouse cursor, involving interpolation and synchronization of its position with the host.

Aurora [Lia+20] provides trusted paths between SGX enclaves and an SMVisor running in System Management Mode (SMM). The SMVisor intercepts device interrupts and invokes the corresponding UEFI device drivers. It furthermore communicates with an enclave to establish a trusted path. Aurora is evaluated on keyboards, serial devices, and USB storage. Using SMM is appealing from a security perspective, but might introduce performance penalty due to full system preemption. To attest the SMVisor to the enclave, Aurora proposes to use TPM attestation features in combination with the SGX launch enclave. However, the SGX launch enclave cannot be used for that purpose, as it has no notion of the TPM. With SGXIO, we present a way to achieve attestation of a hypervisor, which equally applies to SMVisor running in SMM.

Android. Trusted paths on mobile phones are often directly integrated into existing commodity operating systems, such as Android. ScreenPass deploys a trusted software keyboard [Liu+13], and GuarDroid [TE13] hooks password input fields to protect user passwords. Crossover [LL13] tackles secure user interfaces with multiple personalities. TIVOs [Fer+14] are trusted visual I/O paths for Android applications that show a secret image to the user to ensure a correct trusted path setup. All those approaches consider the Android subsystem as trusted, thus significantly enlarging the TCB. In contrast, Brandon et al. implement a trusted display for Android using an FPGA for screen overlays [BT17].

Arm TrustZone. Mobile phones based on Arm processors likely feature TrustZone support. While TrustZone allows running code protected from the Android operating system, it can provide trusted paths as well. As such, TrustZone can isolate hardware peripherals and physical memory via a system MMU (cf. an IOMMU in x86), which can also prevent DMA attacks. Moreover, device interrupts can be routed directly into the secure world.

TrustUI [Li+14] provides secure rendering and touchscreen input on Android using TrustZone. Its secure software keyboard randomizes the keyboard layout to decorrelate touch positions from the entered key. A security LED indicates the validity of the trusted path. Schrodintext [San17] removes glyph rendering from the TCB by performing it in the Android operating system, which is considered untrusted. A monitor running in the TrustZone then stitches together the final view. VButton [Li+18] renders critical UI components necessary for user confirmation in the TrustZone and interposes on touchscreen actions. It sends user actions to a web server in an authenticated way. Trusted user interfaces are already being integrated into off-the-shelf mobile phones by Trustonic [Hay19] and Android [Dan18]. They make use of TrustZone to render security-relevant user interfaces, e.g., for validating transactions.

3.2 Threat Model and Challenges

SGXIO utilizes SGX as one building block to provide isolated execution. However, the threat models of pure SGX and SGXIO differ. This section elaborates on the threat model of SGXIO and shows that, in contrast to pure SGX, physical attacks do not apply to trusted paths. Furthermore, we discuss the challenges arising from the combination of SGX with a trusted hypervisor.

3.2.1 Distinction from SGX

SGX is an isolated execution technology with a small trusted computing base (TCB). The TCB only contains the processor itself, which acts as a trust anchor, and the code running within enclaves. Everything else is considered potentially malicious, including not only all other software components (e.g., operating system, hypervisor) but also the hardware environment. Therefore, SGX considers both logical attacks and physical attacks.

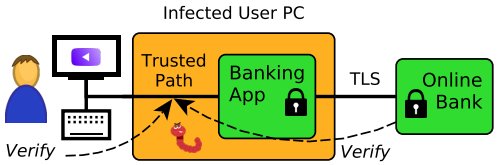

Figure 3.1: In an online banking scenario, malware shall not be able to hijack a banking session. Communication with the bank is encrypted via TLS, while the banking app itself is protected with SGX. Both the user and the bank want to be able to verify the security of the trusted path.

This threat model perfectly fits the requirements for secure cloud computing in which a customer wants to protect enclave code and data against an untrusted cloud provider, controlling the software stack and the hardware. In this use case, all communication with an enclave can be performed using securely encrypted and authenticated channels. Also, content providers can use SGX to enforce a DRM scheme on an untrusted consumer PC.

In a local setting, however, a user wants to benefit from SGX by protecting user-centric applications against a potentially compromised operating system. Especially, communication via the user's I/O devices needs protection from the operating system via a trusted path. This setting contradicts the threat model of SGX, which considers the physical environment, and hence, also the local user, a threat. Currently, in order to achieve a trusted path with SGX, one has to rely on encrypted interfaces like PAVP. However, the prevalence of unencrypted I/O devices in today's computers and the lack of support to securely communicate with these devices demands other, more generic mechanisms.

SGXIO fixes these shortcomings by extending SGX with a generic trusted path. Many user-centric applications can profit from this additional feature. This covers local applications like confidential document viewers, anti-spoofing password prompts, secure password generators, and password safes but also web scenarios, such as secure conferencing and chat applications and online banking.

To take the latter as an example, SGXIO cannot only secure online banking up to the user's browser via TLS but up to the I/O devices via trusted paths, as depicted in Figure 3.1. Hence, SGXIO can protect sensitive information, such as login credentials, the account balance, or the transaction amount, even if other software running on the user's computer, including the operating system, is infected by malware. Moreover, SGXIO's path attestation allows both the user and the online bank to verify that trusted paths are established and functional.

3.2.2 Threat Model

In general, SGXIO establishes trusted paths between a user and a user app. More precisely, SGXIO connects a user's I/O devices with an SGX enclave. Attackers

try to subvert such trusted paths by breaking their confidentiality or authenticity guarantees via logical attacks, as explained below.

Logical attacks are the primary concern of SGXIO. Attackers are assumed to have full control over the operating system and know the whole software configuration, including all enclave code. This is a realistic scenario, addressing both local and remote software attacks, which might even yield kernel privileges to attackers. Attackers can directly attack enclave interfaces visible to the operating system, intercept enclave communication, run enclaves in a fake environment (e.g., within the operating system), etc. Also, attackers can dynamically load and execute custom user apps, enclaves, and drivers and open other trusted paths.

Moreover, indirect attacks on a trusted path can be performed by misconfiguring devices under operating system control, as outlined by Zhou et al. [Zho+12]. The idea of such attacks is to manipulate noninvolved devices to interfere with a trusted path. For example, a PCI device could be misconfigured such that its address range overlaps those of the user device. Also, malicious Direct Memory Access (DMA) requests could be issued, and interrupts could be spoofed.

All code in the TCB (*i.e.*, secure user apps packed into enclaves, secure I/O drivers, and the hypervisor) is assumed to be correct and not vulnerable to logical attacks. Using a formally verified hypervisor such as seL4 [Kle+09] supports this assumption. Also, all hardware (CPU, chipset, peripherals) is expected to work correctly. As with SGX, Denial-of-Service (DoS), as well as side-channel attacks, are out of scope. Note that SGXIO requires an Intel processor with SGX support as well as support for TPM-based trusted boot.

Physical attacks are not considered in SGXIO. A physical attacker has direct access to I/O devices and can impersonate the user without subverting trusted paths. Thus, trusted paths can only protect against logical attacks but cannot provide physical security. To put it differently: SGXIO trusts the user interacting with the system.

3.2.3 Challenges

SGXIO combines SGX with a trusted hypervisor to provide a generic trusted path. However, the hypervisor and SGX form two disjoint security domains with two different trust anchors, which are not designed to collaborate. Subsequently, connecting both domains is a non-trivial task.

This essentially breaks down to two challenges we solve with SGXIO: First, the security domains of the hypervisor and SGX enclaves have to be linked. More concretely, we need a way for SGX enclaves to check the presence and the authenticity of the hypervisor. We name this problem *hypervisor attestation*. Once the hypervisor is attested, it extends trust to any trusted path it establishes.

Second, the SGXIO architecture relies on multiple SGX enclaves that communicate using keys based on local attestation, as discussed in the following sections. These enclaves are executed in different security contexts (trusted hypervisor vs. untrusted operating system). However, in SGX, enclaves are unaware of their context, making them vulnerable to *enclave virtualization attacks*. SGXIO prevents such attacks via a careful interface design between both contexts.

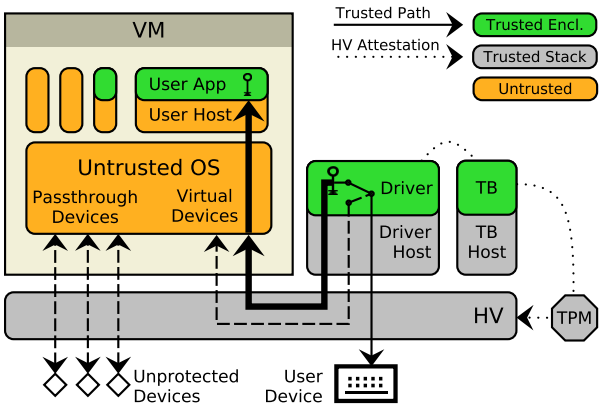

Figure 3.2: The trusted HV stack consists of a hypervisor (HV), a Trusted Boot (TB) enclave, and one or more secure I/O drivers. The virtual machine (VM) operates an untrusted operating system (OS) which hosts secure user apps. The driver obtains data from the user device (thin line) and encrypts it (bold line) for a user app, providing a trusted path (solid line). The TB enclave allows drivers to attest the hypervisor.

3.3 SGXIO Architecture

This section presents SGXIO and elaborates on its isolation guarantees. We discuss the design of secure user apps, I/O drivers, and the hypervisor.

3.3.1 Architecture

SGXIO is composed of two parts: a trusted HV stack and a Virtual Machine (VM), as seen in Figure 3.2. The trusted HV stack contains a small security hypervisor (HV), one or more secure I/O drivers, which we simply call *drivers*, as well as a Trusted Boot (TB) enclave. The VM hosts an untrusted commodity operating system (e.g., Linux), which runs secure user applications, also abbreviated with *user apps*. User apps are protected by means of SGX enclaves.

User apps want to communicate securely with the end user. They open an encrypted and authenticated communication channel to a secure I/O driver to tunnel through the untrusted operating system. Authentication is done via SGX local attestation and indicated with a solid line in Figure 3.2. The driver, in turn, requires secure communication with a generic user I/O device, which we term *user device*. In order to achieve this, the hypervisor exclusively binds user devices to the corresponding drivers. Note that any other device is directly assigned to the VM. I/O on those unprotected devices directly passes through the hypervisor without a performance penalty. The trusted path comprises both the encrypted user-app-to-driver communication and the exclusive driver-to-device binding. Drivers use the TB enclave to get assured of correct trusted path setup

by attesting the hypervisor via trusted boot. Hypervisor attestation is indicated by a dotted line.

3.3.2 Isolation Guarantees

SGXIO establishes a trusted path from a user app to the user device, which requires isolation on several layers. First, all trusted HV stack memory needs to be isolated from the untrusted operating system and Direct Memory Access (DMA). Second, the trusted path itself requires isolation from the operating system. Third, the user device needs isolation from all other devices which are under the control of the operating system. This section outlines how SGXIO meets these isolation requirements.

Trusted memory isolation is a prerequisite for securely executing trusted code in an untrusted environment. It is relevant for user apps as well as the trusted HV stack. We achieve memory isolation for the user app by executing it within an enclave. SGX isolates all enclave memory from the untrusted operating system. In order to achieve memory isolation for the trusted HV stack, the hypervisor confines the untrusted operating system in a VM. Moreover, the hypervisor implements a strict memory partitioning by configuring the Memory Management Unit (MMU) appropriately. The MMU configuration prevents the operating system from escaping the VM and tampering with the trusted HV stack.

Direct Memory Access (DMA) is a more subtle threat to memory isolation [Zho+12]. A DMA-capable device can directly access memory, thus bypassing any MMU protection and potentially violating the integrity and confidentiality of trusted memory. SGX prevents DMA from accessing enclave memory, hence the user app is safe against DMA attacks [Int16a]. Likewise, the trusted HV stack needs protection against such attacks. Modern chipsets typically incorporate an I/O Memory Management Unit (IOMMU), also termed VT-d on Intel systems. The IOMMU restricts device DMA to specific portions of RAM only. By properly configuring the IOMMU, the hypervisor can protect the whole trusted HV stack against device DMA attacks.

Trusted Path Isolation. The trusted path needs protection on two layers, namely the communication between user app and driver as well as the interaction between driver and user device. The user app communicates with the driver via the untrusted operating system stack; hence a cryptographic channel is necessary. The interaction between the driver and the user device is protected by the hypervisor, as follows: The hypervisor establishes an exclusive binding between a driver and the corresponding user device. Moreover, the hypervisor mutually isolates all drivers. Thus, an attacker, loading arbitrary driver code at will, cannot interfere with trusted paths established by other drivers.

User Device Isolation. As outlined before, a malicious operating system could misconfigure devices to interfere with the trusted path. In that way, the operating system could force PCI devices to overlap their MMIO region or I/O port range with those of the user device, or issue forged interrupts on behalf of the user device. To protect against these attacks, Zhou et al. implement several policies in

the hypervisor to detect and prevent malicious device configurations. Thus, their system effectively isolates a user device from other operating system-controlled devices. This approach is also applicable to SGXIO.

3.3.3 User App Design

Secure user applications play a central role in concrete use case scenarios, such as secure online banking. This section outlines principles for designing user apps and shows how user apps securely communicate with drivers. In the end, we elaborate on the enclave programming model.

Design Principles. Key to any secure user-centric application is a trusted path that protects user I/O against advanced malware like keyloggers and likewise. Without a trusted path, an attacker could impersonate the user and act on its behalf, even if the user app itself is not compromised. In order to provide its service, a user app might communicate with other user apps or exchange sensitive data with a remote server using TLS, for example. One should perform any operation on sensitive data within an enclave and keep unproblematic glue code and untrusted libraries, such as file management and network access, outside. All interaction with untrusted code needs careful validation inside the enclave [PG08; CS13; BPH14]. In order to keep state between multiple invocations, enclaves can encrypt sensitive data for offline storage using SGX sealing, for example.

Cryptographic Channel. The user app opens a trusted path by setting up a cryptographic channel to a secure I/O driver. The channel protects the confidentiality and authenticity of sensitive user I/O against the untrusted operating system. In order to open such a channel, the user app needs to exchange an encryption key with the driver. SGX local attestation can assist in the key exchange by providing means to authenticate information between user-app enclaves and driver enclaves. A straight-forward implementation provided by the SGX SDK uses the Diffie-Hellman key exchange [Ana+13; Int16b]. However, local attestation inherently provides a much faster way of exchanging key material.

We give a novel, lightweight key transport scheme, which comes with just a single unidirectional invocation of local attestation. We reuse the already pre-shared SGX report key to derive random 128-bit encryption keys. The scheme works as follows: The user enclave generates a local attestation report over a random salt, targeted at the driver enclave. However, instead of delivering the actual report to the driver enclave, the user enclave keeps it private and uses the report's MAC as a symmetric key. It then sends the salt and its identity to the driver enclave, which can recompute the MAC to obtain the same key. Details of this scheme are given in Section 3.5.

Once a key is established, one can use any authenticated encryption scheme to secure the data stream between the user app and the driver. The use of an authenticated encryption scheme ensures the confidentiality and integrity of the data stream. By doing key exchange via SGX local attestation, the user app and the driver can also mutually authenticate each other. This property is also referred to as origin integrity.

Enclave Programming Model. SGXIO benefits from SGX's simple enclave programming model [Int17b; Ana+15]. User app enclaves are executed directly on a host application running within the untrusted operating system. Hence, secure user applications are treated similarly to normal application processes. The operating system has control over memory management, process management, and scheduling of enclaves, although SGX carefully validates any action that might affect enclave security. Also, integration of multiple enclaves into a bigger user application stack is easy since enclaves share parts of the register set and the virtual address space with their host for communication purposes. Moreover, SGX supports multithreading as well as enclave debugging. Also, SGX natively provides verified launch and attestation of enclaves, which is tedious to implement in software. To support enclave development, Intel provides a software development kit [Int16b] as well as a developer guide [Int17b].

3.3.4 Driver Design

Secure I/O drivers are responsible for connecting user apps and user devices. Drivers are hosted and protected by the hypervisor. Although hypervisor protection is sufficient to isolate drivers from the untrusted operating system, SGXIO also executes actual driver logic inside an enclave. This helps in setting up an encrypted communication channel with user apps, as previously described. Also, driver enclaves are subject to attestation, allowing identification via their `MRENCLAVE` values.

When designing a driver, one has to make specific design choices. We opt for two strategies, namely domain multiplexing, and portability, targeting commodity operating systems. *Domain multiplexing* allows the same driver and, thus, the same user device to be shared across security domains. *Portability* refers to drivers being compatible with different operating systems. Note that SGXIO supports other choices as well.

Domain Multiplexing. A driver handles the data stream from and to a user device and forwards it to the operating system or a user app, respectively. Since many user devices, such as human interface devices or graphic cards, are potentially shared between the untrusted operating system and user apps, the driver has to multiplex the data stream between those security domains. In our example (see Figure 3.2), the driver offers its service to the operating system via two separate virtual devices. During normal operation, the driver simply routes the unmodified data stream to the first virtual device, which matches the device class of the user device. Thus, the operating system has transparent access to the user device. If the user app requests a trusted path, the driver redirects all traffic to the second virtual device, however, in an encrypted fashion. The user app, knowing the proper decryption key, can access this second virtual device to tunnel through the untrusted operating system. This second virtual device can be any standard character device, for example, which just forwards the encrypted data stream.

In our example, the driver implements strict temporal multiplexing between the operating system and the user app. However, one could enforce arbitrary

security policies. For example, the driver could implement spatial partitioning of a graphic card's frame buffer to allow secure screen overlays. Or it could intercept and mask specific keystrokes to react on secure attention sequences [MPR09] and encrypt password entry, for example.

Portability. In our example, we encourage virtual devices as a communication interface between drivers and the operating system, cf. Figure 3.2. They have the advantage of being compatible with commodity operating systems. No changes to the operating system are required since the driver can emulate user devices transparently. Also, the same driver implementation can be reused across different operating systems without porting effort. One can put saved manpower in more robust driver implementations. Note that specific high-throughput user devices might still need cooperation by the operating system.

3.3.5 Hypervisor Design

The hypervisor is responsible for running the untrusted operating system in a VM, as well as loading drivers and binding user devices to them. It can load drivers statically at system boot, which makes sense for permanently installed user devices, such as notebook keyboards and graphic cards. For plug-and-play devices like USB, the hypervisor might dynamically load the corresponding drivers. Note that typically the hypervisor delegates such resource management tasks to a separate Virtual Machine Monitor (VMM).

The hypervisor enforces a bunch of isolation guarantees, as previously outlined: First, it isolates all trusted HV stack memory. Second, it binds a user device exclusively to the corresponding driver and mutually isolates drivers. Third, it isolates user devices from malicious interference with other devices.

Formal Guarantees. The choice of an appropriate hypervisor fulfilling these requirements is crucial for the overall system's security. Since the hypervisor is an essential part of the TCB, it is desirable to have some formal guarantees.

The seL4 microkernel is formally verified [Kle+09; Mur+13]. The proofs cover not only functional correctness of the generic C implementation but also help to find a correct kernel configuration under which isolation guarantees hold. The developers of seL4 claim to have the first general-purpose microkernel with such strong guarantees. Although they conducted initial proofs for the Arm architecture, most of the proven generic code also applies to x86. We recommend using seL4 as a hypervisor, as it allows a straight forward design of SGXIO.

Other kernels and hypervisors comprising at least partial formal proofs are Microsoft Hyper-V [LS09; Coh+09], Verve [YH10], ExpressOS [Mai+13], µC/OS-II [Xu+16], CertiKOS [Gu+16], Hyperkernel [Nel+17], the XMHF hypervisor [Vas+13], and the üXMHF hypervisor [Vas+16].

seL4. seL4 implements a strict resource partitioning, which directly supports the isolation of trusted memory as well as user device binding. seL4 manages resources via capabilities [seL20]. By granting specific capabilities to the VM or a driver, they get access to the underlying resources. With such a capability system in place, isolation breaks down to a correct distribution of memory and device capabilities among the VM and the drivers. For example, the VM, as well

as each driver, gets assigned a disjoint set of memory capabilities, thus enforcing memory isolation. Likewise, each driver gets capabilities to its own user device only. Capabilities to other devices are given to the VM. This ensures trusted path isolation.

To enforce its capability system, seL4 makes heavy use of Intel's VT-x hardware virtualization. Each illegitimate memory or device access, be it via ordinary memory addressing, Memory-Mapped I/O (MMIO), or I/O ports, is intercepted by means of VT-x. In the same way, VT-x helps in blocking device misconfiguration attacks [Zho+12] and achieving user device isolation. Furthermore, seL4 uses Intel VT-d, also referred to as IOMMU, to protect against DMA attacks from misconfigured devices. Thus, seL4 is perfectly suitable to implement all of the above isolation guarantees.

3.4 Domain Binding

The SGXIO concept fundamentally relies on binding the SGX security domain with the trusted hypervisor domain. This section elaborates on the challenges that arise and how to solve them. Specifically, this covers trusted boot and hypervisor attestation. We discuss how to protect hypervisor attestation against remote TPM attacks as well as enclave virtualization attacks. Having a domain binding in place allows remote attestation of trusted paths as well as user verification.

3.4.1 Challenges

SGXIO enables a remote party as well as a local user to verify the security of trusted paths. In the first place, this requires a domain binding between SGX and the trusted hypervisor. In the second place, an appropriate user verification mechanism needs to be in place, which is both secure and usable.

Domain Binding. In order to bind the SGX and the hypervisor domain, the hypervisor must level up to certain security guarantees of SGX regarding secure code execution. In SGX, all enclave memory is isolated from the rest. Moreover, enclave loading is guarded by a verified launch mechanism, which can be attested to other parties. SGXIO rebuilds similar mechanisms for the hypervisor. Isolation of trusted hypervisor memory has already been discussed in Section 3.3.2. Verified launch is implemented via trusted boot of the hypervisor, with the help of the TPM. The TPM, in turn, allows hypervisor attestation.

With trusted boot and hypervisor attestation in place, SGXIO can bind the SGX and the hypervisor domain. The binding needs to be bidirectional, allowing both the hypervisor and SGX enclaves to put trust in the respective other domain. One direction is simple: The hypervisor can extend trust to SGX by running enclaves in a safe, hypervisor-protected environment. These enclaves can, in turn, use local attestation to extend trust to any other enclaves in the system, e.g., driver enclaves and, subsequently, user application enclaves. However, the opposite direction is challenging: On the one hand, user enclaves need confidence that the hypervisor is not compromised and binds user devices correctly to

driver enclaves. Effectively, this requires driver enclaves to invoke hypervisor attestation. SGXIO achieves this with the assistance of the TB enclave. On the other hand, SGX is not designed to cooperate with a trusted hypervisor. Recall that SGX considers all non-enclave code untrusted. In fact, SGX explicitly prohibits the use of any instruction that an enclave might misuse to communicate with the hypervisor [Int17b]. Even if hypervisor attestation succeeds, a driver enclave cannot easily learn whether it is legitimately executed by the hypervisor or virtualized by an attacker who intercepts the enclave's input and output. This makes both driver enclaves and the TB enclave vulnerable to virtualization attacks. SGXIO defends against such attacks by isolating hypervisor attestation from the untrusted operating system.

User Verification. An end user wants to ensure to indeed communicate with the correct user app. This is non-trivial because the user cannot directly evaluate a cryptographic attestation report. Instead, the user requires some form of notification whether a trusted path is present. This notification needs to be unforgeable to prevent the operating system from faking it. Moreover, it needs to help the user distinguish different user apps, not least because an attacker might also run arbitrary fake user apps.

3.4.2 Trusted Boot & Hypervisor Attestation

Trusted boot helps to verify the integrity of the hypervisor. Without it, malware could silently hook into the boot process and disable any protection offered by the hypervisor. As explained in Section 2.2.2, trusted boot makes use of a TPM to measure the whole boot process, starting from a trusted piece of firmware code up to the hypervisor image. Each boot stage measures the next one in a cryptographic log inside the TPM using the `extend` operation. The TPM cumulates all measurements in a Platform Configuration Register (PCR). The final PCR value reflects the whole boot process. If any boot stage deviates from the normal boot process, the PCR will contain a wrong value.

Hypervisor attestation allows enclaves to verify the trusted boot process (*i.e.*, the PCR value) in order to get assured of the hypervisor's integrity. Since the hypervisor is responsible for loading drivers and doing a trusted path setup, its attestation also vouches for the security of all trusted paths.

To ease hypervisor attestation, SGXIO uses a Trusted Boot (TB) enclave, which attests the hypervisor once. Afterward, any driver enclave running on the system can query the TB enclave to get approval if hypervisor attestation succeeded, see Figure 3.3. The driver enclaves in turn can communicate the attestation result to user apps, which can finally implement a mechanism for remote parties or the end user to verify a trusted path.

In order to attest the hypervisor, the TB enclave needs to verify the PCR value obtained during trusted boot. For this, the TB enclave requests a so-called TPM `quote` [TCG14], which contains a cryptographic signature over the PCR value, alongside a fresh nonce. The `quote` ensures not only the integrity of the PCR value but also prevents replay attacks.

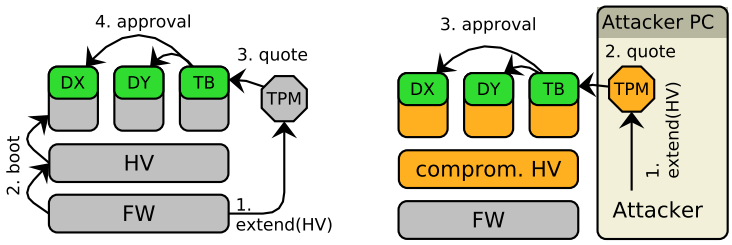

Figure 3.3: Hypervisor attestation (left): During trusted boot, the Firmware (FW) measures the Hypervisor (HV) via a TPM. The TB enclave attests the hypervisor via a TPM `quote` and, in case of success, approves other drivers (DX) and (DY). II) Remote TPM attack (right): An attacker injects a remotely-generated TPM `quote` to hide compromise of the hypervisor.

3.4.3 Attacks

The interaction between the TB enclave and the TPM is crucial for the security of the hypervisor attestation scheme. One has to prevent remote TPM attacks as well as enclave virtualization attacks, which are outlined in the following.

Remote TPM Attacks. If the TB enclave does not identify the TPM correctly, hypervisor attestation becomes vulnerable to remote TPM attacks, also called cuckoo attacks [Par08]. This is depicted in Figure 3.3 on the right side. If the attacker compromises the hypervisor image, the PCR will yield a wrong value during trusted boot, which is detected by the TB enclave. However, the attacker can make hypervisor attestation work again by diverting TB enclave communication to an attacker-controlled TPM. Since the attacker can feed the remote TPM at will to generate a valid `quote`, the TB enclave successfully approves the compromised hypervisor.

Defense. In order to stop remote TPM attacks, the TB enclave needs to be bound to a particular computer. In other words, the TB enclave needs a priori knowledge of the TPM, e.g., in the form of the TPM's Attestation Identity Key (AIK) used for signing the `quote`. Having the AIK enables the TB enclave to verify the origin of the TPM `quote`. To make the AIK known to the TB enclave, one has to provision it to the TB enclave, e.g., during initial system integration. The TB enclave persistently stores the provisioned AIK through SGX sealing. For hypervisor attestation, the TB enclave unseals the AIK and uses it to verify the `quote`. Since sealing uses a CPU-specific encryption key, an attacker cannot trick the TB enclave into unsealing an AIK not sealed by the same CPU. Thus, the TB enclave is effectively bound to the TPM.

It is the responsibility of system integrators to provision AIKs correctly. One has to introduce proper measures to prevent attackers from provisioning arbitrary AIKs. For example, the TB enclave could encode a list of public keys of approved system integrators, which are allowed to provision AIKs.

Enclave Virtualization Attacks. SGX is not designed to cooperate with a trusted hypervisor, making driver enclaves as well as the TB enclave vulnerable

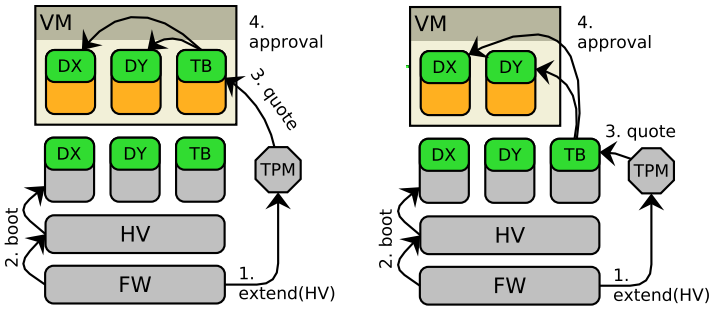

Figure 3.4: Enclave virtualization attacks: An attacker diverts steps 3 to a fake TB enclave (left) or step 4 to fake driver enclaves (right) running in a virtualized, attacker-controlled environment inside the VM.

to enclave virtualization attacks. In an enclave virtualization attack, the attacker does not compromise the actual trusted boot process. Instead, he virtualizes driver enclaves or even the TB enclave in a fake environment on the same computer, as depicted in Figure 3.4. To make hypervisor attestation for the virtualized enclaves succeed, the attacker diverts the legitimate TPM quote or the TB enclave approval to the virtualized TB enclave or driver enclaves, respectively. As shown in Figure 3.5, the attacker can now impersonate the user by rerouting user apps to a virtualized driver, reading the driver's output and providing fake input. Note that the attacker did not change any enclave code. Hence, SGX will generate the same `MRENCLAVE` value and, thus, the same derived cryptographic keys for both legitimate and virtualized enclave instances. Neither the TB enclave nor the driver enclave or a user app can detect such virtualization.

While enclave virtualization attacks compromise the authenticity of secure input, confidentiality is preserved. The attacker does not learn actual user input, which still arrives at the legitimate driver enclave. Conversely, enclave virtualization attacks break the confidentiality of secure output but are unable to violate authenticity. The legitimate driver enclave will refuse to render altered content from the attacker.

Defense. The problem of enclave virtualization stems from the design of SGX, which treats all enclaves equally, regardless of their execution context (VM, hypervisor, etc.). As a defense, SGXIO restricts the communication interface between the hypervisor and the operating system context (*i.e.*, the VM). Hence, the hypervisor hides the TPM as well as the TB enclave from the untrusted operating system. This breaks path 3 on the left and path 4 on the right side of Figure 3.4, respectively. The hypervisor gives TPM access only to the legitimate TB enclave. Thus, the TB enclave might only succeed in hypervisor attestation if the hypervisor has legitimately launched it. Likewise, the hypervisor only grants legitimate driver enclaves access to the TB enclave. A driver enclave might only get approval if it can talk to the legitimate TB enclave, which implies that the driver enclave too has been legitimately launched by the hypervisor.

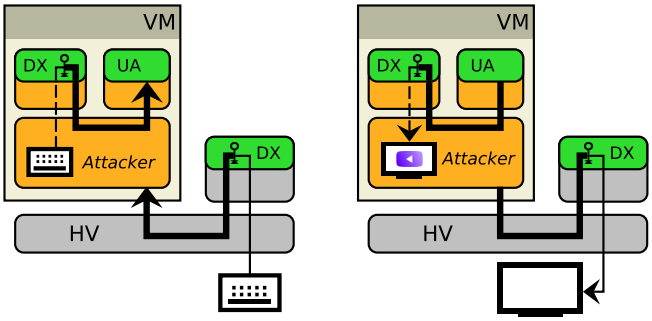

Figure 3.5: Virtualization of driver enclave DX can break the authenticity of secure input (left) and the confidentiality of secure output (right).

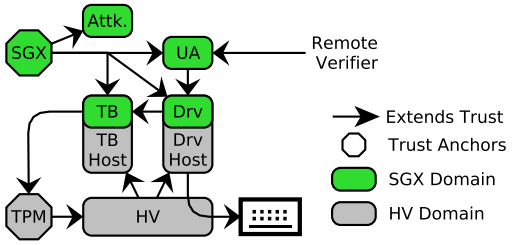

Figure 3.6: Trust hierarchy of SGXIO.

Note that user app enclaves are not subject to enclave virtualization attacks since they are already running in the operating system context and do not exchange sensitive plain data with their untrusted environment.

3.4.4 Remote Trusted Path Attestation

As already mentioned, hypervisor attestation vouches for the security of trusted paths and serves as a basis for remote attestation. This section describes the whole trust hierarchy involved in remote attestation, as shown in Figure 3.6.

SGXIO has two main hardware trust anchors, namely SGX and the TPM. SGX extends trust to all enclaves running on the system via verified launch. This also includes enclaves with attacker-controlled code (Attk.). It is up to a remote verifier and individual enclaves to build a trust hierarchy among "good" enclaves. To do so, the remote verifier can leverage SGX remote attestation and subsequent local attestation. The verifier checks not only the validity of an SGX attestation report but also the `MRENCLAVE` value, which uniquely identifies an enclave codebase.

In a typical scenario, a remote verifier wants to establish a trusted path to a user. It extends trust to a specific user app (UA) under its control, which in turn entrusts appropriate secure I/O drivers (Drv). Drivers extend trust to the TB enclave, which does hypervisor attestation, as previously outlined. If hypervisor

attestation succeeds, trust is implicitly extended to the TB host and all driver host processes, together with all trusted paths to user devices. If attestation fails at any point in the trust hierarchy, the affected entities will terminate trusted path attestation.

3.4.5 User Verification

SGXIO allows a user to assess whether a trusted path is present, *i.e.*, that all interaction is done with the correct user app. Verification does not require additional hardware, such as an external handheld verification device [Zho+12] or similar.

A common approach is to share a secret piece of information between the user and the user app, such as secret welcome messages [Ver] or personalized images [Mar+16]. For the sake of simplicity, we discuss the common scenario of a trusted screen path and a trusted keyboard path. When the user starts the user app, the user app requests a trusted input path to the keyboard and a trusted output path to the screen from the corresponding drivers, including hypervisor attestation. If, for any reason, one or both trusted path setups fail, the user app terminates with an error. In the case of success, the user app displays the pre-shared secret information via the screen driver to the user. The user verifies this information to get assured of a valid trusted path setup for this user app.

This approach requires provisioning of secret information to a user app, which seals it for later usage. Provisioning could be done once at installation time in a safe environment, e.g., with the assistance of the hypervisor, or at any time via SGX's remote attestation feature.

3.5 Key Transport

We present a novel fast non-interactive key transport scheme between two enclaves. It is based on SGX local attestation. First, we outline local attestation and traditional SGX key exchange. Second, we discuss our scheme in detail.

SGX local attestation uses the two instructions, namley `EREPORT` and `EGETKEY`, to generate and verify an attestation report, respectively. For the sake of simplicity, we omit some details in the following description. Furthermore, we denote `MRENCLAVE` of enclave X as X_{ID}, as it serves as an enclave identifier.

Algorithm 3.1: $\texttt{EGETKEY}_{E_{ID}}(key\_type)$

This SGX instruction generates secret keys for enclave E. This involves a CPU-specific secret key as well as subsidiary key attributes.

$key_{E_{ID}} \leftarrow derive\_key(cpukey, E_{ID}, key\_type, attributes)$
return $key_{E_{ID}}$

Local attestation involves a so-called report key that is bound to one specific enclave. As outlined in Algorithm 3.1, this key is derived from a CPU-specific

secret key, which ensures that attestation only succeeds on the same physical CPU. By choosing a key type, the enclave can not only request a report key but also sealing keys. To attest itself to a target enclave T, enclave E generates a report using `EREPORT`. As shown in Algorithm 3.2, this instruction first derives the report key for the target enclave. It then uses the report key to compute a signature over the current enclave's ID as well as enclave-provided data. The signature is essentially a message authentication code (MAC) computed with the AES-based CMAC algorithm [Son+06].

Algorithm 3.2: $\texttt{EREPORT}_{E_{ID}}(T_{ID}, data)$

This SGX instruction generates a report for enclave E, targeted at enclave T, containing additional user $data \in \{0,1\}^{256}$.

$key_{T_{ID}} \leftarrow derive\_key(cpukey, T_{ID}, \text{REPORT\_KEY}, attributes)$
$mac \leftarrow AES_{CMAC}(key_{T_{ID}}, E_{ID}||data)$
return mac

The whole local attestation procedure is outlined in Protocol 1. Enclave E wants to authenticate some data for enclave T. It first generates a report over this data and then sends its identity, the data plus the signature (*i.e.*, the mac) to enclave T. To verify the report, enclave T queries its own report key via `EGETKEY` and manually re-calculates the report signature. Only if both signatures match, enclave T will accept the report and, thus, consider the provided data as authenticated.

Protocol 1: Local Attestation

Enclave E attests itself to target enclave T, providing additional *data* to be authenticated.

E: $mac \leftarrow \texttt{EREPORT}_{E_{ID}}(T_{ID}, data)$
$E \rightarrow T$: $(E_{ID}, data, mac)$
T: $key_{T_{ID}} \leftarrow \texttt{EGETKEY}_{T_{ID}}(\text{REPORT\_KEY})$
$mac2 \leftarrow AES_{CMAC}(key_{T_{ID}}, E_{ID}||data)$
if $mac = mac2$ **then**
  accept report
else
  reject report

To open a cryptographic channel between two enclaves, they need to exchange a cryptographic key. For that purpose, the SGX SDK provides a Diffie-Hellman key exchange. Both enclaves generate a private nonce and compute their public counterpart. In order to avoid man-in-the-middle attacks, the enclaves authenticate their public Diffie-Hellman value towards each other via local attestation. Hence, they can safely compute a shared key.

Our Non-Interactive Key Transport Scheme. The key exchange provided by the SGX SDK is a straight forward application of Diffie-Hellman, which involves expensive exponentiation of big numbers. Using elliptic curve Diffie-Hellman could speed up computation. Nevertheless, we observed that this key

exchange could be completely avoided, as follows: SGX local attestation is a purely symmetric signature scheme using the same key for signature creation and verification. Hence, local attestation already provides a shared symmetric secret between enclaves, namely the report key. Any enclave can sign reports for a target via the `EREPORT` instruction. Although it has no direct access to the target enclave's report key, it can indirectly use the report key by issuing `EREPORT`. In turn, the target enclave can access its own report key via `EGETKEY`. We use this symmetry of report keys to derive fresh encryption keys.

Our key transport scheme is outlined in Protocol 2. To establish a new shared key, enclave A chooses a random nonce and generates an attestation report over this nonce, targeted at enclave B. However, A never transmits this report to B but uses the report's MAC as a 128-bit shared symmetric encryption key. Instead, enclave A sends its identity as well as the nonce to B, which can query its report key and re-calculate the MAC to obtain the shared key.

Protocol 2: Non-interactive, symmetric key transport

Enclave A sends a fresh symmetric key to enclave B.

A: $nonce \xleftarrow{R} \{0,1\}^{256}$ chosen uniformly at random
$mac \leftarrow \texttt{EREPORT}_{A_{ID}}(B_{ID}, nonce)$
$key_{AB} \leftarrow mac$
$A \rightarrow B$: $(A_{ID}, nonce)$
B: $key_{B_{ID}} \leftarrow \texttt{EGETKEY}_{B_{ID}}(\text{REPORT\_KEY})$
$key_{AB} \leftarrow AES_{CMAC}(key_{B_{ID}}, A_{ID}||nonce)$

Our scheme is non-interactive since it only involves a single unidirectional transmission of the nonce. It has zero overhead since local attestation is supported by SGX hardware. The only noteworthy enclave code, namely the AES-CMAC implementation, is typically already part of an enclave codebase for doing local attestation. Moreover, the scheme supports authentication. On the one hand, enclave A binds the key to enclave B through local attestation. On the other hand, enclave B knows the identity of the enclave for which it is deriving a shared key. To also achieve liveness, enclave B could send the encrypted nonce back to A.

3.6 Tweaking Debug Enclaves

Our architecture makes heavy use of SGX enclaves. In order to enable full security of production enclaves, SGX enforces a licensing scheme on enclave code, as outlined in Section 2.3.2. In this section, we show how to level up debug enclaves to behave like production enclaves in the threat model of SGXIO. This requires special handling of SGX remote attestation and sealing. Note that the debug tweak does not apply to a secure cloud computing scenario with an untrusted cloud provider.

The only difference between debug and production enclaves is the presence of SGX debug instructions (`EDBGRD` and `EDBGWR`), which we aim to disable manually. The debug tweak leverages SGX's support for VT-x instruction interception. VT-x supports several configurable bitmaps, making selected instructions trap into the hypervisor when executed from within a VM. The hypervisor configures these bitmaps via the so-called Virtual Machine Control Structures (VMCS). With the release of SGX, Intel added a new bitmap for SGX ENCLS instructions, called ENCLS-exiting bitmap. This bitmap allows the hypervisor to intercept ENCLS instructions selectively. Thus, the hypervisor can intercept all `EDBGRD` and `EDBGWR` instructions that are ever executed from within a VM, by just configuring the VMCS bitmaps accordingly. By doing so, the trusted hypervisor is the only code that can debug enclaves. Since the hypervisor is trusted, we can consider SGX debugging features as disabled. Hence, we have effectively turned all debug enclaves inside the VM into production equivalents.

Tweaked Cloud Enclaves. As already mentioned, this tweak only applies to a setting similar to SGXIO, where a trusted hypervisor is present. In general, this is not the case for cloud scenarios where the cloud provider is untrusted and expected to subvert the hypervisor. In such cases, one has to opt for real production enclaves. Nevertheless, honest server administrators could use the tweak to obtain SGX protection without licensing. This would help in strongly isolating server code and reducing the TCB from the whole system down to the hypervisor and the enclave code.

Remote Attestation. Tweaked debug enclaves require special care during remote attestation. A remote verifier cannot easily determine whether the debug tweak is correctly enabled or not. For example, an attacker could compromise the hypervisor and manipulate (*i.e.*, debug) the TB enclave to issue wrong approvals. Next, the attacker could stealthily debug all enclaves on the system.

To do remote attestation with tweaked debug enclaves securely, one can run only the TB enclave in production mode and do remote attestation towards it. Once a remote party verified the TB enclave, it can be sure that the hypervisor correctly enforces the tweak for all debug enclaves in the system.

Sealing. Both non-tweaked and tweaked debug enclaves share the same sealing keys. This is no problem unless an attacker manages to compromise the hypervisor and disables the tweak. Although hypervisor attestation would fail in that case, the attacker would be able to extract all sealing keys by simply debugging all enclaves. To prevent this, one can delegate sealing key derivation to the TB enclave. The TB enclave, running in production mode, only derives actual sealing keys if hypervisor attestation succeeds.

Local Attestation. Similar to sealing, local attestation for tweaked debug enclaves is not straight forward. If an attacker retrieves report keys from a tweaked debug enclave, e.g., during a compromised system boot, all further usage of these report keys is insecure. This prohibits enclaves from attesting tweaked debug enclaves. However, a tweaked debug enclave can still do local attestation towards the TB enclave. Since the TB enclave is running in production mode, the attacker cannot compromise its keys.

3.7 Further Considerations

This section touches on advanced topics potential users of SGXIO should be aware of, namely driver complexity and side channels.

Driver Complexity. Depending on the bus protocol, driver design might be challenging. Especially multiplexed buses, such as USB, are non-trivial to deal with. One has to identify proper policy rules which guarantee a trusted path. Zhou et al. demonstrate how to establish a trusted path to one specific USB device while keeping other USB devices accessible to the untrusted operating system [ZYG14]. To deal with the complexity of the USB driver stack, they identified all security-relevant parts and either moved them entirely into a trusted domain or at least verified their results. All non-critical operations are kept in the untrusted domain. This approach would, in principle, also be supported by SGXIO with a cooperative design of the operating system and secure I/O drivers.

SGX Side Channels. The threat models of SGX and SGXIO do not consider side channels [Int17b]. SGX enclaves are vulnerable to side channels, and additional precautions need to be taken, as will be explained in Part II. Also, the cryptographic channel between drivers and user app enclaves might leak side-channel information, such as keystroke timings. One could send a constant encrypted data stream to avoid such leakage, as done by [Esk+19].

3.8 Summary

In this chapter, we presented SGXIO, the first SGX-based architecture that supports generic trusted paths. We augmented SGX with a small and trusted hypervisor for setting up a generic trusted path while using SGX for protecting user apps from an untrusted operating system. We solved the challenge of combining the security domains of SGX and the hypervisor. We did so by attesting the hypervisor with the assistance of a TPM towards a special trusted boot enclave, which is bound to the local computer. Special care must be taken to prevent enclave virtualization attacks. With SGXIO, both a remote party and a local user can verify the security of trusted paths.

From an application developer's perspective, SGXIO benefits from SGX's programming model. One can integrate sensitive enclaves directly in existing, untrusted applications and open a trusted path through virtual device I/O. Furthermore, we showed how SGXIO could omit enclave licensing by making debug enclaves behave like production enclaves. In order to achieve this, the trusted hypervisor disables SGX debugging instructions for the untrusted operating system.

SGXIO demonstrates that SGX is not limited to cloud computing and DRM scenarios. It addresses user-centric application security, making generic trusted paths available to SGX enclaves and, thus, protecting against kernel-level key-loggers and screen loggers, for example. SGXIO is compatible with unmodified legacy operating systems and designed for off-the-shelf notebooks.

4

TIMBER-V: Enclaves via Tagged Memory

To be trusted is a greater compliment than being loved.
George MacDonald

Embedded computing devices are used on a large scale in the emerging Internet of Things (IoT). Due to its openness and extensibility, the RISC-V architecture is a particularly promising candidate for wide deployment in the IoT. As IoT devices are built for long service life, means are required to protect sensitive code *in the presence of potential vulnerabilities*, which might be discovered long after deployment. Isolation technologies such as enclaves are a perfect fit for these scenarios. Unfortunately, enclaves for small, constrained RISC-V processors are largely unexplored. This chapter brings enclave technology to small, embedded RISC-V processors utilizing tagged memory.

Research on enclaves has matured over the past years, as we have seen in Section 2.2. The initial focus was on larger systems featuring virtual memory [Suh+03; TLL06; Int15; McC+08; Che+08; YS08; CL10; McC+10; Chh+11; ANZ11; Vas+12; Owu+13; Hof+13; CDA14; Arm09; CL10; WB11; BW12; McK+13; Evt+14; CLD16]. Later on, enclaves were brought to small embedded devices as well [Eld+12; Noo+17; Göt+15; Koe+14; Bra+15; Arm17]. Those devices typically employ MPU-based access protection on physical memory. However, small devices are often resource-constrained and suffer from poor memory utilization due to memory fragmentation and inefficient isolation mechanisms. Tighter integration of trusted memory in the limited physical address space would demand more fine-grain isolation boundaries, which existing schemes either do

not provide or only provide at the expense of high management overhead. Also, more flexibility is beneficial for the dynamic management of trusted memory.

A technique that has the potential to offer fine-grained and flexible isolation boundaries is *tagged memory*. Tagged memory transparently associates blocks of memory with additional metadata. It has been used for dynamic information flow tracking [Suh+04] as well as access control [WCA02] and is still an active subject of research [Son+16; Joa+17] and development [Arm19]. While tagged memory has been shown to support a variety of security policies like protection of control data [CWC06], pointers [Dev+08] or capabilities [Wat+15], strong, efficient, and flexible isolated execution is still an open problem for small embedded systems. In particular, data flow isolation [Son+16] cannot provide strong isolation since tags can be destructively written by untrusted software. Other existing solutions are not appropriate for low-end embedded devices due to their memory overhead stemming from large tags [Zel+08] or fully programmable but expensive tag engines [Ven+08; Che+09; DS12; Dha+15]. Hence, currently, no existing tagged memory schemes support efficient enclave isolation on small embedded devices.

In this chapter, we propose TIMBER-V: Tag-Isolated Memory Bringing fine-grained Enclaves to RISC-V. TIMBER-V targets low-end RISC-V processors in a hardware-software co-design. On the hardware side, we achieve fine-grained in-process isolation with only two tag bits, thus maintaining low memory overhead. Moreover, we combine tagged memory with a memory protection unit (MPU) to support an arbitrary number of processes while avoiding the overhead of large tags [Zel+08]. On the software side, we enforce isolation via a small trust manager, called TagRoot. We isolate privileged from unprivileged security domains, supporting both Intel SGX enclaves [McK+13] and the TrustZone [Arm17] programming model. However, TIMBER-V comes with much finer isolation granularity and more efficient memory utilization, which has several advantages. On the one hand, data locality can be maintained by interleaving trusted and untrusted memory, thus minimizing memory fragmentation. On the other hand, TIMBER-V uses a tag update policy that allows highly flexible dynamic memory management of trusted data. Dynamic memory support has been announced for the upcoming Intel SGXv2 but involves costly interaction with the operating system [Int16a]. In contrast, TIMBER-V enclaves can instantaneously claim memory by using a single instruction.

To demonstrate these advantages, we show heap interleaving and a novel stack interleaving scheme. That is, we use a single heap and stack across different security domains while maintaining strong isolation. Moreover, we demonstrate highly efficient inter-enclave communication over secure shared memory. TIMBER-V supports real-time constraints by making all trusted software interruptible. It is furthermore compatible with existing code. We implement and benchmark TIMBER-V on the RISC-V Spike simulator. We evaluate TIMBER-V under different CPU models, which highlights characteristics of TIMBER-V rather than CPU implementation specifics. We show that the runtime overhead of TIMBER-V is 25.2% for naive implementations, while tag caching reduces the overhead to 2.6%. We have made TIMBER-V open source.

Contributions. In summary, our main contributions are:

- We propose TIMBER-V, the first efficient tagged memory architecture for enclaves on low-end processors
- We present a novel concept called stack interleaving that allows for efficient and dynamic memory management
- We propose lightweight shared memory between enclaves
- We propose an efficient shared MPU design
- We extensively evaluated our proof-of-concept implementation[1] on the RISC-V simulator for different CPU models

This chapter is based on the publication [Wei+19b], of which I am the main author. The rest of this chapter is structured as follows: Section 4.1 gives background information on RISC-V and tagged memory. Section 4.2 specifies our adversary model and design goals. Section 4.3 presents our TIMBER-V design. Section 4.4 explains our trust manager. Section 4.5 discusses dynamic memory management for TIMBER-V. Section 4.6 gives implementation details, and Section 4.7 analyzes the security of TIMBER-V. Section 4.8 evaluates TIMBER-V. Section 4.9 and Section 4.10 discuss related work and possible extensions of TIMBER-V. We summarize in Section 4.11.

4.1 Background

This section gives background information about RISC-V and tagged memory.
RISC-V. RISC-V [WA17a] is an open and extensible instruction set architecture and defines three privilege modes [WA17b], namely machine-mode (M), supervisor-mode (S), and user-mode (U). M-mode has the highest privileges and is used for emulating missing hardware features. S-mode and U-mode are meant to run an operating system and user applications, respectively.
Tagged Memory. The idea behind tagged memory is to extend each memory word with additional bits that store metadata. The concept of tagged memory is very old and can already be found in early computer designs [Feu72]. In these designs, tag bits were used for debugging or dynamically tracking of the data type. Recent commercially available computer architectures hardly support hardware-based tagged memory. Instead, one uses software-based tagging for dynamic analysis tools [Ser+12; SS15; SAB10]. However, recent research on tagged-memory architectures in the system security context [Son+16; Dha+15; BFM14] hints that re-establishing hardware-support can considerably improve security. In fact, Arm announced their memory tagging extensions in 2019 [Arm19].

[1] The source code is available at `https://github.com/IAIK/timber-v`

4.2 Adversary Model and Design Goals

TIMBER-V is designed for stakeholders who want to securely execute pieces of code on a small IoT device. However, the stakeholders distrust the IoT device for various possible reasons. First, the device's operating system might not sufficiently isolate individual tasks to guarantee secure code execution, as is the case for the popular FreeRTOS kernel, for example.[2] Second, even if the operating system provides sufficient task isolation, it might be subject to exploitation, circumventing all isolation guarantees [Iga15]. Third, the operating system might be controlled by a party that the stakeholders distrust and want to protect intellectual property against. We consider the strongest attacker to have complete control over the operating system. Thus, the attacker can not only deny service but also use the system's security features to spawn malicious enclaves in an attempt to subvert benign ones, as depicted in Figure 4.1. However, we assume that benign enclaves are properly protected against direct exploitation via runtime attacks using concepts like memory safety [AHP18], for example. A proper tag isolation architecture shall guarantee the security of benign enclaves in the presence of such attacks. We assume that cryptographic primitives are secure. We do not address physical attacks. The trusted computing base consists of the hardware, including the hardware emulation mode (M-mode), as well as a small trust manager (TagRoot) and the benign enclaves themselves.

TIMBER-V does not prevent software side-channel attacks. While memory interleaving provides the untrusted software with additional information about enclave's memory allocations, an enclave that follows the constant-time paradigm [Ber05; Cop+09] is secure against address-based side-channel attacks.

We demand that a tagged memory architecture designed for enclaves shall meet the following design goals:

G1 Security. It shall guarantee that sensitive code can leverage strong isolation to maintain the confidentiality and integrity of its sensitive data. This demands (i) strong memory isolation, (ii) secure entry points, (iii) secure communication, and (iv) attestation and sealing.

G2 Flexibility. It shall be flexible with respect to fine-grained and dynamically reconfigurable isolation boundaries as well as the programming model.

G3 Compatibility. Untrusted code shall run without modification to support existing operating systems and apps.

G4 Low Overhead. It shall minimize the cost of tagged memory as well as the performance overhead of switching security domains.

G5 Real-time. It shall support hard real-time constraints.

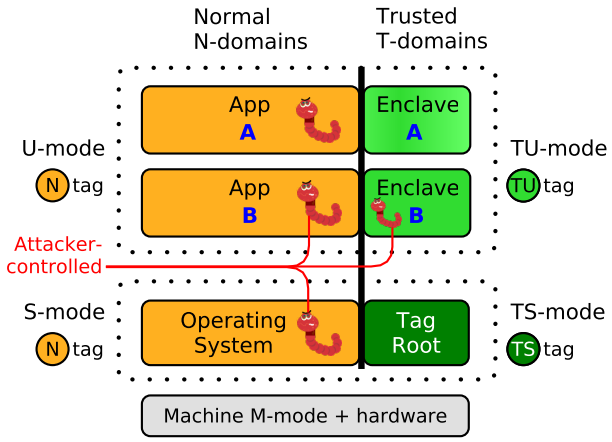

Figure 4.1: TIMBER-V supports four security domains. TIMBER-V extends user apps running in U-mode and the operating system running in supervisor S-mode with trusted memory, namely TU-mode for enclaves and TS-mode for TagRoot. User processes **A** and **B** integrate trusted enclave memory within untrusted apps. The attacker controls all software in the N-domains and can run malicious enclaves (cf. enclave B).

4.3 TIMBER-V Design

TIMBER-V is a novel tagged memory architecture that achieves lightweight, yet powerful enclave isolation on small embedded processors. Specifically, we achieve fine-grained and dynamic in-process isolation. TIMBER-V follows a hardware-software co-design. On the hardware side, TIMBER-V uses tagged memory for enforcing a strong and fine-grained isolation policy and for providing fast domain switches. We augment tagged memory with a Memory Protection Unit (MPU) for lightweight isolation between processes. Dedicated tag instructions allow flexible dynamic memory management. For example, we demonstrate memory interleaving across security domains not only for heap memory but also for stack memory. On the software side, TIMBER-V delegates policy enforcement to a small privileged trust manager called TagRoot, which provides various trusted services to the operating system and to enclaves.

4.3.1 Enclave Isolation

TIMBER-V supports four security domains, as depicted in Figure 4.1. The operating system and apps live in the "normal" N-domains, which are considered untrusted. The N-domains support the traditional split between user mode (U-mode) and supervisor mode (S-mode) and allow existing code to run without modification (goal **G3**). We protect sensitive memory via fine-grained memory

[2]FreeRTOS allows elevating privileges via the prvRaisePrivilege syscall.

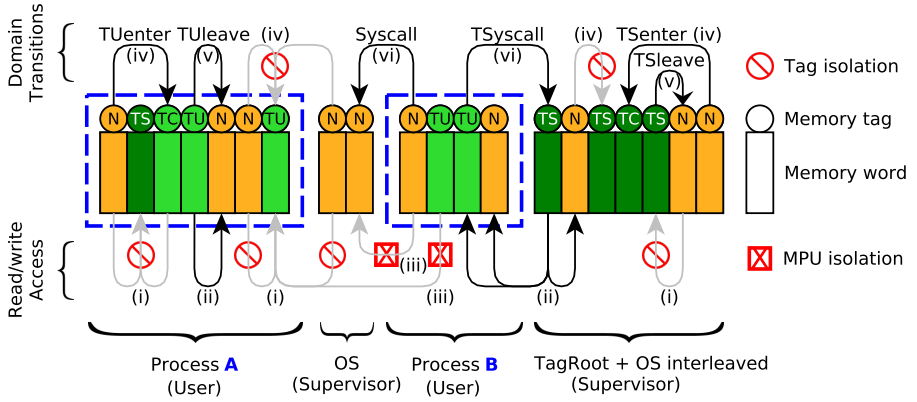

Figure 4.2: Security domains are interleaved in flat physical memory. Tag isolation protects T-domains while MPU isolation encapsulates and protects individual processes across domains. T-domains can only be entered at trusted callable entry points (TC-tag), which allows fast domain transitions.

tagging, which creates islands of trusted memory inside the N-domains. Trusted user mode (TU-mode) can be leveraged for enclaves. Moreover, trusted supervisor mode (TS-mode) allows running a trust manager like TagRoot, augmenting the untrusted operating system with trusted services.

TIMBER-V combines security domain isolation with an MPU. It protects individual processes or enclaves via *MPU isolation*. The trusted domains are protected by a strict tagged memory policy, which we denote as *tag isolation*. Memory accesses are only permitted if *both* mechanisms "agree". Thus, TIMBER-V supports a variety of different programming models, as demanded by goal **G2**. For example, we achieve TrustZone's [Arm17] security split via memory tags; however, with much finer and highly dynamic isolation boundaries. Also, TIMBER-V can embed enclaves directly in user processes, as done in Intel SGX-like designs [McK+13], however, again with the benefits of tagged memory.

Tag Isolation. Tag isolation is depicted with arrows in Figure 4.2 and will be discussed in detail in Section 4.6. TIMBER-V uses a two-bit tag per 32-bit memory word for fine-grained protection of trusted memory (goal **G2**). Having only two tag bits keeps the hardware cost of tagged memory low. This supports goal **G4** and, at the same time, retains the advantages of fine-grained memory isolation. With two tag bits we encode four different tags, namely N-tag, TU-tag, TS-tag, and TC-tag. We use them to identify untrusted memory (N-tag), trusted user memory (TU-tag), trusted supervisor memory (TS-tag) as well as secure entry points via the trusted callable TC-tag.

At every memory access, a hardware tag engine ensures that untrusted code cannot access trusted memory, see Figure 4.2 (i). Moreover, enclaves (TU-tag) cannot access trusted supervisor memory (TS-tag) used for TagRoot. In contrast, trusted domains can access lesser trusted memory (ii), as long as the MPU

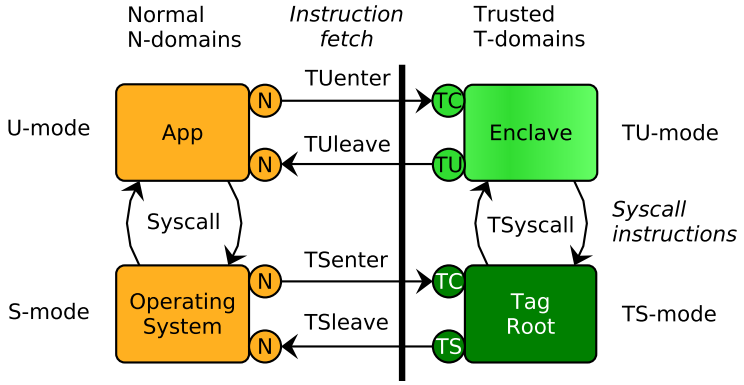

Figure 4.3: TIMBER-V supports horizontal transitions between normal and trusted domains via ordinary instruction fetches, as well as vertical transitions via syscall instructions.

isolation policy allows it. Finally, tag isolation could be directly applied to other peripherals, e.g., preventing DMA accesses to trusted memory.

MPU Isolation. Tag isolation enforces the protection of security domains. However, an embedded system typically runs several independent processes within the *same* security domain. Relying on tag isolation for process isolation would require large tags, which is unacceptable for our goal **G4**. Instead, TIMBER-V isolates individual processes via a memory protection unit (MPU) (see dashed boxes and arrows (iii) in Figure 4.2). This minimizes tagging overhead while supporting fine-grained in-process isolation.

Fast and Secure Domain Transitions. Our system distinguishes horizontal and vertical domain transitions, as visualized in Figure 4.3. Both need to be fast and efficient to achieve goal **G4**. We implement domain transitions without introducing new instructions, thus maintaining compatibility (goal **G3**). Horizontal transitions switch between N and T-domains while maintaining the current privilege mode. In order to avoid code-reuse attacks, trusted domains can only be entered at secure entry points (cf. goal **G1**). Entry points are marked as trusted callable with the TC-tag and are denoted as "TUenter" and "TSenter", depending on the caller's privilege mode. See also (iv) in Figure 4.2. Whenever the CPU fetches an instruction tagged with TC-tag, it switches to the trusted security domains. Likewise, when fetching normal N-tag memory, the CPU switches back to the normal N-domains, leaving trusted execution, denoted as "TUleave" and "TSleave" (v), respectively. More details about how TUenter and TSenter are protected will be discussed in Section 4.6.

Unlike SGX, which involves costly checks in microcode for each domain switch [Int16a], our design imposes zero runtime overhead. Unlike TrustZone-

M [Arm17], it allows even faster and very compact transitions, keeping code locality and compatibility to the maximum extent possible.[3]

Vertical transitions are, in fact, syscalls that transition between user modes and supervisor modes, as seen in Figure 4.3 and (vi) in Figure 4.2. In the N-domains, apps can issue syscalls to the operating system. Likewise, in the T-domains, enclaves can request TagRoot services via trusted TSyscalls. When finished, a syscall or TSyscall can return to the calling app or enclave, respectively. On RISC-V, syscalls are issued with the `ecall` instruction and finished with the `sret` instruction. To cleanly separate vertical transitions, TIMBER-V adds a separate trusted syscall (trap) handler.

MPU Sharing. TIMBER-V shares a single MPU between the N-domains and the T-domains. That is, we can use the same MPU slots for processes executing in U-mode and TU-mode. Thus, TIMBER-V supports not only traditional apps and secure enclaves but also mixed processes, as shown in Figure 4.2. In contrast to using two separate MPUs, our approach reduces hardware and energy costs since fewer MPU slots are required. In order to maintain compatibility (goal **G3**), the operating system can always access and update shared MPU slots. Any such updates are detected by the MPU, which then prevents enclaves from using the updated slots until TS-mode validates the changes. To do so, we augment the MPU with just two additional flags.

4.3.2 Dynamic Memory Management

TIMBER-V supports a highly flexible management of trusted memory. For this, we add new tag-aware instructions. Usage of these instructions is optional, and normal memory-related instructions also adhere to the tag isolation policy. Nevertheless, tag-aware instructions help implement dynamic memory interleaving as well as a simple but effective code hardening transformation.

New Tag-aware Instructions. TIMBER-V adds new checked memory instructions that allow fine-grained and dynamic management of trusted memory. We call them "checked" instructions since they augment ordinary memory instructions adhering to tag isolation with one additional programmable tag check. This additional tag check does not bypass our tag isolation policy but tightens it by constraining memory accesses to a specific security domain. For example, when enclaves process untrusted data, they can use checked instructions to prevent accidentally accessing the wrong security domain.

Tag Update. In addition to the tag checks, checked store instructions allow (de)privileging memory by changing memory tags, as follows: Tags can only be updated within the same or a lower security domain. Checked store instructions cannot be used to elevate privileges. As shown in Table 4.1, TS-mode (and M-mode) have full access to all tags. TU-mode can only change tags between N-tag

[3] For example, one could split an unmodified program binary into untrusted and trusted parts by mere tagging, that is, without the need for changing code or the memory layout. However, in practice, one typically augments the program with secure argument passing, stack handling, and register cleanup.

and TU-tag to support dynamic interleaving of user memory. We prevent TU-mode from manipulating TC-tags, which are reserved for secure entry points. Our tag update policy makes isolation boundaries flexible during runtime (goal **G2**).

Dynamic Memory Interleaving. Checked memory instructions allow to dynamically claim memory across security domains, thus maintaining data locality and reducing management overhead. For example, an enclave can claim untrusted memory during runtime by setting its tags from N-tag to TU-tag. We show that this allows heap interleaving as well as a novel code transformation that we call stack interleaving. That is, an enclave does not need to maintain a separate secure heap or stack. In general, dynamic memory interleaving can help reduce memory requirements to a single heap and a single stack per execution thread. This has not only operational advantages, such as reduced memory fragmentation and, thus, reduced memory consumption; it also improves overall security. One can remove code, which is normally necessary for dynamic memory management, from the trusted computing base (TCB).

Code Hardening Transformation. Checked instructions can be used for additional code hardening against code-reuse attacks. In these attacks, one misuses existing code to perform malicious actions, e.g., leak secrets from trusted to untrusted domains. In contrast, normal code execution usually operates in a single security domain, and this security domain predetermines all accessed memory tags. Our code hardening transformation enforces this property by replacing memory instructions with checked instructions. They check every memory access for the correct tag and, thus, restrain code execution to the current security domain. Only code interacting with untrusted memory on purpose is left unmodified. As discussed later, this transformation adds negligible performance overhead. We apply it to enclaves and TagRoot as an additional layer of defense.

4.3.3 Trusted Services

We provide a small trust manager, called TagRoot. It serves as the trust anchor for bootstrapping secure enclaves and maintaining their isolation properties, as demanded by goal **G1**. TagRoot offers trusted OS services to the untrusted operating system as well as trusted enclave services to the enclaves themselves. Trusted OS services include enclave management, secure entry points, attestation, and sealing. Moreover, in contrast to existing solutions, TagRoot supports fast

Table 4.1: Tag update policy, permitting (✓) or refusing (✗) tag updates from certain security domains.

Can update tag	N-tag	TC-tag	TU-tag	TS-tag
N-domains	✓	✗	✗	✗
TU-mode	✓	✗	✓	✗
TS-mode	✓	✓	✓	✓
M-mode	✓	✓	✓	✓

enclave-to-enclave communication via secure shared memory. It imposes zero copying overhead and allows m:n connectivity. TagRoot and enclaves are fully interruptible, thus meeting goal G5.

Enclave Life Cycle. TIMBER-V enclaves are created and loaded within an ordinary user process at the discretion of the operating system but with the assistance of TagRoot. Once loaded, enclaves can be directly invoked by user apps to carry out security-critical tasks. For freshly generated enclaves, one typically provisions secret data, such as cryptographic keys, to the enclave via a secure remote channel. This channel is authenticated using enclave attestation with the assistance of TagRoot. During their lifetime, enclaves can authenticate and communicate with other enclaves or seal sensitive information for keeping state across reboots, again with the help of TagRoot.

Possible Extensions. Independently of our TagRoot design, TIMBER-V supports other services as well. For example, trusted I/O is straight forward by tagging I/O memory as trusted. Depending on the requirements, I/O memory could either be hard-wired to a particular tag, assigned a tag via memory ranges, similar to a secure attribution unit [Arm17], or be fully configurable. Latter requires additional tag storage, either in RAM or in a cache.

Also, TagRoot could implement services demanding availability, thus realizing safety-critical systems, for example. However, since these additional services enlarge the TCB, we did not implement them in our current prototype. We discuss different design options in Section 4.10.

4.4 TagRoot Trust Manager

We develop a small trust manager for TIMBER-V, called TagRoot. It runs in trusted supervisor mode (TS-mode) and offers privileged trusted services to the untrusted operating system as well as unprivileged trusted services to enclaves. All trusted services are listed in Table 4.2.

Table 4.2: TagRoot trusted OS and enclave services.

Trusted OS services	Trusted enclave services
`create-enclave(ecb)`	`get-key(id)`
`add-region(ecb, region)`	`shm-offer(targetEID, region)`
`add-data(ecb, region)`	`shm-accept(ownerEID)`
`add-entries(ecb, entries)`	`shm-release(region)`
`init-enclave(ecb)`	
`load-enclave(ecb)`	
`destroy-enclave(ecb)`	
`resume()`	

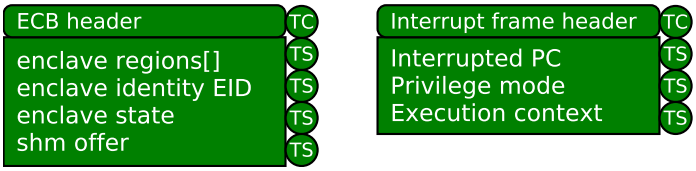

Figure 4.4: TagRoot trusted metadata includes enclave control blocks (ECB) and interrupt frames with unforgeable headers.

4.4.1 Trusted OS Services

The operating system can invoke trusted OS services via TSenter (see Figure 4.3 and (iv) in Figure 4.2). These services provide enclave management such as creation and cleanup, loading as well as handling of interruption. Also, during enclave creation, the sequence of trusted OS service calls define the enclave's identity similar to `MRENCLAVE` for SGX enclaves. This identity is used for subsequent trusted enclave services.

Creation and Cleanup. TagRoot stores enclave metadata in a data structure called enclave control block (ECB). When instantiating a new enclave with `create-enclave`, TagRoot first creates a new ECB. The operating system can choose where this ECB is placed in its memory. TagRoot will re-tag this memory with TS-tag. Hence, an ECB is always kept in secure TS-mode memory, as shown in Figure 4.4.

Once TagRoot created an enclave's ECB, the operating system can add memory regions (*i.e.*, contiguous chunks of memory) to it via `add-region`. These enclave regions will be loaded in the MPU when the enclave is about to run. TagRoot ensures that enclave regions will never overlap with other enclaves but are unique to each enclave. As mentioned before, an enclave region can cover app memory as well. Thus, a single shared MPU region can hold enclave data and app data.

Initially, an enclave region is untrusted and tagged with N-tag. In order to add enclave data, the operating system repeatedly executes `add-data`. TagRoot claims enclave memory by setting TU-tag, as long as the claimed memory is within the enclave's regions. `add-data` works on a word granularity, thus supporting fine-grain memory interleaving. All claimed memory (TU-tag) constitutes the actual enclave (TU-mode), while the rest (N-tag) constitutes the untrusted app (U-mode) (cf. processes in Figure 4.2). While the enclave can access its app counterpart, the opposite direction is prohibited by the tag isolation policy.

Similar to enclave data, the operating system can announce entry points by a call to `add-entries`. TagRoot will mark all entry points with TC-tag, given that they belong to the enclave's regions. Finally, a call to `init-enclave` will cause TagRoot to compute a cryptographic identity over the enclave and mark it as runnable. We discuss this cryptographic identity at the end of this section.

Once the enclave is initialized, it is immutable and cannot be altered using the above trusted service calls. Initialized enclaves can only be loaded, resumed,

or destroyed. At the end of an enclave's life cycle, a call to `destroy-enclave` will unload and invalidate the ECB, preventing the enclave from further execution. TagRoot clears all claimed enclave memory, releases enclaves regions, and clears up ECB memory. It also reverts all enclave tags to N-tag. The operating system can now repurpose the freed enclave memory for other tasks.

Loading enclaves. In order to run an enclave, the operating system first loads the enclave regions into the MPU and then calls `load-enclave`. If another enclave is currently loaded, TagRoot unloads it by invalidating stale enclave MPU slots. Next, TagRoot validates the current MPU configuration, as configured by the operating system. It acknowledges all updated MPU slots that correspond to the enclave. Moreover, TagRoot locks the enclave's ECB to prevent further modifications and restores its runnable or interrupted state in a special register, called `STSTATUS`. The app can now enter the loaded enclave by calling one of its entry points (TUenter). In case of interruption, the app can resume the enclave.

Interruptibility. Trusted code execution is fully interruptible except for a small trusted interrupt handler. Interruptibility is necessary to support real-time tasks reacting to external I/O events or control loops that need to run periodically in order to meet stability criteria, for example. Whenever an interrupt happens during enclave execution, TIMBER-V raises a special "interrupted" CPU flag that prevents re-entering the enclave. The CPU then invokes the trusted trap vector of TagRoot. TagRoot saves the current enclave's execution context in a protected interrupt frame (see Figure 4.4) and erases sensitive CPU registers to avoid accidental leakage of sensitive data. Moreover, it sets the interrupted program counter to a dedicated `resume` function, before giving control to the operating system. When the operating system returns from interrupt handling, `resume` gets executed. TagRoot restores the enclave execution context, clears the "interrupted" CPU flag, and resumes enclave execution. This process is completely transparent and requires no changes to the operating system. Moreover, it also supports the interruption of TagRoot (TS-mode) while processing trusted service calls.

Enclave Identity. From a functional perspective, enclaves are defined by their code base and initial data as well as their entry points. To capture this, all trusted OS service calls from `create-enclave` to `init-enclave` contribute to a continuous SHA256 computation, called measurement. The measurement involves not only the sequence of trusted service calls but also its parameters, that is, enclave regions, data as well as entry points. `init-enclave` stores the final measurement as enclave identity (EID) inside the ECB and marks the enclave state as runnable (see Figure 4.4). Since the EID is immutable, it reliably identifies enclaves. This concept is similar to MRENCLAVE in SGX [Int16a]. Enclave identities are used for trusted enclave services.

4.4.2 Trusted Enclave Services

Enclaves can request trusted enclave services via TSyscalls (see Figure 4.3 and (iv) in Figure 4.2). This includes sealing, attestation, and inter-enclave communication via shared memory.

Sealing and Remote Attestation. An enclave can call `get-key` to generate enclave-specific cryptographic keys. Similar to SGX enclaves (see Algorithm 3.1), those keys are derived from the enclave identity (EID) and a secret platform key K_p, which is only known to TagRoot and remote verifiers. The keys are derived as follows: $k_{\texttt{EID}}^{type} = \text{HMAC}_{K_p}(\texttt{EID}, type)$. By providing an additional key *type*, the enclave can request keys for different purposes. For example, it can derive sealing keys for encrypting and decrypting sensitive data for secure offline storage. Also, it can derive remote attestation keys to compute a message authentication code (MAC) over a challenge given by a remote verifier. The remote verifier knowing the platform key K_p can then recompute the MAC, thus remotely attesting the enclave. TagRoot can also be extended to asymmetric remote attestation protocols [Ana+13].

Secure Shared Memory. TagRoot supports secure shared memory (shm) as a fast and flexible inter-enclave communication method. An enclave can offer another "target" enclave shared memory access to parts of its own enclave memory regions via `shm-offer`. TagRoot creates a special entry in the offering enclave's control block (ECB), covering the offered shm region and the target enclave's EID. For this, the target enclave does not need to exist yet. It can asynchronously accept the shm offer via `shm-accept`, which expects the offering EID as an argument. When accepting shm, TagRoot scans the existing ECBs to find the offering enclave via its EID. In case a valid shm offer exists, TagRoot adds the offered shm region to the target enclave's regions in the ECB and also returns the memory region's pointer back to the enclave to help it use the shared memory. Once an enclave has accepted a new shared memory region, it has to notify the untrusted operating system to load the shm region into the MPU.

The target enclave can close an accepted shm by issuing `shm-release`, which removes the shm from the enclave's memory ranges. An offering enclave can withdraw a pending offer by offering the empty region. However, it cannot close an offer that has been accepted already. This is because for any TSyscall TagRoot only manipulates the ECB of the calling enclave.

Our secure shared memory allows m:n connectivity between enclaves, where m is the number of offers an enclave can make, and n is the number of offers a target enclave can accept. m is unlimited, and n is only limited by the number of enclave regions that can be stored in the ECB, which is an implementation-defined constant. Moreover, TagRoot's shared memory supports a transitive trust model. An owner enclave could subsequently offer the same shared memory to other target enclaves, thus minimizing memory usage in case of broadcast channels, for example.

Local Attestation. TIMBER-V achieves local attestation implicitly by using shared memory without the involvement of cryptographic secrets. By offering and accepting shared memory, both involved enclaves identify their communication partner via its EID, thus mutually attesting each other.

4.5 Dynamic Memory Management

TIMBER-V provides highly flexible and dynamic memory management. Different security domains can claim memory during runtime with fine granularity. Dynamic memory has been an issue for enclaves before. For example, Intel SGX adds dynamic management of enclave pages in SGXv2 via separate trusted service calls in microcode. In contrast to Intel SGX, TIMBER-V naturally supports much finer-grained dynamic memory management by simply updating tags. User software can directly claim or release memory via checked store instruction without the need for trusted service calls. This high flexibility and efficiency enable novel application scenarios, such as dynamic memory interleaving schemes. Memory interleaving minimizes memory fragmentation by keeping data locality across security domains. For example, when passing large untrusted data structures to an enclave, the enclave could avoid copying the data to enclave memory by just updating tags. Thus, the data structures remain interleaved within the untrusted memory. In the same way, memory interleaving can be used for dynamic memory management–the dynamic allocation and deallocation of trusted memory.

In this section, we first explain heap interleaving from which we develop stack interleaving, a novel memory interleaving scheme. We do this for both TagRoot and enclaves, and show that we can entirely outsource dynamic memory from TagRoot to the untrusted operating system, thus reducing the TCB. Finally, we show that stack interleaving supports interrupts with arbitrary nesting levels.

4.5.1 Heap Interleaving

Heap interleaving reuses an untrusted heap to store trusted data. For this, trusted code first instructs untrusted code to allocate a chunk of memory on its heap. The precise heap layout is irrelevant as long as the requested memory chunk lies within N-tagged memory. Since the complex task of memory allocation is now outsourced to the untrusted domains, the TCB can be significantly reduced. Next, the trusted code claims the allocated memory chunk. It does so via checked store instructions, which atomically check memory for N-tag and update it to TS-tag or TU-tag, respectively. This protects the newly created trusted heap object against malicious access from the N-domains. However, care must be taken to identify trusted heap objects reliably during their lifetime. In order to free a trusted heap object, the trusted code simply clears it and reverts its tags to N-tag via checked store instructions, and notifies the untrusted code to do the heap cleanup.

User Heap Interleaving. Typically, an enclave actively requests heap space for trusted heap objects, which it uses internally to satisfy its dynamic memory demand. To reliably identify a trusted heap object, enclaves should always keep a pointer to it inside protected enclave memory and only use this pointer to reference the trusted object. If enclaves would interpret untrusted function arguments as trusted heap pointers, memory corruption attacks become possible.

Supervisor Heap Interleaving. When creating a new enclave, the operating system allocates a trusted enclave control block (ECB) on its heap and calls

`create-enclave`, which claims the ECB for TS-mode. Since most trusted OS service calls take the ECB as an argument from the untrusted OS, TagRoot needs to verify its validity. It does so in two steps: First, TagRoot accesses an ECB only via checked memory instructions, checking for TS-tag. This prevents misinterpreting untrusted data as ECB. Second, since the ECB argument could point to arbitrary TS-tagged memory, TagRoot identifies valid ECBs via an unforgeable header at the start of each ECB, as will be explained in Section 4.5.3.

4.5.2 Stack Interleaving

Stack protection is crucial for enclaves. Typically, an execution thread is given individual stacks for every security domain it can exercise. For example, SGX enclaves use separate secure stacks that are isolated from their hosting app. Also, operating systems usually maintain separate kernel stacks for each app. With TIMBER-V, we can reuse the same stack across different security domains, thus removing the need for maintaining multiple stacks per execution thread. This reduces memory fragmentation, which is particularly relevant to the limited physical address space of low-end embedded systems.

Stack interleaving is a simple program transformation that inserts additional stack allocation code. Whenever allocating a new stack frame, we claim this memory using checked store instructions, checking memory for N-tag and updating it to TS-tag or TU-tag, respectively. When deallocating the stack frame, we clear it and revert the tags to N-tag via checked stores. As with heap interleaving, one needs means to check the validity of dynamic memory, that is, the validity of stack pointers. We show stack interleaving (i) horizontally within supervisor mode, (ii) horizontally within user mode, and (iii) vertically across TSyscalls. We implement stack interleaving in a separate compilation step and defer details to Section 4.6.

Horizontal Supervisor Stack Interleaving. When receiving trusted OS service calls (TSenter), TagRoot reuses the S-mode stack maintained by the untrusted operating system. The validity of the stack pointer is implicitly checked by our stack interleaving transformation, checking untrusted memory for N-tag before claiming it. This prevents TagRoot from accidentally overwriting trusted memory. If the untrusted operating system provides an invalid `sp`, it can only break the system's availability, which it can do in any case. While processing trusted service calls, `sp` cannot be manipulated because TagRoot does not leave TS-mode until the service call is finished (or interrupted).

Horizontal User Stack Interleaving. When transitioning from an untrusted app to an enclave (TUenter), the enclave claims and releases stack frames on the untrusted app's stack. An enclave might call untrusted functions from the outside, e.g., to request dynamic heap memory or file access. Such transitions are named "ocalls" and demand special treatment. First, a finished ocall needs to return to the enclave's call site, denoted as "oret". We achieve this by making the oret sites callable using TC-tag, as depicted in Figure 4.5. Second, orets need to be protected against misuse, as follows: An attacker could directly jump to an oret without a corresponding ocall and thus perform code reuse attacks. We

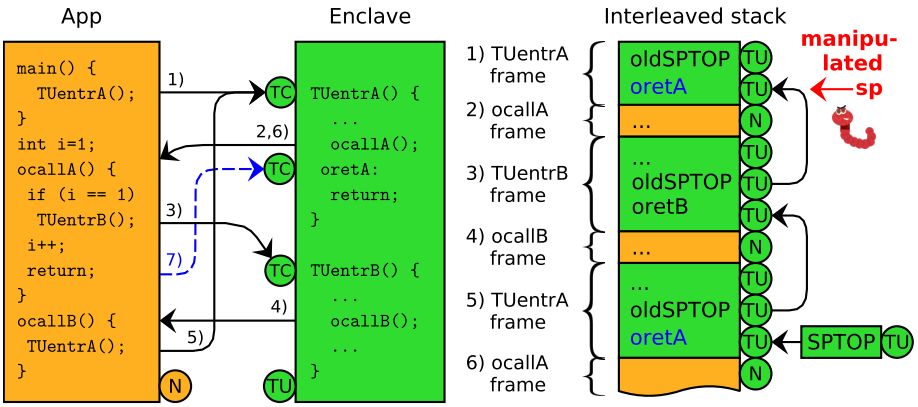

Figure 4.5: User stack interleaving with nested TUenter and ocall.

address this by securely pushing the return address (*i.e.*, the address of oret) onto the stack before doing the ocall and verifying it afterward. Thus, an attacker can only jump into active orets. However, the attacker could point `sp` to arbitrary trusted data that contains a valid return address. E.g., he could confuse the nesting level of multiple ocalls by returning to a previous ocall rather than the latest one. Consider the code in Figure 4.5, where both `TUentrA` and `TUentrB` perform ocalls, leading to a nested call sequence denoted with numbers 1) to 6). When returning from `ocallA` in step 7), an attacker could confuse the context of `oretA` by pointing `sp` to the first `TUentrA` frame instead of the correct fifth one (see upper right corner). We prevent this by verifying the stack pointer `sp` at each enclave oret site against `SPTOP`, which holds the `sp` of the latest ocall in trusted enclave memory. To support nesting, we securely push the previous `SPTOP` onto the stack and restore it afterward.

Vertical Stack Interleaving. When enclaves request trusted services via a TSyscall, TagRoot reuses the enclave's stack in the same way as outlined before. However, care must be taken since the stack is now interleaved across different privilege modes. Before TagRoot uses the enclave stack, it has to ensure that `sp` points into the current enclave's memory and that it has enough space for processing the TSyscall. The stack requirements of TSyscalls can be statically determined using profiling or static code analysis. Besides, the stack needs to be able to hold one interrupt frame, as explained in the following.

Interrupt Handling. Stack interleaving naturally supports interrupt handling. As outlined in Section 4.4.1, on interruption of trusted code, TagRoot stores the current execution context in a secure interrupt frame. With stack interleaving, TagRoot can directly store the interrupt frame on the current stack. As with ocalls, care must be taken since the untrusted operating system can manipulate the stack pointer before resuming from interruption. However, unlike horizontal user stack interleaving, we cannot keep a copy of the last valid `sp` in secure memory (cf. `SPTOP`) because the operating system might resume a different

interrupted enclave first or resume an interrupted TagRoot service call. To allow TagRoot to distinguish valid interrupt frames from other TS-tagged data, we introduce an unforgeable header, which TagRoot can check on every `resume` call.

4.5.3 Unforgeable Headers

Trusted metadata such as ECBs or interrupt frames are protected via unforgeable headers (see Figure 4.4). To make headers unforgeable, they are tagged with TC-tag, which only TagRoot can set. ECB headers and interrupt frame headers contain two distinct magic values. TagRoot uses them to identify valid ECBs and valid interrupt frames. It takes care not to accidentally set the TC-tag on any other data containing these magic values. Since headers are callable via TC-tag, they could be misused as malicious entry points. In order to prevent misuse, the magic values have to fulfill the following property: When interpreted as assembler instruction, they shall divert control flow to some form of secure error handling (e.g., an endless loop "`j .`" or a jump to an error handler).

4.6 TIMBER-V Implementation Details

We implemented TIMBER-V on the RISC-V Spike simulator and used it to run our TagRoot implementation. Subsequently, we give more details about tag isolation and the disambiguation of TUenter and TSenter, our tag-aware instructions, the proposed code transformations, required efforts for enclave developers, our MPU design, and additional CPU registers.

Tag Isolation Policy. Our tag isolation policy is given in Table 4.3. N-domains can only access N-tagged memory. The only way to enter T-domains is by fetching code tagged with TC-tag. Depending on the current privilege mode, TIMBER-V performs a TUenter or a TSenter. When fetching N-tagged memory, the CPU leaves trusted execution and switches back to the N-domains. This is denoted as TUleave and TSleave. Enclaves in TU-mode cannot write TC-tags to prevent the manipulation of secure entry points. TS-tagged memory is exclusive to TS-mode and protects trusted metadata against malicious enclaves and the operating system. For security reasons, we also prevent TS-mode from fetching TU-tagged memory. This technique is well known and implemented as supervisor mode execution prevention (SMEP) in Intel x86 CPUs [Int16a], for example. M-mode has full access to all tags, as it is typically also responsible for emulating missing hardware features.

TUenter vs. TSenter Disambiguation. Both TU-mode and TS-mode use the same TC-tag to specify secure entry points. If not cleanly separated, this would allow confusion attacks between TUenter and TSenter. For example, an attacker could spawn a malicious enclave (TU-mode). While this malicious enclave normally cannot access other benign enclaves, the attacker could invoke the enclave via a TSenter from S-mode rather than a TUenter from U-mode. Hence, the malicious enclave would execute in higher-privileged TS-mode, thus undermining all of TagRoot's security guarantees. We prevent such attacks by constraining

horizontal transitions to MPU regions of the same privilege mode: TUenter is only allowed for user-mode MPU slots, while TSenter can only target MPU slots marked for TS-mode. TS-mode slots cannot be manipulated from the untrusted OS. Again, this resembles supervisor mode execution prevention [Int16a].

MPU Design. Each MPU slot holds not only base and bound information together with rwx access permissions but also a TU and a TS flag. Slots marked as TU are shared between enclaves and untrusted apps. Slots marked as TS cannot be manipulated from untrusted code and are used to distinguish TSenter from TUenter, as outlined before. Only TagRoot can enable these flags. While the untrusted operating system cannot manipulate TS slots, it can overwrite TU slots, which will automatically clear the TU flag. This prevents enclave execution until TagRoot validates changes and reenables the TU flag.

Tag-aware Instructions. We add new instructions for checking and manipulating tags, as listed in Table 4.4. Standard 32-bit RISC-V memory instructions can operate on bytes (`b`), half-words (`h`), and words (`w`), optionally selecting the upper (`u`) byte or half-word. We duplicate those instructions to checked variants with the suffix `ct`. Checked loads preserve the semantics of loading memory from address `src` into the register `dst`. Likewise, checked stores transfer the content of the `src` register to the memory address `dst`. Unlike normal memory accesses, the checked instructions trigger a trap if the memory tag of the memory address being accessed does not match the expected tag, encoded in `etag`. Also, checked stores overwrite the tag at `dst` with a new tag, encoded in `ntag`.

The accessed memory address is determined by a base address stored in a register, and a 12-bit signed address offset, encoded as immediate. Also, `etag` and `ntag` are encoded as immediate, stripping the upper bits of the address offset to 10 bits and 8 bits for checked loads and checked stores, respectively.

For cases where memory tags are unknown, we add a separate load and test tag `ltt` instruction.[4] Similar to a checked load, `ltt` verifies the tag of a memory location (`src`) against a given expected tag (`etag`). However, instead of trapping on a mismatch, `ltt` stores the result in a register (`dst`). This allows subsequent code to take appropriate action. We utilize `ltt` for enclave cleanup in order to discharge TagRoot from keeping track of the exact enclave layout.

[4] Cf. the Test Target (TT) instruction of TrustZone-M [Arm17].

Table 4.3: Tag isolation policy for the memory accesses read (`r`), write (`w`), fetch or execute (`x`) as well as the horizontal transitions TUenter/TSenter (`e`) and TUleave/TSleave (`l`).

Access permitted	N-tag	TC-tag	TU-tag	TS-tag
N-domains	`rwx`	`--e`	`---`	`---`
TU-mode	`rwl`	`r-x`	`rwx`	`---`
TS-mode	`rwl`	`rwx`	`rw-`	`rwx`
M-mode	`rwx`	`rwx`	`rwx`	`rwx`

```
1  lw   t0,24(sp)
2
3  lw   t1,2048(a1)
4
5  add  t0,t0,t1
6  sw   t0,24(sp)
```

(a) Original code.

```
1  lwct ts,t0,24(sp)
2  addi a1,a1,2040
3  lwct ts,t1,8(a1)
4  addi a1,a1,-2040
5  add  t0,t0,t1
6  swct ts,ts,t0,24(sp)
```

(b) Transformed code.

Figure 4.6: Code hardening for TS-mode with overflow correction.

Code Transformations. We implement code transformations in a separate compilation step. We compile source code to assembler code, which we then transform using a custom awk script [Gaw]. The code hardening transformation simply replaces all memory accesses with their checked instruction pedants, as shown for TS-mode in Figure 4.6. In some cases, address overflows occur, namely when the encoding space of memory offsets is insufficient for a direct 1:1 transformation due to the additional `etag` and `ntag` encoding. In these cases, we insert correcting instructions that shift the overflowing part to the instruction's base register (lines 2–4).

For stack interleaving, the script detects stack allocations and deallocations by searching for manipulations of the stack pointer `sp`. It then claims or unclaims the stack frame by inserting checked store instructions accordingly, as seen in Figure 4.7 lines 3–4 and 6–7.

Developer Effort. From a developer's perspective, writing enclaves boils down to placing memory into distinct linker sections, for which we provide macros. One can mix enclave and non-enclave code in the same source file via annotations. Entry points are specified via a simple array. Ocalls, in addition, require to invoke an assembler macro. Code transformations are fully integrated into the macros and the build system. For memory accesses across security domains, we provide dedicated macros setting `etag` accordingly. Edge routines could further reduce efforts, as done in the SGX SDK [Int16b].

Additional CPU registers. We add new control and status registers (CSRs) that are only accessible to TS-mode (and M-mode). `STSTATUS` configures TIMBER-V and controls enclave execution. It holds a flag indicating the current security mode (normal or trusted). Moreover, whenever a running enclave traps

Table 4.4: TIMBER-V tag-aware instructions.

Checked Loads	Checked Stores
`lbct etag, dst, src`	`sbct etag, ntag, src, dst`
`lbuct etag, dst, src`	`shct etag, ntag, src, dst`
`lhct etag, dst, src`	`swct etag, ntag, src, dst`
`lhuct etag, dst, src`	Load Test Tag
`lwct etag, dst, src`	`ltt etag, dst, src`

```
function:
  addi sp,sp,-8


  ...


  addi sp,sp,8
  ret
```

(a) Original code.

```
function:
  addi sp,sp,-8
  swct n,ts,zero,4(sp)
  swct n,ts,zero,0(sp)
  ...
  swct ts,n,zero,4(sp)
  swct ts,n,zero,0(sp)
  addi sp,sp,8
  ret
```

(b) Transformed code.

Figure 4.7: Stack interleaving for TS-mode.

due to an interrupt or exception, `STSTATUS` will raise a flag that prevents enclave execution until resumed by TS-mode. To allow TS-mode to intercept traps, we add a separate trap vector, called `STTVEC`. Whenever the CPU is in trusted mode, traps are redirected to a trusted trap handler pointed to by `STTVEC`. Traps happening in normal mode are forwarded to the standard trap handler, stored in `STVEC`. We implement forwarding in a small M-mode trap delegation code. To help the trusted trap handler in setting up scratch space, we duplicate the supervisor scratch register for the trusted mode, called `STSCRATCH`. Besides, we add a register denoted as `SECB` to hold a pointer to an enclave control block. `SECB` identifies the currently loaded enclave and helps TS-mode in processing trusted enclave service calls.

4.7 Security Analysis

Enclave systems such as TIMBER-V build upon various components to protect sensitive data from being leaked (enclave confidentiality) or corrupted (enclave integrity). As shown in Figure 4.8, this comprises tag isolation, MPU isolation, and secure entry points plus secure interruption on the one hand. On the other hand, enclaves can make use of sealing, attestation, and secure shared memory provided by TagRoot. Since all enclave services are built atop of TagRoot, its integrity is crucial. We reemphasize that a proper implementation of TagRoot, the CPU, and the enclaves is assumed, and cryptographic primitives are considered secure.

In the following, we discuss how TIMBER-V protects enclave integrity and confidentiality against direct and indirect accesses. Furthermore, we discuss the security of enclave shared memory with local attestation, the security of TagRoot, and dynamic memory interleaving. We omit discussion of sealing and refer to SGX instead [Ana+13].

Direct Access. During runtime, the tag isolation policy prevents N-domains from directly accessing or tampering enclave memory. Also, our tag update policy does not allow elevating the current privilege mode. To prevent (malicious) enclaves from accessing other enclave's memory, TagRoot ensures that (i) en-

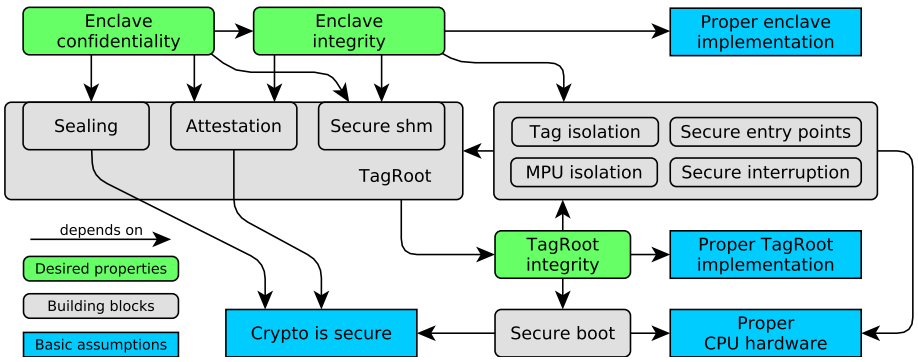

Figure 4.8: Relation graph of TIMBER-V security properties, building blocks, and assumptions.

clave regions do not overlap upon enclave initialization, and (ii) the MPU only holds regions of a single enclave at a time. This corresponds to MPU isolation. (i) ensures exclusiveness, *i.e.*, the only way for having enclave regions overlap is via shared memory, as discussed later. (ii) ensures that enclaves cannot misuse stale MPU entries of other enclaves. Also, our shared MPU design prevents the forging of MPU entries. Whenever the untrusted operating system updates an MPU slot, an enclave cannot use it until TagRoot acknowledges these changes (cf. Section 4.6). This mechanism again contributes to MPU isolation.

Indirect Access. Indirect security violations are prevented by secure entry points, secure interruption, and load-time attestation.

To prevent code-reuse attacks from leaking sensitive enclave data to an attacker, TIMBER-V enforces secure entry points via the TC-tag. Since only TagRoot can set TC-tags, they are tamper-proof. Of course, this does not prevent code-reuse attacks in case of memory safety vulnerabilities in the enclave code itself. Achieving memory safety is an ongoing field of research [AHP18]. If memory safety cannot be guaranteed, our code hardening transformation can make potential code-reuse attacks harder. It prevents an attacker from misusing memory instructions to leak sensitive information.

TagRoot prevents indirect information leakage due to interruption. If an interrupt occurs, TagRoot saves CPU registers and clears sensitive registers before giving control to the operating system. The saved registers are stored in a secure interrupt frame, protected with TS-tag.

During enclave loading, the operating system could manipulate an enclave's code to divulge secret information later on. In order to detect manipulation, enclave loading is measured using a cryptographically strong hash function (SHA256). Thus, whenever the untrusted operating system manipulates the loading procedure, this will yield a different enclave identity (EID). Subsequent attestation or unsealing of secrets will fail since the enclave's cryptographic keys changed.

Shared Memory and Local Attestation. In general, TagRoot prevents altering an enclave's memory regions during runtime except for shared memory (shm). By opening shared memory, enclaves willingly accept memory-region overlaps with other enclaves. Enclaves must always target their shm offer towards another EID. The target enclave needs to accept this shm offer by providing the offering enclave's EID. This enforces mutual authentication and prevents misuse by opening bogus shared memory.

Shared memory also demands temporal isolation to prevent time-of-check vs. time-of-use (TOCTOU) attacks in two directions. First, if an shm-offering enclave gets destroyed, a target enclave still has access to the shm. As long as the target enclave does not release it, the shm cannot be given to a newly created offering enclave because TagRoot prevents enclave region overlaps. Thus, TagRoot supports temporal authenticity of the offering enclave. Second, if the target enclave gets destroyed after having accepted an shm offer, it might get re-instantiated and accept the same shm offer again. This happens without knowledge of the offering enclave, and TOCTOU attacks become possible. In order to avoid TOCTOU issues, the offering enclave needs to close the shm offer after being accepted. Furthermore, it needs to employ a simple handshake to verify the aliveness of the target enclave. For example, both enclaves could agree on a session identifier that changes for each enclave restart. This handshake implicitly provides local attestation between two enclaves.

TagRoot. All of the analysis above critically depends on the integrity of TagRoot. We assume loading of TagRoot itself is protected using secure boot [Rua14a]. Once loaded, TagRoot can protect itself in an isolated execution container similar to enclaves. It uses tag isolation via TS-tag, MPU isolation with TS-mode MPU slots, and secure entry points protected via TC-tag. If TagRoot is interrupted, it saves its own register state in a secure interrupt frame. Various internal data structures are locked. Locking prevents race conditions, should an interrupted TagRoot be re-entered instead of being resumed.

Dynamic Memory Interleaving. Here, untrusted code offers N-tag memory to trusted code. Any untrusted arguments need to be validated by trusted code. In particular, one needs to ensure (i) the validity of the memory when claiming it, and (ii) the validity during usage. By claiming dynamic memory with checked store instructions (`etag` = N-tag) one can ensure (i), namely that trusted code does not accidentally overwrite trusted data in case of bogus memory pointers, for example. Besides, vertical stack interleaving crosses privilege modes and, thus, requires additional enclave region checks, as explained in Section 4.5.2. Point (ii) is different for the various interleaving schemes we presented before. In general, whenever pointers to trusted memory objects could be manipulated by untrusted code, one needs means to validate them. For supervisor heap and stack interleaving, we introduced unforgeable headers, uniquely identifying valid ECBs and interrupt frames. This avoids the need for tracking valid objects. In contrast, for user heap interleaving, we recommended tracking pointers to trusted heap objects inside the enclave. Also, horizontal user stack interleaving with ocalls needs additional checks of the stack pointer `sp` when re-entering the enclave.

Here, we store the last valid stack pointer inside the enclave. By maintaining (i) and (ii), dynamic memory interleaving is secure against corruption and direct information leakage.

4.8 Evaluation

4.8.1 Methodology

We evaluate TIMBER-V by running various macro- and microbenchmarks in the Spike simulator, which we extended to support TIMBER-V. We configure Spike for the `RV32IMAFD` ISA and use it to record histograms of all executed instructions. To estimate the runtime in CPU cycles, we map executed instructions to actual CPU cycles using different pipelined CPU models. We first define a simple baseline model against which we then compare two possible realizations of TIMBER-V. Our Model A captures unoptimized implementations, and Model B represents optimized designs with tag caching.

It should be noted that Model B is by no means an upper bound on the maximum performance achievable. Instead, it presents a conservative performance estimate based on related work about tagged memory architectures [Son+16; Joa+17; Suh+04]. We outline these CPU models in the following and summarize them in Table 4.5.

Baseline CPU Model. As a baseline, we assume that all register (`reg`) or memory instructions (`ld`/`st`) take one CPU cycle. This is reasonable for a load/store architecture as RISC-V with on-chip SRAM commonly used for embedded processors. When instructions stall the execution pipeline, we assume additional latency to refill the pipeline. Stalling applies to conditionally-taken branches for indirect jumps, as well as to syscalls and returns, and is indicated by the column `stall`. We assume that multiplication (`mul`) and division (`div`) instructions also complete within one CPU cycle, which keeps our evaluation results pessimistic. That is, comparing against this baseline will show higher overhead than observed in practice, where multiplication and division typically take multiple cycles.

TIMBER-V CPU Models. Each instruction fetch requires one additional tag fetch for verifying the instruction itself. For the unoptimized Model A, we assume that the prefetcher can effectively hide this additional tag fetch. Thus, linear code fetches do not exhibit overhead, and all non-memory instructions (`reg`, `mul`, `div` and `other`) take one cycle. However, when the execution pipeline stalls, the tag fetching overhead gets visible for the first instruction after the stall. Thus, we add one extra cycle for stalls.

For memory loads and stores, we assume one extra cycle to load and check the tag on the accessed data. A checked memory load (`lct`) does not experience additional overhead since the data's memory tag is already loaded for enforcement of the tag isolation policy and can be readily used for the additional tag check. On the other hand, for checked memory stores (`sct`) we assume one additional cycle to store the new tag to memory.

Table 4.5: Expected CPU cycles per instruction.

CPU model	ld	st	lct	sct	reg	mul	div	other	stall
Baseline Model	1	1	-	-	1	1	1	1	3
TIMBER-V Model A	2	2	2	3	1	1	1	1	4
TIMBER-V Model B	1.1	1.1	1.1	1.1	1	1	1	1	3.1

Model A does not make use of tag caching, which could significantly improve performance. A tag cache can serve tags in parallel to ordinary memory accesses and, thus, hide the tag checking latency for all cached tags. By comparing state-of-the-art literature on tagged memory architectures, we observe that tag caching can reduce the average overhead of tag accesses into the low single-digit range [Son+16; Joa+17; Suh+04]. Considering that our work utilizes two tag bits per word, we conservatively estimate the expected performance impact of the tag operations with 10%, which is reflected in Model B. The resulting costs for the individual instruction classes is depicted in the last line of Table 4.5. Again, the prefetcher hides tag checking latency for instructions, while a stall is prolonged by 10% of a cycle. Likewise, memory loads and stores experience 10% overhead. We assume that checked stores (`sct`) are not slower than ordinary stores (`st`) because the parallel tag cache can absorbe the additional tag update latency.

4.8.2 Macrobenchmarks

To benchmark raw CPU performance, we used the beebs benchmark suite [Ben+] as well as CoreMark [EEM]. We compiled them with GCC version 7.3.0 with "`-O1`". We excluded beebs benchmarks with external dependencies. Also, we filtered `nettle-md5` and `fdct` due to verification mismatches. For `newlib-log` and `ns`, we had to prevent the compiler from optimizing out essential code by adding `volatile` and `noinline` statements. We ran beebs and CoreMark with one iteration since our evaluation does not need warm-up iterations to fill CPU caches but precisely captures all instructions.

Tag Isolation. Our tag isolation policy causes runtime overhead during code execution for both N-domains and T-domains. Figure 4.9 (left) shows an average runtime overhead of 25.2% for TIMBER-V Model A with a peak of 47% for `nsichneu`, which uses frequent lookup table accesses. `insertsort` frequently swaps memory locations, which causes higher overhead. `statemate` implements a state machine with frequent state updates, and `tarai` uses recursion, causing stack accesses to dominate over other operations. Interestingly, `aha-compress` shows significantly less overhead than `compress`, because it benchmarks four different CPU intensive compression algorithms with relatively few memory accesses. The `fibcall` benchmark shows the least runtime overhead (3.4%) because the recursive Fibonacci computation can be kept entirely within the CPU registers. For the optimized TIMBER-V Model B, the average overhead is only 2.6% with a minimum of 0.3% (`fibcall`) and a maximum of 4.7%

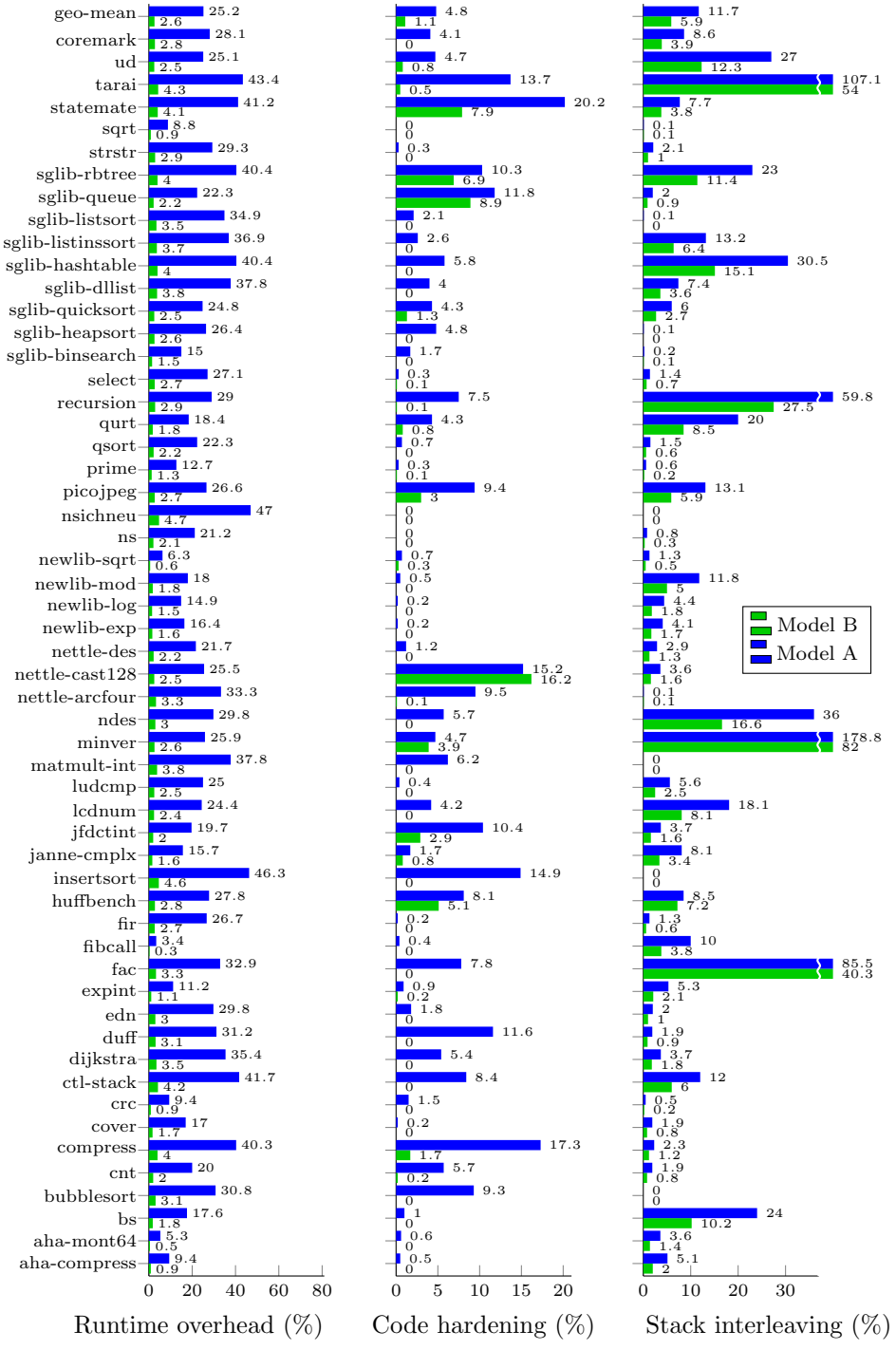

Figure 4.9: Runtime overhead of TIMBER-V tag isolation (left), additional code hardening (middle), and optional stack interleaving (right).

(`nsichneu`). Our results indicate that even for memory-intensive benchmarks Model B incurs small runtime overhead.

Code Hardening Transformation. Our code hardening transformation adds only negligible overhead atop of the above runtime overhead. This is shown in Figure 4.9 (middle). The overhead is low because the checked instructions are almost a drop-in replacement for ordinary memory instructions. Since ordinary memory instructions are subject to tag isolation, the memory tags for accessed data are already loaded. This overhead is included in tag isolation above (see Figure 4.9 (left)), and the additional tag checks are basically for free.

Few benchmarks show noticeable overhead because the code hardening transformation, in some cases, inserts correcting instructions to handle address overflows, as discussed in Section 4.6. By integrating the transformation directly into the compiler, one could leverage compiler optimization to avoid overflow behavior.

Stack Interleaving. To benchmark the additional overhead induced by stack interleaving, we compare each TIMBER-V model without stack interleaving against a compilation with enabled stack interleaving. The results are shown in Figure 4.9 (right). The overhead is highly dependent on good compiler optimization and the used stack space. Many benchmarks (e.g., the memory-intensive `nsichneu`) show zero overhead for stack interleaving since stack frames are optimized out in favor of CPU registers. The highest overhead (178.8%) occurs for `minver`, which allocates a temporary stack buffer of 500 words for computing matrix inverses. The average runtime overhead of stack interleaving is acceptable with 11.7% for Model A and 5.9% for Model B.

We see the potential for improvements in several directions: First, large stack allocations should be avoided. This is considered bad practice in any case since there exists no generic way of handling out-of-memory behavior on stack allocations. We manually adapted `minver` to pre-allocate a large stack buffer in the data segment and observed that the runtime overhead drops from 82% and 178.8% to negligible 2.5% and 5.6% for Model B and Model A, respectively. Second, since stack interleaving implicitly erases new stack frames, one can avoid potential double clearing. We evaluated this for `huffbench` by manually removing the calls to `memset` on stack buffers. This reduces overhead from 7.2% and 8.5% down to 3.9% and 5% for Model B and Model A, respectively. A compiler could assist in avoiding large stack allocations. Third, one could optimize frequent stack frame allocation and deallocation in favor of less frequent pre-allocation. For example, when having frequent calls to the same subfunction inside a loop, one could pre-allocate the subfunction's stack frame at the call site, thus reducing the stack interleaving overhead from N loop iterations to one.

Heap Interleaving. For heap interleaving, we only evaluate benchmarks that use heaps. We use a simple heap implementation provided with FreeRTOS, namely, `heap-4`. Moreover, we wrap (de)allocation routines to claim and unclaim allocated memory using checked store instructions. The runtime overhead of heap interleaving is slightly below 14%, as shown in Figure 4.10, which is comparable to stack interleaving. We believe further improvements are possible since our

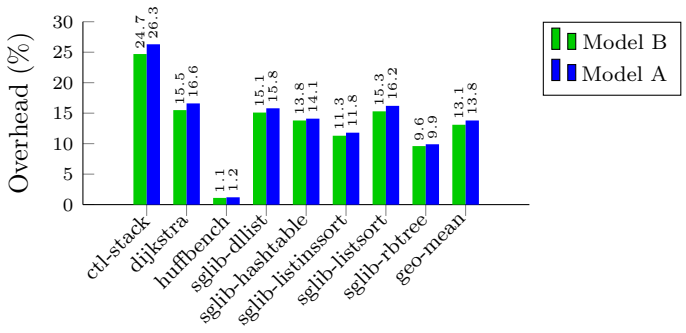

Figure 4.10: Runtime overhead of additional heap interleaving.

`realloc` wrappers currently do not reuse allocations but always request new memory with `malloc`. The `huffbench` test shows negligible overhead because it allocates only one out of many buffers on the heap.

Our secure `malloc` wrapper acts like `calloc`, clearing the whole buffer while changing tags. Likewise, our secure `free` automatically erases all data while restoring the original tags. Thus, for security-critical code that demands such zeroing functionality already, heap interleaving comes virtually for free.

4.8.3 Microbenchmarks

In the following, we discuss the performance of trusted services as well as horizontal transitions between apps and enclaves. The performance numbers are summarized in Table 4.6. We depict SHA256 hashing costs in a separate column.

Trusted OS services. Trusted OS services are invoked like ordinary functions; hence, the transition denoted as TSenter has minimal overhead. When returning from a TSenter via TSleave, TagRoot clears all callee-saved registers to avoid information leakage. The following results show the performance of the individual trusted OS service calls *without* TSenter and Tleave overhead.

The base cost of `create-enclave` is dominated by claiming the ECB. We show the runtime when creating the first enclave. The runtime slightly increases when adding more enclaves since we chain all ECB's in a linked list. For `add-region`, we show the runtime for adding the first region. The runtime grows with the number of regions and the number of enclaves due to the region overlap checks `add-region` performs. This variability is acceptable since overlap checks are cheap. Also, overlap checks are only performed at enclave initialization but not during runtime. For `add-data` and `add-entries`, runtime increases with the size of the added data blob or the number of added entries, respectively. This is because changing memory tags as well as computing the hash measurement depends on the amount of data. Also, performance slightly depends on the position of the associated enclave region in the ECB. The base costs are shown in Table 4.6 when adding one data word or one entry to the first enclave region.

Table 4.6: Enclave performance in expected CPU cycles.

	Functionality	TIMBER-V Model B		TIMBER-V Model A	
		Base cost	Hash cost	Base cost	Hash cost
Trusted OS Services	TSenter	7.1	0.0	9.0	0.0
	TSleave	27.4	0.0	32.0	0.0
	create-enclave	527.5	5647.1	759.0	7175.0
	add-region	396.3	5821.6	606.0	7483.0
	add-data	212.0	11309.4	340.0	14365.0
	add-entries	206.4	5616.2	348.0	7127.0
	init-enclave	123.5	5236.6	208.0	6397.0
	load-enclave	315.6	0.0	437.0	0.0
	destroy-enclave	733.5	0.0	1057.0	0.0
Trusted Encl. Services	TSyscall	68.3	0.0	71.0	0.0
	TSyscall dispatch	71.1	0.0	88.0	0.0
	TSyscall return	49.2	0.0	66.0	0.0
	get-key	337.5	12216.6	457.0	15686.0
	shm-offer	1045.6	0.0	1560.0	0.0
	shm-accept	1455.0	0.0	2062.0	0.0
	shm-release	231.7	0.0	317.0	0.0
	interrupt-enclave	107.7	0.0	175.0	0.0
	resume-enclave	103.0	0.0	200.0	0.0
App-Encl.	TUenter	1.0	0.0	1.0	0.0
	TUleave	4.1	0.0	5.0	0.0
	ocall	16.9	0.0	29.0	0.0
	ocall return	28.4	0.0	45.0	0.0

`init-enclave` has constant overhead. In contrast, `destroy-enclave` unclaims all enclave memory with a linear sweep over the enclave. We show the runtime of destroying an empty enclave. `load-enclave` validates the MPU configuration against the loaded enclave's ECB, hence the moderate overhead. Once an enclave is loaded, horizontal transitions between U-mode (app) and TU-mode (enclave) experience no principled overhead, as discussed later.

Trusted enclave services. Trusted enclave services are implemented as TSyscalls, which experience slight overhead due to M-mode trap delegation. TSyscall dispatching includes validation of the MPU configuration for vertical stack interleaving and jumping to the correct service routine. A return from a TSyscall unwinds the dispatcher context, clears all caller-saved registers, and returns to TU-mode using the RISC-V `sret` instruction. In the following, we exclude TSyscall, dispatch, and return overhead.

`get-key` computes an HMAC using two SHA256 computations, hence the overhead.`shm-offer` needs to check the validity of the arguments–not only their memory tags but also whether the arguments belong to the calling enclave. Apart from that, the performance is constant and independent of other enclaves. `shm-accept` traverses the linked list of ECB's to find a matching shm offer. For our benchmarks, the first enclave in the linked list has a corresponding shm offer. `shm-release` only erases the accepted shm region from the enclave's ECB, in our case, the fifth enclave region.

Interruption and resumption of enclaves (and TagRoot) is quite fast and mainly consists of saving and restoring the execution context in the interrupt frame. As before, the performance numbers of `interrupt-enclave` and `resume` exclude TSyscall latency due to trap delegation, while TSyscall dispatch and return overhead do not apply here.

TUenter and TUleave. As shown in the last rows of Table 4.6, TUenter has no overhead, showing only one jump instruction into the enclave. TUleave only takes longer because of an assumed pipeline stall of the `ret` instruction. When enclaves call untrusted functions on the outside, these ocalls need to securely store and verify the stack pointer, as discussed in Section 4.5.2. Moreover, an enclave must clear sensitive CPU registers on TUleave as well as ocalls, which can be automated, e.g., via so-called edge routines in the SGX SDK [Int16b].

4.8.4 Memory Overhead

TIMBER-V adds two tag bits to each 32-bit memory word, thus introducing 6.25% hardware memory overhead. Our TIMBER-V architecture directly runs unmodified code and, thus, does not introduce software memory overhead. Likewise, our code hardening transformation does not introduce memory overhead, since memory instructions are replaced 1:1 with checked instructions. Slight overhead only occurs if additional instructions are inserted for fixing offset overflows, as discussed in Section 4.8.2. Heap interleaving needs small constant-sized code memory for the allocation hooks but, in turn, avoids the need for secure heap implementations, which in total reduces code size. We do not give actual numbers since this strongly depends on the heap implementation. Stack interleaving needs additional code for stack frame allocation and deallocation. Currently, we insert checked store instructions for each allocated word, thus showing 43% overhead in assembler code lines for the expensive `minver` benchmark. However, when optimizing for code size, one could simply achieve constant overhead per stack (de)allocation by embedding checked stores in a loop. We manually optimized stack interleaving for `minver` and reduced the code overhead to 1%.

TagRoot Code Size. We used `sloccount` to count the number of source code lines as an estimate of TagRoot's complexity. TagRoot consists of 369 lines of assembler code and 1686 lines of C-code, from which 313 lines are used by HMAC and SHA256. This code base is relatively small, which is desirable for a trusted computing base as it reduces the risk of programming bugs. Also, the small size is beneficial for formal verification techniques that could help certify our TagRoot implementation [Kle+10]. As a comparison, the used FreeRTOS operating system has approximately 12 500 lines of code.

4.9 Related Work

In this section, we compare TIMBER-V against related work on enclaves as well as tagged memory.

4.9.1 Enclave Architectures

Many architectures were designed during the evolution of enclaves, which we outlined in Section 2.2. In the following, we compare TIMBER-V against other enclave architectures designed for small embedded devices.

SMART [Eld+12], Sancus [Noo+17], Soteria [Göt+15], TyTAN [Bra+15], and TrustLite [Koe+14] implement program counter-based memory access control for isolating secure tasks. The memory of a secure task is only accessible when the program counter is in the task's code region. Sancus has a hardware-only TCB and isolates a fixed number of small uninterruptible secure tasks stored in predefined memory locations. TyTAN and TrustLite use an execution aware MPU (EA-MPU) with multiple code and data regions per secure task. TrustLite loads all secure tasks at boot time, while TyTAN allows dynamic loading and unloading of secure tasks at runtime. The EA-MPU makes context switches faster but limits the number of concurrently loaded secure tasks. In contrast, TIMBER-V supports an arbitrary number of enclaves with fine-grained, dynamic isolation, and multiple entry points.

Secure communication in TrustLite is done via a simple handshake protocol, where two secure tasks first attest each other and then use cryptographic session tokens to authenticate messages. In TIMBER-V local enclave attestation and communication is done implicitly via shared memory, without any additional encryption. TyTAN uses a dedicated IPC proxy task that forwards messages between secure tasks, introducing copying overhead (1324 CPU cycles). In contrast, our secure shared memory is a fast alternative for exchanging bulk data between enclaves.

TrustZone-M [Arm17] supports four security domains like TIMBER-V. Horizontal and vertical domain transitions require special instructions, while in TIMBER-V domain switches are direct, thus imposing zero runtime overhead. TrustZone-M only supports secure and non-secure tasks. Our architecture supports mixed processes, where enclaves are directly embedded in untrusted processes. Moreover, our tagged memory approach has a finer granularity. TrustZone-M optionally supports two separate MPUs, one for the secure and one for the non-secure world. We reuse the same MPU across security domains, thus saving hardware costs. Also, our dynamic memory interleaving allows for stack (and heap) reuse, while TrustZone-M requires separate stacks for each domain.

4.9.2 Tagged Memory Architectures

The availability of metadata is the foundation for a multitude of runtime monitoring techniques like various sanitizers [Ser+12; SS15], as well as dynamic information flow tracking (DIFT) or taint tracking [SAB10]. Subsequently, many

hardware-based tagged memory architectures have been developed. In particular, DIFT implementations range from single tag bit schemes with fixed policy (e.g., Minos [CWC06] and CHERI [Wat+15]), over multi-bit schemes with partially configurable policy (e.g., Raksha [DKK07], DIFT [Suh+04], DIFT with coprocessor [KDK09]), to schemes with configurable bit width and fully programmable policy and enforcement (e.g., FlexiTaint [Ven+08], instruction-grain lifeguards [Che+09], Harmoni [DS12], PUMP [Dha+15]).

Compared to DIFT architectures, TIMBER-V has notably different characteristics. Firstly, DIFT schemes have a strong focus on performing tag/taint propagation during ALU operations. TIMBER-V, on the other hand, does not perform any tag propagation but utilizes tags for isolation purposes. While it is possible to use a DIFT architecture solely for isolation in some schemes [DKK07; Dha+15], this is needlessly wasteful. Secondly, TIMBER-V introduces a new trusted security domain, and the isolation and update policies depend on the currently active domain. Partially configurable DIFT architectures typically do not support such a domain switch. Finally, even fully programmable DIFT architectures are not necessarily suited for implementing TIMBER-V. Namely, architectures that perform tag operations asynchronously to the main processor [Che+09; KDK09; DS12] introduce a TOCTOU gap that can potentially be used to exfiltrate data from the trusted domain.

Besides DIFT-based architectures, other architectures use tagged-memory for enforcing various kinds of memory protection. HardBound [Dev+08] implements fat pointers in order to prevent spatial memory safety violations. HDFI [Son+16] uses a single tag bit to protect sensitive data words. However, in HDFI, tag checks are only performed when reading the data. This means that destructive write operations on sensitive data can not be prevented but only detected. This property corresponds to the weak *low-watermark policy for objects* of the Biba integrity model [Bib77]. In contrast, TIMBER-V follows the stronger strict integrity policy of the Biba model by refusing untrusted modifications of trusted data. Compared to that, Mondrian Memory Protection [WCA02], which uses two tag bits, and Loki [Zel+08], using up to 32 tag bits per word, are more similar to TIMBER-V. However, both concepts solely use tagged memory to implement word-wise access permissions, which is not sufficient to implement efficient enclave isolation. Additionally, when different permissions are tightly interleaved, Loki's tag size is simply too large for low-end devices that we target.

4.10 Possible Extensions

The concept of TIMBER-V can be directly applied to other system components, such as caches and I/O peripherals. Together with secure interrupts, one can implement flexible safety-critical systems.

Secure Components and Peripherals. One can easily extend CPU caches with our two tag bits and propagate them to main memory on cache eviction. Also, memory-mapped I/O peripherals can benefit from TIMBER-V's tag isolation policy by pinning their tag bits in a tag cache. That way, TIMBER-V can

facilitate secure I/O, that is, secure interaction with end users, sensors, actuators or other networked devices.

Secure Interrupts. Most embedded systems react upon regular timer or irregular I/O interrupts. TIMBER-V supports secure interrupts by modifying the M-mode trap delegation mechanism to always route interrupts directly to the trusted trap handler. Since this trap handler is *not* callable but protected via TS-tag, fake interrupts from S-mode are prevented.

Safety-critical Systems. TIMBER-V is a lightweight enclave architecture designed for security. Extending it for safety-critical systems with availability guarantees is an interesting field of research and should be straight forward. We denote safety-critical enclaves as safeclaves. In order to guarantee real-time behavior, safeclaves must be protected against denial-of-service attacks (DoS). Safeclaves are not triggered by untrusted code but by external I/O events or recurring timer periods. TagRoot can intercept safeclave interrupts as discussed before in order to trigger safeclave execution assuredly. One cannot simply use dynamic memory interleaving for safeclaves, since untrusted code could trigger security violations. Normal enclaves, however, can still benefit from interleaving. Also, by slightly adapting our shared MPU design, one can exclude safeclave MPU slots from being shared, making safeclaves safe against DoS from the operating system.

4.11 Summary

We presented TIMBER-V, a security architecture that brings the concept of enclaves to resource constrained RISC-V processors. TIMBER-V is the first efficient tagged memory architecture for isolated execution of enclaves. It minimizes memory overhead of tagged memory to 6.25% by augmenting tag isolation with MPU isolation. Our MPU supports shared slots between enclaves and their host applications. The runtime overhead varies between 25.2% for naive implementations and 2.6% for CPUs with tag caching.

The flexibility of TIMBER-V enables fine-grained and dynamic management of trusted memory. Dedicated tag-aware memory instructions can be used to simply and instantly check and update memory tags. Thus, TIMBER-V enables heap interleaving as well as interleaving of interrupt frames and other enclave metadata within untrusted memory. We also present a novel stack interleaving scheme that shares the same stack across multiple security domains. Interleaving reduces memory fragmentation, which is particularly relevant to low-end devices.

A small trust manager provides trusted services to the untrusted operating systems and enclaves. It manages the enclave life cycle, sealing, attestation, secure interrupt handling, and shared memory. Our secure shared memory has zero runtime overhead and allows m:n connectivity between enclaves. Shared memory provides an implicit means for local attestation. We implemented and evaluated TIMBER-V on the RISC-V Spike simulator to demonstrate its practicality. TIMBER-V is fully open source.

5

SGXJail: Defeating Enclave Malware

Beware of no man more than of yourself; we carry our worst enemies within us.
Charles Spurgeon

SGX enclaves can effectively shield good code from a bad operating environment. Even if any other part of the software is manipulated or compromised, SGX maintains the security guarantees of enclaves. The strong isolation of SGX enables many new use cases, such as trusted cloud computing, where tenants do not only distrust the other tenants, but also the cloud provider and its hardware and software infrastructure [BPH14; Sch+15; Gje+17].

However, what if the roles of good and bad were flipped? What if an enclave started misbehaving and attacking their environment? Assuming an enclave always behaves "good" seems like distributing a cloak of invisibility to criminals for free. Multiple researchers already warned years ago that SGX's strong isolation guarantees might be abused to develop irreversible malware [Rut13; DF14; CD16]. Unfortunately, this enclave malware threat is no longer pure theory, as we are evidencing functional attacks in the recent past. We have seen not only enclave malware exploiting side channels [Sch+17] but also enclave ransomware and shellcode [Mar18], however, with the help of a colluding host application. Recent research showed that enclaves could effectively hijack and impersonate any benign host application [SWG19], opening up enclaves for various types of userspace malware. Having witnessed those early proof-of-concept attacks, we can expect that more sophisticated and real-world attacks will appear in the future. Hence, it is indispensable to explore the defense space providently, before real-world attacks are being mounted.

Unfortunately, little is known about how to address this emerging threat adequately. While conventional programs can be scanned for misbehavior by anti-virus technology, SGX is a complete game changer when it comes to enclave analysis. On the one hand, SGX prevents runtime inspection of enclaves. On the other hand, SGX allows lazy loading of malicious enclave content at runtime. Thus, malware infection can be totally decoupled from enclave distribution and installation. This renders all static analysis techniques checking for enclave malware useless. In other words, SGX is a viable alternative to malware obfuscation and analysis evasion techniques. So far, no practical defense against enclave malware exists. If Intel chose to allow certified anti-virus software to inspect enclaves, this would undermine essential security guarantees and is in fundamental conflict with the very goal SGX has [Rut13]. Others proposed to detect enclave malware via their I/O behavior [DF14; CD16]. Analyzing all enclave I/O operations is not only prone to false positives and false negatives. The effort for doing so is believed infeasible in practice [Mar18]. Others proposed to embed malware analysis code within the enclave itself [CD16]. Enforcement of such a policy raises several questions regarding its practicality and efficacy. Consequently,

> "[...] the release and adoption of SGX-protected enclaves is likely to require a completely new approach to protecting our machines from the very malware SGX was designed to prevent." [DF14]

In this chapter, we propose the first practical defense mechanism against enclave malware. To do so, we analyze enclave primitives and their resulting attack vectors. We identify the root cause for the enclave malware threat as a too permissive feature set available to enclaves. This forces applications to blindly trust any enclaves they host. Consequently, a proper defense mechanism should give applications means to confine enclave operation to a clearly specified interface. To that end, we propose SGXJail, a lightweight yet effective measure to establish mutual distrust between enclaves and their host applications. SGXJail does so by confining enclave operation to a set of memory pages defined at the discretion of the host application. This mitigates entire classes of runtime attacks (e.g., ROP, JOP, DOP) from the enclave to the host. Further reasoning about enclave misbehavior can be purely based on the legitimate communication interface. We instantiate SGXJail using process sandboxing and syscall filters and demonstrate its efficiency. Furthermore, we propose HSGXJail, a minimal hardware extension to the SGX specification. It leverages Intel memory protection keys to confine enclave execution, which is even more efficient than SGXJail. (H)SGXJail is opt-in, works on unmodified enclaves, and can be easily integrated with the SGX software development kit. We provide the code open-source.[1]

With SGXJail, we expand possible SGX use cases beyond isolated execution. We envision modern software, which is additionally hardened using SGXJail against potentially malicious or misbehaving third-party code. Such hardening

[1] The code is available at https://github.com/IAIK/sgxjail

can be vital for any software making use of third-party plugins and add-ons, such as browsers, mail clients, or password managers.

Contributions. We summarize our contributions as follows:

- We systematically break down the enclave malware threat and identify a number of enclave malware primitives.
- We devise SGXJail, the first practical defense against enclave malware.
- We implement and evaluate SGXJail in software.
- We propose highly efficient HSGXJail via minimal hardware changes.

This chapter is based on the publication [Wei+19a], of which I am the main author, while Michael Schwarz and Luca Mayr contributed a significant part to the implementation of SGXJail. The rest of the chapter is organized as follows: Section 5.1 describes the threat model. Section 5.2 analyzes various enclave primitives and attack vectors. Section 5.3 presents (H)SGXJail. Section 5.4 discusses related work. We summarize our discussion of enclave malware in Section 5.5 and conclude in Section 5.6.

5.1 Threat Model

In this section, we first outline various application scenarios of SGX and argue why the original SGX threat model does not properly address enclave misbehavior. We then present our extended SGX threat model addressing enclave malware.

Scenario A. In the near future, SGX technology will likely permeate consumer systems and create diverse and many-faceted trust relations. Multiple independent software vendors (ISV) can use SGX for mutually protecting their proprietary library code (e.g., multimedia codecs, classification algorithms) or sensitive customer data (e.g., user passwords, encryption keys, or bitcoin wallets) inside third-party enclaves. Applications can embed such third-party enclaves to leverage their functionality.

In this scenario, an attacker develops innocent-looking enclave malware (e.g., disguised as browser plugins) and distributes it as a third-party enclave via existing software stores or repositories. A user installs those third-party enclaves alongside other applications. The attacker defers the installation of malicious payload to enclave runtime via a generic loader [Rut13]. Hence, neither the maintainers of software repositories nor the user can detect malware installation. The enclave can choose when to trigger its malicious activity.

Enclave malware might not only be invasive in nature like ransomware, bots, or rootkits. A malicious enclave can also stealthily track the system by collecting data about users without their knowledge or consent. Since the enclave shields tracking logic, the enclave developer achieves plausible deniability.

Scenario B. As more software is moved into enclaves, chances increase for exploitable vulnerabilities within enclave code. Enclaves are equipped with increasingly complex software, such as fully-fledged TLS stacks [Int19] or library OSes [BPH14]. Thus, it is just a matter of time vulnerabilities in the trusted

code of enclaves are discovered and exploited. In fact, it has already been shown that enclaves are prone to various attacks [Wei+16; Lee+17a; Sch+18a; Bul+19]. Such vulnerabilities could be used to infiltrate trusted enclaves with a malicious payload at runtime.

A Holistic Threat Model. The original threat model of Intel SGX considers all non-enclave code as untrusted, including application code hosting enclaves. This model might be well-suited from an enclave's perspective. However, it does not fit more advanced application scenarios outlined before, leaving applications completely unprotected against misbehaving third-party enclaves. This creates a dangerous asymmetry, as also outlined by Schwarz et al. [SWG19].

In this chapter, we introduce a more holistic threat model that does not violate the original threat model of SGX but augments it to address misbehaving enclaves explicitly. We consider a commodity system running software from various independent software vendors. On the one hand, third-party library vendors protect their secret data (e.g., cryptographic keys or intellectual property) inside enclaves. On the other hand, application developers include third-party enclaves in their applications for implementing various tasks. They also want some form of protection against third-party enclaves that are misbehaving, for the reasons outlined before. SGXJail provides this protection mechanism. From a user's perspective, the computer (including the operating system and most applications) are trusted. SGXJail needs to protect applications (and, subsequently, the computer) from potential enclave misbehavior, even if a dedicated attacker fully controls such enclaves (e.g., enclave malware).

SGXJail protects applications against any inspection or alteration of their state (memory, CPU registers) by enclaves, apart from what they are exposing to the enclaves via the ECALL/OCALL interface. SGXJail does not prevent API attacks that exploit too permissive OCALLs or poorly designed interfaces. Avoiding Iago attacks and confused deputy attacks is a separate, yet important line of research, as we discuss later. SGXJail focuses on security rather than availability. Hence, malware that exploits system resources such as CPU power (e.g., cryptocurrency miners) is not prevented. Moreover, this work does not focus on microarchitectural side channels, although SGXJail prevents specific classes of side-channel attacks. Finally, the CPU hardware is considered trusted.

5.2 Analyzing the Enclave Malware Threat

In this section, we analyze enclave primitives leading to various memory safety attack vectors on the application. Those primitives help us design a proper defense mechanism and solve the enclave malware threat at the level of memory safety. Attackers are left with high-level API attacks, as discussed at the end of this section.

Table 5.1: Enclave primitives leading to various attack vectors on the host application.

Attack \ Requirement	Arbitrary Read	Arbitrary Write	Arbitrary EEXIT
Information disclosure	✓	✗	✗
Control-flow attacks	(✓)	✓	(✓)
Data-only attacks	(✓)	✓	✗

5.2.1 Enclave Primitives

Intel SGX entrusts enclaves with powerful primitives leading to different memory safety attacks, as depicted in Table 5.1. We outline them in the following.

Arbitrary read. An enclave can read arbitrary memory of the host application, which is intended for exchanging data between enclave and host. Furthermore, an enclave can use hardware transactions to suppress exceptions stemming from reading inaccessible memory [SWG19]. Attackers can misuse hardware transactions as powerful fault-resistant arbitrary read primitive.

Arbitrary write. An enclave can write arbitrary writable host memory, which, again, is intended for data exchange. Furthermore, it can use hardware transactions to suppress exceptions while writing inaccessible or non-writable memory [SWG19]. This yields a fault-resistant arbitrary write primitive.

Arbitrary EEXIT. An enclave can choose the precise code location in the application where execution shall continue after leaving enclave execution via the `EEXIT` instruction. Moreover, the enclave has control over many CPU registers immediately after an `EEXIT`, in particular the stack pointer. Hence, an enclave can configure the application's CPU state before resuming application execution.

5.2.2 Attack Vectors

Given the above primitives, a malicious enclave can mount a broad range of attacks violating memory safety of the host application. In the following, we cluster them into information disclosure, control-flow attacks as well as data-only attacks and give representative examples of these attacks. A detailed overview of attacks violating memory safety was presented by Szekeres et al. [Sze+13].

Information disclosure. A malicious enclave can use the arbitrary read primitive to exfiltrate sensitive user data, such as cryptographic keys or passwords, from the host application. Even if the application contains no such user secrets, an enclave can disclose other sensitive information, e.g., as used in various runtime protection mechanisms. For example, an enclave can derandomize application protection schemes like ASLR [PaX03], stack canaries [Cow98], code randomization [PPK12], or randomization-based control-flow integrity schemes [Kuz+14; Mas+15]. The enclave can furthermore disclose the host application's codebase and, subsequently, generate targeted exploitation payload via ROP chains on the fly. Thus, information disclosure is a powerful tool often used for subsequent exploitation.

Control-flow attacks. A malicious enclave can deliberately tamper with the application's control flow in several ways. For example, it can directly corrupt code pointers, use rogue EEXITs and bypass various mitigation mechanisms.

Code pointer corruption. An enclave can manipulate an arbitrary code pointer of the host using the write primitive. It might corrupt, e.g., return addresses on the stack or virtual function pointers on the heap. As soon as the application fetches a corrupted code pointer, execution diverts to an attacker-chosen address. By carefully crafting a so-called ROP chain (cf. Section 2.1.1) and diverting execution to it, the attacker can gain arbitrary code execution with the privileges of the application, allowing to execute arbitrary syscalls in lieu of the application. In order to prepare a ROP chain, the enclave scans the host application for ROP gadgets using the arbitrary read primitive. It then writes the corresponding addresses on a fake stack using the arbitrary write primitive [SWG19].

A malicious enclave is by no means restricted to ROP attacks only. Similar to ROP, it can craft jump-oriented programming (JOP) attacks, loop-oriented programming (LOP) attacks, or call-oriented programming (COP) by overwriting indirect function pointers [Che+10; Ble+11; Lan+15; CW14; Gök+14]. COOP attacks are also possible by overwriting virtual function pointers in C++ applications [Sch+15]. SROP attacks [BB14] hijack a signal handler.

Rogue EEXIT. A malicious enclave can also mount control-flow attacks without corrupting a single code pointer. By using the arbitrary EEXIT primitive, the enclave can directly corrupt the CPU state. For example, it can manipulate the stack-pointer register to point to an attacker-crafted ROP chain. By doing an `EEXIT` instruction towards an arbitrary `ret` instruction of the host, the enclave can immediately trigger the ROP chain. Thus, rogue EEXITs lead to the same security implications as ROP.

Bypassing Defenses. Several defense mechanisms seek to protect the application's control flow. Stack canaries [Cow98] protect against linear buffer overflows overwriting return addresses on the stack. ASLR [PaX03] hides code addresses via randomization, while others randomize code itself [PPK12]. Both make the generation of ROP gadgets harder. More elaborate mechanisms enforce control-flow integrity (CFI), arguably at different granularity. CPI [Kuz+14] hides code pointers in a shadow stack[2] while CCFI [Mas+15] encrypts code pointers. As these mechanisms rely on randomization, they can be easily broken by the enclave via information disclosure. If CFI metadata is involved, it can be easily corrupted using the write primitive. Stronger hardware-enforced CFI schemes like CET [Int17a] are still unavailable on modern x86 CPUs, and it is unclear to what extent they consider rogue EEXIT attacks.

Data-only attacks. Apart from control-flow attacks, enclaves can corrupt application data other than code pointers or CFI metadata. For example, they can corrupt loop counters, function arguments, or syscall arguments [Car+15; Isp+18] using the arbitrary read/write primitives. Typically, data-only attacks

[2] This corresponds to the weaker randomization scheme since the stronger segment-based isolation is unavailable for 64-bit execution mode.

are much more restricted than control-flow attacks. For example, they can only reuse code reachable in the normal control flow. However, data-only attacks are agnostic to CFI protection schemes and can even achieve Turing-complete computation in many cases by chaining together valid execution paths [Isp+18].

5.2.3 API attacks

The previous attack vectors all violate memory safety of the application by reading, writing, and illegitimately executing application memory. Defeating these attacks is paramount to protecting an application from misbehaving enclaves. Only with such protections in place, it makes sense to reason about the application's security on the API level. SGXJail does not defend against too permissive OCALLs, e.g., giving an enclave the ability to access arbitrary files. We need to ask to what extent an enclave can attack its host application purely via the ECALL/OCALL interface, that is, without relying on the above SGX attack primitives. For example, an enclave can seek to attack the application by crafting invalid API calls or returning malformed data. For a successful attack, either the API itself needs to be flawed, or the underlying implementation misses important validation steps (e.g., confused deputy attacks [Har88] and Iago attacks [CS13]). Since such API-based attacks are highly application-specific, they cannot be addressed by a generic defense mechanism anticipated in this chapter. We discuss proper mitigation strategies in Section 5.5. Also, we do not address the misuse of computational power (e.g., for cryptocurrency mining).

5.3 SGXJail

In this section, we present SGXJail, a novel mechanism to protect host applications from untrusted (third-party) enclaves. SGXJail defeats entire classes of attacks by prohibiting enclave primitives outlined in Section 5.2 at the discretion of the host application. SGXJail can be implemented purely in user space. It relies on process isolation and syscall filters, similar to other sandboxing techniques, such as Docker [Mer14]. We evaluate SGXJail under different workloads to demonstrate its efficiency. Finally, we show how SGXJail can also be implemented via minimal changes to the SGX specifications and corresponding hardware, which we call HSGXJail.

5.3.1 SGXJail via Software Confinement

SGXJail defeats enclave malware by breaking all three enclave primitives described in Section 5.2.1. SGXJail does so by confining enclave operation to a strict set of memory pages.

Figure 5.1 illustrates the basic idea of SGXJail. To break the arbitrary read and write primitives, we rely on the operating system's ability to isolate

processes.[3] Namely, we run potentially malicious or misbehaving enclaves in a separate *sandbox process* that does not have access to the host application's memory. To still allow benign ECALL/OCALL interaction, we establish shared memory between the sandbox process and the host application to implement a form of inter-process communication.

Even with the above process isolation in place, a malicious enclave can perform attacks on the control flow of the sandbox process. An enclave can choose between rogue EEXIT attacks and code-reuse attacks (e.g., ROP) that directly manipulate the host stack [SWG19]. Once successful, the enclave can issue arbitrary syscalls on behalf of the sandbox process. Breaking the primitives that allow an attacker to change the control flow is not trivial. EEXIT can jump to any executable page, and the target address cannot be restricted. Similarly, if the enclave rewrites the saved return address on the stack, the sandbox process cannot detect this modification.

A possible but rather expensive solution is to mark all executable pages of the sandbox process (except for trampoline code) as non-executable before entering the enclave. When leaving the enclave, the sandbox immediately traps to the kernel. The kernel can then assess the legitimacy of the control flow and remap the pages as executable. However, the extra kernel interaction and frequent, expensive page remappings might add considerable runtime overhead.

Instead of trying to protect the control flow in the sandbox process, we confine the damage of hijacking attacks. In particular, SGXJail restricts the syscall interface of the sandbox process by using seccomp syscall filters [Lin17] to whitelist only absolutely necessary syscalls. Even if a malicious enclave gains arbitrary code execution inside the sandbox process, it can no longer perform malicious actions. In contrast to sandboxing techniques such as Docker, which isolate the entire system (e.g., via cgroups), we only need to restrict a single user process for which syscall filters are the appropriate choice.

Life Cycle. A complete SGXJail life cycle works as follows: First, SGXJail creates a new process, the *sandbox process*. It then loads the third-party enclave within the sandbox process. Moreover, SGXJail creates a shared memory between the host application and the sandbox process and installs dispatchers for routing all ECALLs and OCALLs through this shared memory. Afterward, SGXJail activates seccomp filters to restrict the syscalls of the sandbox process to an absolute minimum. Only syscalls required for the communication between application and sandbox process, as well as syscalls required to terminate the sandbox process, are whitelisted. After initialization, the application can issue ECALLs and receive OCALLs, as follows: The application dispatcher automatically encapsulates ECALLs into messages and transfers them via the shared memory to the sandbox process. It copies ECALL function arguments from the host application to the shared memory. The sandbox process dispatcher listens for incoming messages, decapsulates arriving messages, and performs the actual

[3] Conforming with Intel's and our extended SGX threat model, software-based side-channel attacks circumventing such isolation, e.g., Meltdown [Lip+18] or Rowhammer [Kim+14], are out of scope.

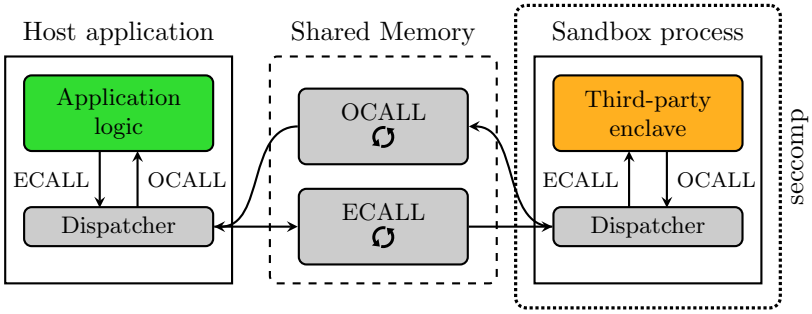

Figure 5.1: With SGXJail, third-party enclaves are isolated within a separate sandbox process. They can communicate with the host application only via shared memory. Also, the sandbox is confined using seccomp filters.

ECALL towards the enclave. Results are returned to the application, again via message passing over shared memory. The application dispatcher finally copies ECALL results from the shared memory to application memory and hands over control to the application. In the same way, OCALLs are routed from the sandbox process through the shared memory to the application host and vice versa. Upon termination of the application, the sandbox process is simply destroyed. Multiple enclaves can be isolated via separate sandbox processes with individual shared memory segments.

Compatibility. SGXJail is a transparent enclave confinement mechanism. It does not require any changes to third-party enclaves themselves, *i.e.*, it is binary-compatible with existing enclaves and their existing cryptographic signatures. Also, no enclave source code needs to be available. Instead, SGXJail is tightly integrated within the SGX SDK [Int16b]. All glue code for dispatching and redirecting ECALLs and OCALLs via shared memory is automatically generated from an enclave's EDL file [Int16b], which needs to be shipped together alongside each pre-compiled third-party enclave. Also, SGXJail provides code for instantiating the sandbox process, the shared memory, and activating seccomp filters. When using SGXJail, one only has to recompile untrusted application code under the SGXJail toolchain.

The installation of seccomp filters is independent of the enclave itself. Since enclaves are not entitled to issue syscalls, the selection of proper syscall filters solely depends on SGXJail and does not affect compatibility with enclaves.

SGXJail enforces benign enclave communication to follow the ECALL/OCALL interface specified in the enclave's EDL file. An enclave implementing other communication methods (e.g., by directly accessing host memory) breaks as soon as SGXJail is active. This is intentional, as enclave developers are strongly encouraged to clearly define the enclave's API via ECALLs and OCALLs. In particular, SGXJail breaks unsafe usage of ECALLs and OCALLs where enclave and host application exchange and dereference raw, unchecked pointers rather than buffered data. For example, if one marks an ECALL function parameter with the so-called `user_check` attribute within the EDL file [Int16b], the SDK

passes this function parameter without further checking and copying into the enclave. A quick code inspection revealed usage of `user_check` in some Intel architectural enclaves and remote attestation code, all for performance reasons. They could be updated to avoid `user_check` at the cost of slight performance loss. To yet support `user_check`, one would need to manually share (*i.e.*, map) host memory with the sandbox process to which an enclave shall have unrestricted access. Also, host application pointers passed to the enclave need to be translated to the sandbox process due to ASLR. SGXJail could provide simple helper functions for sharing host memory and translating pointers.

5.3.2 Implementation Details

For generating dispatcher code, we extend the `edger8r` tool [Int16b] accordingly. An enclave always copies arguments to enclave memory before processing it, because it distrusts the host. Conversely, before accepting an OCALL, our dispatchers copy arguments from distrusted enclaves to host memory. Argument copying prevents TOCTOU vulnerabilities, such as double-fetch bugs [Wan+18], by design.

ECALLs and OCALLs are routed between the application and the sandbox process via two distinct shared memory regions, one for each direction. The dispatchers synchronize ECALL/OCALL interaction via shared semaphores. Semaphores hav the advantage that processes (application and sandbox) are consuming no CPU time while waiting for the other communication partner. For receiving OCALLs, the application installs a separate listener thread that only gets active upon incoming OCALLs.

Selection of appropriate syscall filters is crucial for the security of SGXJail, as a malicious enclave can directly exploit a lax configuration (e.g., via rogue EEXIT attacks). It is favorable to restrict both the number of syscalls as well as their complexity to reduce the attack surface given by the whitelisted syscalls. Syscall filtering also has an impact on the type of inter-process communication between the sandbox and the application process. By choosing shared memory as a communication channel, we do not require any syscall for the actual communication and only one syscall (`futex`) for synchronization. In summary, we configure seccomp [Lin17] only to allow the syscalls `futex` necessary for semaphores as well as `exit_group` for terminating the sandbox process. Thus, the shared memory approach results in only one whitelisted syscall in addition to the required `exit_group` syscall. Unless the implementation of these two syscalls is buggy, they cannot cause a security violation when issued by a malicious enclave.

The SGX SDK passes OCALL function arguments from the enclave to the application via the application's stack. The enclave knows the application's stack location via the stack pointer (`RSP` register), which is preserved by the `EENTER` instruction. Hence, it can allocate a stack frame on the host stack via a function called `sgx_ocalloc` and store any outgoing OCALL arguments there. One can leverage this mechanism for reducing SGXJail overhead, as follows: Currently, when doing an OCALL, our sandbox dispatchers copy OCALL arguments from the sandbox to the shared memory. By modifying `RSP` immediately before an `EENTER`

to point to the shared memory, one can instruct the enclave to write OCALL arguments directly to the shared memory instead of the sandbox application's stack. When the enclave `EEXIT`s, one can simply restore the original sandbox stack (namely, `RSP`).

In our current implementation, the size of the shared memory is hard-coded to three pages for each direction. For ECALL/OCALL arguments exceeding the shared memory, one can dynamically resize the shared memory on demand. Although multithreaded enclaves are currently not supported by our prototype implementation, support can be easily added: one would install separate semaphores and shared buffers for all enclave threads that are enumerated in a public enclave XML configuration file. Also, support for nested calls (OCALLs issuing ECALLs) can be added by adapting the synchronization mechanism appropriately.

An interesting question arises whether SGXJail should be integrated with the SGX SDK in a way that does not demand recompilation of the application. Thus, system administrators could globally enforce SGXJail by just installing corresponding shared libraries. Since the enclave's EDL file is public already and will be distributed alongside third-party enclaves, the generation of dispatcher code is straight forward. One would also need to hook the enclave API of the unmodified application binary and inject dispatcher code. This could be done by preloading SGX SDK libraries (in particular, `sgx_urts.so`).

5.3.3 Evaluation

SGXJail does not affect native runtime performance of host applications or enclaves. That is, as long as no interaction between enclave and application takes place, they can run without performance loss. The only performance overhead occurs when doing ECALLs and OCALLs due to the message passing via shared memory and the necessary synchronization. To evaluate this effect, we first present microbenchmarks for bare metal ECALL and OCALL latency, which are followed by macrobenchmarks on more representative workloads.

Test Setup. Evaluations are done on a commodity notebook featuring an Intel i5-6200U CPU, a Samsung SM951 SSD, and running Ubuntu 16.04 Desktop with SGX SDK version 2.4. For the benchmarks, we disabled the screen as well as network interfaces to reduce noise from screen redrawing or external interrupts. Also, we fixed the CPU frequency to its maximum (2.3 GHz) and pinned benchmarks to a single core. The benchmarks include a warm-up phase.

Microbenchmarks. To measure the ECALL latency, we implemented a simple ECALL and measured its execution time from within the host application. That is, the ECALL latency includes EENTER, EEXIT, all glue code for the enclave and the host, as well as context switching and synchronization between application and sandbox for SGXJail. To measure the OCALL latency, we, in addition, perform one simple OCALL from within the ECALL and subtract the ECALL latency. We repeated the measurement 500 times. The resulting latencies are shown in Table 5.2. The raw ECALL latency increases from $15.6 \cdot 10^3$ cycles to $22.1 \cdot 10^3$ cycles while the OCALL latency increases from $13.4 \cdot 10^3$ cycles to $19.5 \cdot 10^3$ cycles. Hence, the absolute latency remains small. Since

Table 5.2: ECALL and OCALL latency in CPU cycles of SGXJail compared to the unprotected Vanilla version. The standard deviation is shown in braces.

Latency	ECALL	OCALL
Vanilla	15 624 (± 301)	13 438 (± 1046)
SGXJail	22 094 (± 814)	19 515 (± 1360)

many practical usage scenarios of SGX involve somewhat complex computations inside the enclave, the actual runtime overhead is much lower than the pure ECALL/OCALL overhead.

Macrobenchmarks. Quantifying the performance of enclaves is highly application-specific. Unfortunately, enclaves are not widely deployed yet, and standardized benchmarking suites are unavailable to the best of our knowledge. A common approach is to port existing programs to an enclave [WBA17]. While this sounds appealing, it tends to introduce many unnecessary OCALLs to the standard library, which well-designed enclaves would not perform, e.g., the `getpid` syscall in openVPN [WBA17].

Instead, we quantify the performance of SGXJail as follows: First, we benchmark a synthetic workload under different OCALL frequencies. The results of this benchmark are generic and can be applied to any enclave for which the OCALL frequency can be determined. Second, we benchmark storage of sensitive enclave data to disk via the Intel protected filesystem (PFS) [Sel16]. The PFS is integrated within the SGX SDK and is likely to be used by a vast number of enclaves.

For our first benchmark, we observe that an enclave typically issues OCALLs to perform syscalls, e.g., writing to files. Our benchmarked OCALL performs a `close` syscall on an invalid file descriptor. Such a fast syscall gives an upper bound on the performance overhead since longer syscalls decrease the influence of the OCALL overhead. We repeated each measurement 100 times. The OCALL-to-enclave ratio (w.r.t. their runtime), as well as the overhead of SGXJail, compared to unprotected Vanilla applications, is given in Figure 5.2. The simple standard deviation is shown as an area under the curves. We execute a fixed baseline workload inside the enclave, which corresponds to 2201.44 (± 25.67) $\cdot 10^6$ cycles, or 0.96 (± 0.011) s on our 2.3 GHz CPU. As this workload runs within the enclave, we quantify it as enclave seconds or Esec. While we keep the enclave workload constant, we issue OCALLs at different frequencies and measure the additional OCALL work. The corresponding ratio is shown on the left axis of Figure 5.2. It allows us to decouple the OCALL overhead from the OCALL frequency, which we quantify as OCALLs/Esec.

One can see that the overhead of SGXJail is virtually non-existent for low-frequency OCALLs, meaning that pure enclave execution is not impeded by SGXJail at all. Even for 10 000 OCALLs/Esec, the overhead is below 3%, and for a large number of 50 000 OCALLs/Esec the overhead is only around 11%. To put these numbers into perspective, Netflix observed a maximum of 50 000 OCALLs/s across their systems [GHG18]. For even higher OCALL

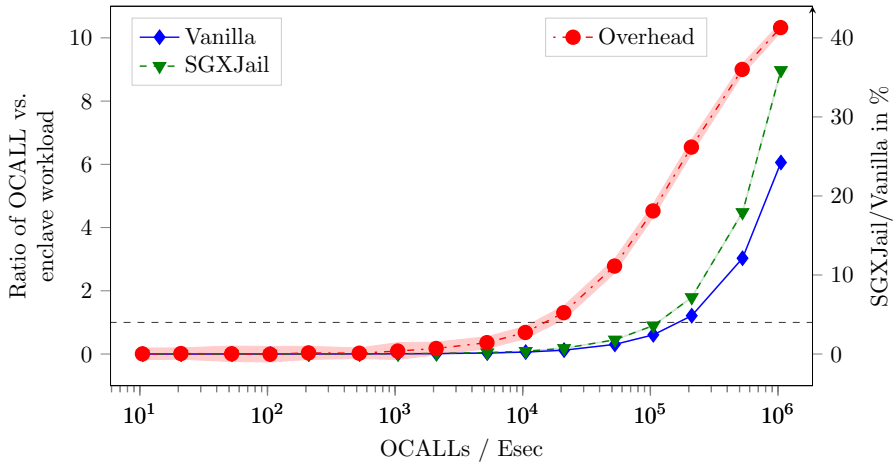

Figure 5.2: Benchmark on unprotected (Vanilla) and hardened (SGXJail) applications, plotted over different numbers of OCALLs per enclave second (Esec).

frequencies, the OCALL workload starts to exceed the enclave workload in the vanilla version already. With SGXJail, enclaves can issue up to 113 000 OCALLs/Esec before OCALL processing exceeds actual enclave computations (ratio=1). For unprotected apps, this point is reached for 164 000 OCALLs/Esec. Such situations should be dealt with in practice by redesigning the enclave API and reducing or removing unnecessary OCALLs. Nevertheless, SGXJail only introduces around 20% overhead, even in this extreme case.

Our first benchmark measures raw OCALL performance. However, this does not reflect the performance of copying OCALL arguments between enclave and application. To evaluate the maximum overhead of a real-world scenario, we benchmark an enclave that only accesses files via the Intel protected file system (PFS) library [Sel16]. PFS is shipped with the SGX SDK and is intended for sealing sensitive enclave data on the host file system for persisting state across reboots. To resemble a worst-case scenario of PFS, we implement and benchmark a single ECALL that opens a new file (`sgx_fopen_auto_key`), writes a fixed-size buffer (`sgx_fwrite`), and immediately closes the file again (`sgx_fclose`). We repeat the measurements 200 times. After each run, we delete the file and synchronize the file system to capture the overhead of PFS reliably. Figure 5.3 shows the PFS performance for different payload sizes up to 1MB. The runtime includes enclave as well as OCALL computation. The simple standard deviation is shown as an area under the curves.

The maximum overhead for protecting PFS with SGXJail is roughly around 20%. There is almost constant runtime up to 2 kB payloads for SGXJail and the unprotected vanilla enclave, with a sudden increase at 4 kB payloads. The reason is that the PFS library caches smaller chunks of data and defers actual file writing to closing the file with `sgx_fclose` with 8 OCALLs in total. We manually checked the PFS library and found that a PFS block can hold up

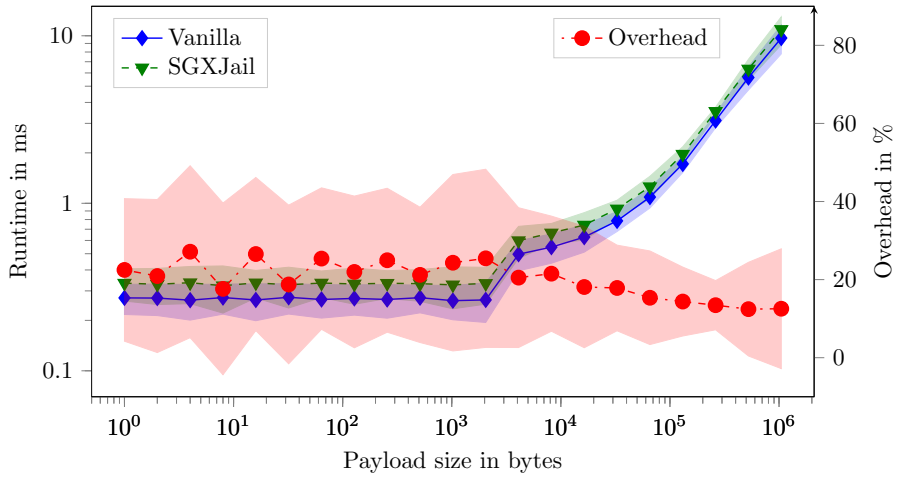

Figure 5.3: PFS runtime of SGXJail compared to unprotected Vanilla enclaves for different payload sizes.

to 3072 bytes (specified via `MD_USER_DATA_SIZE`). When exceeding the internal buffer of 3072 bytes, the PFS library flushes data to the file system using seven more OCALLs, resulting in the sudden increase of the absolute runtimes for SGXJail and the vanilla enclave.

For larger payloads (4 kB and more), the overall overhead does not increase but falls below 20%. This suggests that argument copying itself is not the bottleneck of PFS. We verified this by manually removing argument copying in the sandbox for the actual file write OCALL. Using 1 MB payloads, the overhead dropped by roughly 3%. Rather than argument copying, the runtime overhead of SGXJail is dominated by the OCALL overhead since the PFS implementation chops larger payloads into a sequence of smaller OCALLs. In fact, for 1 MB payloads, we observed 313 OCALLs in total.

We have shown that SGXJail does not impede pure enclave computation (0% overhead). For real-world workloads up to 10 000 OCALLs/Esec, the overhead is below 3% (cf. Figure 5.2). Even for uncommonly high OCALL frequencies (100 000 OCALLs/Esec), the overhead of SGXJail is still below 20%, whereas plain writing of protected files with high OCALL interaction comes at only 20% overhead. To further improve performance, SGXJail could use HotCalls for faster enclave communication [WBA17]. Alternatively, we propose a lightweight hardware extension (HSGXJail) that provides SGXJail isolation at virtually no overhead.

Memory overhead. SGXJail requires one additional process for the sandbox. As for site isolation in browsers [Rei18], this incurs a constant memory overhead sandbox and the shared memory.

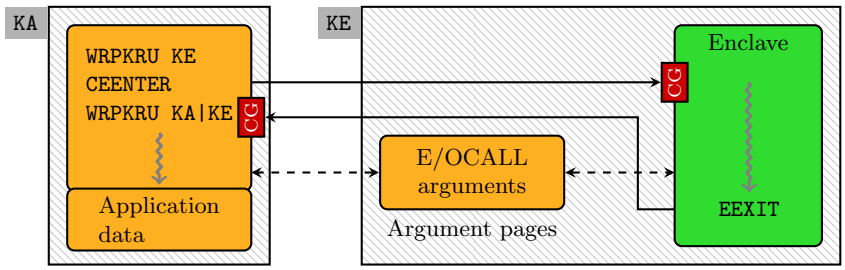

Figure 5.4: HSGXJail confines the enclave to pages marked with memory protection key `KE`. Thus, the application can protect its pages via a disjoint memory protection key `KA`. ECALL/OCALL interaction is constrained to shared `KE` pages (dashed lines). Moreover, `EEXIT` can only target a single exit point, namely the instruction following a `CEENTER` (a new confined `EENTER` instruction). This resembles an exit call gate (CG).

5.3.4 HSGXJail via Hardware Confinement

In this section, we propose a more efficient defense mechanism via a minimal change to the SGX specification concerning Intel memory protection keys (MPK), *i.e.*, disallowing one MPK instruction in SGX.

To prevent an enclave from accessing host application memory, we propose a stricter page access policy. To that end, HSGXJail introduces two extensions: first, data confinement and second, control confinement. First, memory regions that are not supposed to be used by the enclave shall be inaccessible to the enclave. Data confinement limits memory pages an enclave can read or write, thus breaking the arbitrary read and write primitives. Second, `EEXIT` shall be only allowed on well-defined exit points. Control confinement prevents the enclave from misusing `EEXIT` to jump to arbitrary host application code, thus breaking the arbitrary `EEXIT` primitive.

Data Confinement with Intel Memory Protection Keys. The central issue of enclave malware is an asymmetry in the memory access policy, granting enclaves unrestricted access to host-application memory. Data confinement uses a recent protection mechanism called memory protection keys (MPK) [Int16a] to partition virtual memory into enclave-accessible memory and protected application memory. If the enclave attempts to access protected application memory, the CPU raises a page fault. To prevent the enclave from reconfiguring MPK, HSGXJail disallows specific MPK instructions in enclave execution mode. Similar to SGXJail (cf. Section 5.3.1), we use this mechanism to confine enclave execution to a narrow ECALL/OCALL interface, as shown in Figure 5.4.

Memory protection keys work as follows: they augment page-based read, write, and execute permissions with additional access policies. Each application page can be assigned one particular memory protection key. This protection key is stored directly in the corresponding page table entry (PTE). By assigning different protection keys to different pages, MPK allows to partition virtual

memory pages into 16 disjoint protection domains. The PKRU CPU register controls which access policy is applied to those protection domains. For each of the 16 protection keys, PKRU allows to selectively disable write and read access for the current execution thread. The PKRU register can be updated via the unprivileged `WRPKRU` instruction, enabling frequent switching of protection domains within the application. Since each CPU thread maintains its own local PKRU register, MPK supports multithreading.

For HSGXJail, we partition the application into protection key `KA` comprising all application pages and `KE`, covering enclave memory as well as argument pages, as shown in Figure 5.4. Immediately before entering an enclave, the application configures PKRU to confine memory accesses to the enclave only (`WRPKRU KE`). During enclave operation, the enclave can only access argument pages for ECALL/OCALL arguments. After leaving the enclave, the application re-enables full access to the application itself (`KA`) as well as the argument pages (`KE`) via `WRPKRU KA|KE`.

To prevent the enclave from manipulating MPK by reconfiguring the PKRU register, HSGXJail demands a slight modification to the SGX specification. Whenever HSGXJail is active, the `WRPKRU` instruction is disallowed for the enclave and raises an invalid opcode exception instead. This change should be simple to deploy via a microcode update to the CPU.

HSGXJail poses no limit on the number of applications using third-party enclaves. However, the number of enclaves within a single application is restricted. Since MPK supports up to 16 different protection domains, HSGXJail can natively secure applications utilizing up to 15 distinct enclaves. Note that one protection domain is needed for the application itself. To support more enclaves per application, one can follow various approaches: First, in many cases enclaves provide simple functionality, e.g., ECALLs without OCALLs, or OCALLs for issuing syscalls but not towards other enclaves. In these cases, enclaves are never called in an interleaved way and, thus, are never concurrently active. Hence, the application can safely share the same argument pages and also the same protection key among those enclaves. This increases the number of supported enclaves by the degree of enclaves which are not interleaved with other enclaves. Second, memory protection keys can be dynamically updated and scheduled among different enclaves. While this supports an arbitrary large number of enclaves per application, it incurs additional performance penalty in updating protection keys in the PTEs.

Control Confinement. Whenever leaving enclave execution (via ECALLs and OCALLs), the enclave jumps into the host application via an `EEXIT` instruction. However, since the enclave can freely choose the jump target of `EEXIT`, a variety of code-reuse attacks become possible (cf. Section 5.2).

Data confinement already limits an enclave's read and write access using MPK. While MPK protects data accesses, it does not prevent fetching code from other protection domains. This design choice is intentional to enable application code to update protection domains without accidentally removing access to its

own code. Hence, data confinement does nothing to protect an application from rogue EEXITs.

To break the arbitrary EEXIT primitive, HSGXJail restricts EEXIT to a single valid *exit point*. In particular, EEXIT can only target the instruction immediately following a so-called CEENTER instruction. This *exit point* is similar to the enclave *entry points* used to protect an enclave from malicious applications, both of which are shown as call gates (CG) in Figure 5.4.

Control confinement can be easily implemented via small changes to SGX. We propose to extend the semantics of EENTER via a novel confined CEENTER instruction. From the enclave's perspective, CEENTER behaves exactly as EENTER. EENTER already stores the exit point (*i.e.*, the address of the instruction immediately following EENTER) in register RCX. However, SGX leaves it up to the enclave to store this exit point and later on pass it to EEXIT. In contrast, our CEENTER instruction additionally stores the exit point in a protected, thread-local storage called OEXIT which is inaccessible to the enclave. To make use of this exit point, we propose to adapt the semantics of the EEXIT instruction, as follows: Instead of jumping to a target provided by the enclave via register RBX, our EEXIT ignores RBX and instead directly jumps to the address stored in the protected OEXIT register. Both, CEENTER and EEXIT can be implemented in CPU microcode.

Compatibility. To be fully compatible with existing enclave software, we activate HSGXJail only on demand. If the application issues a normal EENTER instruction, HSGXJail is inactive, and SGX behaves as usual. When entering the enclave via our new confined CEENTER instruction, HSGXJail is active until EEXIT. Moreover, HSGXJail's slim design is fully compatible with advanced SGX features, such as multithreading, dynamic memory management, and virtualization [Int16a]. The availability of HSGXJail could be indicated via a model-specific register.

Software Considerations. HSGXJail protects applications from existing, unmodified third-party enclaves. HSGXJail can be integrated entirely within the SGX SDK [Int16b], thus being fully transparent to existing application code. This allows using HSGXJail by recompiling applications without the need to rewrite any application code.

To use HSGXJail, the SDK needs the following slight adaptations. First, the SDK replaces EENTER with CEENTER in the untrusted urts library. The urts library already uses a single exit point, which is the address immediately following EENTER. The corresponding trusted trts library belonging to the enclave performs EEXIT only towards this single exit point. Since our modified EEXIT instruction enforces the same exit point, it does not change the behavior of benign enclaves. No changes to the trts library are required. Benign enclaves compiled under the original trts library work out of the box.

For data confinement, the SGX SDK needs to establish enclave-accessible argument pages reflecting the ECALL/OCALL interface, and configure memory protection keys accordingly. By default, all application code runs with protection key *zero*. Thus, the SDK assigns protection keys, starting with *one*, to all enclave

pages as well as the corresponding argument pages. Similar to the software-only variant, SGXJail, the SDK can do this once when loading a new enclave.

When doing an ECALL, the SDK additionally copies all input arguments from application memory to an enclave-accessible argument page. In the same way, the SDK copies back any output arguments from the argument page to application memory at the end of an ECALL. The same applies to OCALLs. While argument copying causes some overhead, it is deemed necessary to generically prevent TOCTOU attacks and guarantee the security of the application. For the same reason, the enclave copies untrusted application arguments to enclave memory before operating on it.

Before entering the enclave, the SDK saves all necessary CPU registers in application memory, clears sensitive content from the registers, and configures the application's stack pointer `RSP` to point to one of the argument pages. Configuring `RSP` in that way causes the enclave to read and write any OCALL arguments directly from/to the argument page, which is enclave-accessible, without additional copying overhead. After leaving the enclave, the SDK restores the application's CPU registers, including the stack pointer.

Performance Estimates. The functional changes for HSGXJail are minimal. For data confinement, `CEENTER` disallows usage of the `WRPKRU` instruction, which can be easily implemented in the CPU. Data confinement via MPK shows the same performance as for MPK without HSGXJail. For control confinement, the microcode changes we propose to `CEENTER` and `EEXIT` are minimal as well. They only comprise saving and restoring the exit point `OEXIT` in trusted thread-local storage. One could store `OEXIT` in the thread control structure (TCS) [Int16a]. Hence, it is reasonable to expect a negligible overhead of HSGXJail in every aspect, far lower than the overhead of the software-based SGXJail variant.

5.4 Related Work

Defense by Detection. Researchers proposed to detect enclave malware by monitoring their I/O behavior [DF14; CD16]. However, this is believed to be infeasible in practice [Mar18]. Others proposed analyzing enclave code before actually running it [CD16], which is not feasible for generic loaders. Generic loaders can remotely fetch arbitrary malicious code at runtime. Refusing such generic loaders would annihilate all use cases for protecting intellectual property. Instead, Costan et al. [CD16] proposed to force generic loader enclaves to embed malware analysis code within the enclave. However, it is unclear how effective this technique is in detecting malicious code. It also raises the question of who decides which analysis code to embed and to ensure the analysis code does not leak enclave secrets. Also, the analysis code cannot be easily updated, and enclaves without analysis code cannot be executed without risk.

Defense by Prevention. While applying control-flow integrity (CFI) to the host application sounds appealing, it does not close all attack vectors outlined in Section 5.2. Although hardware-assisted CFI can prevent some control-flow attacks [Int17a], they are not yet available and might miss rogue EEXIT attacks.

Software CFI schemes [Kuz+14; Mas+15] can simply be bypassed by leaking secrets and corrupting CFI metadata via the arbitrary read and write primitives. Moreover, no CFI scheme can prevent data-only attacks.

Readactor [Cra+15], Heisenbyte [TSS15], and NEAR [Wer+16] severely limit the arbitrary read primitive necessary for many attacks by forcing page faults when trying to access sensitive code. However, they have significantly larger overhead than SGXJail, and blind ROP attacks might still be possible [Bit+14]. Ryoan [Hun+16] executes malicious enclaves inside a software sandbox using software fault isolation (SFI). However, Ryoan demands recompilation of the enclave with SFI, which cannot be applied in our setting. Also, Ryoan severely restricts the enclave life cycle to a single stateless invocation, which is incompatible with generic third-party enclaves.

5.5 Discussion

Since the very first blog post in 2013 [Rut13], the enclave malware threat has been discussed at a high level but was mostly disregarded by the research community. With recent attacks showing powerful and practical enclave malware, research on proper defense mechanisms becomes pressing.

In this chapter, we identified three enclave primitives, namely arbitrary memory reads, writes, and `EEXIT`s. We believe these primitives lie at the heart of the enclave malware threat by exposing an application to a variety of runtime attacks originating from misbehaving enclaves. Although these primitives might increase the flexibility of different SGX programming models, they give rise to enclave malware. In fact, they are unnecessary in practice, as enclaves ought to strictly comply with the defined ECALL/OCALL interface. In particular, the enclave runtime services offered by the SGX SDK demand precise EDL specification of the data exchanged, and bypassing this specification is considered bad practice. Moreover, the SDK uses only a single enclave exit point, from which all ECALLs and OCALLs are dispatched.

Based on these observations, we proposed (H)SGXJail to confine enclave primitives to the narrow interface specified by the EDL file. (H)SGXJail applies the principle of least privileges [SS75] also to enclaves and closes an entire class of runtime attacks, including information disclosure, control-flow attacks, as well as data-only attacks. Even more, by automatically copying ECALL/OCALL arguments from and to application memory, (H)SGXJail prevents double-fetch bugs [Wan+18] by design.

SGXJail paves the way for reasoning about application security based on application code only (*i.e.*, without trusting any enclave code), and the ECALL/OCALL interface in particular. While SGXJail defeats an entire class of runtime attacks, it cannot solve the problem of too permissive host interfaces, e.g., a syscall proxy [Mar18], which allows executing arbitrary syscalls. Further research on designing and validating ECALL/OCALL interfaces is needed to avoid API-level attacks via too permissive OCALLs or confused deputy [Har88] and Iago attacks [CS13]. In general, one has to consider enclave-to-host commu-

nication not as asymmetric (cf. the kernel's syscall interface) but as part of a mutually distrusted API where both communication parties distrust each other. Mutual distrust is an integral part of designing secure web APIs. Since enclave malware raises similar threats as web applications, we also see some overlap in defense strategies. In particular, input validation or sanitization [OWA19, Section V5] can help prevent Iago-style attacks while verification of the logical execution flow [OWA19, Section V11] can prevent confused deputy attacks.

Closing Side Channels. Several side-channel attacks mounted against benign SGX enclaves have been shown [Göt+17; Bra+17b; Lee+17b; Wan+17b; MIE17; XCP15]. Moreover, malicious enclaves themselves can mount side-channel attacks [Sch+17; Gru+18; SWG19]. Although not the primary focus of this work, SGXJail prevents a variety of side-channel attacks that rely on accessing host application memory, e.g., Flush+Reload on shared host libraries used by the host application from within enclaves, Prime+Probe using host application arrays [Sch+17], Rowhammer attacks from within enclaves [Gru+18] as well as TSX-based address probing [SWG19].

5.6 Summary

While designed to increase the security of a computing system, secure enclave technology, such as Intel SGX, might also be misused for shielding malware inside enclaves. However, research on potential enclave malware is still in its beginnings, and practical defense mechanisms are virtually non-existent.

In this chapter, we identified the root cause of enclave malware as insufficient enclave-to-host isolation. We proposed SGXJail as a generic defense against a wide range of enclave malware threats. It enforces mutual isolation between host applications and enclaves, thus protecting applications from potentially misbehaving or malicious third-party enclaves. SGXJail is an efficient and transparent software defense, running third-party enclaves in an isolated sandbox. Our proof-of-concept implementation shows zero overhead for pure enclave computation and less than 3% for realistic workloads. SGXJail is tightly integrated within the SGX SDK and can be used out of the box. Furthermore, we proposed SGXJail directly in hardware. Our HSGXJail mechanism provides enclave confinement utilizing Intel MPK with slim extensions to the SGX specification at virtually no cost. We believe HSGXJail should be immediately rolled out via a microcode update to SGX-enabled CPUs to enable our SGX malware defense proactively. However, support for MPK is still rare. Although some server CPUs support MPK [Zha19], it is unclear when x86-based desktop CPUs catch up.

Apart from defending against enclave malware, (H)SGXJail opens up new use cases for Intel SGX and similar isolation technologies. For example, we envision that (H)SGXJail can be used as a lightweight and secure sandboxing mechanism for browser site isolation or plugin management, where third-party code has proven to be both potentially malicious and potentially security-critical.

Part II

Address-based Side Channels

✳ ✳ ✳

6
Software Side Channels

> But when you give to the needy, do not let your left hand know what your right hand is doing, so that your giving may be in secret. Then your Father, who sees what is done in secret, will reward you.
>
> *Jesus Christ* – Gospel of Matthew

Side channels are a subtle yet dangerous threat to security architectures and enclaves, in particular. A side-channel vulnerability might leak cryptographic keys or other secret information and, thus, completely circumvent traditional isolation boundaries. To understand their nature, we first introduce the notion of the main channel. In Section 2.1, we gave two prominent classes of threat models. In the first example, an attacker provides malformed input to trigger vulnerabilities in a piece of software and gain arbitrary code execution. In the second example, the attacker wants to elevate privileges further by corrupting other components on the system. In either case, the attacker misuses a designated communication interface, *i.e.*, the main channel, in order to subvert a system.

Side channels, on the other hand, are not designed on purpose but unintended. They rather occur as a side effect of various optimizations, resource sharing, or other design decisions. Also, side channels do not violate the integrity but the confidentiality of certain operations. A prominent example is the timing channel that leaks the amount of time an algorithm or program takes to complete or react. Cache timing channels exploit memory access patterns on a shared cache, which again influences execution timing. Enclave architectures, such as Intel SGX, even exhibit strong page-fault side channels by design.

In the following, we introduce prominent side-channel attacks that are relevant to this thesis. We discuss many software side-channel vulnerabilities found in literature and their consequences. We also put various vulnerabilities we

discovered and described in Chapters 7 to 9 into the context of related work. Furthermore, we outline how to defend against side-channel attacks on three layers. First, one can try to harden the architecture against the exploitation of side channels. Second, one can close side-channel vulnerabilities in software to prevent secret information leakage. This requires knowledge of where in the code actual vulnerabilities reside. Third, we discuss several tools that were developed to ease the search for and analysis of side-channel vulnerabilities. Again, we compare our analysis tool developed in Chapter 8 against related work.

6.1 Side-channel Attacks

Various side channels exist. They range from physical side effects, such as power consumption [KJJ99], electromagnetic [Eck85], or acoustic emanations [Bac+10], over side channels leaking keystroke timings [SWT01], and action-based side channels in the IoT [BBS18] towards powerful software side channels undermining the security of cryptographic implementations. In this thesis, we focus on the latter.

Address-based Side Channel. Most side-channel attacks analyze timing behavior to derive secret input. Measuring and analyzing the overall execution time of cryptographic primitives has been shown to leak private keys of asymmetric encryption schemes [Koc96] as well as symmetric encryption schemes [Ber05]. The timing channel might be purely on an algorithmic level [Koc96] or caused by the underlying microarchitecture [Ber05].

In this thesis, we primarily focus on side channels that stem from memory access patterns. We generalize them as so-called *address-based side channels* in Chapter 8 since different memory addresses are accessed, depending on some secret value. We distinguish between data and control-flow leakage. Data leakage occurs if accessed memory locations depend on secret inputs. Control-flow leakage occurs if code execution depends on secret inputs. Many microarchitectural attacks [Ber05] operate on an address-based side channel. Also, algorithmic channels, such as [Koc96], fall into this category, as executed code addresses depend on the secret key.

Data-based Side Channels. Orthogonal to address-based side channels are attacks that exploit data leakage. For example, instructions with variable execution time can leak the processed data. Some Arm processors have variable time multiplication [Gro+09] and division [PPM17] instructions, where the execution time depends on the input operands. On x86, variable-time behavior was observed for floating-point instructions [And+15], as well as integer division [CCS12]. Tsai et al. [Tsa+20] showed that compression of cache lines leaks data stored therein and allows so-called Pack+Probe attacks. Also, transient execution attacks [Lip+18; Koc+19; Che+19; Kor+18; Bul+18; Sch+19a; Sch+19b; Sch+20a; Mur+20] fall outside the category of pure address-based side channels. They exploit speculation on actual data (e.g., branch decisions, jump targets, or written data) rather than addresses only.

6.1.1 Microarchitectural Attacks

Microarchitectural side-channel attacks rely on the exploitation of information leaks resulting from the contention of shared hardware resources. Especially microarchitectural components, such as the CPU cache, the DRAM, and the branch prediction unit, enable powerful attacks that can be conducted from software only. In these examples, the contention is based on memory addresses. For instance, attacks exploiting the different memory access times to CPU caches (aka cache attacks) range from pure timing-based attacks [Ber05] to more fine-grained attacks that infer accesses to specific memory locations by manipulating cache state. Cache attacks can target code access patterns via Flush+Reload [GBK11; YF14] or data accesses via Prime+Probe [Per05; OST06; TOS10]. Flush+Reload flushes individual cache lines. Thus, an attacker can observe memory accesses on a cache-line granularity, which is typically 64 bytes. However, Flush+Reload demands that both the victim and the attacker process share some common code pages, such as a shared library, for example. In contrast, Prime+Probe also works on data memory. It can even be carried out over the last level cache [Liu+15b] across different virtual machines. However, Prime+Probe has lesser accuracy and requires knowledge of physical addresses. In a hyper-threaded setting, the level one cache lines are further divided into cache banks. Hence, attacks on a sub-cache-line granularity become possible [YGH16]. DRAM row buffers have been used to launch side-channel attacks [Pes+16] by exploiting row buffer conflicts of different memory addresses. Other shared components can be exploited as well, such as translation lookaside buffers [HWH13] or the branch predictor [AGS07; AKS07b; AKS07a]. Branch prediction represents a special type of cache attack that exploits the branch target buffer (BTB) cache in order to learn information about executed branches. For a detailed overview of microarchitectural attacks, we refer to recent survey papers [Ge+18; Sze19].

6.1.2 Attacks in SGX Settings

Most microarchitectural attacks also apply in an SGX setting. Even worse, SGX promotes existing side channels and creates new ones since an SGX attacker typically has full control over the operating system.

Cache Attacks. Since enclaves do not share the memory with other processes, Flush+Reload attacks are not directly possible against enclaved programs. An enclave could, however, perform Flush+Reload attacks via its host application memory towards another non-enclaved program. Nevertheless, other techniques such as Prime+Probe can be applied to enclaves. Götzfried et al. [Göt+17] demonstrated a Prime+Probe attack on an SGX enclave running an AES T-table implementation. They queried the performance monitoring unit (PMU) in order to observe the number of cache hits and cache misses precisely. While the PMU does not directly monitor performance metrics of enclaves, Götzfried et al. inferred enclave cache behavior indirectly by probing their own memory accesses with the PMU. Similarly, Moghimi et al. [MIE17] demonstrated a Prime+Probe attack against enclaves running an AES T-table implementation as well as an

S-box implementation. They increased the accuracy of the cache observations by interrupting the enclave repeatedly using the Advanced Programmable Interrupt Controller (APIC). These cache attacks naturally suffer from false positives and false negatives, even though the operating system can be instrumented to minimize the influence of noise. Brasser et al. [Bra+17b] performed a Prime+Probe attack against an SGX enclave running RSA decryption. Schwarz et al. [Sch+17] showed that the Prime+Probe attack against an enclave itself could be hidden inside another enclave.

TLB. Although Flush+Reload cannot be applied to enclaved programs directly, van Bulck et al. [Bul+17] proposed to use Flush+Reload to attack the page table entries managed by the operating system. This reveals which pages have been accessed by the enclave. Thereby, they defeat countermeasures that aim to detect page faults [Shi+17a; Shi+16] or that mask the *accessed* and *dirty* flags of page table entries. However, their attack comes at the cost of an even coarser granularity (32 KB) since one cache line holds eight PTEs.

DRAM. Wang et al. [Wan+17b] showed that DRAM-based side-channel attacks [Pes+16] could also be applied in an SGX setting across different SGX enclaves. They improved DRAM-based attacks by combining them with a Prime+Probe attack. Prime+Probe ensures that enclave memory accesses bypass the enclave page cache (EPC) and reach DRAM.

Branch Prediction. Lee et al. [Lee+17b] observed that SGX does not clear the branch history when switching between enclave mode and non-enclave mode, which enables branch shadowing attacks. Branch shadowing represents an enhanced version of branch prediction analysis (cf. [AKS07a]), which relies on the last branch record (LBR). They used APIC timer interrupts to increase precision. Evtyushkin et al. [Evt+18] target the directional predictor rather than the branch target buffer (BTB). Huo et al. [Huo+20] expand this attack on the two-level directional predictor.

Hyper-threading. On hyper-threaded systems, multiple logical cores share the same execution units. Aldaya et al. [Ald+18] exploit contention of execution ports to attack enclaved programs running on the same physical core. Moghimi et al. [MES18] show that level one caches can be exploited to attack hyper-threaded enclaves. Unlike [YGH16], they exploit false dependencies due to read-after-write operations. By accessing aliased addresses, they introduce false dependencies in the CPU pipeline, causing an artificial delay if the victim enclave accesses specific target addresses.

Controlled-Channel Attacks. Xu et al. [XCP15] demonstrated a new class of attacks on Intel SGX, called controlled-channel attacks. They are also referred to as pigeonhole attacks [Shi+16] or page-level attacks [Xia+17]. Controlled-channel attacks rely on the fact that the untrusted operating system controls the mapping between virtual and physical pages for all processes, including enclaves. Hence, the OS can modify the *present* bit for page table entries (PTEs) to induce page faults into enclaves. Whenever an enclaved process accesses unmapped pages, the CPU delivers a page fault to the OS. Thus, the OS can observe the memory accesses or executed code paths of an enclave at page granularity. Unlike prior

side channels, this controlled channel yields noise-free observations. Instead of using the *present* bit, page faults can also be triggered by making pages non-executable [Xia+17] using the *non-executable* (NX) bit, or by setting a *reserved* bit [XCP15; Xia+17].

Interrupt-driven Attacks. Previous page-fault based attacks could not monitor the execution of single instructions on a page. Hähnel et al. [HCP17] and van Bulck et al. [BPS17] relied on frequent timer interrupts of the Advanced Programmable Interrupt Controller (APIC) in order to suspend enclave execution with an asynchronous exit event (AEX). By reading and clearing the *accessed* bit of the enclave's PTEs, they can even single-step page table accesses during enclave execution. As an example, they suggested attacking a string comparison function, where the APIC interrupts the SGX enclave after every single memory access (byte granularity). Those interrupt-driven attacks were further refined [BPS18; Mog+20] to single-step enclave execution with zero noise in practice, which constitutes an instruction-granular side channel. Interrupt-driven attacks can also be used to identify software versions running inside an enclave [Kim+19]. Gyselinck et al. [Gys+18] further showed how segmentation could be misused in specific configurations to leak individual instructions executed in an enclave, even on a byte level. He et al. [He+18] showed that also the interrupt latency could be misused as a side channel to distinguish code patterns.

6.2 Side-channel Vulnerabilities

In the previous section, we have seen various side channels and exploitation techniques to observe runtime behavior of normal programs as well as enclaves. A successful attack, however, needs to be able to extract useful information from side-channel observations. If an attacked program does not leak sensitive information via side channels, it is secure against side-channel attacks. This property is often referred to as constant time. On the other hand, a software side-channel vulnerability might leak cryptographic keys, for example. For address-based side channels, this means that either code access patterns or data access patterns leak secret information. In the following, we survey various address-based side-channel vulnerabilities in popular cryptographic software.

Recall that transient execution attacks fall outside this category. Since they exploit not only address leakage but also data leakage, it is tough to make a piece of code constant time in the presence of transient execution attacks. Talking about mere software vulnerabilities here would do software developers injustice. We believe that a proper defense mechanism against transient execution attacks inevitably needs to be rooted in the CPU design [Yu+19].

6.2.1 Modular Exponentiation

Square&Multiply is a common technique for computing modular exponentiations bit-by-bit. Aciiçmez et al. [AKS07b] exploited the different branches of squaring and multiplication in the OpenSSL RSA implementation. Their

branch prediction attack recovers most exponent bits from a single RSA computation. Yarom and Falkner [YF14] introduced Flush+Reload by attacking RSA square&multiply in GnuPG. Similarly, Prime+Probe attacks have been launched against ElGamal decryption implemented via square&multiply for both GnuPG [Liu+15b] and libgcrypt [Zha+12].

Sliding Window. A faster alternative than square&multiply is the sliding window approach [BC89]. Here, multiple exponent bits are grouped together to reduce the total number of multiplications. Percival [Per05] attacked OpenSSL's sliding window implementation of RSA, thereby introducing the concepts behind the Prime+Probe technique [TOS10]. Aciiçmez [Aci07] proposed the first attack exploiting the instruction cache (I-cache) to infer executed instruction paths taken by square&multiply operations. He attacked sliding window exponentiations in the OpenSSL RSA implementation. Similarly, the sliding window implementations of GnuPG ElGamal [Liu+15b] and libgcrypt RSA [Ber+17] have been attacked.

Fixed Window. Using fixed exponent windows eradicates leakage due to conditional code execution in the sliding window approach. However, Aciiçmez and Schindler [AS08] managed to attack the extra reduction step of the Montgomery multiplication routine by exploiting the I-cache. Also, an implementation flaw in OpenSSL allowed bypassing the fixed window implementation for DSA signing operations [GBY16]. Fixed window exponentiation does not prevent Prime+Probe attacks on the window multipliers [Liu+15b]. To close this data leakage, OpenSSL implemented the scatter-gather technique, which aligns multipliers in memory such that the same cache lines are accessed irrespective of the active multiplier. Scatter-gather has been further improved in [Gop+09; Gue12]. Yarom et al. [YGH16] attacked OpenSSL's scatter-gather implementation by exploiting cache-bank conflicts within a cache line [Ber05; TOS10]. For a 4 096-bit RSA modulus, they recovered the key from 16 000 decryptions.

SGX. Brasser et al. [Bra+17b] launched Prime+Probe attacks from the operating system against an enclave running the Intel IPP library. They attacked the fixed-window exponentiation during RSA decryption. Schwarz et al. [Sch+17] demonstrated Prime+Probe attacks from one SGX enclave against another SGX enclave in order to extract an RSA key from mbedTLS [Sch+17]. They extract 96% of a 4096-bit RSA key within a single trace. To reduce measurement noise, both attacks gathered several traces [Bra+17b; Sch+17].

In Chapter 8, we discover a side-channel vulnerability that leaks DSA keys via unprotected modular exponentiation to an SGX attacker during the process of key loading. In Chapter 9, we identify multiple novel side-channel vulnerabilities in OpenSSL and LibreSSL that leak the length of secret exponents (*i.e.*, secret nonces) in a preparatory step for constant-time exponentiation. They are tied to the internal data representation in so-called big numbers. We demonstrate a full key recovery attack on DSA-256 with 36 traces by using an SGX controlled-channel attack.

6.2.2 ECDSA Scalar Multiplication

Double&Add is a technique analogous to square&multiple for computing scalar multiplications over elliptic curves. Brumley et al. [BT11] targeted constant-time double&add in OpenSSL ECDSA by measuring the total number of iterations. Yarom et al. [YB14] exploited conditional code during double&add via Flush+Reload, bypassing the constant-time implementation. In Chapter 9, we identify similar side-channel vulnerabilities as for modular exponentiation also in ECDSA implementations of OpenSSL and LibreSSL.

Windowed Multiplication. Brumley et al. [BH09] attacked the windowed Non-Adjacent Form (wNAF) multiplication of OpenSSL on the secp160 curve via Flush+Reload. Similar attacks on the popular secp256k1 curve leverage better side-channel observations and better recovery methods [Ben+14; PSY15; FWC16; All+16]. Dall et al. [Dal+18] attacked a fixed-window scatter-gather version of Intel EPID by exploiting a leak in the number of iterations. In Chapter 9, we also present a new side-channel vulnerability in the constant-time point addition of OpenSSL and BoringSSL, which leaks whether a nonce window is all-zero. It could be exploited in an SGX setting [Mog+20].

6.2.3 GCD

The Euclidean algorithm is a key ingredient for public-key cryptography. It can be used to compute the greatest common divisor (GCD) of two numbers. OpenSSL uses an optimized binary Euclidean algorithm (BEA) to test co-primality RSA parameters during key generation. The susceptibility of BEA to side-channel attacks has already been shown before [AGS07; AT07; GB17].

In Chapter 7, we introduce a controlled-channel attack against RSA key generation by exploiting conditional branches in the BEA implementation of OpenSSL. We managed to recover a full RSA key by observing a single key generation operation inside an SGX enclave. Concurrently to our work, Aldaya [Ald+19] mounted a Flush+Reload attack on the vulnerable BEA implementation in a non-enclave setting, with a success rate of 27%.

6.2.4 Modular Inversion

The Euclidean algorithm cannot only be used for computing a GCD. The extended Euclidean algorithm (EEA) also allows doing modular inversion in prime fields. Modular inversions are used to compute RSA private keys, or (EC)DSA signatures, amongst others. In the past, software implementations relied on an optimized variant of EEA, namely the binary extended Euclidean algorithm (BEEA) [MOV96, Algorithm 14.57]. Similar to the BEA, the BEEA executes input-dependent (*i.e.*, secret-dependent) branches. Aciiçmez et al. [AGS07] suggested attacking the modular inversion during RSA computations by means of branch prediction analysis (cf. [AKS07a]). They proposed an attack that relies on precise monitoring of all executed branches. Concurrently, Aravamuthan and Thumparthy [AT07] analyzed BEEA with respect to power analysis attacks.

Later on, García and Brumley [GB17] performed a Flush+Reload attack on the BEEA used in OpenSSL ECDSA. They recovered parts of the secret nonce, which allowed them to recover the full secret key from 50 traces. In order to mitigate these attacks, OpenSSL introduced a Euclidean modular inversion algorithm that reduces secret-dependent branches.

In Chapter 8, we identify a programming bug similar to [GB17] where the initialization of RSA keys leaks secret prime numbers to an SGX attacker. In Chapter 9, we demonstrate that the improved Euclidean inversion is not free of side-channel leakage. In fact, it leaks the topmost bit of secret DSA nonces, which could be used in an advanced Bleichenbacher attack [Ble00] to recover secret keys. In response, OpenSSL adopted Fermat inversion, which replaces the Euclidean algorithm with modular exponentiation. Recently, Aldaya and Brumley attacked BEEA on mbedTLS with an SGX-stepping attack [AB20]. Similarly, Moghimi et al. [Mog+20] performed an interrupt-driven SGX-stepping attack on BEEA in WolfSSL.

6.2.5 Modular Reduction

Aciiçmez and Schindler [AS08] attacked the final reduction step of the Montgomery multiplication routine. Ryan [Rya19] discovered an early-abort condition in OpenSSL's modular reduction and exploited it with a Flush+Reload attack to recover ECDSA private keys.

6.3 Side-channel Defenses

Side-channel attacks can be tackled on different layers, which we address in the following. First, one could attempt to close side channels by redesigning a microarchitecture or the operating system, in particular with respect to shared resources. Second, one could remove the source of leakage by closing side-channel vulnerabilities in software. The first approach is likely necessary to address transient execution attacks. However, we believe that closing side-channel vulnerabilities in software is the preferred way to go for address-based side channels. This requires the detection of side-channel vulnerabilities for which appropriate tool support is needed.

6.3.1 Closing Side Channels

Cache attacks can be addressed by redesigning caches, as partially surveyed in [DXS19]. Page [Pag05] described a partitioned cache architecture in which the operating system can segregate critical applications in the cache. Partitioned caches were further explored in [WL08; Dom+12]. Liu et al. [Liu+16] relied on Intel's cache allocation technology (CAT) to partition the last-level cache and, thus, prevent cache-line sharing. Others [Raj+09; Shi+11; ZRZ16] used software-only page coloring in order to partition the last-level cache via a careful page mapping. Hardware transactional memory can be used to mitigate cache attacks

by keeping all sensitive data in the cache during the computation [Gru+17]. To thwart attacks in a hyper-threaded setting, coscheduling [Cor18a] can group mutually trusted processes on the same CPU cores. The above partitioning approach can be expanded to avoid any resource sharing in hardware while executing sensitive programs [AA18].

Researchers called for designing better instruction set architectures (ISA) and OS abstractions reflect side-channel properties accurately [Hei18; GYH18; Ge+19]. Yu et al. [Yu+19] designed and implemented a data-oblivious ISA atop of the RISC-V BOOM processor. Gao et al. [Gao+20] proposed and implemented an ISA extension for addressing physical side channels.

Randomized caches can help thwart cache attacks as well [WL07; Kon+08; LL14; Qur18; Tri+18; Wer+19; Ram+19; Tan+20]. Moreover, oblivious RAM (ORAM) [GO96] has been proposed as a generic countermeasure against data leaks by hiding memory access patterns on the address bus. While ORAM can address DRAM-based attacks [Pes+16], it does not necessarily stop cache attacks. Several research prototypes integrate ORAM-based techniques in their processor design [ZZP04; FDD12; Maa+13; Liu+15a; Nay+17; Ren+19]. Smart memory that has built-in computing capabilities can be used to relax the overhead of ORAM [AN17; Awa+17; Lia+18].

SGX. To thwart cache attacks on enclaves, the Sanctum processor designed by Costan et al. [CLD16] assigns enclaves DRAM regions that are also segregated in the cache. Dessouky et al. [DFS20] proposed a hybrid cache design, which selectively enables side-channel defenses only for enclaves.

To address controlled-channel attacks, Shinde et al. [Shi+16] proposed hardware support that guarantees to deliver enclave page faults directly into the enclave. Similarly, Aga et al. [AN19] proposed that enclaves should handle their own page faults and manage their page tables, which demands changes to the CPU. SGX-LAPD [Fu+17] considers large pages (*i.e.*, 2 MB instead of the usual 4 KB) in order to reduce the overall number of page faults. The enclave relies on the `EXINFO` data structure, which tracks page fault addresses of an enclave, to verify that the OS indeed provides large pages. Strackx et al. [SP17] proposed hardware modifications to preload all critical page mappings in the translation lookaside buffer (TLB) whenever entering the enclave. Moreover, they protect the TLB mapping from being tampered during enclave execution.

Shih et al. [Shi+17a] observed that transactional synchronization extensions (TSX) could be used to detect exceptions, such as page faults, and report them to enclave-internal code only rather than to the OS. They proposed T-SGX, in which they execute blocks of enclave code inside TSX transactions. If an exception is thrown, the transaction aborts, and the enclave decides whether or not to terminate its execution. Chen et al. [Che+17] proposed to detect side-channel attacks within enclaves by detecting frequent page faults. They rely on the execution time within the enclave as an indicator of an ongoing side-channel attack. Their in-enclave timing source (*i.e.*, a timer variable) is protected via TSX. TSX can also be used to detect cache misses inside SGX enclaves [CTZ17] to thwart cache attacks.

Detection of page faults or cache misses does not prevent stealthier attacks [Bul+17; Wan+17b]. These attacks derive page access patterns either by monitoring the *accessed* and *dirty* bits of page table entries or by mounting cache-attacks like Flush+Reload attacks on page table entries.

To thwart hyper-threading attacks on enclaves, Chen et al. [Che+18] proposed that enclaves can occupy a whole physical CPU using shadow threads. Since the operating system cannot be obliged to schedule shadow threads on the same CPU, they establish repeated side-channel measurements to detect co-residency.

6.3.2 Closing Side-channel Vulnerabilities

Software side channels can be closed by randomization or equalization, also referred to as constant time.

Randomization. For cryptographic operations, randomization is a common technique to hide leakage of secrets. Kocher [Koc96] proposed blinding as a means to prevent timing attacks on modular exponentiation. OpenSSL also employs base point blinding in their generic double&add code. mbedTLS adopted exponent blinding in their RSA implementation in response to [Sch+17]. To fix leakage in modular reductions [Rya19], OpenSSL also blinds DSA calculations done after modular exponentiation.

Randomization cannot only be applied on an algorithmic level but also on the generated code. Seo et al. [Seo+17] proposed SGX-Shield, which randomizes the memory layout of enclaves via ASLR in a multi-stage loading step. While primarily intended as a countermeasure against runtime attacks, it also raises the bar for controlled-channel attacks. Shih [Shi19] discussed weaknesses of SGX-Shield against interrupt-driven attacks such as [BPS17]. He proposed a side-channel resistant defense named SGX-Armor that randomizes the code layout in a switching network via a primitive called oblivious swap. Shih furthermore developed a generic SGX framework called Pridwen [Shi19] for applying various side-channel defenses to enclaved programs. Priwden applies side-channel defenses on Web Assembly, before compiling enclaves to native binaries.

ORAM. Costa et al. [Cos+17] compared various ORAM schemes in an SGX setting and observed runtime overheads well above 1000x. Sasy et al. [SGF18] developed the ZeroTrace SGX library, which hides data access patterns via ORAM. However, they do not report macrobenchmark results. Ahmad et al. [Ahm+19] optimized ORAM for enclaves. Their Obfuscuro defense protects against data leakage and control-flow leakage. However, it still shows an average overhead of 51x–83x. Brasser et al. [Bra+19] proposed Dr. SGX, which hides data access patterns via continued light-weight re-randomization, as opposed to ORAM. Dr. SGX reduces the average overhead to 4.36x; however it does not hide code access patterns. Zhang et al. [Zha+20] designed an ORAM scheme for SGX called Klotski. It hides both code and data access patterns behind a virtualized memory subsystem, managed in software. Klotski shows runtime overheads between 16.7x and 50% on web servers and image compression, respectively.

Constant Time. Leakage of data accesses frequently occurred due to lookup table implementations of symmetric primitives. They can be mitigated by bit-

slicing [RSD06; Kön08; KS09]. Bitslicing reformulates symmetric encryption operations as boolean equations that can be parallelized via vectorized instructions, for example. This can even improve overall throughput [KS09]. To hide data accesses in fixed-window RSA exponentiation, the scatter-gather technique interleaves data buffers in memory such that the same cache lines are accessed irrespective of the used data buffer. For attacks in a hyper-threaded setting, a finer-grained block size needs to be chosen [YGH16].

A rich body of literature exists on automated code transformations that mitigate side-channel leakage. Agat [Aga00] proposed a program transformation that pads unequal branches to avoid timing leakage. Hedin et al. [HS05] extended Agat's work towards a larger subset of Java bytecode. However, padding does not take into account stronger attacks leveraging address leakage (e.g., cache attacks) rather than pure timing leakage. Barthe et al. [BRW06] proposed a program transformation for sequential, object-oriented programs with support for exceptions. Their transactional branching executes both branches but only commits the desired one. Still, transactional branching shows control-flow leakage. Coppens et al. [Cop+09] proposed compiler transformations to eliminate key-dependent control flows. More specifically, control-flow dependencies are transformed into data-flow dependencies by means of conditional instructions. Rane et al. [RLT15] expanded the idea of transactional branching. Their Raccoon compiler always executes all secret-dependent branches in a leakage-free fashion. Data leaks are mitigated by always streaming across the whole data buffer in question. This is much faster than ORAM, according to their evaluation. Average runtime overheads are 16.1x, with peaks of more than 600x. The threat model of Raccoon closely resembles our notion of address-based side-channel attacks.

SGX. All the above equalization methods also work in an SGX setting. If applied correctly, this closes controlled-channel and cache attacks on enclaves (e.g., by using Raccoon). Nevertheless, researchers explicitly tailored defenses for SGX enclaves. To mitigate controlled-channel attacks in software, Shinde et al. [Shi+16] introduced the notion of page-fault obliviousness. This means that the page-fault pattern observed by the operating system is independent of the secret input. They proposed a compiler-based approach for code balancing similar to [Cop+09]. van Bulck et al. [Bul+17] demonstrated an interrupt-based attack bypassing page-fault obliviousness. Sinha et al. [SRS17] developed a compiler for achieving page obliviousness inside enclaves, together with a verifier. They resort to balancing the number of instructions in each secret-dependent branch by inserting nop instructions. This neither prevents cache attacks nor does it remove timing leakage, since nop instructions execute much faster than regular ones. To address branch shadowing attacks, Lee et al. [Lee+17b] proposed a system called Zigzagger, which obfuscates conditional branches via indirect branches. Zigzagger does not withstand fine interrupt-driven attacks [BPS17].

Constant Time by Proof or Design. In order to check whether a program is indeed constant time, several verification methods have been developed [Alm+13; Bar+14; Alm+16; DK17; BPT19]. They typically demand programmer annotations. Bond et al. [Bon+17] developed the Vale framework for programming

side-channel secure cryptographic primitives in low-level language constructs. Similarly, Almeida et al. [Alm+17] developed a programming language and a compiler called Jasmin, which allows developing high-performance cryptography that is side-channel secure.

Also, the compiler itself might introduce side-channel vulnerabilities and, thus, ruin verification guarantees given on the source code or some intermediate representation. Barthe et al. [BGL18; Bar+20] and Besson et al. [BDJ19] studied whether compilation passes provably preserve constant-time properties.

6.3.3 Detecting Side-channel Vulnerabilities

In order to apply constant-time techniques to a program, one requires means to find actual side-channel vulnerabilities in the code. Tools for verifying constant-time properties alone do not provide comprehensive information about existing vulnerabilities. In the following, we survey existing detection tools, as listed in Table 6.1.

Terminology. We consider a program secure if it does not contain address-based information leaks. We further distinguish between deterministic and non-deterministic programs. Non-determinism might be due to the randomization of intermediates (blinding) or results (probabilistic constructions). Latter include any kind of non-determinism, such as randomization of intermediates (blinding) or results (probabilistic constructions). A *false positive* denotes an identified information leak that is, in fact, none. A *false negative* denotes an information leak that was not identified.

Timing Leakage Detection. Reparaz et al. [RBV17] measures the overall execution time of implementations in a blackbox fashion for different classes of inputs. Their dudect tool relies on statistical tests to infer whether or not the implementation leaks information. More advanced approaches use symbolic execution to give upper leakage bounds [PPM16]. Themis [CFD17] is a static analysis tool for assessing timing leakage (or leakage of other resources) in Java applications. However, these approaches fall short for more fine-grained address-based attacks, such as cache attacks.

Static Address Leakage Detection. CacheAudit [KMO12; Doy+13], as well as follow-up works [DK17; MWK17], are static analysis tools that symbolically evaluate all program paths via abstract interpretation. They detect leakage on a cache line or byte granularity. Rather than pinpointing the leakage origin, CacheAudit accumulates potential leakage into a single metric, which represents an upper-bound on the maximum leakage possible. While a zero leakage bound guarantees the absence of side channels, a non-zero leakage bound could become somewhat imprecise (false positives) due to abstractions made on the data of the program. Abstraction also fundamentally prohibits analysis of interpreted code, which is encoded in the data plane of the interpreter. Analyzing large code bases such as OpenSSL with many potential leaks demands higher precision.

CacheS [Wan+19] uses abstract interpretation for finding secret-dependent branches. CacheS increases the precision of full abstract interpretation (e.g., as done by CacheAudit) by only tracking secrets in a precise way.

Dynamic Address Leakage Detection. Dynamic analysis relies on concrete rather than symbolic execution. This increases precision (and typically also performance) but introduces false negatives if leakage is not triggered during concrete execution.

Ctgrind [Lan10] dynamically tracks unsafe usage of secrets with the Valgrind memory error detector on annotated secrets. It detects control-flow leaks and data leaks. Ctgrind suffers from false positives as well as false negatives [Alm+16].

CacheD [Wan+17a] combines static and dynamic analysis. It taint-tracks instructions accessing secret data and evaluates them symbolically to find potential data leaks. CacheD does not model control-flow leaks. Since it only analyzes a single execution, CacheD miss leakage in other execution paths (false negatives). Its static analysis could further introduce false positives.

Zankl et al. [ZHS16] use binary instrumentation to build a histogram of all executed instructions. They correlate the histogram against the Hamming weight of the private key, thus finding control-flow leaks in modular exponentiation.

Stacco [Xia+17] uses binary instrumentation to record instruction traces rather than histograms only. Stacco specifically finds padding oracle vulnerabilities used for Bleichenbacher attacks [Ble98]. It does not consider data leakage, and it does not consider reducing false negatives, *i.e.*, finding multiple control-flow leaks within the traces. If Stacco did, it would suffer from false positives due to improper trace alignment (it uses the Linux diff tool).

In Chapter 8, we present DATA. We introduce the notion of more generic address traces, capturing instruction and data addresses. Similar to Stacco, DATA records address traces via binary instrumentation. By matching address traces, DATA finds potential control-flow and data leaks. DATA aligns traces in a way to also detect nested control-flow leaks. Moreover, DATA provides statistical methods for distinguishing secret-dependent leaks from unrelated ones due to non-determinism (e.g., blinding), and it supports dedicated leakage models similar to [ZHS16]. The statistical methods are not part of this thesis but can be looked up in the original publication [Wei+18a].

MicroWalk [Wic+18] also records all accessed addresses by means of binary instrumentation to detect control-flow and data leakage. Instead of processing the whole trace, MicroWalk collapses the execution context via hash functions. This improves performance at the expense of losing, e.g., call stack information. MicroWalk performs Mutual Information (MI) estimation to assess the amount of leaked bits. It could find unknown leakage in proprietary, closed-source programs.

TriggerFlow [Gri+19] is a side-channel regression testing tool that uses selective source-code annotations and the GDB debugger to check critical functions that were subject to side-channel vulnerabilities in the past. If a breakpoint is hit, a potentially dangerous code path is being executed. Accuracy mainly depends on proper annotation.

Attack-based Approaches. Some tools base their analysis on concrete attacks. Hence, they do not generalize to other attacks. For instance, Brumley and Hakala [BH09], as well as Gruss et al. [GSM15], suggested detecting implementations vulnerable to cache attacks by relying on template attacks.

Table 6.1: Comparison of leakage detection tools. ● means that the tool suffers from false positives/negatives. ○ means that the tool's design does not suffer from false positives/negatives (although the prototype implementation might do). $○_S$ denotes statistical guarantees.

Analysis Tool	Method	Granularity	Control-flow leakage	Data leakage	Deterministic false pos.	Non-determ. false pos.	False negatives	Output	Source code required	Tool available
CacheAudit [Doy+13]	Static	Cacheline	✓	✓	●	●	○	Leakage bound	no	✓
CacheAudit 2 [DK17]	Static	Byte	✓	✓	●	●	○	Leakage bound	no	✓
CacheS [Wan+19]	Static	Cacheline	✓	✓	●	●	○	Leak origin	no	✓
CacheD [Wan+17a]	Both	Cacheline	✗	✓	●	●	●	Leak origin	no	✗
ctgrind [Lan10]	Dynamic	Byte	✓	✓	●	●	●	Leak origin	yes	✓
Zankl et al. [ZHS16]	Dynamic	Byte	✓	✗	$○_S$	$○_S$	●	Leak origin, HW	no	✓
Stacco [Xia+17]	Dynamic	Byte	✓	✗	○[a]	●	●	Leak origin	no	✗
MI-Tool [Ira+17]	Dynamic[b]	Cacheline	✓	✓	$○_S$	$○_S$	●	Leak origin, MI	yes	✗
DATA [Wei+18a]	Dynamic	Byte	✓	✓	$○_S$	$○_S$	◐	Leak origin, FR, HW[c]	no	✓
MicroWalk [Wic+18]	Dynamic	Byte	✓	✓	$○_S$	$○_S$	●	Leak origin, MI	no	✓
TriggerFlow [Gri+19]	Dynamic	Breakpoint	✓	✓	●	●	●	Hit breakpoints	yes	✓

[a] Only the first control-flow leak is reliably identified. Reporting multiple leaks could cause false positives.

[b] Attack-based rather than trace-based.

[c] DATA uses a generic fixed-vs-random (FR) test. Apart from the Hamming Weight (HW), analysts can also program their own leakage models.

Irazoqui et al. [Ira+17] use cache observations and a mutual information metric to identify control-flow and data leaks. Basu et al. [BC17] and Chattopadhyay et al. [Cha+17b] quantify the amount of information leaked via cache attacks.

✳ ✳ ✳

7

RSA Enclave Side-channel Leakage

A gossip betrays a confidence, but a trustworthy person keeps a secret.

Solomon – Proverbs

Side-channel attacks represent a serious threat to cryptographic implementations, especially in the setting of enclaves. We outlined several attacks in Chapter 6. Popular cryptographic libraries such as OpenSSL [Ope] are often hardened against software side-channel attacks on secret key operations, including decryption and signature generation of digital signature schemes. However, the process of key generation has been mostly neglected in these analyses. This is especially fatal if, for example, RSA signature keys are generated inside an SGX enclave, as we demonstrate in this chapter.

In the past, side-channel attacks against RSA key generation routines relied on power analysis [CC07; FGS09; VEW12; Bau+14] and targeted the prime generation procedure. Prime generation is usually sped up with a sieving process that shows secret-dependent branches. Recent work [Lee+19] performed a simple power analysis attack (SPA) on the Miller-Rabin primality test. All these side-channel attacks either target the primality test or the prime generation itself. They were purely based on observing the power consumption as side channel, although software side-channel attacks might be possible as well.

Prior to the publication of our original paper, software-based side-channel attacks on key generation have been mostly considered out of scope for at least two reasons. On the one hand, key generation is usually a one-time operation, limiting possible attack observations to a minimum. Especially in case of noisy side channels, e.g., timing attacks and cache attacks are hard to conduct, given a single observation. Recently, and concurrently to our work, Aldaya [Ald+19]

managed to mount the first Flush+Reload attack on RSA key generation, although with a success rate of only 27%. On the other hand, key generation might be done in a trusted environment that is entirely inaccessible to a side-channel attacker.

The situation, however, has changed with the introduction of enclaves that aim to support secure software execution in untrusted environments. As discussed in Chapter 6, SGX enclaves enable new attack techniques such as controlled-channel attacks [XCP15; Shi+16; BPS17]. By monitoring enclave page faults, the operating system can gather noiseless measurement traces of executed code paths and accessed data at page granularity. In order to thwart controlled-channel attacks, the SGX documentation demands enclave code to be side-channel resistant, namely to avoid leaking information through page access patterns [Int17b].

In light of this powerful attack technique, we investigated the RSA key generation routine of the Intel SGX SSL library, which is based on OpenSSL. Surprisingly, we identified a critical vulnerability that allows us to fully recover the generated private key. The identified vulnerability is due to an optimized binary Euclidean algorithm (BEA). The BEA features input-dependent branches for checking the correctness of the generated prime factors p and q, *i.e.*, whether $p-1$ and $q-1$ are coprime to the public exponent. By launching a controlled-channel attack, we recover the executed branches of the BEA running inside an enclave program and establish linear equations on the secret input, *i.e.*, the prime factors p or q. Based on these equations, we factor the modulus $N = pq$ with minor computational effort on a commodity PC, *i.e.*, in less than 12 seconds for a 8 192 bit modulus, which trivially allows recovering the private key.

Differentiation from Existing Attacks. The attack presented in this chapter differs from previous attacks on RSA key generation as follows: First, contrary to related work which target the prime generation itself [VEW12] or the primality tests [CC07; FGS09; Bau+14], we target the subsequent parameter checking routine. Clavier et al. [CC07] described a side-channel attack on the fast prime generation algorithm proposed by Joye and Paillier [JP06]. By observing branching behavior in the generation of candidate primes, e.g., with a simple power analysis attack (SPA), one can recover their parity bits and, subsequently, the least significant bits of the generated prime. Finke et al. [FGS09] performed an SPA on another fast prime generation procedure [BDL91]. They exploited leakage of the number of trial divisions before the Miller-Rabin primality test [MOV96, Algorithm 4.24] is applied. Vuillaume et al. [VEW12] considered differential power analysis (DPA), template attacks, and fault attacks on the prime generation procedure. They attacked the Fermat primality test [MOV96, Algorithm 4.9], which is rarely used in practice due to false positives. Bauer et al. [Bau+14] attacked the prime sieve procedure during prime number generation. Recently, Lee et al. [Lee+19] performed an SPA on the Miller-Rabin primality test. They exploited collisions in a side-channel protected square&multiply exponentiation.

Second, the above attacks rely on power analysis while we use a pure software attack. To the best of our knowledge, software side-channel attacks on the RSA key generation procedure have not been demonstrated before.

Different from other microarchitectural attacks on RSA implementations that targeted modular exponentiations [Aci07; AS08; Bra+17b; Sch+17], the attack presented in this chapter targets the binary Euclidean Algorithm (BEA). The extended BEA (BEEA) is used for modular inversion. It has already been attacked before in the setting of RSA decryption [AGS07] or ECDSA signature generation [GB17], which typically requires several measurement traces. In contrast, we target the BEA implementation used for RSA key generation, for which, by definition, only one measurement trace is available.

Concurrent to our work, Aldaya et al. [Ald+19] attacked the same vulnerable RSA implementation, thus demonstrating the first cache attack on RSA key generation. Their setting considers a Flush+Reload attacker and applies only to non-enclaved programs. Due to measurement noise, they performed extensive pruning steps to narrow down the search space. For example, over a period of 45 days of CPU time, they recovered 2285 out of 9434 RSA keys. In contrast, we demonstrate the first controlled-channel attack on RSA key generation, which succeeds within seconds with a 100% success probability.

Controlled-channel attacks have been studied before. Xu et al. [XCP15] used them to extract sensitive data such as images and processed texts from enclaved programs. Shinde et al. [Shi+16] studied known information leaks in cryptographic primitives of OpenSSL and Libgcrypt with respect to controlled-channel attacks. However, they did not analyze RSA key generation routines. Xiao et al. [Xia+17] used controlled-channel attacks to mount Bleichenbacher and padding oracle attacks on various TLS implementations.

Contributions. We sum the contributions presented this chapter as follows:

1. We identify a critical controlled-channel vulnerability in the RSA key generation routine of Intel SGX SSL/OpenSSL, which relies on the binary Euclidean algorithm (BEA) to check the validity of generated parameters.
2. We present an attack to recover most of the bits of one RSA prime factor, which allows us to factor $N = pq$ and, thus, recover the generated private key.
3. We implement a proof of concept attack that recovers generated RSA keys with a single observation only and with 100% success probability.
4. We provide a patch to mitigate the vulnerability, which is even faster than the original implementation.[1]

This chapter is based on the publication [WSB18a], of which I am the main author, while Raphael Spreitzer and Lukas Bodner contributed a significant part to the practical evaluation. The remainder of the chapter is structured as follows: In Section 7.1, we present the threat model underlying our attack. In Section 7.2, we describe the RSA key generation procedure and the binary Euclidean algorithm as implemented in OpenSSL. In Section 7.3, we discuss the identified vulnerability and our key recovery attack on RSA. In Section 7.4, we evaluate our attack in a real-world setting. In Section 7.5, we present our software

[1] The patch is already merged upstream by OpenSSL.

patch to fix the identified vulnerability. We discuss further vulnerabilities in Section 7.6 and conclude in Section 7.7.

7.1 Threat Model

We consider an enclave that dynamically generates RSA keys, which are intended never to leave the enclave. Dynamic key generation has already broad applications in other trusted execution environments, such as trusted platform modules and smart cards. Also, dynamic key generation is a fundamental operation for most SGX applications. For example, scenarios like audio and video streaming with SGX [Hoe+13] fall into our threat model. Here, a streaming enclave dynamically generates an RSA key pair and registers the public key at its streaming counterpart. Latter delivers all streaming content encrypted under this key, allowing the enclave to decrypt it securely and to display it to the user. Another example is a document signing enclave, generating its own signature keys inside the enclave and issuing a certificate signing request to an external certification authority. Thereby, the enclave protects the signing key against malware. In any case, the compromise of a private key could lead to signature forgery, espionage, or video piracy with all its legal and financial consequences.

In line with SGX's threat model, the operating system (OS) is considered untrusted and compromised, trying to extract secret keys from the enclave. In general, attackers in SGX settings are considered to be able to trigger enclave operations arbitrarily often by repeatedly invoking the enclave with a fresh state.[2] However, our attacker is naturally limited to at most one observation of the enclave's key generation, as the next invocation will generate a different, independent key.

Using a noiseless controlled-channel attack [XCP15; Shi+16; Xia+17], the attacker can observe page access patterns of the executing enclave. While this is sufficient for the attack presented in this chapter, we note that, without loss of generality, an attacker could also resort to different techniques. Among them are side channels using branch shadowing [Lee+17b], single-step approaches based on the APIC timer interrupts [HCP17; BPS17], or even attacks with fewer or no page faults [Wan+17b; Bul+17], given that enough information can be extracted from a single execution.

7.2 RSA Key Generation

The Intel SGX SSL library [Int19] is a cryptographic library for SGX enclaves. It is built on top of OpenSSL [Ope], a widely-used toolkit for cryptographic purposes. Since Intel SGX SSL operates on OpenSSL, it inherits all of OpenSSL's side-channel properties, including mitigation techniques but also potential vulnerabilities. In particular, OpenSSL employs several side-channel countermeasures to thwart traditional side-channel attacks such as cache attacks.

[2] SGX does not prevent roll-back attacks, which would require persistent storage [SP16].

The RSA public-key cryptosystem [RSA78] provides public key encryption as well as digital signatures. The RSA key generation routine of OpenSSL–implemented in `rsa_gen.c`–starts by generating two primes p and q, which are then used to compute the public modulus $N = pq$. While p and q are chosen randomly during the key generation procedure, it is common practice that the public exponent is fixed to $e = 65\,537_{10} = \texttt{0x010001}_{16}$ (cf. [Bon99]). The private key is later computed as $d \equiv e^{-1} \mod \phi(N)$, with ϕ being Euler's totient function. For two prime numbers p and q, $\phi(N) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$.

Among other checks, the key generation routine ensures that $(p-1)$ and $(q-1)$ are coprime to e, *i.e.*, that the greatest common divisor (GCD) of the public exponent e and $(p-1)$ as well as $(q-1)$ is one. These checks are performed by relying on a variant of the Euclidean algorithm, which we attack.

7.2.1 Binary Euclidean Algorithm

A well-known algorithm to compute the GCD is the Euclidean algorithm [MOV96, Algorithm 2.104]. For two positive integers $a > b$, it holds that $\gcd(a, b) = \gcd(b, a \mod b)$. Since this algorithm relies on costly multi-precision divisions, a more efficient variant is usually implemented for architectures with no dedicated division unit, using simple (and more efficient) shift operations and subtractions.

Listing 7.1 depicts an excerpt of the Euclidean algorithm as implemented in OpenSSL. It is an optimized version denoted as binary GCD [MOV96, Algorithm 14.54] that has been introduced by Stein [Ste67]. As can be seen in Listing 7.1, OpenSSL uses the BIGNUM implementation for arbitrary-precision arithmetic. The functionality of each BIGNUM procedure is indicated with comments.

```
1  BIGNUM *euclid (BIGNUM *a, BIGNUM *b) {
2    BIGNUM *t;
3    int s = 0;
4    while (!BN_is_zero(b)) {      // b != 0
5      if (BN_is_odd(a)) {
6        if (BN_is_odd(b)) {       // a is odd, b is odd
7          BN_sub(a, a, b);        // a = a-b
8          BN_rshift1(a, a);       // a = a/2
9          if (BN_cmp(a, b) < 0) {
10           t = a; a = b; b = t; // swap a and b
11         }
12       } else {                  // a is odd, b is even
13         BN_rshift1(b, b);       // b = b/2
14         if (BN_cmp(a, b) < 0) {
15           t = a; a = b; b = t; // swap a and b
16         }
17       }
18     } else {
19       if (BN_is_odd(b)) {       // a is even, b is odd
20         BN_rshift1(a, a);       // a = a/2
21         if (BN_cmp(a, b) < 0) {
22           t = a; a = b; b = t; // swap a and b
23         }
24       } else {                  // a is even, b is even
25         BN_rshift1(a, a);       // a = a/2
26         BN_rshift1(b, b);       // b = b/2
27         s++;
28       }
29     }
30   }
31
32   if (s)
33     BN_lshift(a, a, s);         // a = a * 2^s;
34   return a;
35 }
```

Listing 7.1: Binary GCD (a.k.a. Stein's algorithm) in OpenSSL.

The binary GCD works as follows: If b is zero, a holds the GCD, and the algorithm terminates. Otherwise, the algorithm distinguishes the following cases in a loop:

- **Branch 1** (lines 7–10): If a and b are odd, the $\gcd(a, b) = \gcd((a-b)/2, b)$. The division by 2 (implemented as a right shift) accounts for the fact that the difference of two odd numbers is always even, but 2 does not divide odd numbers.
- **Branch 2** (lines 13–15) and **branch 3** (lines 20–22): If either a or b is odd, then the even number is divided by 2 through a right shift since 2 is not a common divisor.
- **Branch 4** (lines 25–27): If both a and b are even, then 2 is a common divisor and, therefore, both a and b are divided by 2. In this case, the resulting GCD is a multiple of 2, and the variable s holds the number of times this branch is executed.

During the execution, the algorithm always ensures that $a > b$. It swaps a and b as soon as this condition is not satisfied anymore (see lines 9–10, 14–15 and 21–22).

A Note on the Implementation. In the OpenSSL source code, the function BN_gcd(...) used to compute the GCD calls the function euclid(...) as depicted in Listing 7.1, but the compiler inlines the corresponding function into BN_gcd(...). Hence, in the remainder of this chapter, we will refer to BN_gcd(...) when talking about the vulnerable code.

7.3 Attacking RSA Key Generation

During RSA key generation, the binary GCD variant described in Section 7.2 is used to ensure that $p - 1$ and e are coprime. In order to do so, the algorithm depicted in Listing 7.1 is executed with $a = p - 1$ (with p being the secret prime) and $b = e$ (the public exponent). The crucial observation is that the binary GCD executes different branches depending on the input parameters. An attacker who is able to observe the executed branches can recover the secret input value $a = p - 1$ and, hence, the secret prime factor p.

Without loss of generality, we describe the attack by targeting the prime factor p, but the presented attack can also be applied to recover the prime factor q. Once we recovered either of the two prime factors, N can be factored trivially, which also allows us to compute the private exponent d.

7.3.1 Idealized Attacker

We first consider an attacker who can precisely distinguish all executed branches of the binary GCD algorithm (BEA), including the swapping operations in lines 10, 15, and 22. This attacker model, for example, accounts for branch shadowing attacks [Lee+17b] and the generalized attack described in Section 7.3.4.

Let a be the unknown secret input to be recovered, b the known input, and $a_i, b_i, i \geq 0$ all intermediate values calculated by the BEA. To recover the secret a, we build a system of linear equations, starting with $a = a_0$ and $b = b_0$. We then iteratively add equations, depending on the executed branches, as follows:

First branch: $a_{i+1} = \frac{a_i - b_i}{2}$

Second branch: $b_{i+1} = \frac{b_i}{2}$

Third branch: $a_{i+1} = \frac{a_i}{2}$

Fourth branch: $a_{i+1} = \frac{a_i}{2}$ and $b_{i+1} = \frac{b_i}{2}$

We increment i by one before proceeding with the next iteration. In addition, if a and b are swapped, *i.e.*, BN_cmp(a, b) < 0 yields true, we add the following two equations and increment i again: $a_{i+1} = b_i$ and $b_{i+1} = a_i$. The algorithm finishes after n steps with $a_n = \gcd(a, b)$ and $b_n = 0$. By recursively substituting

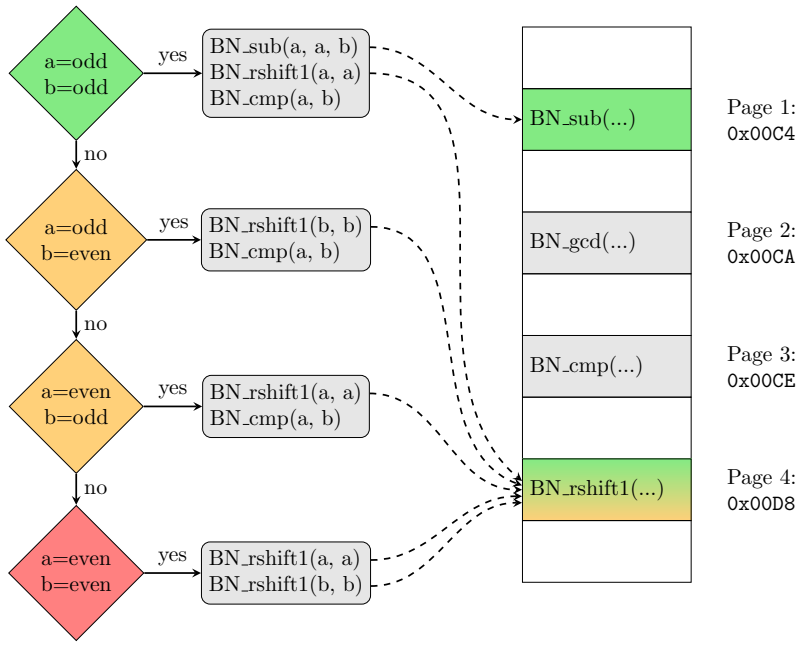

Figure 7.1: Relevant control flow of the BEA (left) and the page layout (right).

all equations one can express the unknown a as a linear equation $a = f(a_n, b_n) = f(\gcd(a, b), 0)$, which is trivial to solve, given that $\gcd(a, b)$ is known to be 1 in case of valid RSA parameters.

7.3.2 Controlled-channel Attacker

Considering our powerful idealized attacker can indeed be a realistic assumption [Lee+17b]. However, we resort to a weaker assumption in the rest of this chapter. We consider a controlled-channel attacker [XCP15; Shi+16], who recovers the secret input a from even fewer observations (up to the point where the two variables are swapped) and with a coarser granularity (page level).

Figure 7.1 illustrates an excerpt of the control flow of the binary GCD for the four branches in question. For illustration purposes, also the mapping of specific functions to their corresponding code pages are drawn.[3] If an attacker can distinguish executed branches based on page-access observations, the BEA can be reverted, and the secret input a can be recovered. Indeed, the functions BN_sub(...) and BN_rshift1(...) reside on different pages within memory, namely page 1 and page 4, while BN_gcd(...) is on page 2.

Observations. If this algorithm is executed with RSA parameters ($a = p - 1$ and $b = e$), we observe the following:

[3] The mapping depicts the actual offsets of the commit 899e62d of OpenSSL 1.1.0g.

1. Since p is a prime number, $a = p - 1$ is even. The second parameter ($b = e$) is always odd. Otherwise, we have invalid RSA parameters. Hence, the first iteration always executes the second branch.
2. The execution of the first branch can be observed by consecutive accesses to the code pages of BN_sub(...) and BN_rshift1(...), *i.e.*, page 1 and 4.
3. The second or the third branch are executed if either a or b is odd. These two branches, however, cannot be distinguished based on code page accesses since both branches execute the functions BN_rshift1(...) and BN_cmp(...) in the same order. Nevertheless, recall that in our setting, the algorithm is always executed with an odd $b = 65\,537$, which is much smaller than a. Thus, in the beginning, the algorithm will only execute the third (and the first) branch, reducing the value of a_i, but b_i remains an unchanged odd value. This is true until a_i and b_i are swapped for the first time, which is the case if $a_i < b_i$. Since each iteration reduces a_i by one bit (in general) due to the right shift operation, the first swap will approximately occur after $\log_2(p-1) - \log_2(e)$ iterations. Until then, every time we observe a single access to code page 4, we can be sure that branch 3 has been executed.
4. The fourth branch will only be executed if the greatest common divisor of the parameters a and b is a multiple of 2. Since the parameter $a = p - 1$ is even and $b = e$ is odd, this branch will never be executed (indicated as a red branch), as otherwise, we would have invalid RSA parameters.
5. The end of a branch and the start of the next iteration can be detected by monitoring accesses to BN_gcd(...) on page 2.
6. Although a controlled-channel attacker can observe when the BN_cmp(...) function is executed, our restricted controlled-channel attacker cannot decide whether or not the variables are swapped (*i.e.*, whether or not the conditional branch depending on the result of BN_cmp(...) is executed).[4] This is because the corresponding code for swapping the two numbers is on the same page as the binary GCD algorithm BN_gcd(...) itself. More specifically, our controlled-channel attacker cannot decide whether BN_gcd(...) directly continues with the next iteration, or whether a and b are being swapped first.

These observations, combined with the fact that the public exponent e is known, allow us to "revert" the computations for all bits of $a = p-1$, except about $\log_2(e)$ bits. As mentioned before, the public exponent is fixed to $e = 65\,537$.[5] This means that about $\log_2(65\,537) \approx 16$ bits of $a = p - 1$ cannot be recovered based on the accessed code pages. However, they can be easily determined based on the relations established from these observations.

Memory Layout. As mentioned, the functions BN_sub(...) and BN_rshift1(...) reside on different pages within the memory. In our tested implementation, they

[4] See Section 7.3.4 for a more generalized controlled-channel attacker.

[5] This choice of the public exponent has been widely established as quasi-standard among RSA cryptosystems (cf. [Bon99]).

are even 20 pages apart. Thus, it is very unlikely that a different compiler setting would link them to the same page, which would make them indistinguishable to a controlled-channel attacker monitoring these functions only. Even if this would happen, one could easily distinguish them by monitoring the sub-functions called by BN_sub(...) only, *i.e.*, BN_wexpand(...), BN_ucmp(...), BN_usub(...), etc.

7.3.3 Exploiting the Information Leak

We denote the sequence of page accesses observed by an attacker as $P = (p_0, \ldots, p_n)$. Without loss of generality, let us assume the same mapping from functions to code pages, as in the previous example. For instance, the function BN_sub(...) resides on page 1 (`0x00C4`), BN_gcd(...) resides on page 2 (`0x00CA`), and the function BN_rshift1(...) resides on page 4 (`0x00D8`). That is, the sequence of interesting page accesses consists of pages $p_i \in \{P_1, P_2, P_4\}$.

In order to recover the prime factor p (or $p-1$ respectively), we observe a sequence of page accesses up to the point where the two variables are swapped for the first time. All later page accesses are discarded. We denote this number of iterations as m. Given the modulus N or its bit size $\log_2(N)$, we denote the bit size of p and q as $K = \log_2(N)/2$. Thus, m is upper-bounded by $\lceil K - \log_2(e) \rceil$. Similar as before, we build a system of linear equations based on a_i, starting with the unknown input $a = a_0$. Since $i < m$, b will remain unchanged and we only need to distinguish two branches:

Access to page 1, and page 4: $a_{i+1} = \frac{a_i - b}{2}$

Access to page 4: $a_{i+1} = \frac{a_i}{2}$

Accesses to page 2 allow distinguishing different iterations. After m iterations, we express these equations by recursive substitution as a linear equation $a = f(a_m, b)$, or, more precisely

$$a = a_m \cdot c_a + b \cdot c_b$$

with known constants c_a and c_b, which result from the substitution.

Both, a and a_m are unknown. However, we additionally know that swapping occurred after m iterations, *i.e.*, $a_m < b$. Hence, we can determine the correct a by iterating over values $a_m \in [1, e)$ and evaluating the above equation. We use the resulting values a to check the GCD of $(p = a + 1)$ and N. In case the GCD is greater than 1, we recovered a as well as the corresponding prime factor p. We can then factor the modulus N by computing $q = N/p$.

As mentioned before, the iteration counter m is upper-bounded by the value $\lceil K - \log_2(e) \rceil$ with K being the bit size of the prime numbers. This is because each iteration reduces a_i by at least one bit due to the right shift operation. For example, a 4 096 bit RSA key will have prime numbers of length $K = 2\,048$ bits, yielding $m = 2\,032$ iterations to consider. However, a prime number which is closer to 2^{K-1} than to 2^K combined with the subtraction in branch 1 could reduce a_i by one additional bit. This would make swapping occur one iteration

Table 7.1: Executed and recovered operations when calling BN_gcd(...) for $a = 11082$ and $b = 17$.

a	b	Performed operation	$a < b$ swap	Page observation	Recovered operation	Substituted equation
0010 1011 0100 1010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_1 = \frac{a}{2}$
0001 0101 1010 0101	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_2 = \frac{a}{4} - \frac{b}{2}$
0000 1010 1100 1010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_3 = \frac{a}{8} - \frac{b}{4}$
0000 0101 0110 0101	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_4 = \frac{a}{16} - \frac{5b}{8}$
0000 0010 1010 1010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_5 = \frac{a}{32} - \frac{5b}{16}$
0000 0001 0101 0101	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_6 = \frac{a}{64} - \frac{21b}{32}$
0000 0000 1010 0010	0001 0001	$a_{i+1} = \frac{a_i}{2}$	no	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_7 = \frac{a}{128} - \frac{21b}{64}$
0000 0000 0101 0001	0001 0001	$a_{i+1} = \frac{a_i - b_i}{2}$	no	P_1, P_2, P_4, P_2	$a_{i+1} = \frac{a_i - b}{2}$	$a_8 = \frac{a}{256} - \frac{85b}{128}$
0000 0000 0010 0000	0001 0001	$a_{i+1} = \frac{a_i}{2}$	yes	P_4, P_2	$a_{i+1} = \frac{a_i}{2}$	$a_9 = \frac{a}{512} - \frac{85b}{256}$
0000 0000 0001 0001	0001 0000	$\mathbf{b_{i+1} = \frac{b_i}{2}}$	no	P_4, P_2	$\mathbf{a_{i+1} = \frac{a_i}{2}}$	$\mathbf{a_{10} = \frac{a}{1024} - \frac{85b}{512}}$
⋮	⋮	⋮	⋮	discard		
0000 0000 0000 0001	0000 0000	Return a as the GCD				

earlier. We would erroneously consider an incorrect equation due to swapping, and determining the correct a might fail. In this case, we simply omit the last erroneous equation a_m from the recursive substitution and try to determine a again by iterating over values $a_{m-1} \in [1, e)$. As we will see in Section 7.4, this happens in approximately 25% of all runs, meaning that about 75% of the generated RSA keys can be recovered in the first run.

In case $p - 1$ is not coprime to e–which is the reason why the binary GCD algorithm is executed–the RSA key generation will discard this prime factor candidate p and re-generate another prime factor candidate p. Nevertheless, by observing the page fault pattern, an attacker is also able to detect this (extremely rare) case, and we run the same attack on the newly generated p.

Example. For an illustrative example let us assume the following hypothetical parameters: the public exponent is $e = 17 = \texttt{0x11}_{16}$ and the two 14-bit primes are $p = 11083 = \texttt{0x2B4B}_{16}$ and $q = 9941 = \texttt{0x26D5}_{16}$, respectively. In the course of validating the selected parameters, the OpenSSL implementation calls the binary GCD function with $a = 11082$ and $b = 17$. Table 7.1 illustrates the executed operations for the given input parameters a and b. In the first loop iteration, a is even and b is odd, which means that the function BN_rshift1(...) will be called. In the second loop iteration, a is odd and b is odd, which means that BN_sub(...) followed by BN_rshift1(...) will be executed, and so on. Finally, the algorithm returns 1 as the GCD of $a = 11082$ and $b = 17$.

Based on a controlled-channel attack, we are able to observe accesses to pages P_1, P_2, and P_4, and to precisely recover the executed operations up to the point where a and b are swapped. We recursively substitute the recovered operations on a_i, which leads to the equations shown in the last column of Table 7.1. Recall that the first swap will happen *at latest* after $m = \lceil 14 - \log_2(17) \rceil = 10$ iterations. In our example, swapping is done already in iteration 9 due to a smaller p and additional subtractions. This leads to the erroneously recovered operation marked bold in Table 7.1. To recover the secret a, we start with the m-th substituted

equation a_{10}, not knowing that it is erroneous. If the attempt to recover a based on a_{10} fails, we would need to fall back to equation a_9. However, in this particular case, the error cancels out, and we already succeed with a_{10}. Recall that $a_{10} = \frac{a}{1024} - \frac{85b}{512}$. With $b = 17$, we can rearrange it to

$$a = 1024 \cdot a_{10} + 2890 \tag{7.1}$$

The unknown variable a_{10} is bound by the parameter b. Since a and b have been swapped, a_{10} must be smaller than b. We try to solve this equation by iterating over $a_{10} \in [1, b)$ and checking the GCD of $a + 1$ and N. If the GCD is greater than 1, we can factor N. Indeed, for $a_{10} = 8$ the equation yields $a = 11\,082$ and $\gcd(a + 1, N) > 1$. Thus, we recovered the first prime $p = 11\,083$, which allows to factor N ($q = N/p = 9941$) and to recover the secret exponent $d \equiv e^{-1} \mod (p - 1)(q - 1)$. To see why recovery on the erroneous equation a_{10} works in this case, we compare it to the valid equation $a_9 = \frac{a}{512} - \frac{85b}{256}$, which can be rewritten as

$$a = 512 \cdot a_9 + 2890 \tag{7.2}$$

Here, recovering a succeeds for $a_9 = 16$. Observe that in equations (7.1) and (7.2) the first constants are only off by a factor of 2 because the erroneous operation does not introduce a subtraction but only a right shift. Hence, we hit the correct guess with $a_{10} = a_9/2 = 8$.

7.3.4 Generalization

The proposed attack on RSA key generation is not limited to code pages only. One could also monitor accesses to data pages, especially those on which the heap buffers a and b reside. If a and b are located on different heap data pages, we can distinguish which of these buffers is accessed and, thus, which arguments are provided to the BIGNUM functions. This allows to distinguish all relevant branches, enabling the idealized attack described in Section 7.3.1. For example, one can distinguish branch 2 and 3 based on the input of BN_rshift1(...) in lines 13 (accessing b only) and line 20 (accessing a only) of Listing 7.1. Also, one can detect swapping of a and b, after which their pointers map to the opposite page, respectively. For example, if BN_is_zero(...) in line 4 accesses buffer a instead of b, or the call to BN_cmp(...) (line 9, 14 or 21) accesses b *before* a, one can infer that swapping occurred in the previous iteration. Thus, one could derive equations over all iterations and recover the key without the need for guessing values for $a_m \in [1, e)$.

Even if a and b are located on the same heap page, attacks might still be possible. An enclave might copy variable-sized user input onto the heap such as messages to sign, for example. By carefully crafting this user input, an attacker can shift the targeted buffers a and b onto different heap pages. We did not investigate such generalized attacks further, since our attack already recovers the full key by monitoring page faults on code memory.

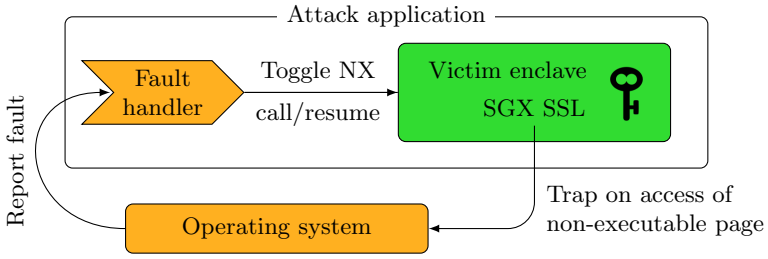

Figure 7.2: Basic principle of the performed attack.

7.4 Attack Evaluation

We evaluate the presented attack on an Intel Core i7-6700K 4.00 GHz platform running Ubuntu 17.10 (Linux kernel 4.13.0-37). In order to do so, we developed an SGX application that generates an RSA key based on the latest version of Intel SSL SGX.[6] We used the Linux Intel SGX software stack v1.9, consisting of the Intel SGX driver, the Intel SGX software development kit (SDK), and the Intel SGX platform software (PSW).[7] For controlling the page mapping, we used the SGX-Step kernel module as well as the corresponding SGX-Step library functions (cf. [BPS17]). Note that we do not use the single-stepping feature of SGX-Step but rather its page mapping capability. Since Intel SGX considers an untrusted OS, the application of SGX-Step is in line with the threat model. We describe the implementation details below.

7.4.1 Implementation Details

We consider a victim enclave using the Intel SGX SSL library to generate an RSA key pair. The enclave is hosted by a malicious attack application that interacts with the OS to manipulate page mappings and to record page accesses within the corresponding fault handler. Figure 7.2 depicts the principle of the attack. After this recording step, we evaluate the collected trace of page accesses in order to recover the secret key.

SGX Enclave Application (Victim Enclave). We developed an enclave program that generates a single RSA key using the Intel SGX SSL library and outputs the public parts only, *i.e.*, the modulus N. We implemented an ECALL function for invoking key generation, as well as an OCALL function that prints the modulus of the generated key to the standard output. Recall that the public exponent is fixed to $e = 65\,537$. The project is built in pre-release hardware mode, *i.e.*, it uses the same compiler optimizations as a production enclave in release mode and yields the same memory layout.

[6] We relied on the most recent commit 654f94d of Intel SSL SGX, which in turn is based on OpenSSL version 1.1.0g (https://www.openssl.org/source/openssl-1.1.0g.tar.gz).

[7] https://github.com/01org/linux-sgx.

Attack Application. Based on the SGX-Step framework [BPS17], we developed an attack application that enables and disables executable regions (pages) of the enclave program. More precisely, it toggles the NX bit of the page table entries belonging to the code pages to be traced. One could also use the present bit or a reserved bit [XCP15; Xia+17] for the same purpose. The application registers a fault handler (via a `sigaction` standard library function call), which is executed whenever the enclave encounters a segmentation fault due to a non-executable page. This fault handler conveniently serves as the basis to monitor page faults, which later on allow recovering the executed code paths.

7.4.2 Mounting the Attack

In order to determine the pages of interest, *i.e.*, the ones where the BN_gcd(...), BN_sub(...), and BN_rshift(...) functions are located, we dissect the enclave binary by means of `objdump`. In our case, `objdump` reveals the following page frame numbers: `0x00CA` for BN_gcd(...), `0x00C4` for BN_sub(...), and `0x00D8` for BN_rshift1(...). When starting the victim enclave, the attack application disables execution of the BN_gcd(...) page by setting the non-executable (NX) bit in the corresponding page table entry. This causes the enclave to trap as soon as it attempts to execute this page.

When the fault handler function is executed for the first time, *i.e.*, when a page fault (segmentation fault) occurs, we start recording subsequent page faults. On the one hand, we enable execution of the current page, which caused the page fault by clearing its NX bit in order to allow the enclave to continue. On the other hand, we also disable the other pages of interest by setting their NX bits. Whenever the page fault handler is triggered, we record the accessed page and toggle the non-executable bits accordingly. Thus, we are able to monitor each access to these pages precisely.

Our practical evaluation confirmed that we observe the following page fault patterns. Executing branch 1 leads to consecutive page faults on `0x00C4` (BN_sub(...)) and `0x00D8` (BN_rshift1(...)), interleaved with page faults on `0x00CA` (BN_gcd(...)), whereas executing branch 3 leads to a page fault on `0x00D8` (BN_rshift1(...)) only. When the attack application finished gathering the page faults, we process the page-fault sequence from left to right and build up an equation system according to the rules in Section 7.3.3. That is, whenever we observe consecutive accesses to pages `0x00C4` and `0x00D8`, we add $a_{i+1} = (a_i - b)/2$. For a single access to page `0x00D8` we add $a_{i+1} = a_i/2$. Based on these equations, we run a SageMath script that recursively substitutes the equations, recovers the remaining bits by solving the equation for a_m, and finally recovers the private key.

The execution time of the attack, including the gathering and the parsing of the page-fault trace, is negligible, even when attacking larger RSA keys. Causing page faults on the above-mentioned pages increases runtime slightly. Compared to normal key generation, running the attack causes moderate overall slowdowns of 65 ms (15,5%) for 4,096 bit keys and 248 ms (5,87%) for 8,192 bit keys due to the intentionally induced page faults. The largest share of the execution time is

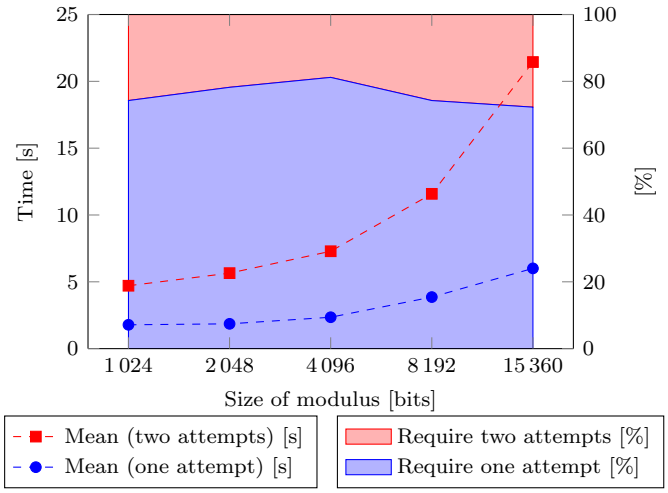

Figure 7.3: Key recovery complexity for different bit sizes of the modulus N.

consumed by the generation of the two random primes, *i.e.*, the random number generation and the primality test, during RSA key generation.

7.4.3 Key Recovery Complexity

We developed a simple script for SageMath[8] that iterates over all possible values for $1 \leq a_m < 65\,537$, evaluates $a = f(a_m)$, and checks the GCD of $a + 1$ and N. Figure 7.3 illustrates the complexity for the task of recovering the remaining bits. The complexity has been averaged over 100 runs per modulus size and the computations are evaluated with SageMath on an Intel Xeon E5-2660 v3 (2.60GHz). The area plot (right x-axis) indicates that in about 75%–80% of all cases, the prime factors can be recovered at the first attempt, considering $m = \lceil K - \log_2(e) \rceil$ equations. In only about 20%–25% of all cases the first attempt fails due to an early swapping in the binary GCD algorithm. In this case, we need to remove the last equation a_m and restart the search in the range $1 \leq a_{m-1} < 65\,537$. The asymptotic complexity of the key recovery is $\mathcal{O}(1)$. This means that the number of iterations is bound by the public exponent e, which is a constant value. In contrast, the computation time of the GCD for candidates a increases due to the larger bit sizes of the modulus N. In 75% of all cases, a 8 192-bit modulus can be factored in less than 5 seconds on average, after gathering the measurement trace. In only 25% of all cases, we need approximately 12 seconds on average. Although 15 360-bit RSA keys (providing 256-bit security according to NIST [Bar16]) are currently not being used in practice, we provide the results here for the sake of completeness.

[8] http://www.sagemath.org/

```
diff --git a/crypto/rsa/rsa_gen.c b/crypto/rsa/rsa_gen.c
index 4ced965..4051933 100644
--- a/crypto/rsa/rsa_gen.c
+++ b/crypto/rsa/rsa_gen.c
@@ -41,6 +41,7 @@ static int \
 rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
 {
     BIGNUM *r0 = NULL, *r1 = NULL, *r2 = NULL, \
            *r3 = NULL, *tmp;
     int bitsp, bitsq, ok = -1, n = 0;
+    unsigned long error = 0;
     BN_CTX *ctx = NULL;

     /*
@@ -88,16 +89,25 @@ static int \
 rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
     if (BN_copy(rsa->e, e_value) == NULL)
         goto err;

+    BN_set_flags(rsa->e, BN_FLG_CONSTTIME);
+
     /* generate p and q */
     for (;;) {
         if (!BN_generate_prime_ex(rsa->p, bitsp, 0, \
             NULL, NULL, cb))
             goto err;
         if (!BN_sub(r2, rsa->p, BN_value_one()))
             goto err;
-        if (!BN_gcd(r1, r2, rsa->e, ctx))
-            goto err;
-        if (BN_is_one(r1))
-            break;
+        // Inverse only exists if GCD = 1
+        if (BN_mod_inverse(r1, r2, rsa->e, ctx))
+            break; // GCD = 1
+        else {
+            error = ERR_peek_last_error();
+            if (ERR_GET_LIB(error) == ERR_LIB_BN &&
+                ERR_GET_REASON(error) == BN_R_NO_INVERSE)
+                    ERR_clear_error(); // GCD != 1
+            else
+               goto err; // Another error occurred
+        }
         if (!BN_GENCB_call(cb, 2, n++))
             goto err;
     }
@@ -110,10 +120,17 @@ static int \
 rsa_builtin_keygen(RSA *rsa, int bits, BIGNUM *e_value,
         } while (BN_cmp(rsa->p, rsa->q) == 0);
         if (!BN_sub(r2, rsa->q, BN_value_one()))
             goto err;
-        if (!BN_gcd(r1, r2, rsa->e, ctx))
-            goto err;
-        if (BN_is_one(r1))
-            break;
+        // Inverse only exists if GCD = 1
+        if (BN_mod_inverse(r1, r2, rsa->e, ctx))
+            break; // GCD is 1
+        else {
+            error = ERR_peek_last_error();
+            if (ERR_GET_LIB(error) == ERR_LIB_BN &&
+                ERR_GET_REASON(error) == BN_R_NO_INVERSE)
+                    ERR_clear_error(); // GCD != 1
+            else
+                goto err; // Another error occurred
+        }
         if (!BN_GENCB_call(cb, 2, n++))
             goto err;
     }
```

Listing 7.2: Patch for RSA key generation in OpenSSL.

7.5 Patching OpenSSL

We discussed several possible countermeasures against controlled-channel attacks in Chapter 6. The most straightforward approach to prevent the attack described in this work is to fix the RSA key generation procedure at the implementation level. We propose an appropriate patch in Listing 7.2.

Instead of relying on BN_gcd(...) to ensure that $p-1$ and e are coprime, we compute the modular inverse of $p-1$ modulo e using a side-channel protected modular inversion algorithm (BN_mod_inverse(...)). The inverse only exists if $\gcd(p-1,e)=1$. Hence, if BN_mod_inverse(...) signals through an error that the inverse does not exist, we know that $\gcd(p-1,e)\neq 1$. This corresponds to lines 33–43 and 56–66 in Listing 7.2.

In order to ensure that the side-channel protected inversion is called, we need to set the BN_FLG_CONSTTIME flag on the public modulus e (see line 20). In this case, BN_mod_inverse(...) internally calls the protected function BN_mod_inverse_no_branch(...), which has some side-channel protection in place. In Chapter 9, we will show that also this side-channel protected inversion can leak one bit of information in some cases. While this is problematic for DSA-based signature schemes where an attacker can query a large number of "leaky" signatures, it does not threaten the security of RSA key generation.

Performance Impact. An appealing benefit of our proposed patch is that it is even faster than the vulnerable implementation.[9] We benchmarked 10 000 coprimality checks for a random number a and $e = 65\,537$, and provide the corresponding cumulative execution times in Table 7.2. As can be seen, our patch is by one to two orders of magnitudes faster than the original implementation on our first test machine. On an Intel Core i7-5600U 2.6 GHz CPU (notebook), the speedup exceeds even a factor of 500 for 8 192 bit numbers.

The reason for this massive speedup is that inversion, as implemented in OpenSSL, uses the original Euclidean algorithm with $\gcd(a,b)=\gcd(b,a \mod b)$. This algorithm requires far fewer loop iterations (e.g., between 5 and 13 iterations for 8 192-bit numbers) than the binary GCD (≈ 8192 iterations). The original Euclidean algorithm relies on a costly modular reduction in each iteration, which was the initial motivation to use the binary GCD instead. The binary GCD replaces modular reductions with shift operations and subtractions. However, on our test machines, the original Euclidean algorithm is significantly faster because OpenSSL leverages the x86 `div` instruction to perform the expensive modular reductions directly in hardware.

In any case, the coprimality check only contributes a small share to the overall runtime, as opposed to the prime generation itself. It handles a corner case in RSA key generation, which is highly unlikely to happen in practice. Hence, the corresponding check is, in general, only executed once per generated prime factor and, thus, two times during the RSA key generation.

[9] Note that we do not need to compute the GCD but only check whether or not it is 1.

Table 7.2: Performance comparison for 10 000 runs on an Intel Core i7-6700K (upper half) and an i7-5600U (lower half).

Bit size of a	BN_gcd(a, e)	BN_mod_inverse(a, e)
1 024	0.25 s	0.02 s
2 048	0.69 s	0.03 s
4 096	2.07 s	0.03 s
8 192	6.97 s	0.05 s
1 024	1.18 s	0.03 s
2 048	3.78 s	0.04 s
4 096	14.04 s	0.06 s
8 192	54.64 s	0.10 s

7.6 Further Vulnerabilities

RSA X9.31. Further investigation of the OpenSSL source code revealed that the prime derivation function based on the ANSI X9.31 standard [Ins98] (BN_X931_derive_prime_ex(...)) is also vulnerable to the presented attack. Similar as in the default RSA key generation procedure implemented in `rsa_gen.c`, the generated primes p and q are verified, *i.e.*, that $p-1$ and $q-1$ are coprime to the public modulus e. Hence, the same attack technique also applies to the X9.31 implementation. Irrespective of whether or not this implementation is actually used (ANSI X9.31 has already been withdrawn in [BR15]), we suggest patching this implementation, similar to Section 7.5.

Furthermore, there are two additional usages of the vulnerable BN_gcd(...) function, namely in RSA_X931_derive_ex(...) and RSA_check_key_ex(...). In these cases, the GCD is not used as a mere security check but to factor out the GCD of the product $(p-1)(q-1)$. Since the calculated GCD is never 1, our patch using the inversion algorithm cannot be applied here. Instead, we suggest adding a constant-time implementation of the GCD algorithm, which is resistant against software side-channel attacks. Ideally, this implementation is even faster than the binary GCD implementation (cf. the performance analysis of our proposed patch in Section 7.5).

RSA Blinding. While our attack highlights a critical vulnerability in RSA key generation, other algorithms also need careful evaluation with respect to single-trace attacks. For example, we found a vulnerability in the generation of RSA blinding values used to thwart side-channel attacks on sensitive RSA exponentiation. In OpenSSL, blinding is created via `BN_BLINDING_create_param`, which uses `BN_mod_exp` to prepare the inverse blinding value for later unblinding. However, we found that `BN_mod_exp` does not check all necessary parameters for the constant-time flag, falling back to unprotected exponentiation in this case. Similar to the attack presented in this chapter, a controlled-channel attacker could attempt to recover the blinding value from a single trace and subsequently peel off the side-channel protection offered by blinding. The OpenSSL team

fixed this issue in response to our findings by using the side-channel protected exponentiation algorithm appropriately.

7.6.1 Responsible Disclosure

We responsibly notified Intel as well as OpenSSL about our findings and provided a patch to fix the RSA key generation, as shown in Listing 7.2. In response, OpenSSL patched the RSA key generation vulnerability in commit `8db7946e`. Also, the RSA blinding vulnerability was fixed in commit `e913d11f`.[10]

7.7 Summary

In this chapter, we investigated the RSA key generation routine executed inside SGX enclaves under the aspect of software side-channel attacks. Our investigations revealed a critical vulnerability inside Intel SGX SSL that allows recovering the generated RSA secret key with a single observation using a controlled-channel attack. More specifically, the observable page fault patterns during the RSA key generation help recover the prime factor p and, thus, to factor the modulus N. To the best of our knowledge, this represents the first software-based side-channel attack targeting the RSA key generation process.

Ironically, the discovered vulnerability is due to an optimized binary GCD algorithm that should improve the performance compared to the original Euclidean algorithm but, in fact, is significantly slower on Intel x86 platforms. Moreover, the vulnerable GCD computation itself is only a security check to cover rare cases in which $p - 1$ or $q - 1$ share a common factor with e.

Nevertheless, our work demonstrates that cryptographic operations, which might fall outside the threat model of cache attacks, can be very well subject to side-channel attacks in an SGX setting. Many implementations, such as OpenSSL, were only protected against cache attacks. Before porting them to an SGX enclave, they have to be re-evaluated under stronger controlled-channel attacks. In general, more tool support is desired to assist software developers in the painful process of analyzing their cryptographic implementations against side-channel vulnerabilities.

[10] https://github.com/openssl/openssl.git

✳ ✳ ✳

8

DATA: Differential Address Trace Analysis

Love earns the right to speak the truth, but truth proves that you really love.

Mark Hall

Side-channel attacks can infer sensitive information by monitoring inadvertent information leaks of computing devices. Especially cryptographic implementations are a valuable attack target, as we discussed in Section 6.2. The class of software-based side-channel attacks (e.g., cache attacks, DRAM attacks, branch-prediction attacks, and controlled-channel attacks) are particularly dangerous since they can be launched from software and, thus, without the need for physical access to the target device. At its core, all these attacks exploit leakage of memory access patterns. In other words, they exploit *address leakage*, which occurs due to an *address-based side channel*. In this chapter, we introduce a new methodology to detect address leakage to help counteract side-channel attacks.

Various countermeasures against address leakage have been proposed (cf. Section 6.3). While some attempt to close the side channel, a more promising line of defense is to remove address-based information leaks from the software itself, thus fixing the side-channel vulnerability. To fix side-channel vulnerabilities, one needs to eliminate secret-dependent memory accesses for both data accesses and code fetches. For example, data leakage can be thwarted through bitslicing [RSD06; Kön08; KS09] and control-flow leakage by unifying the control flow [Cop+09]. Even though software countermeasures are already well studied, in practice their adoption to cryptographic libraries is often partial, error-prone,

or non-transparent, as demonstrated by recent attacks on OpenSSL [YGH16; GBY16; GB17].

In order to address these issues, leakage detection tools have been developed that allow developers and security analysts to identify side-channel vulnerabilities. They can be classified into static and dynamic approaches. Many static analysis methods use abstract interpretation [KMO12; Doy+13; DK17; MWK17] to give upper leakage bounds. They ideally prove the absence of information leaks in already secured implementations, e.g., the evaluation of Salsa20 [Doy+13]. However, these approaches struggle to describe and pinpoint information leaks accurately due to over-approximation [Doy+13, page 443], rendering leakage bounds meaningless in the worst case. Moreover, their approximations of the program's data plane fundamentally prohibit the analysis of interpreted code.

In contrast, dynamic approaches focus on concrete program executions to reduce false positives. Contrary to static analysis, dynamic analysis cannot prove the absence of leakage without exhaustive input search, which is infeasible for large input spaces. However, in the case of cryptographic algorithms, testing a subset of inputs is enough to encounter information leaks with a high probability, because cryptographic primitives heavily diffuse the secret input during processing. Thus, there is a fundamental trade-off between static analysis (minimizing false negatives) and dynamic analysis (minimizing false positives).

We aim for a pragmatic approach towards minimizing false positives. This is necessary for helping developers to identify information leaks in real-world applications. In this chapter, we focus on dynamic analysis and tackle the limitations of existing tools. In particular, existing tools either focus on control-flow leaks or data leaks, but not both at the same time [ZHS16; Wan+17a; Xia+17]; they could suffer from false positives and require source code annotations [Lan10], or they consider the strongest adversary to observe cache-line accesses only [Ira+17], which is too coarse-grained in light of recent attacks (CacheBleed [YGH16]). A more detailed discussion of those tools is given in Section 6.3.3. Based on these shortcomings, we argue that tools designed to identify address-based information leaks must tackle the following challenges:

1. *Leakage origin*: Detect the exact location of both data and control-flow leaks on byte-address granularity instead of cache-line granularity.
2. *Detection accuracy*: Minimize false positives and provide reasonable strategies to also reduce false negatives.
3. *Practicality*: Report information leaks (i) fully automated, *i.e.*, without requiring manual intervention, (ii) using only the program binary, *i.e.*, without requiring the source code, and (iii) efficiently in terms of performance.

In this work, we tackle these challenges with *differential address trace analysis* (DATA), a methodology and tool to identify address-based information leaks in application binaries.[1] DATA targets programs processing secret input, e.g., keys or passwords, and reveals dependencies between the secret and the program execution. Every leak that DATA identifies in a program is potentially exposed

[1] DATA is open-source and can be retrieved from `https://github.com/Fraunhofer-AISEC/DATA`.

to side-channel attacks. DATA works in three phases, of which only the first phase is included in this thesis.

First Phase: Difference Detection. The first phase generates noiseless address traces by executing the target program with binary instrumentation. It identifies differences in these traces on a byte-address granularity. This accounts for all address-based side-channel attacks, such as cache attacks [OST06; YF14; Per05], DRAM attacks [Pes+16], branch-prediction attacks [AKS07b], controlled-channel attacks [XCP15], and many blackbox timing attacks [Ber05]. Since we execute the program under test with concrete input, we avoid false positives by design, given that the program is deterministic. To reduce false negatives, we repeatedly execute the same program under different inputs.

Second and Third Phase: Statistical Tests. DATA also employs two statistical phases for further analysis. They are not part of this thesis but can be looked up in the original publication [Wei+18a]. In short, the second phase explicitly addresses non-deterministic program behavior, such as cryptographic blinding, and helps distinguish it from actual key dependencies. Finally, the third phase helps classify information leakage according to a particular leakage model chosen by the analyst. A leakage model helps assess the severity and exploitability of a leak, as we will also see in Chapter 9.

Evaluation. We implement DATA in a fully automated evaluation tool that allows analyzing large software stacks, including initialization operations, such as key loading and parsing, as well as cryptographic operations. We use DATA to analyze OpenSSL and PyCrypto, confirming existing and identifying new vulnerabilities. Among several expected leaks in symmetric ciphers (AES, Blowfish, Camellia, CAST, Triple DES, ARC4), DATA also reveals known and previously unknown leaks in asymmetric primitives (RSA, DSA, ECDSA) and identifies erroneous bug fixes of supposedly resolved vulnerabilities.

Contributions. Our contributions are summarized as follows:

- We propose a method for discovering address-based side-channel vulnerabilities called differential address trace analysis (DATA), which captures both data leaks and control flow leaks.
- We implement DATA as a fully automated framework for analyzing unannotated production binaries.
- We evaluate DATA on common cryptographic libraries and, thereby, confirm existing and identify new vulnerabilities. To the best of our knowledge, we perform the first address-based analysis of interpreted code (PyCrypto) and the interpreter (CPython).

This chapter is based on parts of the original publication [Wei+18a], namely the difference detection phase, which is my core contribution, alongside its implementation and the vulnerability analysis. As mentioned, the statistical phases two and three are not a contribution of this thesis but can be looked up in [Wei+18a]. The remainder of this chapter is organized as follows. In Section 8.1, we discuss related work. In Section 8.2, we introduce the methodology behind DATA. In Section 8.3, we give implementation details and optimizations. In

Section 8.4, we evaluate DATA on OpenSSL and PyCrypto and discuss our findings. Finally, we summarize in Section 8.5.

8.1 Related Work

We already covered related side-channel analysis tools in Section 6.3.3 and Table 6.1, in particular. In the following, we show how DATA overcomes several shortcomings of existing approaches, thus meeting the challenges identified before.
Leakage Origin. DATA follows a dynamic trace-based approach to identify both control flow and data leakage on byte-address granularity. Having a fine granularity avoids wrong assumptions about attackers, e.g., only observing memory accesses at cache-line granularity [Wan+17a; Doy+13; Ira+17; Wan+19], which were disproved by more advanced attacks [YGH16; AKS07b; Mog+20]. Nevertheless, identifying information leaks on a byte granularity still allows to map them to more coarse-grained attacks.
Detection Accuracy. Static approaches like CacheAudit [KMO12; Doy+13] suffer from false positives. In contrast, DATA's phase one avoids false positives for deterministic programs. However, as with all dynamic approaches, DATA could theoretically miss leakage that is not triggered during execution. Nevertheless, we found that few traces already suffice in practice, e.g., ≤ 10 for asymmetric algorithms, and ≤ 3 for symmetric algorithms, due to the high diffusion provided by these algorithms. Although without a formal guarantee, this gives evidence that DATA reduces false negatives successfully. Compared to others, we take multiple measures to reduce false negatives in DATA. In contrast to CacheD [Wan+17a] and ctgrind [Lan10], which analyze a single execution path only, we analyze several execution paths. Compared to Stacco [Xia+17], which has improper trace alignment, we report all leaks visible in the address traces. Contrary to Irazoqui et al. [Ira+17], we do not only focus on a specific attack technique (e.g., cache attacks). This advantage of DATA is indicated by ◐ in Table 6.1.

DATA also measures up to more recent work in terms of detection accuracy. MicroWalk [Wic+18] compresses execution contexts, thus trading detection accuracy against analysis speed. DATA does not perform any compression. By operating on the raw address traces, DATA can detect even nested leaks (leaks within other leaks). CacheS [Wan+19] applies abstract interpretation to larger programs, however, only at cache-line granularity. Also, they miss leakage due to implicit information flows, which DATA is able to detect.
Practicality. DATA analyzes information leaks fully automatically. It does so on the program binary without the need for source code, allowing analysis of proprietary software. As will be outlined in our evaluation, we achieve competitive performance and support analysis of large software stacks and even interpreted code (PyCrypto and CPython). Finally, DATA is open source.

8.2 Differential Address Trace Analysis

DATA is a methodology and a tool to identify address-based information leaks in program binaries. In the following, we introduce our threat model and outline our methodology for detecting address differences. We then go into details of our methodology, as follows: First, we introduce address-based information leaks. Second, we show how address traces are recorded. Third, we explain the process of finding differences in the recorded traces.

8.2.1 Threat Model

To cover a wide variety of possible attacks, we consider a powerful adversary who attempts to recover secret information from side-channel observations. In practice, attackers will likely face noisy observations because side channels typically stem from shared resources affected by noise from system load. Also, practical attacks only monitor a limited number of addresses or memory blocks. For DATA, we assume that the attacker can accurately observe full, noise-free address traces. More precisely, the attacker does not only learn the sequence of instruction pointers [Mol+05], *i.e.*, the addresses of instructions, but also the addresses of the operands that are accessed by each instruction. This is a strong attacker model that covers many side-channel attacks targeting the processor microarchitecture (e.g., branch prediction) and the memory hierarchy (e.g., various CPU caches, prefetching, DRAM). A strong model is preferable here to detect as many vulnerabilities as possible. In line with [Gru+16], we consider defenses, such as address space layout randomization (ASLR) and code obfuscation, as ineffective against powerful attackers.

Limitations. DATA covers software side channels of components that operate on address information only, e.g., cache prefetching and replacement, and branch prediction based on the branch history. In contrast, the recent transient execution attacks, such as Spectre [Koc+19] and Meltdown [Lip+18], exploit not only *address* information but actual *data* that is speculatively processed but insufficiently isolated across different execution contexts. In these attacks, sensitive data spills over to the address bus. These hardware bugs cannot be detected by analyzing software binaries with tools listed in Table 6.1. While software-only defenses exist for specific CPU models [Tur18; Cor18b], a generic solution should fix the hardware.

Relation to Similar Concepts. The idea of DATA is similar to differential power analysis (DPA) [KJJ99], which works on power traces. However, power traces are often noisy due to measurement uncertainty and the underlying physics. Hence, DPA often requires several thousands of measurements, and non-constant-time implementations demand heavy pre-processing to align power traces correctly [MOP07]. In contrast, address traces are noise-free, which minimizes the number of required measurements and allows perfect re-alignment for traces that are non-constant time due to control-flow leaks.

DATA is also related to differential computation analysis (DCA) [Bos+16]. DCA relies on software execution traces to attack white-box crypto implemen-

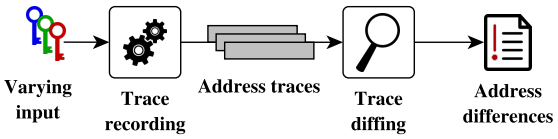

Figure 8.1: Differential address trace analysis searches for differences in recorded address traces.

tations. While DCA is conceptually similar to DATA, the white-box model considers a much stronger adversary who can read the actual content of accessed memory locations rather than addresses only.

8.2.2 Methodology

DATA detects address differences in a multi-step process, as depicted in Figure 8.1. First, we execute the target program with a binary instrumentation tool and record all accessed code addresses as well as memory addresses.Thereby, we ensure to capture both control flow and data leakages at their exact origin. By executing the program multiple times with varying secret input values (e.g., cryptographic keys or messages), we obtain multiple address traces. Second, DATA compares the recorded address traces using a dedicated trace diffing algorithm. Differences in these traces suggest potential information leakage and are summarized in a report. For deterministic programs, all reported differences are secret-dependent and, thus, true positives.

8.2.3 Address-based Information Leakage

We now introduce address-based information leakage and give descriptive examples of data leakage as well as control-flow leakage. DATA analyzes a program binary P with respect to address leakage of secret input k. Let $P(k)$ denote the execution of a program with controllable secret input k. We write $t = \mathsf{trace}(P(k))$ to record a trace of accessed addresses during program execution. We define an address trace $t = [a_0, a_1, a_2, a_3...]$ as a sequence of executed instructions, augmented with memory addresses. For instructions operating on CPU registers, $a_i = ip$ holds the current instruction pointer ip. In case of memory operations, $a_i = (ip, d)$ also holds the accessed memory address d. Information leaks appear as differences in address traces. We develop an algorithm $\mathsf{diff}(t_1, t_2)$ that, given a pair of traces (t_1, t_2), identifies all differences. If the traces are equal, $\mathsf{diff}(t_1, t_2) = \varnothing$. A deterministic program P is leakage free if and only if no differences show up for any pair of secret inputs (k_i, k_j):

$$\forall k_i, k_j : \mathsf{diff}(\mathsf{trace}(P(k_i)), \mathsf{trace}(P(k_j))) = \varnothing \tag{8.1}$$

```
0  program entry: // call process with user-input key
1  unsigned char LUT[16] = { 0x52,
2                            0x19,
                             ...
16                           0x37 };
17 int transform(int kval) { return LUT[kval%16]; }
18 int process(int key[3]) {
19   int val = transform(0);
20   val += transform(key[0]);
21   val += transform(key[1]);
22   val += transform(key[2]);
23   return val;
   }
```

Listing 8.1: Vulnerable table look-up causing data leakage of `key`.

Data leakage is characterized by the same instruction (ip) accessing different memory locations (d). Consider the code snippet in Listing 8.1. Without loss of generality, in our examples, line numbers correspond to code addresses. Execution with two different keys $key_A = [10, 11, 12]$ and $key_B = [16, 17, 18]$ yields two address traces $t_A = \mathsf{trace}(P(key_A))$ and $t_B = \mathsf{trace}(P(key_B))$.

Starting at the program entry 0, we obtain the following traces, with address differences marked bold:

$$t_A = [0, 18, 19, (17, 1), 20, (17, \mathbf{11}), 21, (17, \mathbf{12}), 22, (17, \mathbf{13}), 23]$$
$$t_B = [0, 18, 19, (17, 1), 20, (17, \mathbf{01}), 21, (17, \mathbf{02}), 22, (17, \mathbf{03}), 23]$$

The function `transform` leaks the argument `kval` (line 17), which is used as an index into the array table `LUT` (lines 1–16). Since the base address of `LUT` is 1, this operation leaks memory address `kval` + 1. The first call to `transform` (line 19) with `kval` = 0 always results in $a_2 = (17, 1)$. Subsequent calls to `transform`, however, leak sensitive key bytes (line 20–22). The differences in the traces–marked bold–reveal key dependencies. Note that the key bytes of key_B are reduced modulo 16 in line 17. Hence, an attacker can observe at most 16 different addresses every time this data leak is triggered. This corresponds to four bits of leakage since $log_2(16) = 4$.

To accurately report data leakage and to distinguish non-leaking cases (line 19) from leaking cases (line 20–22), we take the call stack into account. We formalize data leaks as tuples (ip, cs, ev) of the leaking instruction ip, its call stack cs, and the evidence ev. The call stack is a list of caller addresses leading to the leaking function. For example, the first leak has the call stack $cs = [0, 20]$. The evidence is a set of leaking data addresses d. The larger the evidence set, the more information is leaked. For example, $ev = \{11, 01\}$ for the first leak, $ev = \{12, 02\}$ for the second one, etc. Our diff algorithm would report:

$$\begin{aligned}\mathsf{diff}(t_A, t_B) = \{&(17, [0, 20], \{11, 01\}),\\ &(17, [0, 21], \{12, 02\}),\\ &(17, [0, 22], \{13, 03\})\}\end{aligned}$$

```
0 program entry: // call exp with user-input key and public p
1 function exp(key, *p) {
    ...
2   foreach (bit b in key)
3     if (b)
4       mul(r, p);
      else
5       mul(t, p);
6   return r;
  }
  function mul(*a,*b) {
7   tmpA = *a;
8   tmpB = *b;
    // calculate res = tmpA * tmpB
9   *a = res;
  }
```

Listing 8.2: Secret-dependent branch causing control-flow leakage of `key`.

Control-flow leakage is caused by key-dependent branches or indirect jumps. As an example, we study a square&multiply exponentiation common for RSA decryption, depicted in Listing 8.2. For simplicity, we left out the squaring step. To avoid simple timing attacks, the algorithm always performs multiplication, either on the real buffer r (line 4), or on a temporary dummy buffer t (line 5). Nevertheless, the algorithm is prone to address leakage. Consider this algorithm to be executed with two keys $k_A = 4 = 100_2$ and $k_B = 7 = 111_2$. In our example, let R, P, and T denote the data addresses of the variables r, p, and t, respectively. Trace recording will yield the following address traces:

$$\begin{aligned}\mathsf{trace}(P(k_A)) = t_A = [&0, 1, 2, 3, 4, (7, R), (8, P), (9, R),\\ &2, 3, \mathbf{5, (7, T), (8, P), (9, T)},\\ &2, 3, \mathbf{5, (7, T), (8, P), (9, T)}, 2, 6]\\ \mathsf{trace}(P(k_B)) = t_B = [&0, 1, 2, 3, 4, (7, R), (8, P), (9, R),\\ &2, 3, \mathbf{4, (7, R), (8, P), (9, R)},\\ &2, 3, \mathbf{4, (7, R), (8, P), (9, R)}, 2, 6]\end{aligned}$$

There are two differences in the traces; both marked bold. The differences occur due to the `if` statement in line 3, which branches to line 4 or 5, depending on the key bit b. Moreover, it causes operations in line 7 and 9 to be done either on the intermediate variable r or a temporary variable t.

We can observe that the control-flow leak is characterized by its branch point, where the control flow diverges, and its merge point, where branches coalesce again. In this example, the branch point is at line 3 and the merge point at line 2, when the next loop iteration starts. The merge point is mainly of interest for trace alignment. We model control-flow leaks as tuples (ip, cs, ev) of branch point ip, call stack cs, and evidence ev. In our example, both differences occur at the same call stack $cs = [0]$. Hence, they are reported as one and the same

leak. The evidence is a set of sub-traces (*i.e.*, traces between the branch point and the merge point) corresponding to the two branches.

It is worth noting that the present code snippet also contains two data leaks in lines 7 and 9. However, since the leaky multiplication is wrapped in a control-flow leak, these data leaks do not reveal more information—they are part of the control-flow leak. Our diff algorithm would report:

$$\begin{aligned}\mathsf{diff}(t_A, t_B) = \{(3, [0], \{[4, (7, R), (8, P), (9, R)],\\ [5, (7, T), (8, P), (9, T)]\}\ )\}\end{aligned}$$

8.2.4 Recording Address Traces

To obtain address traces, we execute the program on a dynamic binary instrumentation (DBI) framework, namely Intel Pin [Luk+05]. We instrument the program to store all accessed code and data addresses in an address trace. To execute the program in a clean and noise-free environment, we disable ASLR and keep public inputs (e.g., command-line arguments, environment variables) to the program fixed. Disabling ASLR does not introduce false positives or negatives but simplifies trace analysis. As shown in Figure 8.1, we repeat trace recording multiple times with varying inputs, causing address leaks to show up as differences in the address traces.

The concept of DATA is agnostic to concrete recording tools and, hence, could also rely on other tools [Son+08] or hardware acceleration, such as Intel Processor Trace (IPT) [Int16a]. Since the recording time is short compared to trace analysis, we did not investigate other tools further.

8.2.5 Finding Trace Differences

To find address leaks, DATA compares recorded address traces. We developed a dedicated address trace comparison algorithm denoted as diff. It sequentially scans pairs of traces (t_A, t_B) for address differences, while continuously re-aligning traces in the same pass, should control-flow leaks occur. Thus, DATA not only discovers multiple subsequent control-flow leaks but also nested leaks (*i.e.*, leaks within control-flow leaks).

We run our diff algorithm pairwise on all recorded traces and accumulate the results in one report. Testing multiple traces helps capture nested leakage, that is, leakage, which appears conditionally, depending on which branches are taken in a superordinate control-flow leak. Nested leakage would remain hidden when only testing trace pairs that take different superordinate branches. By running diff on all pairwise combinations of traces, we increase the probability of discovering nested leaks. Thus, the number of diff invocations increases quadratically with the number of recorded traces. For our evaluation, we use up to ten traces.

Algorithm 8.1: Identifying address trace differences (diff).

input : t_A, t_B ... the two traces
output : rep ... the report of all differences

```
1  rep = ∅, i = 0, j = 0
2  while i < |t_A| ∧ j < |t_B| do
3  |  a = t_A[i], b = t_B[i]
4  |  if a.ip = b.ip then
5  |  |  if a.d ≠ b.d then
6  |  |  |  rep = rep ∪ report_data_diff(t_A, t_B, i, j)
7  |  |  end
8  |  |  i++, j++
9  |  else
10 |  |  rep = rep ∪ report_cf_diff(t_A, t_B, i, j)
11 |  |  (i, j) = find_merge_point(t_A, t_B, i, j)
12 |  end
13 end
14 return rep
```

Our trace comparison is given in Algorithm 8.1. Whenever *ip* values match, but data addresses (d) do not, a data difference is detected (lines 4–6). After reporting it, both traces are advanced to the next instruction (line 8). Control-flow differences occur when *ip*s start to differ (line 9–11). Differences are reported using report_data_diff and report_cf_diff using the format specified in Section 8.2.3. After a control-flow difference is reported, both traces need to be re-aligned to the merge point (line 11).

Trace Alignment. For control-flow differences, it is crucial to determine the correct merge point. Knowledge of the merge point allows the algorithm to continue sequential scanning and to detect further leaks until the end of one or both traces is reached.

In principle, merge points correspond to post-dominators of the diverged branches in the control-flow graph (CFG) [Geo+04]. We recover merge points using a simple set intersection approach, as shown in Algorithm 8.2. Starting from the branch point, the algorithm sequentially scans both traces and extends two sets S_A and S_B (lines 7–8) with the scanned instructions. If their intersection M becomes non-empty (lines 9–10), M holds the merge point's *ip*. We then determine the first occurrence of M in both branches using find (lines 11–12) and report them back as merge point to our diff algorithm. Eventually, both traces are aligned to the merge point in Algorithm 8.1 line 11, and the difference search is continued.

Context-Sensitivity. Since control-flow leaks could incorporate additional function calls (cf. function mul in Listing 8.2), we need to exclude those from the merge point search. If we do not, we could erroneously identify the merge point to be in the sub-function (e.g., line 7 of Listing 8.2).

Algorithm 8.2: find_merge_point

input : t_A, t_B ... the two traces
input : i, j ... the trace indices of the branches
output : k, l ... the indices of the merge point

```
1  k = i, l = j, C_A = 0, C_B = 0, S_A = ∅, S_B = ∅
2  while k < |t_A| ∧ l < |t_B| do
3  |  if isCall(t_A[k]) then C_A++ ;
4  |  if isRet(t_A[k]) then C_A-- ;
5  |  if isCall(t_B[l]) then C_B++ ;
6  |  if isRet(t_B[l]) then C_B-- ;
7  |  if C_A <= 0 then S_A = S_A ∪ t_A[k].ip ;
8  |  if C_B <= 0 then S_B = S_B ∪ t_B[l].ip ;
9  |  M = S_A ∩ S_B
10 |  if M ≠ ∅ then
11 |  |  k = find(t_A[i...k], M)
12 |  |  l = find(t_B[j...l], M)
13 |  |  return (k, l)
14 |  end
15 |  if C_A >= 0 then k++ ;
16 |  if C_B >= 0 then l++ ;
17 end
18 error No merge point found
```

Therefore, we maintain the current calling depth in counters C_A and C_B (Algorithm 8.2 lines 3–6). The functions isCall(a) and isRet(a) return `true` iff the assembler instruction at address $a.ip$ is a function call or return, respectively. A calling depth greater than zero occurs when inspecting instructions of sub-functions. Such instructions are ignored in lines 7–8. If the calling depth drops below zero, the trace returned to the function's call-site. We stop scanning this trace (lines 15–17) and wait for the other trace to hit a merge point. As an example, consider the control-flow leak in Listing 8.3 line 2. Depending on the value of `key`, the function either exits at line 2 or line 3. Thus, the correct merge point is line 0 at the call-site.

Applicability. The presented trace alignment method is designed for applications following the call-return paradigm introduced with procedural pro-

```
0 program entry: // call process with user-input key
1 int process(int key) {
2   if (key == 0) return 256;
3   else return key;
  }
```

Listing 8.3: Merge point is at call-site.

gramming. Thus, our context-sensitive alignment also works for techniques like retpoline [Tur18] that aim to prevent Spectre attacks, since they just add additional call/ret layers. Code directly manipulating the stack pointer (return stack refill [Tur18], setjmp/longjmp, exceptions, etc.) could be supported by detecting such stack pointer manipulations alongside calls and returns.

Comparison to Related Work. Trace alignment has been studied before as the problem of correspondence between different execution points. Several approaches for identifying execution points exist [SZ13]. Instruction-counter-based approaches [ML89] uniquely identify points in one execution but fail to establish a correspondence between different executions. Using calling contexts as a correspondence metric could introduce temporal ambiguity in distinguishing loop iterations [Sum+10]. Xin et al. [XSZ08] formalize the problem of relating execution points across different executions as execution indexing (EI). They propose structural EI (SEI), which uses taken program paths for indexing but could lose comprehensiveness by mismatching execution points that should correspond [SZ13]. Other approaches combine call stacks with loop counting to avoid problems of ambiguity and comprehensiveness [SZ13]. Many demand recompilation [XSZ08; Sum+10; SZ13], which prohibits their usage in our setting. Specifically, EI requires knowledge of post-dominators, typically extracted from control flow graphs (CFGs) [Geo+04], which are not necessarily available (e.g., obfuscated binaries or dynamic code generation). Using EI, Johnson et al. [Joh+11] align traces in order to propagate differences back to their originating input.

We use a similar intuition as Johnson et al. in processing and aligning traces in a single pass, however, without the need to make program execution indices explicit. By constantly re-aligning traces, we inherently maintain the correspondence of execution points. Our set-based approach does not require CFG or post-dominator information.

In contrast to EI, we do not explicitly recover loops. This could cause imprecision when merging control-flow leaks embedded within loops. If the two branches are significantly asymmetric in length, we might match multiple shorter loop iterations against one longer iteration. This case introduces an artificial control-flow leak (false positive) when one branch leaves the loop while the other does not. Should such leaks occur, they would be dismissed as key independent during the statistical phases of DATA. Note that correspondence (*i.e.*, correct alignment) would be automatically restored as soon as both branches leave the loop. Also, this is not a fundamental limitation of DATA, as other trace alignment methods could be implemented as well.

8.3 Implementation and Optimizations

DATA offers a convenient command-line interface to invoke all analysis steps shown in Figure 8.1. This includes key generation, trace recording and trace diffing. To speed up analysis, DATA supports parallel trace recording as well as parallel trace diffing. To test a given program, the analyst only needs to provide a bash script that generates keys in an appropriate format and invokes

the program with the generated keys. If the program is compiled with debug symbols, the final report will in addition show the symbol names, alongside the memory addresses.

Trace Recording. The concept of DATA is platform independent. For our prototype we chose to implement trace recording on top of the Intel Pin framework [Int] for analyzing x86 binaries. We developed our own Pintool to record address traces in separate trace files. To reduce their size, we only monitor instructions with branching behavior and their target branch as well as instructions performing memory operations. In particular, pure register instructions are not recorded. This suffices to detect control-flow and data leakage. Furthermore, we store traces in a custom binary format that can be directly accessed as C++ vector. This reduces not only the trace size but also avoids conversion costs. Our Pintool also offers an option to specify at which function call recording shall start.

Trace Diffing. We implement the difference detection in Python. To speed up analysis, we use a fast-forwarding method that compares large chunks of trace data at once. As soon as differences occur, we fall back to the instruction-granular `diff` algorithm explained before. In the end, our Python script condenses all findings into a human-readable leakage report in XML format. This report structures information leaks by libraries, functions, and call stacks.

Tracking Heap Allocations. Data leakage of heap objects demands special treatment. Depending on the utilization of the heap, identical memory objects could get assigned different addresses by the memory allocator, especially if previous heap allocations vary in size. While these previous variable-sized heap allocations might constitute actual data leakage, all subsequent heap allocations would also be shifted in memory and cause address differences. During trace analysis, one could misinterpret these subsequent heap objects as different ones, while, in fact, they just expose the same previous data leakage. We encountered such behavior for OpenSSL, which dynamically allocates big numbers on the heap and resizes them on demand. This causes frequent re-allocations and big numbers hopping between different heap addresses for different program executions. Our Pintool can, therefore, be configured to detect heap objects and replace their virtual address with its relative address offset. Our current heap tracking refrains from labeling different heap objects. While this gives us more readable results, we might miss data leakage in which different memory objects are accessed, depending on the secret. More elaborate approaches such as memory indexing [SZ10] are left as future work.

8.4 Evaluation and Results

For our evaluation we used Pin version 3.2-81205 for instrumentation and compiled glibc 2.24 as well as OpenSSL 1.1.0f[2] in a default configuration with additional debug information, using GCC version 6.3.0. Although DATA does not require

[2]Specifically, we tested commit `7477c83e15`.

Table 8.1: Leakage summary of algorithms.

	Algorithm	Differences
OpenSSL	AES-NI	0 (2)
	AES-VP	0
	AES bitsliced	4
	AES T-table	20
	Blowfish	194
	Camellia	82
	CAST	202
	DES	138
	Triple DES	410
	ECDSA (secp256k1)	515
	DSA	781
	RSA	2248
PyCrypto	AES	96
	ARC4	5
	Blowfish	384
	CAST	284
	Triple DES	108

debug symbols, it incorporates them in the final report, if available. Debug symbols help mapping detected leaks back to functions and data symbols.

8.4.1 Analysis Results

Table 8.1 shows the address differences reported by DATA. For symmetric algorithms in OpenSSL, we recorded up to 10 traces in the difference detection phase. We found that three traces are sufficient as more traces did not uncover additional differences. The low number of traces results from the high diffusion and the regular design of symmetric ciphers, which yields a high probability for quickly hitting all variations in the program execution. This suggests that the difference detection phase achieves good accuracy for symmetric ciphers. Symmetric ciphers are typically deterministic. Thus, all differences are key-dependent.

Non-deterministic algorithms, often included in asymmetric ciphers, also show address differences that are independent of the secret key. This explains the high number of differences, especially for RSA in Table 8.1. In [Wei+18a], we present statistical phases to automate the process of distinguishing noise from actual secret-dependent leaks. For this chapter, we manually separated non-deterministic differences from key-dependent leaks by analyzing the call stack in the reports. If a function does not take a secret input, we consider any address differences it shows as noise. In [Wei+18a], we further argue that in the case of OpenSSL, ten traces should suffice to analyze asymmetric primitives.

We analyzed the reported differences and discovered numerous known as well as unknown side-channel vulnerabilities. In the following, we present our analysis results for OpenSSL symmetric and asymmetric primitives as well as for PyCrypto.

OpenSSL (Symmetric Primitives). To analyze AES, we implemented a wrapper that calls the algorithm directly. For other algorithms, we used the `openssl enc` command-line tool with keys in hex format.

Our analysis shows that AES-NI (AES new instructions [Gue10]) as well as AES-VP (vector permutations based on SSSE3 extensions) do not leak. However, when using AES-NI (and other ciphers) via the OpenSSL command-line tool, the key parsing yields two data leaks, as indicated in brackets. In particular, the leaks occur in function `set_hex`, which uses stdlib's `isxdigit` function that performs leaky table lookups. Besides, `OPENSSL_hexchar2int` uses a leaky switch-case statement to convert key characters to integers. One should be aware of such leaks and favor storing symmetric keys in binary format only.

Besides, we identified four data leaks in the bitsliced AES. While OpenSSL uses the protected implementation by Käspar and Schwabe [KS09] for the actual encryption, they use the same unprotected key expansion as used in T-table implementations. In particular, the bitsliced AES implementation uses the vulnerable `_x86_64_AES_set_encrypt_key` function for the key schedule.

All other tested symmetric implementations yield a significant number of data leaks since they rely on lookup tables with key-dependent memory accesses, which makes them vulnerable to cache attacks [TOS10; Ber05]. The unprotected AES leaks in function `_x86_64_AES_encrypt_compact`. Blowfish leaks at `BF_encrypt`, Camellia leaks vie the `LCamellia_SBOX` at `Camellia_Ekeygen` and `_x86_64_Camellia_encrypt`, CAST leaks via the `CAST_S_table0` to `7` at `CAST_set_key` as well as `CAST_encrypt`, DES leaks via the `des_skb` lookup table at `DES_set_key_unchecked` as well as via `DES_SPtrans` at `DES_encrypt2`.

OpenSSL (Asymmetric Primitives). For the analysis of asymmetric ciphers, we use OpenSSL to generate keys in PEM format and then invoke the `openssl pkeyutl` command-line tool to create signatures with those keys.

Similar to symmetric ciphers, asymmetric implementations leak during key loading and parsing. We found leaks in `EVP_DecodeUpdate`, in `EVP_DecodeBlock` via lookup table `data_ascii2bin`, in `c2i_ASN1_INTEGER` that uses `c2i_ibuf` and in `BN_bin2bn`. Although the key is typically loaded only once at program startup, this has direct implications on applications using Intel SGX SSL. Controlled-channel attacks (cf. Chapter 7) or interrupt-driven attacks [BPS17] could be used to exploit the key initialization steps.

The asymmetric primitives show significant non-deterministic behavior, which would be dismissed in the statistical phases (see [Wei+18a]). For example, OpenSSL uses RSA base blinding with a random blinding value. By analyzing the leaks, we found two constant-time vulnerabilities in RSA and DSA, respectively, which bypass constant-time implementations in favor of vulnerable implementations. This could allow key recovery attacks similar to [WSB18a; Ald+19]. The first vulnerability leaks during the initialization of Montgomery

```
1 int BN_MONT_CTX_set(BN_MONT_CTX *mont, BIGNUM *mod, BN_CTX *ctx)
2 {
3   ...
4   BN_copy(&(mont->N), mod);
5   ...
6   BN_mod_inverse(Ri, R, &mont->N, ctx);
7   ...
8 }
```

Listing 8.4: OpenSSL RSA vulnerability.

constants for secret RSA primes `p` and `q`. This is a programming bug: the so-called constant-time flag is set for `p` and `q` in function `rsa_ossl_mod_exp` but not propagated to temporary working copies inside `BN_MONT_CTX_set`, as shown in Listing 8.4, since the function `BN_copy` in line 4 does not propagate the consttime-flag from `mod` to `mont->N`. This causes the inversion in line 6 to fall back to non-constant-time implementations (`int_bn_mod_inverse` and `BN_div`). The second vulnerability is a missing constant-time flag for the DSA private key inside `dsa_priv_decode`. This causes the DSA key loading to use the unprotected exponentiation function `BN_mod_exp_mont`.

Moreover, DATA reconfirms address differences in the ECDSA wNAF implementation. ECDSA still uses the vulnerable point multiplication in `ec_wNAF_mul`, which was exploited in [Ben+14; PSY15; FWC16]. Finally, we found that the majority of information leaks reported for OpenSSL are leaking the length of the key or intermediate variables. For example, we reconfirm the leak in `BN_num_bits_word` [Wan+17a], which leaks the number of bits of the upper word of big numbers. There are several examples where the key length in bytes is leaked, e.g., via `ASN1_STRING_set`, `BN_bin2bn`, `strlen` of glibc as well as via heap allocation.

Python. We tested PyCrypto 2.6.1 running on CPython 2.7.13. We wrote a wrapper to invoke PyCrypto with randomly generated keys and excluded the wrapper from trace recording to remove leakage stemming from key parsing.

PyCrypto incorporates native shared libraries for most cryptographic operations. From a side-channel perspective, this is desirable since those native libraries could be tightened against side-channel attacks, independently of the used interpreter. However, we found that all ciphers leak key bytes via unprotected lookup table implementations within those shared libraries, as indicated by the byte leakage model. In particular, AES leaks the tables `Te0` to `Te4` and `Td0` to `Td3` in functions `ALGnew`, `rijndaelKeySetupEnc` and `rijndaelEncrypt`. Blowfish leaks in functions `ALGnew` and `Blowfish_encrypt`. CAST leaks the tables `S1` to `S4` in function `block_encrypt` and the tables `S5` to `S8` in `schedulekeys_half`. Triple DES leaks the table `des_ip` in function `desfunc` as well as `deskey`. ARC4 leaks in function `ALGnew`.

Leakage-free Crypto. As a sanity check, we applied DATA to ciphers that were designed to be constant time. We analyzed Curve25519 in NaCl [BLS] as well as the corresponding Diffie-Hellman variant of OpenSSL (X25519). DATA

Table 8.2: Performance of DATA during the analysis of OpenSSL (top) and PyCrypto (bottom). Sizes are per trace. Time is in CPU minutes.

	Algorithm	Traces	Size (MB)	Time (min.)
OpenSSL	AES-NI	3	0.5	0.1
	AES-VP	3	0.5	0.1
	AES bitsliced	3	0.5	0.4
	AES T-table	3	0.5	0.4
	Blowfish	3	28.2	0.8
	Camellia	3	27.3	0.6
	CAST	3	27.3	0.6
	DES	3	27.3	0.6
	Triple DES	3	27.3	0.7
	ECDSA (secp256k1)	10	54.1	79.8
	DSA	10	35.6	29.8
	RSA	10	44.2	55.0
PyCrypto	AES	3	1081.6	4.0
	ARC4	3	1081.5	3.9
	Blowfish	3	1082.3	5.0
	CAST	3	1081.6	4.0
	Triple DES	3	1082.4	4.2

found no address-based information leakage (apart from OpenSSL's key parsing), approving their side-channel security.

8.4.2 Performance

We ran our experiments on a Xeon E5-2630v3 with 386 GB RAM. DATA achieves good performance, adapting its runtime to the number of discovered leaks, as summarized in Table 8.2. Unless stated otherwise, all timings reflect the runtime in CPU minutes (single-core) and thus represent a fair and conservative metric. If tasks are parallelized, the actual runtime can be significantly reduced.

For OpenSSL symmetric ciphers, DATA completes analysis in less than a minute. Analysis of the leakage-free AES-NI and AES-VP only took around 6 seconds. OpenSSL's asymmetric ciphers need between 29.8 CPU minutes for DSA and 79.8 CPU minutes for ECDSA, for two reasons. First, they require more traces to be recorded and analyzed. As we compare traces pairwise, the runtime of trace diffing grows quadratically in the number of traces. Second, asymmetric ciphers yield significantly more differences that need to be analyzed. Especially control-flow differences demand costly re-alignment of traces. Nevertheless, these results are quite encouraging, especially since the automated analysis of large real-world software stacks is out of reach for many existing tools. By exploiting parallelism, the actual execution time can be significantly reduced, e.g., from 55 CPU minutes to approximately 250 s for RSA on our test machine.

For OpenSSL, the trace size is $< 30\,\text{MB}$ for symmetric and $< 55\,\text{MB}$ for asymmetric ciphers. For PyCrypto, each trace has approximately 1 GB, because the execution of the interpreter is included. Despite the large trace sizes, DATA finishes analysis of PyCrypto ciphers in 5 minutes or less.

8.4.3 Discussion

The adoption of side-channel countermeasures is often partial, error-prone, and non-transparent in practice. Even though countermeasures have been known for over a decade [RSD06], most OpenSSL symmetric ciphers, as well as PyCrypto, do not rely on protected implementations such as bitslicing. Also, the bitsliced AES adopted by OpenSSL leaks during the key schedule. It was integrated only partially [KS09] since practical attacks have not been shown yet. Moreover, we discovered two new vulnerabilities, bypassing OpenSSL's constant-time implementations for RSA and DSA initialization. Considering incomplete bug fixes of similar vulnerabilities identified by Garcia et al. [GBY16; GB17], this sums up to four implementation bugs related to the same countermeasure. This clearly shows that the tedious and error-prone task of implementing countermeasures should be backed by appropriate tools such as DATA to detect and appropriately fix vulnerabilities as early as possible.

We found issues in loading and parsing cryptographic keys as well as initialization routines. Finding these issues demands analysis of the full program execution, from the program start to exit, which is out of reach for many existing tools. Also, analysis often neglects these information leaks because an attacker typically has no way to trigger key loading and other single events in practice. However, when using OpenSSL inside SGX enclaves (cf. Intel's SGX SSL library [Int19] and Chapter 7), the attacker can trigger arbitrarily many program executions, making single-event leakage practically relevant.

Responsible Disclosure. We informed the library developers as well as Intel of our findings. We furthermore provided patches for the critical constant-time vulnerabilities, which were merged by the OpenSSL team.[3]

Outlook. The generic design of DATA also allows detecting other types of leakage, such as variable-time floating point instructions, by including the instruction operands in the recorded address traces. DATA also paves the way for analyzing other interpreted languages and quantifying the effects of interpretation and just-in-time compilation on side-channel security. Moreover, DATA could be extended to analyze multi-threaded programs by recording and analyzing individual traces per execution thread.

[3] See commits `3de81a59`, `9f944291` and `6364475a` in `https://github.com/openssl/openssl.git`.

8.5 Summary

In this work, we proposed differential address trace analysis (DATA), a methodology to identify address-based information leaks underlying most software-based side-channel attacks. Our practical implementation of DATA is efficient enough to analyze real-world software – from program start to exit. Thereby, we include key loading and parsing in the analysis and found leakage, which has been missed before. Based on DATA, we confirmed existing and identified several unknown information leaks as well as already (supposedly) fixed vulnerabilities in OpenSSL. In addition, we showed that DATA is capable of analyzing interpreted code (PyCrypto), including the underlying interpreter, which is conceptually impossible with current static methods. This shows the practical relevance of DATA in assisting security analysts in identifying information leaks as well as developers in the tedious task of correctly implementing countermeasures. We open-source DATA in order to encourage analysis of real-world cryptographic libraries.

✳ ✳ ✳

9

Big Numbers – Big Troubles. On Nonce Leakage in (EC)DSA

For whatever is hidden is meant to be disclosed, and whatever is concealed is meant to be brought out into the open.

Jesus Christ – Gospel of Mark

Digital signatures are an essential building block for encrypted communication channels, e.g., via Transport Layer Security (TLS) and the underlying public-key infrastructures, SSH, as well as for cryptocurrencies. The extensive and ubiquitous usage of digital signature schemes demands good security arguments, not only from a cryptanalytic perspective but also regarding their implementation, as a single implementation vulnerability can completely break the scheme [BH19].

Most digital signature schemes used today are susceptible to attacks on their so-called nonces [NS00]. Even partial knowledge of nonces leads to full recovery of private keys, thus allowing an attacker to issue fake signatures, impersonate users, intercept communication channels, steal money, etc. In light of these threats, digital signature implementations need extensive hardening against nonce leakage. While biased random number generation [BH19] is a common implementation pitfall, also side channels [BH09] have been proven a powerful way of leaking nonce bits. Especially side-channel attacks constantly improve along several axes. This includes advanced side-channel observation methods, a reduction of required knowledge, faster key recovery attacks, and, most importantly, the continued discovery and disclosure of new side-channel leakage that might have been hidden for years.

Modern cryptographic libraries already explicitly address nonce leakage by relying on constant-time code execution. Unfortunately, efforts to make implementations side-channel resistant are not being evaluated thoroughly enough, leading to a continuous cycle of vulnerability disclosure and patching. To break this cycle, a more systematic approach for nonce leakage analysis is required. However, this seems to be a challenging endeavor for the following reasons:

1. Although side-channel evaluation techniques are actively researched, complex code bases such as OpenSSL are hard to evaluate.
2. Popular libraries use randomization, e.g., blinding, to avoid leakage in vulnerable non-constant-time code. However, analyzing blinded computations for side channels is non-trivial; and insufficient blinding is exploitable.
3. Cryptographic libraries use non-constant-time code when computing on public data. Although legitimate, this puts an additional burden on code analysis to avoid false positives.
4. While tool support for side-channel analysis is growing, existing tools do not address nonce leakage.

In this chapter, we address these challenges by extending the DATA framework introduced in Chapter 8 and [Wei+18b]. In particular, we adapt DATA to recognize nonces as additional secrets in a backward manner and develop leakage models tailored for detecting nonce leakage. These leakage models operate on DATA's statistical leakage detection phase described in [Wei+18b]. We also develop a graphical user interface for visualizing leakage results. Our GUI helped us to systematically analyze three popular cryptographic libraries for (EC)DSA nonce leakage, namely OpenSSL, LibreSSL, and BoringSSL.

We systematically study the whole lifetime of a nonce, *i.e.*, from its generation to its final use. Rather than proving code secure–which would typically require formal models and static analysis approaches–we focus on finding actual side-channel vulnerabilities. In fact, we uncovered numerous unknown vulnerabilities leaking nonce bits, and thereby highlight a fundamental problem in the Bignumber representation in OpenSSL and LibreSSL. In particular, if the nonce is close to a machine word boundary, the Bignumber implementations possibly leak whether the nonce crosses this boundary in either direction. We found that lazy resize operations involving the nonce leak several nonce bits via Flush+Reload [YF14], which has been acknowledged and documented under CVE-2018-0734 and CVE-2018-0735. Surprisingly, this leakage occurs due to a side-channel defense mechanism. We also found that small nonces can leak nine nonce bits at once for the secp521r1 curve. The Bignumber implementation of BoringSSL[1] prevents size-related Bignumber issues by design. Yet, we found a tiny but expressive leak in the constant-time scalar multiplication of BoringSSL and OpenSSL. During responsible disclosure, we identified a flaw in the OpenSSL patches that would have downgraded exponentiation to a vulnerable implementation (cf. [GBY16]). We report residual leakage in the patched OpenSSL version, which we exploit via controlled-channel attacks [XCP15] for full key recovery.

[1] See https://github.com/openssl/openssl/issues/6640.

Due to our findings, the OpenSSL team decided to rework Bignumber arithmetic, similar to BoringSSL [Dal19].

This chapter provides a snapshot of the current situation of nonce leakage in popular cryptographic libraries. With the help of our GUI we analyzed known and unknown vulnerabilities and document their potential damage, exploitability, and patching state. We open-source both our tool and the GUI to facilitate reproducibility and future side-channel analysis.[2]

Contributions. Our contributions are as follows:

- We expand The DATA side-channel analysis framework for automated nonce leakage detection, and present results in an intuitive GUI.
- We systematically analyze nonce leakage in three popular crypto libraries: OpenSSL, LibreSSL, and BoringSSL.
- We uncover and document several unknown side-channel vulnerabilities resulting from fundamental flaws in the Bignumbers representation of OpenSSL and LibreSSL, among others.
- We responsibly disclosed vulnerabilities, proposed fixes, and document residual leakage that remains unfixed.

This chapter is based on the publication [Wei+20], of which I am the main author, while Lukas Bodner contributed most of the GUI design. The remainder of this chapter is structured as follows: Section 9.1 gives background information. Section 9.2 presents our automated side-channel analysis tool. Section 9.3 outlines analysis results and Section 9.4 discusses the vulnerabilities in detail. Section 9.5 presents an actual key recovery attack on one of the vulnerabilities. Section 9.6 evaluates our leakage models. We discuss the implications of our work in Section 9.7 and summarize in Section 9.8.

9.1 Background

9.1.1 Digital Signatures

Digital signature schemes consist of three algorithms, namely KeyGen, Sign, and Verify. KeyGen generates a long-term public/private key pair from a given set of public parameters $param$; see Equation (9.1). Using the private key prv, one can sign arbitrary messages M, which gives a digital signature S, as seen in Equation (9.2). Finally, the signature S can be verified against the original message M and the public key pub, as seen in Equation (9.3).

$$(pub, prv) \leftarrow KeyGen(param) \tag{9.1}$$

$$S \leftarrow Sign(M, prv, param) \tag{9.2}$$

$$\top or \bot \leftarrow Verify(M, S, pub, param) \tag{9.3}$$

Digital signatures can be built from a trapdoor, a mathematical problem that can only be solved efficiently in one direction unless one knows additional

[2] Our tool and the GUI is available under https://github.com/Fraunhofer-AISEC/DATA and https://github.com/IAIK/data-gui

information. An example of such a trapdoor is the RSA problem, with two instantiations, namely RSA-PKCS#1.5 [Kal98] and RSA-PSS [BR98]. However, computing these trapdoors is costly.

Alternatively, one can build signature schemes on top of the discrete logarithm problem. One example is the ElGamal [Gam84] signature scheme, dating back to 1984. In 1986, Fiat and Shamir proposed a generic heuristic to transform an interactive proof-of-knowledge to a (non-interactive) signature scheme [FS86]. Applying this transformation to Schnorr's identification scheme yields efficient Schnorr signatures [Sch89]. Unfortunately, the Schnorr signatures did not find wide adoption due to patent issues [Cod98]. Instead, the Digital Signature Algorithm (DSA) [Kra91] was standardized as Digital Signature Standard (DSS) [KG13].

DSA. The Digital Signature Algorithm (DSA) [KG13] is based on prime fields. It relies on two primes p and q, where q divides $p-1$. Parameter g serves as generator over p such that $g^q \equiv 1 \mod p$. Keys are generated as follows:

$$x \xleftarrow{R} [1, q-1] \tag{9.4}$$
$$y \leftarrow g^x \mod p \tag{9.5}$$

The private key x is sampled uniformly from $[1, q-1]$. The public key y is obtained by Equation (9.5). Due to the hardness of the discrete logarithm problem, the private key x cannot be efficiently recovered from public knowledge of (y, g, p). To sign a message M, DSA first compresses M using a hash function H, which yields the message digest m. Next, DSA generates a random secret nonce k (Equation (9.6)) and computes r by modular exponentiation (Equation (9.7)). It then inverts the nonce in Equation (9.8) and multiplies it to derive s, as shown in Equation (9.9). The final signature is formed by the tuple (r, s).

$$k \xleftarrow{R} [1, q-1] \tag{9.6}$$
$$r \leftarrow g^k \mod q \tag{9.7}$$
$$kinv \leftarrow k^{-1} \mod q \tag{9.8}$$
$$s \leftarrow kinv \cdot (m + xr) \mod q \tag{9.9}$$

Verification of a signature is done by recomputing r' from public information and matching it against r.

$$u_1 \leftarrow m \cdot s^{-1} \mod q \tag{9.10}$$
$$u_2 \leftarrow r \cdot s^{-1} \mod q \tag{9.11}$$
$$r' \leftarrow g^{u_1} y^{u_2} \mod q \tag{9.12}$$

Observe that inverting Equation (9.9) yields Equation (9.13). Now, one can expand Equation (9.7) into Equation (9.14).

$$k \equiv (m + xr) \cdot s^{-1} \mod q \tag{9.13}$$
$$r \equiv g^k \equiv g^{m \cdot s^{-1}} g^{x \cdot r \cdot s^{-1}} \equiv g^{u_1} y^{u_2} \equiv r' \mod q \tag{9.14}$$

Other DSA Constructions. Several variants of DSA exist. Schnorr signatures [Sch89] omit the inversion step in Equation (9.8). Deterministic schemes such as [Por13] and EdDSA [JL17] derive unique nonces from the message input instead of using random numbers in Equation (9.6). ECDSA [KG13] is one of the most widely used signature algorithms nowadays. It computes r in Equation (9.7) via scalar multiplication over an elliptic curve generator G, as follows:

$$r = k \cdot G \tag{9.15}$$

Nonce Attacks. DSA-like cryptosystems strongly rely on the secrecy and the uniformity of the nonce k. For instance, if the same nonce is reused for two signatures (r, s) and (r, s'), full key recovery is trivially possible by subtracting s' from s and recovering the shared nonce k. It has been shown that even partial knowledge of the nonce suffices to break the scheme [NS00]. By collecting enough "leaky" signatures, one can formulate a so-called Hidden Number Problem (HNP) [BV96] and recover the private key with lattice or Bleichenbacher attacks. This knowledge about nonces can be obtained by weak nonce generation algorithms [BGM97] or side channels [BH09]. Thus, an implementation needs to adequately address both unpredictability of nonces and side-channel resistance and protect nonces throughout their whole lifetime (cf. Equations (9.6) to (9.9)).

9.1.2 The Hidden Number Problem

Nonce leakage can be encoded as a Hidden Number Problem (HNP). Solving the HNP via lattice attacks or more generic Bleichenbacher attacks reveals the private key.

HNP. Following [Ben+14; Rya19], we denote $\lfloor \cdot \rfloor_q$ as the value modulo q and $|\cdot|_q$ as reducing the argument modulo q into the range $[-q/2, q/2]$ and then taking the absolute value. $MSB_{L,q}(k)$ denotes knowledge about the L most significant bits of k, *i.e.*, an integer u satisfying $|k - u|_q < q/2^{L+1}$.

The HNP [BV96; BV97] denotes the problem of finding a hidden number, given partial information about multiples of the hidden number. In particular, the HNP attempts to recover a hidden number $x \in [1, q-1]$, given knowledge of its multiples $t_1, ..., t_d \in \mathbb{F}_q$ for a known prime q as well as knowledge about $u_i = MSB_{L,q}(\lfloor t_i x \rfloor_q)$. This yields a system of d inequalities:

$$|\lfloor t_i x \rfloor_q - u_i|_q < q/2^{L_i+1} \text{ for all } i \in \{1, ..., d\} \tag{9.16}$$

(EC)DSA can be encoded as an instance of the HNP to recover the private key x from signatures (r, s) and known nonce bits $u = MSB_{L,q}(k)$. Using Equation (9.13) gives:

$$|k - u|_q < q/2^{L+1} \tag{9.17}$$

$$|\lfloor (m + xr) \cdot s^{-1} \rfloor_q - u|_q < q/2^{L+1} \tag{9.18}$$

$$|\lfloor \lfloor s^{-1} r \rfloor_q \cdot x \rfloor_q - \lfloor u - s^{-1} m \rfloor_q|_q < q/2^{L+1} \tag{9.19}$$

Applying Equation (9.19) to d signatures (r_i, s_i) and nonce bits u_i yields a hidden number problem

$$|\lfloor t_i x \rfloor_q - v_i|_q < q/2^{L_i+1} \text{ for all } i \in \{1, ..., d\} \tag{9.20}$$

with $t_i = \lfloor s_i^{-1} r_i \rfloor_q$ and $v_i = \lfloor u_i - s_i^{-1} m_i \rfloor_q$. The HNP can also be applied when leaking inverse nonces, least significant nonce bits, or a block of contiguous [HS01] or non-contiguous bits [HR06].

Lattice. Boneh et al. [BV96] mapped the HNP to a Closest Vector Problem (CVP), for which efficient algorithms are available. Let $\mathbf{t} = (t_1, ..., t_d, 1)$ and $\mathbf{t}x = (t_1 x, ..., t_d x, x)$. According to the HNP, $\lfloor \mathbf{t}x \rfloor_q$ will be a close vector to $\mathbf{u} = (u_1, ..., u_d, 0)$ with a distance smaller than $q/2^{L_i+1}$ for the first d components, *i.e.*, $\lfloor \mathbf{t}x \rfloor_q - \mathbf{u}$ will be *small* multiples of q. By constructing a lattice basis from $\mathbf{t}$ and solving the CVP, the closest vector reveals the private key x. Boneh et al. solved the CVP by using LLL [LLL82] lattice reduction and Babai's nearest plane algorithm [Bab86] to recover Diffie-Hellman keys. Instead of using Babai, it is also possible to embed the CVP into a Shortest Vector Problem (SVP) and solve it directly via lattice reduction [NS00; FGR12; Won15]. The idea is to include a scaled version of $\mathbf{u}$ in the lattice basis. In particular, the first d components of $\mathbf{t}$ and $\mathbf{u}$ are scaled by 2^{L_i+1}. Following [Ben+14], this gives a $d+2$-dimensional row-wise lattice basis $\mathbf{B}$ in Equation (9.21). With $\mathbf{x} = (\lambda_1, \ldots, \lambda_d, x, 1)$ one can write $\mathbf{x} \cdot \mathbf{B} = \mathbf{y}$. Reducing this basis yields the shortest vector $\mathbf{y}$, which holds the private key x in its second last entry, see Equation (9.22).

$$\mathbf{B} = \begin{pmatrix} 2^{L_1+1}q & & & 0 & 0 \\ & \ddots & & \vdots & \vdots \\ & & 2^{L_d+1}q & 0 & 0 \\ 2^{L_1+1}t_1 & \dots & 2^{L_d+1}t_d & 1 & 0 \\ -2^{L_1+1}u_1 & \dots & -2^{L_d+1}u_d & 0 & q \end{pmatrix} \tag{9.21}$$

$$\mathbf{y} = \left(2^{L_1+1}(t_1 x - u_1 + \lambda_1 q), \ldots, 2^{L_d+1}(t_d x - u_d + \lambda_d q), x, q\right) \tag{9.22}$$

Boneh et al. [BV96] showed that lattice reduction requires at least $L = \log_2 \log_2 q$ bit leakage. Howegrave-Graham and Smart [HS01] recovered the private key for 160-bit DSA given 30 signatures and knowledge of 8 bits for each nonce. Naccache et al. [Nac+05] only required 27 signatures for the same leakage using the block Korkin-Zolotarev (BKZ) algorithm. Given 200 signatures and two shared LSBs of the nonce, Faugère et al. [FGR12] recovered the private key using a lattice attack. Besides, they recovered the private key with a probability of 90% with just a single shared LSB and 400 signatures.

Bleichenbacher proposed an FFT-based attack using exponential sums to detect influences of small biases [Ble00]. This requires more samples than lattice attacks, but is noise-tolerant and works even with fractional bit leaks [Mul+13; Mul+14]. Aranha et al. [Ara+14] exploited a single-bit bias for 160-bit ECDSA using 2^{33} signatures. De Mulder et al. [Mul+13] used a BKZ-based method to exploit 5-bit leakage of 384-bit ECDSA using 4000 signatures.

9.2 Automated Nonce Leakage Detection

Tool support is essential for effective and accurate side-channel analysis. In particular, a high degree of automation and a proper representation of results is imperative for productive analysis. However, none of the tools discussed in Section 6.3.3 were designed or used to detect address-based leakage of (EC)DSA nonces. In this section, we present our methodology for detecting nonce leakage in a fully automated way. In particular, we extended the automated side-channel analysis tool DATA to also identify nonce leakage. We furthermore developed an intuitive GUI for visualizing leakage results.

9.2.1 Methodology

Our analysis is based on Differential Address Trace Analysis (DATA), as presented in Chapter 8. As such, it inherits the threat model and limitations of DATA. In short, DATA detects address leakage on byte granularity to cover not only attacks on memory pages [XCP15], cache lines [YF14] and cache banks [YGH16] but even single-byte addresses [BPS17; Mog+20].

The first phase of DATA – the *difference detection* – identifies address differences, which indicate potential leaks. However, analyzing randomized (e.g., blinded) algorithms yields various address differences that do not leak secret information but represent a form of non-determinism. Also, many differences stem from public input and are also uncritical. To filter these false positives, DATA employs statistical tests in phase two and three, as presented in [Wei+18a]. The second phase – the *leakage detection* – tests if the found address differences depend on the private key. It does so by comparing traces generated from a fixed key with traces generated from varying keys. This fixed-vs-random testing requires control over the secret variable. Since nonces are not controllable from the outside but generated randomly (internally), this phase cannot be used for detecting nonce leakage. The third phase – the *leakage classification* – classifies information leakage based on a pre-defined leakage model. Leakage models correlate the observed leakage (*i.e.*, the address traces) with the secret. In this chapter, we define leakage model suitable to detect nonce leakage. However, a high correlation does not necessarily imply actual leakage but could also stem from public values (e.g., the modulus). This is a fundamental issue of statistical testing and implies that an analyst should always carefully review potential leakage reported by DATA in a semi-automated fashion.

To help analysts rule out potential false positives and assess actual exploitability of leaks, we spend quite some effort in visualizing leakage reports in a comprehensible and intuitive GUI. We found the GUI to be essential during our analysis, especially if the number of address differences reported by DATA is large. Also, constant-time code and side-channel patches can be easily tested for their efficacy, preventing the reintroduction of previously known leaks.

9.2.2 Detecting Nonce Leakage

We tweak the implementation of DATA to fit our use for nonce leakage detection, as follows: We bypass the second phase and make the third phase run independently, *i.e.*, without relying on phase two results. As mentioned, the leakage classification phase correlates leakage to a secret value via leakage models. However, secret nonces are generated internally and are not exposed to the outside. To overcome this limitation, we adapt DATA to recognize nonces as an additional secret in a backward manner. That is, we recover the nonce from the private key, the message, and the signature using Equation (9.13) and provide the recovered nonce to the third phase. Furthermore, the original DATA tool only provided multiprocessing for phase one but not for phase three. We significantly improve the performance of phase three by analyzing different leaks in a parallelized fashion.

Leakage Models. Definition of proper leakage models is essential for finding nonce leaks. This, however, demands knowledge of potential leaks to search for. Based on initial manual inspection of OpenSSL's source code, we developed leakage models tailored for detecting nonce leakage. This was no straightforward process but involved extending the leakage models the more issues we found. In particular, we searched for Bignumber issues by testing the bit length of the nonce k and its variants. We chose to test for the nonce k and its inverse k^{-1}, as they are part of the DSA specification (Equations (9.6) to (9.9)). Furthermore, we tested for $k+q$ and $k+2q$, which are computed as part of a special nonce padding scheme during the exponentiation step, which we denote as k-padding.

Our first leakage model for finding nonce leaks retrieves the bit length of the nonce or its derivatives (*i.e.*, the position of the highest non-zero bit). This leakage model is denoted as $num\_bits(\cdot)$ and finds leakage, e.g., due to lazy resizing of Bignumbers. Furthermore, we used the Hamming weight model denoted as $HW(\cdot)$ to search for leaks in DSA modular exponentiation (square-and-multiply) and ECDSA scalar multiplication (double-and-add), respectively. In total, we defined and tested eight different leakage models, as follows:

$$num\_bits(k) \tag{9.23}$$
$$num\_bits(k+q) \tag{9.24}$$
$$num\_bits(k+2q) \tag{9.25}$$
$$num\_bits(k^{-1}) \tag{9.26}$$
$$HW(k) \tag{9.27}$$
$$HW(k+q) \tag{9.28}$$
$$HW(k+2q) \tag{9.29}$$
$$HW(k^{-1}) \tag{9.30}$$

With these models, we were able to reduce the number of unrelated differences greatly. E.g., the leakage models typically filter well above 90% of the differences, focusing the analyst's attention to critical leaks.

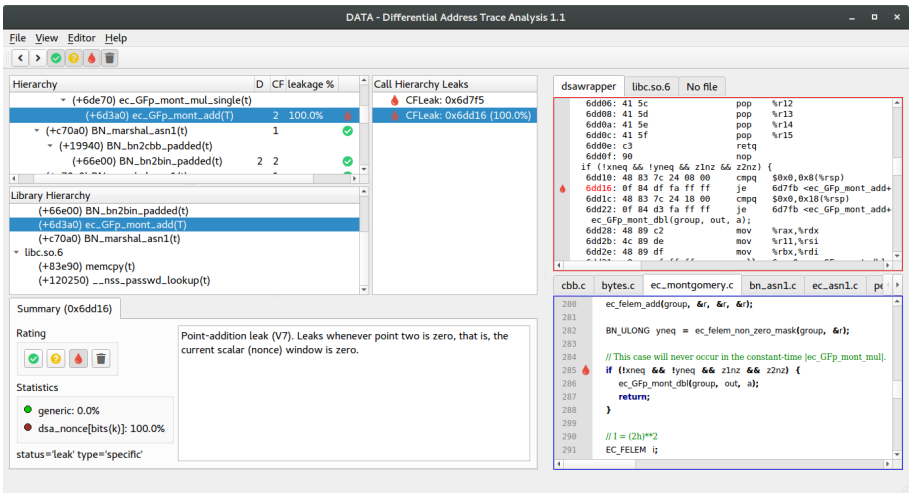

Figure 9.1: DATA GUI showing the point addition vulnerability (V7) in BoringSSL where the ECDSA scalar multiplication is leaking $num\_bits(k)$ with 100%.

9.2.3 DATA GUI

Since analyzing leakage reports of DATA is cumbersome, we developed a graphical user interface called DATA GUI. The DATA GUI allows to quickly navigate leakage reports, together with the source code and disassembly, and rate or comment potential leaks. As we need to test different cryptographic libraries under different configurations repeatedly, the DATA GUI was essential in mastering the amount of data we collected. To facilitate analysis and reporting, we extended DATA to generate an accompanying file archive that contains all necessary object files, disassemblies, and source code files alongside the regular leakage report. This approach decouples the test phases of DATA from GUI-aided analysis, which now may be done on a completely different computer.

Figure 9.1 depicts the DATA GUI, showing a discovered control-flow leak in BoringSSL. The GUI consists of several views: The left side sorts all leaks according to their call stack (top) and library (middle). Moreover, it shows for each function the number of data (D) and control-flow (CF) leaks as well as the maximum correlation with the leakage models in percent. One can see several other potential (false-positive) leaks which do not correlate with any of the predefined leakage-models. The center box gives a list of data and control-flow leaks for the selected function. The right side highlights leaks in the disassembly and the source code, if available, which is crucial for the analysis. The summary tab on the bottom left gives details about a particular leak, including correlations for various leakage models. Also, it allows the analyst to comment and rate leaks for documentation and reporting purposes. Clickable elements automatically synchronize the different views (*i.e.*, source code, assembler, and hierarchical views) in order to help quickly navigate through complex reports.

Table 9.1: Handling of secret nonces is either secure ○ or vulnerable ● to address-based side-channel attacks, according to our analysis.

Algorithm	Generate				Exp.		S. mul.		Invert		Mod.mul.		
Library	minimal rep.	rej. sampling	truncation	+ private key	k-padding	fixed window	k-pad + blind.	fixed window	Ext. Euclid	Little Fermat	Unprotected	Blinding	Const-time
OpenSSL	●		●	○	●	○	●	●	●			○	
LibreSSL	●	●			●	○	●		●		●		
BoringSSL		○				○		●		○			○

9.3 Vulnerability Analysis Overview

In this section, we present an overview of our analysis results for (EC)DSA nonce leakage in OpenSSL, LibreSSL, and BoringSSL. We include the whole life cycle of nonces in the analysis, *i.e.*, nonce generation in Equation (9.6), modular exponentiation for DSA in Equation (9.7) or scalar multiplication for ECDSA in Equation (9.15), modular inversion in Equation (9.8), and the final modular multiplication in Equation (9.9). Our findings are summarized in Table 9.1 and outlined in the following. As mentioned in Section 9.2, DATA cannot prove an implementation secure in a mathematical sense, and our analysis might have missed more side-channel vulnerabilities.

Nonce representation is based on Bignumbers. OpenSSL and LibreSSL minimize memory usage; *i.e.*, small numbers use fewer memory words than larger ones. This *minimal representation* of Bignumbers leaks the length of small nonces in several subsequent computation steps. BoringSSL, on the other hand, does not shrink sensitive Bignumbers, avoiding all Bignumber-related vulnerabilities we found by design.

Generation of nonces is done via rejection sampling in LibreSSL and BoringSSL, which gives uniformly distributed nonces. In contrast, OpenSSL truncates a large random number to the target nonce, introducing a negligible bias. Only OpenSSL includes the private key in the nonce generation to address potential weaknesses in random number generators.

DSA modular exponentiation itself did not reveal any leaks, as the fixed-window implementations are constant time. However, for OpenSSL and LibreSSL, we found several critical leaks due to padding the nonce *prior* to exponentiation. This enables easy-to-mount cache attacks, leading to full key recovery. Although the patched OpenSSL version closes the cache-attack vulnerability, it is still vulnerable to more sophisticated attacks, which we demonstrate in Section 9.5.

ECDSA scalar multiplication leaks in OpenSSL and LibreSSL in the same way as DSA exponentiation, namely when padding the nonce. On the other hand, the default multiplication uses blinding to make side-channel leakage independent

of the nonce. Additionally, OpenSSL and BoringSSL provide optimized constant-time windowed multiplication routines for several NIST curves. We discovered a tiny but severe side-channel leakage in their constant-time point addition, which leaks whenever a nonce multiplication window is all zero. For OpenSSL, we identified additional nonce leakage due to Bignumber handling, which was partly known before.

Modular inversion in OpenSSL and LibreSSL is done via a variant of Euclid's algorithm, claiming some side-channel security. Nevertheless, we found an easy-to-exploit vulnerability leaking the topmost nonce bit during a division step. Moreover, Euclid's algorithm inherently leaks the number of iterations, which correlates to the nonce itself. While we could not find a way to exploit this non-constant-time behavior, our tool reported another leak in a final negation step that helps an attacker again to learn the topmost nonce bit. BoringSSL employs Fermat's little theorem to invert nonces securely. Due to our findings, OpenSSL also switched to Fermat inversion.

Modular Multiplication. While OpenSSL uses blinding to alleviate non-constant-time code, LibreSSL removes blinding too early, leaking the length of the inverse nonce in some cases.

9.4 Detailed Analysis

In the following, we present our analysis strategy and discuss results and discovered vulnerabilities in detail.

Analysis Strategy. The process of tool-aided side-channel analysis comprises a proper selection of algorithms to test, the actual analysis phase, and interpretation of the results. Since OpenSSL supports over 80 different elliptic curves and countless compiler options, exhaustive testing of each combination is impractical. We selected the default configuration as a basis for our analysis, and selectively enabled different implementations of popular NIST curves. We focused on ECDSA curves operating close to a machine word boundary. For DSA, we tested all three parameter sets available. For the actual analysis, we used our tool alongside manual code review to specifically test relevant portions in the code. While the tool helps uncover leakage, interpreting the results remains a manual task. In particular, leakage models might not trigger if they do not match the actual leakage. In this case, the leakage might still show up in the *phase one* differences reported by DATA, and an extension of the leakage models is required. Also, leakage models might show a correlation without causation, e.g., via public values. Such cases can be eliminated by tracing the leakage back to its sources in our DATA GUI.

Table 9.2: Disclosed vulnerabilities in OpenSSL, LibreSSL, and BoringSSL and whether they are fixed ✓as of October 2019, currently being patched 🔧, or unpatched ✗. Side channel attacks can be easy ●, medium ◐, or hard ○.

	Vulnerability	OpenSSL	LibreSSL	BoringSSL	Leakage	SC	Comments
Generate:	(V1) Small k (top)	EC✗	EC✗	–	Topmost 0-limbs of k	●	Leaks in several subsequent steps
Exp. / Scalar Mult.	(V2) k-padding resize	DSA✓EC✓	DSA✗EC✗	–	Topmost 0-bits of k	●	CVE-2018-0734, CVE-2018-0735
	(V3) consttime-swap	DSA✓EC✓	DSA✗EC✗	–	same as (V2)	◐	Already known before
	(V4) Downgrade	DSA✓	–	–	same as (V2) + [GBY16]	●	Introduced while fixing (V2)
	(V5) k-padding (top)	DSA✗EC✗	DSA✗EC✗	–	same as (V2)	○	In BN_add/is_bit_set, cf. Section 9.5.
	(V6) Buffer conversion	EC✓	–	–	Topmost 0-bytes of k	○	
	(V7) Point addition	EC🔧	–	EC✓	All 0-windows of k	○	
Invert	(V8) Euclid BN_div	DSA✓	DSA✗	–	Topmost bit of k	●	Leaks via resize, similar to (V2)
	(V9) Euclid negation	DSA✓	DSA✗	–	Topmost 0-bit of k^{-1}	●	Leaks via conditional negation
Multiply:	(V10) Small k^{-1} (top)	–	EC✓	–	Topmost 0-limbs of k^{-1}	◐	

Following this strategy helped us uncover numerous vulnerabilities, as summarized in Table 9.2. To give an intuition about their exploitability, we rank them as easy to exploit ● if a Flush+Reload attack suffices for extracting nonce bits, medium ◐ for more elaborate attacks requiring performance degradation or Prime+Probe, or hard ○ for tiny leakage (e.g., few assembler instructions on a single cache line) which might be only exploitable in an SGX setting [BPS17; Mog+20]. The amount of leaked bits indicates complexity for a full key recovery.

9.4.1 Nonce Representation

OpenSSL and LibreSSL represent cryptographic values, such as nonces, via Bignumbers. Rather than being constrained to a fixed size, Bignumbers can store arbitrarily large values. For asymmetric cryptography, it is not uncommon to compute on variables comprising several hundred or thousands of bits.

Each Bignumber is stored in a `BIGNUM` struct that contains a lazily allocated array of limbs (e.g., 64-bit words). The number of allocated limbs is tracked via the field `dmax`. Bignumbers are represented in their minimal form, *i.e.*, each `BIGNUM` tracks the actually used limbs in a separate `top` field. As seen in Figure 9.2, `top` can be smaller than `dmax`. Whenever space is exhausted, a `BIGNUM` is dynamically resized via a call to `bn_wexpand`.

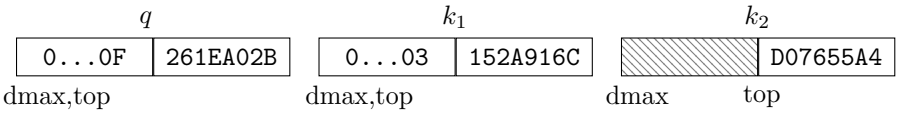

Figure 9.2: OpenSSL/LibreSSL (V1): some nonces (k_2) are smaller than the average (k_1) and the modulus q.

To maintain the minimal representation, OpenSSL and LibreSSL constantly realign `top` via a call to `bn_fix_top` by excluding leading zero limbs. This has two advantages: First, it avoids unnecessary computations and increases performance. Second, the programmer does not need to know the maximum size of Bignumbers in advance. However, it is also a source for unintended information leakage, leading to various side-channel vulnerabilities.

BoringSSL, in contrast, has hardened their implementation against such leaks by abandoning the minimal representation invariant of Bignumbers. They introduced a `width` field, which fixes `top` to the maximum width in advance.[3] Hence, it is immune to the Bignumber-related leaks we found.

Small Nonce Vulnerability (V1). Nonces are generated in the range $[1, q-1]$, as shown in Equation (9.6). If the length of the modulus q is slightly above a word boundary, it may happen that the generated nonce uses fewer limbs than q. In Figure 9.2, the first nonce k_1 uses two limbs, whereas the second nonce k_2 is represented in one limb, as indicated by `top`. A side-channel attacker learning the value of `top` can distinguish small nonces from large ones and mount a key recovery attack. In this example, q uses only four bits (0xF) of the topmost limb. Thus, an attacker learns whether the four topmost bits of k are zero. Consider w as the word size, *i.e.*, the size of one limb. For i386, w=32 and for x86_64, w=64. Thus, a small nonce leaks $L = log_2(q) \mod w$ bits, which occurs every 2^Lth signature on average. By collecting enough leaky signatures, an attacker can recover the private key via lattice or Bleichenbacher attacks (see Section 9.1.2).

In general, both DSA and ECDSA are affected by small nonces. However, if L is too large, leaky signatures occur too rarely to be practically exploitable. Since DSA moduli are always (half)word-aligned, $L = 32$ or $L = 64$ and attacks are impractical. On the other hand, for ECDSA, several curves have a modulus (group order) that is slightly above a word boundary. Table 9.3 lists all affected curves with $L < 20$, and curves affected on 64-bit systems are marked bold. For example, the sect131 curves leak 2 bits approximately every 4th signature, while secp521r1 leaks 9 bits every 512th signature.

In order to exploit the small nonce vulnerability, an attacker needs to learn the nonce length (*i.e.*, the value of `top`). Since the nonce is involved in many different computation steps, there are plenty of opportunities for an attacker to observe its length. We found small nonce leakage in the nonce generation, scalar multiplication, and nonce inversion (Equations (9.6), (9.8) and (9.15)). In the following, we focus on the most critical leakage present in the OpenSSL

[3] https://github.com/openssl/openssl/issues/6640

Table 9.3: OpenSSL/LibreSSL curves leaking L bits of small (inverse) nonces (V1),(V10) on 32/64-bit systems.

Curve	L_{32}	L_{64}	Curve	L_{32}	Curve	L_{32}
secp112r1	15.8	–	sect163r1	2.0	c2tnb359v1	0.8
secp112r2	13.8	–	sect163r2	2.0	c2tnb431r1	1.7
secp521r1	9.0	**9.0**	sect233k1	7.0	wap-wtls1	16.0
prime239v1	15.0	–	sect233r1	8.0	wap-wtls3	2.0
prime239v2	15.0	–	sect239k1	13.0	wap-wtls4	16.0
prime239v3	15.0	–	c2pnb163v1	2.0	wap-wtls5	2.0
sect113r1	16.0	–	c2pnb163v2	2.0	wap-wtls6	15.8
sect113r2	16.0	–	c2pnb163v3	2.0	wap-wtls8	16.0
sect131r1	2.0	**2.0**	c2tnb239v1	13.0	wap-wtls10	7.0
sect131r2	2.0	**2.0**	c2tnb239v2	12.4	wap-wtls11	8.0
sect163k1	2.0	–	c2tnb239v3	11.7		

version patched against (V8). The leaky code in Listing 9.1 converts the nonce stored in `BIGNUM a` into its Montgomery representation. `BIGNUM b` holds a Montgomery conversion factor. If both `a` and `b` have the full word length of q, denoted as `num`, the `if` branch will execute an assembler-optimized multiplication (`bn_mul_mont` in line 4) and terminate in line 5. If, however, the nonce `a` is one limb smaller, OpenSSL falls back to the functions `bn_mul_fixed_top` and `bn_from_montgomery_word`. By probing any of those functions, e.g., with a Flush+Reload attack, an attacker can distinguish small nonces from larger ones.

```
1  if (a->top == num && b->top == num) {
2      if (bn_wexpand(r, num) == NULL)
3          return 0;
4      if (bn_mul_mont(...))
5          return 1;
6  }
7  ...
8  if (!bn_mul_fixed_top(tmp, a, b, ctx))
9      goto err;
10 if (!bn_from_montgomery_word(r, tmp, mont))
11     goto err;
```

Listing 9.1: Simplified OpenSSL Little Fermat inversion leaking small nonces (V1) via conditional branching.

Unfortunately, this vulnerability is not only easy to exploit, but patching is hard as small nonces leak in several places. On June 25, 2019, we reported this issue to OpenSSL. The OpenSSL team decided to target a fix in OpenSSL version 3.0, as it requires a major redesign of OpenSSL's Bignumber implementation.

Small Nonce Leakage Details. OpenSSL shows leakage as follows: Nonce generation in `BN_generate_dsa_nonce` relies on `BN_div` for nonce reduction, which is non-constant time and leaks the length of small nonces, e.g., via `BN_rshift`. Also, the nonce is stripped by skipping leading zero limbs via `bn_correct_top`, which causes leakage in subsequent steps. OpenSSL uses a blinded double&add for

default scalar multiplication in `ec_GF2m_simple_points_mul`. Before blinding is applied, the nonce length leaks when being copied from `scalar` to `k` via `BN_copy`, when checking its bit length via `BN_num_bits`, and during the first addition of the nonce with the cardinality via `BN_add`. Also, the NIST-optimized curves call `BN_num_bits` with the nonce as input, e.g., in `ec_GFp_nistp521_points_mul`, which is already known to leak the length of the input.[4] During nonce inversion done via `BN_mod_exp_mont`, which is invoked by `ec_group_do_inverse_ord` and `ec_field_inverse_mod_ord`, there is an early abort when comparing the Bignumbers k and q via `BN_ucmp`. While exploitation might be tricky, we also found an easy-to-exploit leak, which we described in Section 9.4.1.

For LibreSSL, nonce generation leaks the nonce length via an early abort condition during rejection sampling via `BN_ucmp`. LibreSSL also leaks the nonce length during the first addition of the nonce and the group order in `BN_add`. Moreover, they used an old non-constant-time version of `BN_num_bits_word`, which was patched in OpenSSL already in January 2018 via commit `972c87df`. Due to our reporting, LibreSSL patched this issue in commit `9046ac5`.

9.4.2 Nonce Generation

In the following, we analyze nonce generation for different libraries. DSA and ECDSA nonces are generated both in the same way.

Rejection Sampling. To generate a nonce k uniformly at random in the interval $[1, q-1]$, LibreSSL and BoringSSL implement rejection sampling. They sample k in the interval $[1, 2^{qbits}-1]$, where $qbits = \lfloor \log_2 q \rfloor + 1$. If k exceeds $q-1$, it is rejected, and the procedure is repeated. The final k is uniformly distributed, assuming an unbiased random number generator. Although rejection sampling is inherently non-constant time, it only leaks information about rejected nonces. While we did not find issues for BoringSSL, small nonces leak for LibreSSL. Interestingly, during our analysis, one condition in the rejection sampling procedure of LibreSSL was reported to slightly correlate with the nonce length. However, the code in question did not depend on the final nonce. Instead, this leak occurred due to a correlation with the length of the public q, which also upper-bounds the length of the nonce. This false positive indicates that tool-aided analysis still requires careful analysis of the leakage.

Truncation. Rejection sampling has no guaranteed upper execution time. There is always a non-zero probability of rejecting the current k, which demands repeating the sampling phase. OpenSSL takes a different approach. It first generates a large number k' in the interval $[0, 2^{qbits+64}-1]$, as seen in Algorithm 9.1 lines 2–6. To compute the final nonce, k' is truncated to the target interval $[0, q-1]$ via modular reduction (line 8). As with LibreSSL, small nonces leak during truncation, as detailed in Section 9.4.1. Moreover, truncation introduces a tiny bias in k since q does not exactly divide $2^{qbits+64}$. However, since k' is 64 bits larger than q, this bias is impractical to exploit.

[4] See https://github.com/openssl/openssl/pull/5001#discussion_r159935593.

Algorithm 9.1: OpenSSL nonce generation by truncation

```
   input  : x, q ;                    // Private key and modulus
   input  : m ;                       // Message digest
   output : k ;                       // Nonce
1  k' ← []
2  while num_bits(k') < num_bits(q) + 64 do
3  |  rnd ←R [0, 2^512 − 1]
4  |  digest ← SHA512(x|m|rnd)
5  |  k'.append(digest) ;             // Up to num_bits(q) + 64 bits
6  end
7  k'' ← BN_bin2bn(k') ;              // Convert to BIGNUM
8  k ← k'' mod q ;                    // Reduce via BN_div
```

Before reducing k', OpenSSL converts it to a Bignumber representation via `BN_bin2bn` in line 7, which introduces a tiny side-channel leakage on k'. In particular, `BN_bin2bn` removes leading zeros, leaking the byte length of k' to a side-channel attacker. Our tool revealed another leakage in `BN_div` called in line 8, which leaks the length of k'. Luckily, both issues are impractical to exploit due to the 64-bit margin of k'.

Private Key Inclusion. Biases in the nonce generation are fatal. For that reason, some variants of (EC)DSA [Por13; JL17] compute the nonce deterministically from the message via hash functions rather than using randomness. Similarly, OpenSSL uses a cryptographic hash function to merge the private key as additional input into the nonce generation procedure.[5] By applying the hash function to the random number itself, the message m and the private key x (Algorithm 9.1 line 4), the resulting nonce is unpredictable to an attacker, even for biased random numbers. Moreover, this approach also protects against side-channel leaks. We found that OpenSSL uses a leaky AES[6] during random number generation when compiled with the `no-asm` flag. The hash function in line 4 decorrelates these leaks from the nonce.

BoringSSL and LibreSSL do not include the private key in the nonce computation, which makes them susceptible to biased random number generators. However, we did not analyze the uniformity or unpredictability of the random number generators themselves.

9.4.3 DSA Exponentiation

K-padding Vulnerabilities (V2)-(V5). Bignumber computation has been a source for nonce leakage in the past. For example, the fixed window exponentiation of OpenSSL leaks the bit length of the secret exponent k (Algorithm 9.2 line 5). This leakage was fixed by padding nonce k with q until it has a fixed length, namely

[5] This change was introduced in OpenSSL commit `8a99cb2` in 2013.
[6] It leaks several intermediate values via lookup tables `Te0 - Te3`.

Algorithm 9.2: Exponentiation with k-padding

```
   input  : k ;                          // Nonce
   output : r ;                          // Signature part
1  k ← k + q ;                           // Expand k to fixed num_bits(q) + 1
2  if num_bits(k) <= num_bits(q) then
3  |  k ← k + q
4  end
5  r ← g^k  mod q
```

$num\_bits(q)+1$, as shown in lines 1–3. The initial k-padding[7] executed the second addition in line 3 conditionally. To prevent attacking this conditional execution, it was made constant time.[8] OpenSSL k-padding is shown in Listing 9.2. In lines 13–14, OpenSSL unconditionally computes both additions inside `BIGNUM`s `l` and `m`, while line 15 copies the correct result to `k`.

```
1   q_bits = BN_num_bits(dsa->q);
2  -if (!BN_set_bit(k, q_bits)
3  -    || !BN_set_bit(l, q_bits)
4  -    || !BN_set_bit(m, q_bits))
5  + q_words = bn_get_top(dsa->q);
6  +if (!bn_wexpand(k, q_words + 2)
7  +    || !bn_wexpand(l, q_words + 2))
8       goto err;
9   ...
10  BN_set_flags(k, BN_FLG_CONSTTIME);
11 +BN_set_flags(l, BN_FLG_CONSTTIME);
12  ...
13  if (!BN_add(l, k, dsa->q)
14 -    || !BN_add(m, l, dsa->q)
15 -    || !BN_copy(k, BN_num_bits(l) > q_bits ? l : m))
16 +    || !BN_add(k, l, dsa->q)
17      goto err;
18 +BN_consttime_swap(BN_is_bit_set(l, q_bits), k, l,...);
```

Listing 9.2: Vulnerable k-padding in OpenSSL, with code added (+) and removed (-) during the patching process.

By analyzing OpenSSL, we found that k-padding leaks in several ways. First, we discovered an easy-to-exploit vulnerability leaking the size of the nonce via `dmax` inside the second `BN_add` (Listing 9.2 line 14). This leakage denoted as (V2) allows full key recovery. Second, our tool also reported data leakage in line 15, which was already known before and is denoted as (V3). By distinguishing whether buffer `l` or `m` is copied, one learns the same information as before. Third, we found the same information leaking via the nonce's `top` variable, denoted as (V5). This leakage exists in all patched versions and occurs when `k` is processed in lines 16 and 18. Although harder to exploit, we show an end-to-end attack in an SGX setting in Section 9.5.

[7] Nonce padding was introduced in OpenSSL commit `0ebfcc8` in 2005.
[8] Constant-time padding was introduced in OpenSSL commit `c0caa94`.

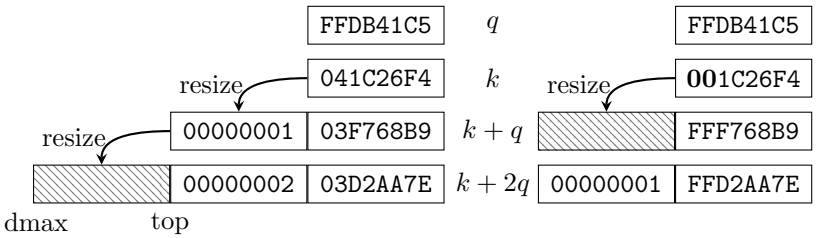

Figure 9.3: OpenSSL/LibreSSL k-padding causes Bignumber resize, depending on the topmost nonce bits (V2).

K-padding Resize Vulnerability (V2). As mentioned before, OpenSSL lazily resizes Bignumbers whenever their space is exhausted. E.g., when adding two `BIGNUM`s with `BN_add`, the resulting `BIGNUM` is expanded to the largest `top` value of the summands plus one limb for holding a potential carry. Unfortunately, lazy resizing happens during nonce padding in lines 13 and 14 of Listing 9.2. Consider the example in Figure 9.3, where the `BIGNUM`s k and q contain one limb each. On the left side, the first addition $k + q$ resizes the result buffer to two limbs in order to hold the additional carry exceeding the first limb. The second addition $k + 2q$ resizes to three limbs, although only two limbs are actually used since the carry is zero. In contrast, on the right-hand side, the first addition does not overflow, and the second addition only requests two limbs. Since the result `BIGNUM` already has two limbs, no actual resize happens.

By distinguishing whether one or two resize operations happened, a side-channel attacker can learn information about k. The second resize only happens if the first addition overflowed into the carry limb. In practice, such an overflow can only happen if q is close to a word boundary, that is, the topmost bits are set. Again, consider w as word size. Then, $Q = \lfloor log_{2^w}(q) \rfloor + 1$ is the number of words needed to represent q, and $qbound = (2^w)^Q > q$ is the upper bound (exclusive) of q representable with Q words. No resize happens if $k + q < qbound$, which occurs with probability $(qbound - q)/q$. Thus, for each such situation, an attacker can learn L nonce bits at once:

$$L = log_2(q) - log_2(qbound - q) \tag{9.31}$$

Since k is chosen uniformly at random, this happens for approximately every 2^Lth signature. In the previous example, $qbound = 0x1\,0000\,0000$ and $q = 0xFFDB\,41C5$, hence an attacker can learn $L = 10.8$ nonce bits for one out of 1783 signatures on average. By collecting enough leaky signatures, an attacker can recover the private key, as shown in Section 9.1.2.

Only DSA moduli close to the word boundary are susceptible. OpenSSL supports DSA moduli in the ranges 160, 224, or 256 bits, respectively. Since these parameters are all at a 32-bit boundary, they are all susceptible on a 32-bit system. For 64-bit systems, only DSA with 256-bit is on a word boundary and, thus, susceptible. The modulus q is a prime generated randomly for each key

with its topmost bit set. Hence, every 2^Lth key is susceptible to $L+1$-bit nonce leakage.

Exploitation of the vulnerability is straight forward. An attacker needs to monitor Bignumber resize operations during k-padding. Each Bignumber resize triggers several nested allocation routines of OpenSSL, which in turn invoke malloc/realloc from the standard library. Hence, a Flush+Reload attacker has plenty of opportunities to observe a resize with little noise. This attack is practical in terms of easy-to-obtain side-channel observations and low complexity for key recovery, which caused OpenSSL to issue CVE-2018-0734.

Consttime-swap Vulnerability (V3). Our tool showed another k-padding issue, which was already documented in the source code comments. After the two additions, copying the correct result to the target Bignumber `k` accesses different Bignumbers `l` or `m`, as shown in Listing 9.2 line 15. This leaks the same information as (V2) and could be exploited via a Prime+Probe attack on the Bignumber `l` or `m`, respectively.

Patching (V2) and (V3). Our reports triggered immediate discussion and patching[9] by the OpenSSL team. To avoid lazy reallocation, the patch enlarges the preallocation of the nonce buffers (lines 6–7). To hold the padded nonce, one additional limb would suffice. Since `BN_add` allocates an additional carry limb, this totals two additional limbs to preallocate. To fix the consttime issue, the patch replaces Bignumber `m` with `k` in line 16 and introduces `BN_consttime_swap` in line 18.

LibreSSL adopted similar patches for ECDSA, but insufficiently, as explained in Section 9.4.4. We contacted LibreSSL on May 17, 2019, but they did not apply these patches to DSA.

Downgrade Vulnerability (V4). By analyzing the OpenSSL patches for (V2) and (V3) with our tool, we immediately recognized a flaw bypassing constant-time exponentiation. While Bignumber `k` has the flag `BN_FLG_CONSTTIME` set, Bignumber `l` has not. The consttime-swap introduced in Listing 9.2 line 18 also swaps these flags between `l` and `k`, making `k` lose its `BN_FLG_CONSTTIME` flag. This causes every other subsequent exponentiation (Equation (9.7)) to downgrade to the unprotected variant. As shown in [GBY16], this can be exploited to recover DSA keys, e.g., from OpenSSH handshakes.

As we discuss in the following, erroneous flag propagation has a long history, since manual detection within the complex code base of OpenSSL is non-trivial. Luckily, our systematic tool-aided approach uncovered this issue straight away, avoiding another exploit-patch cycle. The final patch[10] applies the `BN_FLG_CONSTTIME` flag also to the Bignumber `l` in line 11.

Related Vulnerabilities to (V4). Issues with the `BN_FLG_CONSTTIME` are not uncommon.[11] Garcia et al. [GBY16] exploited a vulnerability similar to (V4) where `BN_copy` lost the `BN_FLG_CONSTTIME` flag on the secret nonce. Garcia et al. [GB17] also exploit a missing `BN_FLG_CONSTTIME` flag in OpenSSL

[9] See OpenSSL commit `a9cfb8c`.
[10] See OpenSSL commit `00496b6`.
[11] `https://github.com/openssl/openssl/issues/6078`

Table 9.4: OpenSSL/LibreSSL curves leaking L nonce bits via k-padding (V2)–(V5) on 32-bit and 64-bit systems.

Curve	L_{32}	L_{64}	Curve	L_{32}	L_{64}
brainpoolP160	3.4	–	brainpoolP320	2.2	2.2
brainpoolP192	1.7	1.7	brainpoolP384	0.3	0.3
brainpoolP224	2.4	–	brainpoolP512	1.0	1.0
brainpoolP256	1.0	1.0			

to recover ECDSA keys via unprotected modular inversion (Equation (9.8)). In Chapter 8, we reported similar defects in RSA key initialization where `BN_MONT_CTX_set` lost the `BN_FLG_CONSTTIME` flag after a `BN_copy`, causing subsequent modular inversion to be unprotected. Also, we reported a missing `BN_FLG_CONSTTIME` flag in DSA key loading `dsa_priv_decode`, leading to unprotected modular exponentiation.

K-padding Top Vulnerability (V5). Fixing the resize vulnerability (V2) does not mitigate the Bignumber minimal representation issue. That is, even if the buffer size (`dmax`) is independent of `k`, the number of used limbs (`top`) still depends on the nonce (cf. Figure 9.3). In particular, the second addition `BN_add` in Listing 9.2 line 16 leaks the value of `l->top` via the number of limb-wise additions carried out. Also, `BN_is_bit_set` (line 18) leaks via an early abort, as detailed in Section 9.5. This has the same implications as vulnerability (V2).

Naturally, exploitation is harder than (V2), as the leaky code is only a few instructions. Nevertheless, we reported this residual leakage already back in October 2018. Since we could not observe any progress, we developed an end-to-end SGX attack, as outlined in Section 9.5. Reporting our attack on May 8, 2019 triggered a pull request with our proposed patch [Dal19]. However, the pull request was closed, since the OpenSSL team decided for a long-term mitigation abandoning the minimal representation invariant similar to BoringSSL. While the decision for a complete fix is encouraging, this vulnerability remains unpatched until then.

9.4.4 ECDSA Scalar Multiplication

K-padding Resize Vulnerability (V2). Similar to DSA, our investigations revealed the same Bignumber resize vulnerability also in ECDSA, leading to CVE-2018-0735. Only curves with a word-aligned modulus (*i.e.*, a word-aligned curve cardinality) are vulnerable. We found that all Brainpool curves are exploitable and leak up to 3.4 bits, as listed in Table 9.4. Luckily, other curves that have a word-aligned modulus are not practically exploitable. For example, the curve secp128r1 has cardinality 0xFFFFFFFD FFFFFFFE F80091C8 184ED68C. By using Equation (9.31), an attacker could learn $L = 31$ nonce bits at once. However, only every 2^{31}th signature will be vulnerable, which renders actual attacks impractical.

Fixing this issue for ECDSA is analogous to DSA.[12] Although LibreSSL adopted the patch[13], our tool still reported leakage. Further analysis revealed that the patched LibreSSL version uses k-padding twice, once correctly during multiplication `ec_GFp_simple_mul_ct` and a second time inside `ecdsa_sign_setup`. The second k-padding was not only unpatched, leading to another instance of (V2); it even created additional leakage. In particular, the multiplication routine performs an additional leaky modular reduction if the nonce (the scalar) is larger than the group order. This issue again highlights the importance of tool-aided side-analysis during the patching process. Although we reported this issue to LibreSSL on May 20, 2019, it is still unpatched.

Related issues (V3) and (V5). As with DSA, the issues with consttime swap (V3) and k-paddding top (V5) as well as their patches equally apply to ECDSA for the curves listed in Table 9.4. Since the patched LibreSSL uses k-padding twice for ECDSA, it is still vulnerable not only to (V2) but also to (V3).

Buffer Conversion (V6). We uncovered distinct vulnerabilities in some ECDSA scalar multiplication routines of OpenSSL[14] leaking the byte length of the nonce. Before the actual scalar multiplication, the nonce is converted from a Bignumber to a byte array with `BN_bn2bin` and `flip_endian`. In contrast to Bignumber-related issues subject to word-granular leakage, those functions operate on bytes. By stripping leading zero bytes, they leak the byte length of a nonce. For secp224r1 and secp256k1, $L = 8$ bits leak every 256th signature, and $L = 16$ bits every 65536th signature. secp521r1 is not byte aligned and leaks $L = 1$ bit every 2nd signature, or $L = 9$ bits every 512th signature, etc. Since the side channel only comprises a few instructions and data bytes, we rate it as hard to exploit. Yet, an SGX attack similar to Section 9.5 could target the stripped nonce buffer. This issue was patched on August 3, 2019.[15]

Point Addition Vulnerability (V7). For ECDSA signatures, the nonce k is multiplied with the generator G in Equation (9.15). Analyzing OpenSSL and BoringSSL showed that the constant-time scalar multiplication uses a non-constant-time point addition. This leaks nonce windows which are zero. We uncovered this leakage with our tool showing 100% correlation on the bit length of k, as shown in Section 9.2.3 Figure 9.1.

The scalar multiplication in question uses a fixed window approach. This means that the scalar is split into multiple fixed-size windows. Each window is used as an index into a precomputed table to select the point to be added. If the window is all-zero, the first point is selected from the table. This first point represents infinity and has all-zero coordinates.

[12] See OpenSSL commit `99540ec`.
[13] See LibreSSL commit `34b4fb9`.
[14] This applies to the optimized NIST curve implementations, which are obtained via the `enable-ec_nistp_64_gcc_128` compilation flag.
[15] See `https://github.com/openssl/openssl/pull/9511` as well as commits `8b44198b` and `805315d3`

Table 9.5: Curves vulnerable (●) to ECDSA point addition leak (V7) in constant-time scalar multiplication for base point (BP) or arbitrary point (AP).

	Curve	BP	AP	Compile configuration
OpenSSL	secp224r1	○	●	`enable-ec_nistp_64_gcc_128`
	secp256k1	○	●	
	secp256k1	●	●	`enable-ec_nistp_64_gcc_128 no-asm`
	secp521r1	●	●	`enable-ec_nistp_64_gcc_128`
BoringSSL	secp224r1	●	●	`OPENSSL_SMALL`
	secp256k1	○	●	
	secp384r1	●	●	
	secp521r1	●	●	

Point addition has a special treatment for cases in which both points to be added are equal. In these cases, the point is doubled in Listing 9.3 line 2. Although doubling itself is never performed for ECDSA, the check in line 1 reveals whether the added point is infinity or not. Hence, an attacker can learn whether the current nonce window is zero.

```
1 if (x_equal && y_equal && !z1_is_zero && !z2_is_zero)
2     point_double(...)
```

Listing 9.3: Simplified excerpt from vulnerable point_add (V7) in OpenSSL and BoringSSL scalar multiplication.

The leak occurs due to the order in which the branching condition is evaluated. The `if` statement in line 1 consists of four separate conditions, which are compiled into multiple compare and jump instructions (cf. Figure 9.1 in Section 9.2.3). This creates a tiny leakage because a different number of instructions are executed, depending on the secret scalar. When the added point is not infinity, already the first comparison (`x_equal`) fails, since the added points are unequal. If the added point is infinity, the flags `x_equal` and `y_equal` are set to `true` because infinity is represented with all-zero projective (x,y,z) coordinates. Only the last flag `!z2_is_zero` fails, which results in a few more executed instructions.

With a window size of w bits, roughly 2^{-w}th of the nonce is leaked per sign operation. E.g., for the common window size of 5, around 3.2% of the nonce is leaked. Exploiting this leakage with a cache attack seems infeasible due to the tiny difference in the executed code. However, in an SGX setting, [BPS17; Mog+20] could be used to single-step instructions.

We systematically analyzed various point multiplication implementations and list affected ones in Table 9.5. Base-point multiplication is used in ECDSA, whereas arbitrary point multiplication is used in Elliptic Curve Diffie-Hellman. In OpenSSL, only optimized NIST implementations are affected. Other configurations and curve settings are unaffected because they use a blinded double-and-add implementation. In BoringSSL, all curves are vulnerable at least under one configuration. Since LibreSSL only uses blinded double-and-add for scalar multiplication, it is also unaffected.

Algorithm 9.3: OpenSSL/LibreSSL leaky inversion

```
   input  : a, n
   output : inv ;                    // Inverse of a mod n
1  (A, B, X, Y, sign) ← (a, n, 1, 0, −1)
2  while B > 0 do
3  |  (D, M) ← (A/B, A%B) ;          // Leaky division (V8)
4  |  (A, B) ← (B, M)
5  |  (X, Y) ← (D · X + Y, X)
6  |  sign ← −sign
7  end
8  ensure A = 1
9  if sign < 0 then
10 |  Y ← n − Y;                     // Leaky negation (V9)
11 end
12 inv ← Y  mod n
```

Our report led to an immediate fix[16] by BoringSSL, which replaces the evaluation of the branching condition with bit-wise operations, such that a short-circuit evaluation is no longer possible. OpenSSL is currently in the process of patching[17], since our responsible disclosure on May 31, 2019.

9.4.5 Modular Inversion

Euclid BN_div (V8). OpenSSL and LibreSSL implement modular inversion via the Extended Euclidean algorithm. In contrast to the binary extended Euclidean algorithm (BEEA), which is known to be vulnerable [GB17; WSB18b; Ald+19], the inversion used for DSA is denoted as constant time in the source code. With our tool, we uncovered a leak hidden deeply in this constant-time modular inversion of OpenSSL. In particular, the first Euclidean iteration leaks the topmost nonce bit of every signature to a side-channel attacker.

Since DATA accumulates leakage not only over the first but all Euclidean iterations, our leakage models did not show a high correlation. Instead, we found this leak by carefully analyzing the differences reported by DATA's phase one.

Algorithm 9.3 shows the leaky Extended Euclidean inversion. The division `BN_div` in line 3 is not constant time, although the `BN_FLG_CONSTTIME` flag is used. Note that `BN_div` computes both the integer division D and the remainder M. In the first iteration, `A` holds the public modulus `q`, and `B` holds the secret nonce k. Inside `BN_div` the `BIGNUM`s are aligned before the actual division, as follows: The divisor (nonce k) is shifted to the left such that its highest word is filled, having no leading zero bits. The numerator (modulus) is shifted left by the same amount of bits (modulo the word size). Normally, the nonce has the same bit length as the modulus, and the numerator also gets word-aligned. If the

[16] See BoringSSL commit `12d9ed6`.
[17] `https://github.com/openssl/openssl/pull/9239`

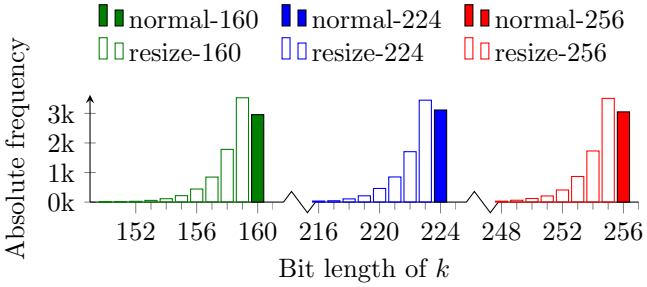

Figure 9.4: OpenSSL DSA leaks the topmost bit of the nonce during Euclidean inversion (V8).

nonce, however, has fewer bits than the modulus, this shift operation causes the numerator `BIGNUM` to spill over to the next limb, and `top` is incremented. This will cause a `BIGNUM` resize operation. Observing such resize operations allows an attacker to distinguish nonces whose most significant bit is cleared.

To evaluate the leakage further, we generated 100 DSA keys and computed 100 DSA signatures per key. Figure 9.4 plots the resulting bit length of k for each of the standard DSA settings $qbits \in \{160, 224, 256\}$. There is a clear separation between nonces with a zero MSB causing a resize, and "normal" nonces whose topmost bit is set. Since the modulus is chosen randomly per key, the probability of having the MSB of the nonce set is only around 30%, whereas the probability of a zero MSB is around 70%. Thus, an attacker can effectively learn approximately 0.88 bits[18] for each signature.

To exploit the vulnerability, an attacker probes for leaky resize operations in `BN_div` during the first Euclidean inversion. A simple Flush+Reload attack on the corresponding Bignumber allocation routines suffices, as with (V2). Since $L = 1$, a Bleichenbacher attack is needed to recover the private key.

We proposed to abandon Euclid inversion in favor of a safer method. One could either use blinding to decorrelate leakage from the nonce or use Fermat's little theorem, as done by BoringSSL. OpenSSL decided to implement Fermat's little theorem[19] by computing the inverse as $k^{q-2} \mod q$. Although we reported this vulnerability also to LibreSSL on May 17, 2019, they did not apply the patch.

Euclid Negation (V9). The Euclidean algorithm is inherently non-constant time and leaks the number of iterations. We initially tried to correlate the number of iterations to the nonce length. By doing simulations, we found that the number of iterations fluctuate significantly, and cannot be used as a reliable side channel for learning the nonce length. However, after applying our automated statistical methods, DATA reported a significant correlation on the bit length of the *inverse* nonce k^{-1}. In particular, the Euclidean algorithm keeps track of the inverse's sign bit and conditionally negates Y in the end, as shown in Algorithm 9.3 line 10.

[18] Computed via the information entropy
[19] The patch was introduced in OpenSSL commit `415c335`.

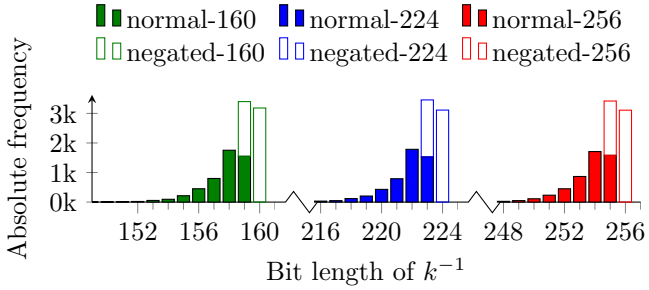

Figure 9.5: OpenSSL DSA leaks the topmost bit of the inverse nonce after Euclidean inversion (V9).

We found that negation causes larger inverses on average, presenting a useful side channel.

To visualize the leakage, we repeat the experiment from (V8). We plot the bit length of inverse nonces k^{-1} in Figure 9.5. In our experiments, negation gives a large k^{-1}; however, the topmost bit is not necessarily one ($num\_bits(k^{-1}) \geq qbits - 1$). In contrast, "normal" inversion without negation causes the MSB of k^{-1} to be zero, which happens in around 70% of the cases, giving 0.88 bits of leakage per signature.

This vulnerability can be exploited via a Flush+Reload attack on the leaky `BN_sub` function and only collecting signatures where no negation happens. A Bleichenbacher attack can be used to recover the actual private key. The patch introduced in (V8) also fixes this vulnerability. LibreSSL remains vulnerable, as they did not apply this patch.

9.4.6 Modular Multiplication (V10)

As Bignumber primitives are not constant time in several places [Rya19], OpenSSL blinds[20] the actual computation of the signature s in Equation (9.9) to avoid leaking the private key x. This works by applying a random blinding value b, as follows:

$$b \xleftarrow{R} [1, q-1] \tag{9.32}$$

$$s \leftarrow (bm + bxr) \mod q \tag{9.33}$$

$$s \leftarrow s \cdot k^{-1} \mod q \tag{9.34}$$

$$s \leftarrow s \cdot b^{-1} \mod q \tag{9.35}$$

This makes leakage during addition, modular reduction, and multiplication in Equation (9.33) independent of the private key as well as the inverse nonce in Equation (9.34). Unfortunately, when LibreSSL applied the patch,[21] they

[20] Blinding was introduced via OpenSSL commit `7f9822a`.
[21] See LibreSSL commits `2cd28f9` and `2a937ef`.

swapped Equation (9.34) and Equation (9.35), causing the multiplication with k^{-1} to be unprotected. In particular, the routine `BN_mul` leaks the value of `top` at various locations.

This vulnerability is conceptually the same as the small nonce vulnerability (V1), affecting the same curves listed in Table 9.3. It leaks whether the inverse nonce is one limb smaller than the modulus. Since leakage of the inverse nonce is equally dangerous as leakage of the nonce itself, an attacker can mount the same key recovery attack as for (V1). In response to our disclosure, LibreSSL fixed this issue.[22]

9.5 SGX Controlled-Channel Attack on (V5)

Fixing some of our reported vulnerabilities demand significant changes to the code base. For example, k-padding (V2) was fixed in OpenSSL, while the underlying problem of minimal Bignumbers still persists until OpenSSL has reworked the Bignumber implementation. In this section, we show how to exploit residual leakage via the k-padding top vulnerability (V5).

During k-padding, OpenSSL calls the function `BN_is_bit_set` with the intermediate nonce buffer `l`, as shown in Listing 9.2 line 18. If `l->top` is smaller than `q_bits`, this causes an early abort in Listing 9.4 line 4. In order to exploit this leakage, an attacker needs to detect whether or not line 5 is executed.

```
1 int BN_is_bit_set(const BIGNUM *a, int n) {
2     ...
3     if (a->top <= i)
4         return 0;
5     return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
```

Listing 9.4: OpenSSL k-padding leaks `k->top` (V5).

While Flush+Reload might not work due to the small amount of leaky code, we demonstrate a controlled-channel attack [XCP15] on an SGX enclave running the vulnerable DSA sign operation from the SGX SSL library [Int19]. Controlled-channel attacks detect individual memory accesses on a page granularity, be it code or data. Since the vulnerable function is likely on a single *code* page, probing this page does not suffice. Although more elaborate techniques to single-step enclave execution exist [BPS17], we distinguish whether line 5 accesses the *data* page covering buffer `a->d`.

For the attack, we need to trace execution to the vulnerable k-padding. We do this with the SGX-Step framework [BPS17] without using its single-stepping functionality. We unmap all relevant enclave code pages on which the following functions reside: `dsa_do_sign`, `BN_generate_dsa_nonce`, `BN_MONT_CTX_set_locked`, `BN_add`, and `BN_is_bit_set`. As soon as one of those pages is fetched by the enclave, a page fault is triggered, which we capture in user space via a custom signal handler. Then, we selectively enable only the faulted page until we hit the vulnerable `BN_is_bit_set` function. Now we also unmap the data page holding

[22] See LibreSSL commits `1f6b35b` and `159fbd1`.

the nonce buffer `a->d`. If the next step throws a page fault on `a->d`, we know that line 5 has been executed. If not, we know that the early abort in line 4 has been triggered. In that case the nonce was not resized in the first addition of k-padding (line 13 of Listing 9.2) and, thus, is smaller than the average. We only collect such signatures and mount a lattice attack.

We build the lattice according to Equation (9.21) and gradually fill it with leaky signatures until the lattice reduction reveals the private key. For the actual reduction, we use the BKZ algorithm with a block size of 30. For a DSA-256 modulus leaking $L = 8$ bits, recovery succeeded with 36 signatures within 3.3s. For $L = 6$ bit leakage, recovery took 47 signatures and 7.8s. $L = 4$ required 79 signatures and took 111 hours with an increased BKZ block size of 50, since it is closer to the estimated bound in Section 9.1.2, demanding at least $L = 3$.

For the attack to work, `a->top` in line 4 needs to be on a different page than `a->d`. This can be easily achieved if the enclave copies variably-sized attacker-controlled arguments, such as messages to sign, to the enclave heap. By changing the argument's size, Bignumber `a` can be shifted appropriately. If this is not possible, one can resort to stronger interrupt-driven attacks [BPS17; Mog+20] in order to single-step execution of the `BN_is_bit_set` function.

9.6 Evaluation

Having detailed all vulnerabilities, we now evaluate our analysis strategy as well as the effectivity of our leakage models.

Analysis Strategy. Investigating the leakage reports of DATA represents a chicken-and-egg problem. The results of DATA phase one cover all discovered differences (*i.e.*, potential leaks), but are tedious to analyze. Developing precise leakage models to filter those results requires an intuition about the nature of leakage, which in turn demands some manual analysis of phase one results. As described in Section 9.4, we concurrently followed both approaches. By manually analyzing phase one results, we gained an understanding of the libraries. Although we found vulnerabilities related to k-padding as well as (V8) that way, this task is tedious. Thus, we derived the leakage model $num\_bits(\cdot)$, which captures the bit length of k, $k + q$, and $k + 2q$ to detect k-padding leaks automatically. We used the gained knowledge to search for other Bignumber-related leaks, and also included inverse nonces k^{-1} in our models. Our leakage models confirmed initial results and helped us discover more Bignumber-related vulnerabilities such as (V1), (V9), and (V10). Moreover, since $num\_bits(\cdot)$ correlates with the bit length rather than the word length of the nonce, we also found leakage on a byte granularity (V6) and window granularity (V7).

The choice of library configurations and algorithm parameters is essential. E.g., we realized that (V2) does not show up for DSA-160 on a 64-bit system, while 32-bit systems leak for all parameter sets. Also, the choice of the tested modulus `q` is essential in causing leakage to show up. In order to confirm (V2) also for ECDSA, we analyzed all ECDSA moduli offline and found that only Brainpool curves are vulnerable. Similarly, discovering and analyzing leakage of

Table 9.6: Evaluation of leakage models. Depending on the triggered vulnerabilities, the differences (Diffs) found by DATA are filtered via our leakage models. The overall reduction is computed when filtering almost non-matching leaks (<1%), somewhat matching leaks (<50%), or all leaks except for perfect correlation (<100%).

Tested config	Vulnerabilities	Diffs	Leakage model (max. correlation in %)								Reduction		
			$num\_bits(\cdot)$				$HW(\cdot)$				Diffs–leaks (%)		
			k	$k+q$	$k+2q$	k^{-1}	k	$k+q$	$k+2q$	k^{-1}	<1	<50	<100
LibreSSL sect131r1	(V1),(V9) (V10)	1450	**100**	0.0	0.0	**100**	7.4	18.0	9.4	10.0	90.2	97.9	99.0
OpenSSL DSA-256	(V2),(V5) (V8),(V9)	663	**100**	**100**	**100**	79.8	0.0	2.7	17.8	0.0	23.7	26.4	27.5
OpenSSL secp521r1[a]	(V6)	88	**100**	0.0	0.0	1.5	11.4	20.3	0.0	1.8	84.1	94.3	94.3
BoringSSL secp521r1	(V7)	26	**100**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	96.2	96.2	96.2
OpenSSL secp521r1	artificial leak	535	32.4	0.0	0.0	8.3	14.0	**100**	13.5	0.0	98.1	99.8	99.8

[a]compiled with `enable-ec_nistp_64_gcc_128`

small nonces (V1) demanded careful investigations of (V10). Both issues depend on ECDSA curve parameters that are slightly above a word boundary, which led us to specifically testing the sect131r1 curve showing small nonces every fourth signature. Thus, we were able to find numerous instances of (V1) with the help of our tool. Also, we could generalize these results to other curves. E.g., for secp521r1, the (V1) vulnerability only shows up every 512th signature on average, which cannot be easily discovered by DATA within a reasonable time.

Leakage Models. We evaluate the leakage models on OpenSSL 1.1.1, BoringSSL `chromium-stable` commit `2e0d354`, and LibreSSL 3.0.0. We use GCC 6.3.0 and test DATA phase one with 16 and phase three with 200 traces.

Table 9.6 summarizes our results. We benchmark different configurations to trigger all major vulnerabilities and count all potential leaks (differences, or Diffs) found by the original DATA phase one. For each implemented leakage model, we print the maximum correlation, which reveals the strongest leak found by a leakage model. In order to capture how often leakage models match, the last three columns represent the overall reduction of phase one differences when filtered by the models. In particular, we discard leaks with less correlation than the thresholds 1%, 50%, and 100%. For example, the 100% threshold only preserves leaks that fully match the model.

LibreSSL sect131r1 leaks small nonces via the $num\_bits(k)$ model in several places with 100%. Moreover, LibreSSL uses leaky Euclidean inversion also for ECDSA, resulting in 100% leakage for $num\_bits(k^{-1})$. Since LibreSSL does not work with the so-called heap tracking of DATA phase one, it has over 1000 differences, most of which are filtered by our leakage models. Thus, the overall reduction is over 90%. Analyzing those leaks by hand would be quite tedious.

For OpenSSL DSA-256, the leaky k-padding addition (Listing 9.2 line 14) is captured by the $num\_bits(\cdot)$ models on $k+q$ and $k+2q$, showing 100% correlation. The corresponding leaky resize operation influences the heap layout and causes several subsequent Bignumber operations to leak via data accesses. Due to the high number of these actual data leaks, which are all instantiations of (V2), the reduction is "only" around 25%.

To trigger (V6), we compiled OpenSSL to use the optimized secp521r1 implementation. Indeed, $num\_bits(k)$ shows 100% correlation during the conversion of the nonce buffer and during scalar multiplication, as this implementation is also vulnerable to (V7). We also triggered (V7) for BoringSSL, showing 100% correlation. Other leakage models remain insignificant, and the overall reduction for OpenSSL is above 84% and for BoringSSL above 96%.

The Hamming weight model $HW(\cdot)$ did not show a high correlation in any of our tested configurations. It is designed for square&multiply and double&add, respectively. However, the tested DSA implementations use fixed window multiplication. ECDSA uses a blinded double&add by default, for which $HW(\cdot)$ would apply. However, the actual computation does not leak. To test the correctness of the $HW(\cdot)$ model, we artificially introduced a conditional code execution during double&add, leaking the current nonce bit. Indeed, we obtain 100% correlation on the padded nonce $k + q$.

9.7 Discussion

Proper tool support significantly improves side-channel analysis and facilitates the discovery of unknown weaknesses. However, tools do not fully discharge an analyst from thorough investigations. Knowledge of the nature of expected leakage is required to leverage tool support and interpret the results. Together with a visual representation of the results, we see this as a valuable path to follow.

The process of vulnerability patching has been tedious in the past, as evidenced by numerous issues involving the `BN_FLG_CONSTTIME` flag [GBY16; GB17; Wei+18b]. Also, patching of (V2) introduced new leakage in OpenSSL (V4) and LibreSSL (another instance of (V2) for ECDSA). We believe this is due to a lack of practical tools for developers to test their patches thoroughly. Luckily, our tool uncovered both issues with little effort. Also, regression testing of already discovered leakage is promising in this regard [Gri+19].

While most OpenSSL vulnerabilities were patched or are in the patching process, the issues (V1) and (V5) related to minimal Bignumbers (`top`) remain unpatched. The OpenSSL team decided to target a fix in version 3.0, as it requires a major redesign of their Bignumber primitives. According to [Dal19], reworking Bignumber arithmetic in BoringSSL prior to our disclosure took between one and two months. While BoringSSL immediately fixed (V7), LibreSSL only fixed (V10) and (V2) partially. We also were in contact with the developers of libgcrypt for fixing (V2), and the ring library for fixing (V7) in their code, however, without further in-depth analysis.

Due to a change in their security policy in May 2019, OpenSSL does not consider Flush+Reload attacks in their threat model anymore, since they are mounted on the *same physical system* [Ope19]. We see this downgrading questionable, as it tempers not only efforts to analyze OpenSSL's side-channel security but also undermines software relying on the previous threat model. For example, Intel SGX SSL [Int19] faces adversarial code on the *same physical system* by design. Also, vendors notified of (V2) by the CVE system were not notified of the equally dangerous (V1) due to this policy update.[23]

In the long term, more compiler support for side channels is needed [SCA18]. As of today, compilers might optimize constant-time code in a way that reintroduces side-channel leakage. Thus, a notion of side-channel invariants like constant-time guarantees is needed on a language level. For an overview of existing literature, we refer the interested reader to Section 6.3.2.

9.8 Summary

In this chapter, we showed that nonce leakage is far from being abandoned and requires attention both from academia and practitioners. For our systematic study, we extended the DATA framework to detect nonce leakage and developed an easy-to-use GUI. We found that having an intuitive GUI representation of the discovered leakage is imperative for a productive analysis of complex reports. E.g., it helped us to quickly determine whether a leaky function deeply nested in the call stack is given public or secret input. The visualization of leakage model results furthermore helped to identify hotspots, especially if the number of potential leaks is large.

For OpenSSL and LibreSSL, we found numerous side-channel vulnerabilities leaking secret (EC)DSA nonce bits that allow full key recovery in many cases. They mostly result from weaknesses in the underlying Bignumber implementation. We disclosed all our findings to the library developers and assisted them in the patching process. We furthermore document vulnerabilities that were not yet patched. We open-source our tools to help developers and analysts foster their efforts in strengthening the cryptographic libraries we use on an everyday basis against software side-channel attacks.

[23] https://www.cvedetails.com/cve/CVE-2018-0734/

Conclusion and Outlook

Good does not triumph unless good people rise to the challenges around them.
Alister E. McGrath

In this thesis, we studied enclaves as a promising technology for guaranteeing secure code execution in the midst of an insecure and complex software ecosystem. We tackled various challenges; however, many more are yet to be addressed. In the following, we summarize our insights and open challenges along three properties which we believe lie at the core of research on enclaves, namely user-centeredness, security, and openness.

User-centeredness. Enclaves are an ideal fit for computation tasks that shall be outsourced to the cloud, as well as Digital Rights Management (DRM). Unfortunately, DRM schemes tend to infantilize end customers by restricting their control over their own computers. It cannot be denied that this blanket distrust towards users is counterproductive for healthy customer relationships. Instead of abandoning enclaves for their potential use in DRM schemes, we promote their utilization in user-centric applications to improve the overall security of user data. For example, secure chat applications or video conferencing could be advanced from end-to-end security towards user-to-user security. Online payment systems could enforce stronger transaction authentication, and password stores could shield user passwords even in case of a system compromise. Other everyday use cases involve not only the protection of company data on a bring-your-own-device but also the secure usage of private data on a company device.

To realize this vision, we need more research on secure interaction between enclaves and users. We have shown that trusted hypervisors can provide secure enclave I/O in a generic way. However, the current trend for secure I/O with SGX goes towards specific I/O interception devices. In contrast, the solution provided by Arm TrustZone allows one to link a Trusted Execution Environment (TEE) directly with hardware peripherals. Future work on enclaves should attempt to combine these concepts by augmenting user enclaves with some form of kernel enclaves that can take temporary but exclusive control over particular devices. More importantly, the success of enclaves for the consumer market crucially depends on the usability of enclave systems. In particular, helping users without a security background to discern legitimate enclaves presented to them from a phishing app is a challenge on its own.

Security. Enclaves are inherently subject to stronger attackers controlling the operating system. However, the security assumptions and implications of

enclaves are not as well understood as those of other isolation technologies. In this work, we have addressed common misconceptions and wrong assumptions about misbehaving enclaves. We have shown that shielding against enclave malware is not only necessary but also doable with little effort. Still, more work needs to be done to explore the remaining attack vectors of enclave malware, e.g., via the API between enclaves and their host applications.

The threat of side channels for enclaves has been underestimated as well. Many creative attacks have exploited the various novel side channels SGX exposes. We have demonstrated that moving existing software into enclaves can have fatal consequences. In particular, single-trace attacks on key generation algorithms become pressing, as we demonstrated with our RSA attack. To ease the search for side-channel vulnerabilities, we have introduced the notion of differential address trace analysis (DATA), which captures most of the prominent side-channel attacks. We have advanced state-of-the-art side-channel analysis by developing our automated DATA tool. We have used DATA to identify and fix various unknown side-channel vulnerabilities in OpenSSL, amongst others. As the landscape of cryptographic libraries is vast, much more side-channel analysis is needed. Also, protecting enclaves against transient execution attacks is still an open issue, and it is unclear to what extent transient execution leakage is tolerable.

Openness. To promote wide deployment of enclaves, an open ecosystem is indispensable. On the one hand, an enclave system needs to encourage transparent assessment, e.g., by providing an open and verifiable design. We developed an open-source enclave system for the popular RISC-V architecture that fits the needs of resource-constrained environments in the IoT. On the other hand, more support is needed to propagate enclave technology towards more architectures. In the future, it will become decisive to establish common terminology and security standards for enclaves in order to make trust interchangeable among different architectures. In particular, attestation mechanisms for verifying a device's security state are usually highly dependent on the device manufacturer. The practical utility of enclaves for the open-source community, however, critically depends on the openness of the provided attestation infrastructure. Intel already took a step towards opening SGX via a flexible launch control mechanism that also enables third parties to deploy independent attestation services. We are currently lacking a proper understanding of the security implications of flexible launch control. On the other hand, decentralized attestation schemes could address many objections, but it is unclear how to design an enclave system to support decentralized attestation. To conclude, we want to encourage the research community to design and prototype the enhanced enclaves we envision on the open RISC-V architecture.

Epilogue

Do not let your hearts be troubled. Trust in God; trust also in me.
Jesus Christ – Gospel of John

Everything in IT security is based on trust, of which enclaves are a prime example. In this thesis, we studied enclaves, an exciting new building block for modern security architectures, from various low-level aspects. Let us take a step back and reason about their use from the perspective of individuals. Enclaves can effectively help minimize our trust in the systems. As such, they can restore what has been lost in previous years – digital control over one's own data. However, enclaves can equally remove trust from people. When managing digital rights become a monopoly of the few, enclave technology could serve an instrument to restrict people's control over their own devices. It is up to us, the security community, to discuss how technological advances can relax our trust assumptions on the technology itself, while at the same time strengthening trust between individuals.

In the end, trust is not at all a technological term; it is inherently relational. As such, trust can never be made obsolete. It will remain not only the currency of the future; trust is the substance of human nature itself. We shall not ask the question of whether or not to trust. It is a matter of whom we trust.

While writing this thesis during the corona outbreak, I witnessed an earthquake that shook our society with (un)controlled fear and tragic losses, while simultaneously bringing an overdue slowdown. It is healthy to acknowledge that technology alone cannot solve social issues. Let us not fall prey to the illusion a corona app would rescue us but take the courage to put aside any technology, if necessary, and exercise both common sense and altruism to restore trust in each other.

✳ ✳ ✳

List of Contributions

Publications

In the following, we list the author's scientific publications that are part of this thesis.

[Wei+18a] 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'18*. USENIX Association, 2018, pp. 603–620.

[Wei+19a] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGXJail: Defeating Enclave Malware via Confinement." In: *Research in Attacks, Intrusions and Defenses – RAID'19*. 2019, pp. 353–366.

[Wei+19b] 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'19*. 2019.

[Wei+20] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations." In: *USENIX Security'20*. In press. 2020.

[WSB18a] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. "Single Trace Attack Against RSA Key Generation in Intel SGX SSL." In: *Asia Conference on Computer and Communications Security – AsiaCCS'18*. 2018, pp. 575–586.

[WW17a] Samuel Weiser and Mario Werner. "SGXIO: Generic Trusted I/O Path for Intel SGX." In: *Conference on Data and Application Security and Privacy – CODASPY'17*. 2017, pp. 261–268.

Further Collaborations

In the following, we list further collaborations that led to scientific publications which are *not* part of this thesis.

[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'17*. 2017, pp. 3–24.

[Sch+18b] 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'18*. 2018.

[Sch+20b] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. "Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86." In: *USENIX Security'20*. Accepted for publication with minor revision at the time of writing. 2020.

[SWG19] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'19*. Ed. by Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, pp. 177–196.

Bibliography

[AA18] Zelalem Birhanu Aweke and Todd M. Austin. "Øzone: Efficient execution with zero timing leakage for modern microarchitectures." In: *Design, Automation & Test in Europe – DATE'18*. IEEE, 2018, pp. 1123–1128. ISBN: 978-3-9819263-0-9.

[AB20] Alejandro Cabrera Aldaya and Billy Bob Brumley. "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020 (2020), pp. 196–221.

[Aba+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity." In: *Conference on Computer and Communications Security – CCS'05*. ACM, 2005, pp. 340–353. ISBN: 1-59593-226-7.

[Aci07] Onur Aciiçmez. "Yet another MicroArchitectural Attack: : exploiting I-Cache." In: *Computer Security Architecture Workshop – CSAW*. ACM, 2007, pp. 11–18. ISBN: 978-1-59593-890-9.

[Aga00] Johan Agat. "Transforming Out Timing Leaks." In: *Principles of Programming Languages – POPL'00*. ACM, 2000, pp. 40–53. ISBN: 1-58113-125-9.

[AGS07] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures." In: *Cryptography and Coding – IMA'07*. Vol. 4887. LNCS. Springer, 2007, pp. 185–203. ISBN: 978-3-540-77271-2.

[Ahm+19] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. "OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX." In: *Network and Distributed System Security Symposium – NDSS'19*. The Internet Society, 2019. ISBN: 1-891562-55-X.

[AHP18] Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. "The Meaning of Memory Safety." In: *Principles of Security and Trust – POST'18*. Vol. 10804. LNCS. Springer, 2018, pp. 79–105. ISBN: 978-3-319-89721-9.

[AKS07a] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "On the Power of Simple Branch Prediction Analysis." In: *Asia Conference on Computer and Communications Security – AsiaCCS 2007*. ACM, 2007, pp. 312–320.

[AKS07b] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "Predicting Secret Keys Via Branch Prediction." In: *Topics in Cryptology – CT-RSA'07*. Vol. 4377. LNCS. Springer, 2007, pp. 225–242. ISBN: 3-540-69327-0.

[Ald+18] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit." In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 1060.

[Ald+19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. "Cache-Timing Attacks on RSA Key Generation." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019 (2019), pp. 213–242.

[All+16] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. "Amplifying side channels through performance degradation." In: *Annual Computer Security Applications Conference – ACSAC'16*. ACM, 2016, pp. 422–435. ISBN: 978-1-4503-4771-6.

[Alm+13] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. "Formal verification of side-channel countermeasures using self-composition." In: *Sci. Comput. Program.* 78 (2013), pp. 796–812.

[Alm+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations." In: *USENIX Security Symposium'16*. USENIX Association, 2016, pp. 53–70.

[Alm+17] José Bacelar Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography." In: *Conference on Computer and Communications Security – CCS'17*. ACM, 2017, pp. 1807–1823. ISBN: 978-1-4503-4946-8.

[AN17] Shaizeen Aga and Satish Narayanasamy. "InvisiMem: Smart Memory Defenses for Memory Bus Side Channel." In: *International Symposium on Computer Architecture – ISCA'17*. ACM, 2017, pp. 94–106. ISBN: 978-1-4503-4892-8.

[AN19] Shaizeen Aga and Satish Narayanasamy. "InvisiPage: oblivious demand paging for secure enclaves." In: *International Symposium on Computer Architecture – ISCA'19*. ACM, 2019, pp. 372–384. ISBN: 978-1-4503-6669-4.

[Ana+13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative Technology for CPU Based Attestation and Sealing." In: *Hardware and Architectural Support for Security and Privacy*. Vol. 13. HASP'13. Aug. 2013.

[Ana+15] Ittai Anati, Frank McKeen, Shay Gueron, Haitao Huang, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos V. Rozas, and Hisham Shafi. *Intel Software Guard Extensions (Intel SGX)*. Tutorial Slides presented at ICSA 2015. https://software.intel.com/sites/default/files/332680-002.pdf. (Accessed 2020/03/11). 2015.

[And+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "On Subnormal Floating Point and Abnormal Timing." In: *IEEE Symposium on Security and Privacy – S&P'15*. IEEE Computer Society, 2015, pp. 623–639. ISBN: 978-1-4673-6949-7.

[ANZ11] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms." In: *Conference on Computer and Communications Security – CCS'11*. ACM, 2011, pp. 375–388. ISBN: 978-1-4503-0948-6.

[Ara+14] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. "GLV/GLS Decomposition, Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias." In: *Advances in Cryptology – ASIACRYPT'14*. Vol. 8873. LNCS. Springer, 2014, pp. 262–281. ISBN: 978-3-662-45610-1.

[Arb05] William Arbaugh. *Treacherous or Trusted Computing: Black Helicopters, an Increase in Assurance, or Both?* Invited Talk at USENIX Security'05. 2005.

[Arn+16] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX." In: *Operating Systems Design and Implementation – OSDI'16*. USENIX Association, 2016, pp. 689–703.

[AS08] Onur Aciiçmez and Werner Schindler. "A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL." In: *Topics in Cryptology – CT-RSA'08*. Vol. 4964. LNCS. Springer, 2008, pp. 256–273. ISBN: 978-3-540-79262-8.

[AT07] Sarang Aravamuthan and Viswanatha Rao Thumparthy. "A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks." In: *Communication System Software and Middleware – COMSWARE'07*. IEEE, 2007. ISBN: 1-4244-0614-5.

[Awa+17] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories." In: *International Symposium on Computer Architecture – ISCA'17*. ACM, 2017, pp. 107–119. ISBN: 978-1-4503-4892-8.

[Bab86] László Babai. "On Lovász' lattice reduction and the nearest lattice point problem." In: *Combinatorica* 6.1 (1986), pp. 1–13.

[Bac+10] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. "Acoustic Side-Channel Attacks on Printers." In: *USENIX Security Symposium'10*. USENIX Association, 2010, pp. 307–322.

[Bar+14] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. "System-level Non-interference for Constant-time Cryptography." In: *Conference on Computer and Communications Security – CCS'14*. ACM, 2014, pp. 1267–1279. ISBN: 978-1-4503-2957-6.

[Bar+20] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. "Formal verification of a constant-time preserving C compiler." In: *PACMPL* 4 (2020), 7:1–7:30.

[Bar16] Elaine Barker. *NIST. Recommendation for Key Management, Part 1: General*. http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4. (Accessed 2020/03/11). 2016.

[Bau+14] Aurélie Bauer, Éliane Jaulmes, Victor Lomné, Emmanuel Prouff, and Thomas Roche. "Side-Channel Attack against RSA Key Generation Algorithms." In: *Cryptographic Hardware and Embedded Systems – CHES'14*. Vol. 8731. LNCS. Springer, 2014, pp. 223–241. ISBN: 978-3-662-44708-6.

[BB14] Erik Bosman and Herbert Bos. "Framing Signals - A Return to Portable Shellcode." In: *IEEE Symposium on Security and Privacy – S&P'14*. IEEE Computer Society, 2014, pp. 243–258. ISBN: 978-1-4799-4686-0.

[BBS18] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. "If This Then What?: Controlling Flows in IoT Apps." In: *Conference on Computer and Communications Security – CCS'18*. ACM, 2018, pp. 1102–1119. ISBN: 978-1-4503-5693-0.

[BC17] Tiyash Basu and Sudipta Chattopadhyay. "Testing Cache Side-Channel Leakage." In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 51–60. ISBN: 978-1-5090-6676-6.

[BC89] Jurjen N. Bos and Matthijs J. Coster. "Addition Chain Heuristics." In: *Advances in Cryptology – CRYPTO'89*. Vol. 435. LNCS. Springer, 1989, pp. 400–407. ISBN: 3-540-97317-6.

[BDJ19] Frédéric Besson, Alexandre Dang, and Thomas P. Jensen. "Information-Flow Preservation in Compiler Optimisations." In: *Computer Security Foundations – CSF'19*. IEEE, 2019, pp. 230–242. ISBN: 978-1-7281-1407-1.

[BDL91] Jørgen Brandt, Ivan Damgård, and Peter Landrock. "Speeding up Prime Number Generation." In: *Advances in Cryptology – ASIACRYPT'91*. Vol. 739. LNCS. Springer, 1991, pp. 440–449. ISBN: 3-540-57332-1.

[Bee15] Jethro Beekman. *Intel Has Full Control over SGX*. https://jbeekman.nl/blog/2015/10/intel-has-full-control-over-sgx/. (Accessed 2020/03/11). Oct. 2015.

[Ben+] Jeremy Bennett, Andrew Burgess, Simon Cook, Kerstin Eder, Simon Hollis, and James Pallister. *Bristol/Embecosm Embedded Benchmark Suite*. http://beebs.eu/. (Accessed 2020/03/11).

[Ben+14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. ""Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way." In: *Cryptographic Hardware and Embedded Systems – CHES'14*. Vol. 8731. LNCS. Springer, 2014, pp. 75–92. ISBN: 978-3-662-44708-6.

[Ber+17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. "Sliding Right into Disaster: Left-to-Right Sliding Windows Leak." In: *Cryptographic Hardware and Embedded Systems – CHES'17*. Vol. 10529. LNCS. Springer, 2017, pp. 555–576. ISBN: 978-3-319-66786-7.

[Ber05] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. (Accessed 2020/03/11). 2005.

[BFM14] Alex Bradbury, Gavin Ferris, and Robert Mullins. *Tagged Memory and Minion Cores in the lowRISC SoC*. lowRISC-MEMO 2014-001. 2014.

[BGL18] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"." In: *Computer Security Foundations – CSF'18*. IEEE Computer Society, 2018, pp. 328–343. ISBN: 978-1-5386-6680-7.

[BGM97] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. ""Pseudo-Random" Number Generation Within Cryptographic Algorithms: The DDS Case." In: *Advances in Cryptology – CRYPTO'97*. Vol. 1294. LNCS. Springer, 1997, pp. 277–291. ISBN: 3-540-63384-7.

[BH09] Billy Bob Brumley and Risto M. Hakala. "Cache-Timing Template Attacks." In: *Advances in Cryptology – ASIACRYPT'09*. Vol. 5912. LNCS. Springer, 2009, pp. 667–684. ISBN: 978-3-642-10365-0.

[BH19] Joachim Breitner and Nadia Heninger. "Biased Nonce Sense: Lattice Attacks Against Weak ECDSA Signatures in Cryptocurrencies." In: *Financial Cryptography – FC'19*. Vol. 11598. LNCS. Springer, 2019, pp. 3–20. ISBN: 978-3-030-32100-0.

[Bib77] K. J. Biba. *Integrity Considerations for Secure Computer Systems*. Tech Report. The MITRE Corporation. Tech. Report ESD-TR-76-372. 1977.

[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'18*. USENIX Association, 2018, pp. 1213–1227.

[Bit+14] Andrea Bittau, Adam Belay, Ali José Mashtizadeh, David Mazières, and Dan Boneh. "Hacking Blind." In: *IEEE Symposium on Security and Privacy – S&P'14*. IEEE Computer Society, 2014, pp. 227–242. ISBN: 978-1-4799-4686-0.

[BL16] Erick Bauman and Zhiqiang Lin. "A Case for Protecting Computer Games With SGX." In: *System Software for Trusted Execution – SysTEX*. ACM, 2016, 4:1–4:6. ISBN: 978-1-4503-4670-2.

[Ble+11] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2011, pp. 30–40. ISBN: 978-1-4503-0564-8.

[Ble00] Daniel Bleichenbacher. "On the generation of one-time keys in DL signature schemes." In: *Presentation at IEEE P1363 working group meeting*. 2000, p. 81.

[Ble98] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1." In: *Advances in Cryptology – CRYPTO'98*. Vol. 1462. LNCS. Springer, 1998, pp. 1–12. ISBN: 3-540-64892-5.

[BLS] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. *NaCl: Networking and Cryptography library*. https://nacl.cr.yp.to/. (Accessed 2020/03/11).

[Bon+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. "Vale: Verifying High-Performance Cryptographic Assembly Code." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 917–934.

[Bon99] Dan Boneh. "Twenty Years of Attacks on the RSA Cryptosystem." In: *Notices of the American Mathematical Society (AMS)* 46 (1999), pp. 203–213.

[Bos+16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. "Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough." In: *Cryptographic Hardware and Embedded Systems – CHES'16*. Vol. 9813. LNCS. Springer, 2016, pp. 215–236. ISBN: 978-3-662-53139-6.

[BP07] Kevin Borders and Atul Prakash. "Securing Network Input via a Trusted Input Proxy." In: *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association, 2007.

[BPH14] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. "Shielding Applications from an Untrusted Cloud with Haven." In: *Operating Systems Design and Implementation – OSDI'14*. USENIX Association, 2014, pp. 267–283.

[BPH15] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. "Shielding Applications from an Untrusted Cloud with Haven." In: *ACM Trans. Comput. Syst.* 33 (2015), 8:1–8:26.

[BPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control." In: *System Software for Trusted Execution – SysTEX*. ACM, 2017, 4:1–4:6. ISBN: 978-1-4503-5097-6.

[BPS18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic." In: *Conference on Computer and Communications Security – CCS'18*. ACM, 2018, pp. 178–195. ISBN: 978-1-4503-5693-0.

[BPT19] Sandrine Blazy, David Pichardie, and Alix Trieu. "Verifying constant-time implementations by abstract interpretation." In: *Journal of Computer Security* 27 (2019), pp. 137–163.

[BR15] Elaine Barker and Allen Roginsky. *NIST. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. http://doi.org/10.6028/NIST.SP.800-131Ar1. (Accessed 2020/03/11). NIST Special Publication 800-131A, Revision 1. 2015.

[BR98] Mihir Bellare and Phillip Rogaway. "PSS: Provably secure encoding method for digital signatures." In: *Submission to IEEE P1363* (1998).

[Bra+11] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation." In: *;login:* 36 (2011).

[Bra+15] Franz Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. "TyTAN: tiny trust anchor for tiny devices." In: *Design Automation Conference – DAC'15*. ACM, 2015, 34:1–34:6. ISBN: 978-1-4503-3520-1.

[Bra+17a] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. "Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory." In: *Dependable Systems and Networks – DSN'17*. IEEE Computer Society, 2017, pp. 157–168. ISBN: 978-1-5386-0542-4.

[Bra+17b] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *Workshop on Offensive Technologies – WOOT'17*. USENIX Association, 2017.

[Bra+19] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. "DR.SGX: automated and adjustable side-channel protection for SGX using data location randomization." In: *Annual Computer Security Applications Conference – ACSAC'19*. ACM, 2019, pp. 788–800. ISBN: 978-1-4503-7628-0.

[Bre+17] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. "Secure Cloud Micro Services Using Intel SGX." In: *Distributed Applications and Interoperable Systems – DAIS'17*. Vol. 10320. LNCS. Springer, 2017, pp. 177–191. ISBN: 978-3-319-59664-8.

[BRW06] Gilles Barthe, Tamara Rezk, and Martijn Warnier. "Preventing Timing Leaks Through Transactional Branching Instructions." In: *Electron. Notes Theor. Comput. Sci.* 153 (2006), pp. 33–55.

[BT11] Billy Bob Brumley and Nicola Tuveri. "Remote Timing Attacks Are Still Practical." In: *European Symposium on Research in Computer Security – ESORICS'11*. Vol. 6879. LNCS. Springer, 2011, pp. 355–371. ISBN: 978-3-642-23821-5.

[BT17] Anthony Brandon and Michael Trimarchi. "Trusted display and input using screen overlays." In: *ReConFigurable Computing and FPGAs – ReConFig'17*. IEEE, 2017, pp. 1–6. ISBN: 978-1-5386-3797-5.

[Bul+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 1041–1056.

[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'18*. USENIX Association, 2018, pp. 991–1008.

[Bul+19] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes." In: *Conference on Computer and Communications Security – CCS'19*. ACM, 2019, pp. 1741–1758. ISBN: 978-1-4503-6747-9.

[BV96] Dan Boneh and Ramarathnam Venkatesan. "Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes." In: *Advances in Cryptology – CRYPTO'96*. Vol. 1109. LNCS. Springer, 1996, pp. 129–142. ISBN: 3-540-61512-1.

[BV97] Dan Boneh and Ramarathnam Venkatesan. "Rounding in Lattices and its Cryptographic Applications." In: *Symposium on Discrete Algorithms – SODA'97*. ACM/SIAM, 1997, pp. 675–681. ISBN: 0-89871-390-0.

[BW12] R. Boivie and P. Williams. *SecureBlue++: CPU Support for Secure Executables*. IBM research report no. RC25369. http://domino.research.ibm.com/library/cyberdig.nsf/papers/BE73A643EFE8274B85257B51006760C0. (Accessed 2020/03/11). 2012.

[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'15*. USENIX Association, 2015, pp. 161–176.

[CC07] Christophe Clavier and Jean-Sébastien Coron. "On the Implementation of a Fast Prime Generation Algorithm." In: *Cryptographic Hardware and Embedded Systems – CHES'07*. Vol. 4727. LNCS. Springer, 2007, pp. 443–449. ISBN: 978-3-540-74734-5.

[CCS12] Jeroen Van Cleemput, Bart Coppens, and Bjorn De Sutter. "Compiler mitigations for time attacks on modern x86 processors." In: *TACO* 8 (2012), 23:1–23:20.

[CD16] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[CDA14] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. "Virtual ghost: protecting applications from hostile operating systems." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'14*. ACM, 2014, pp. 81–96. ISBN: 978-1-4503-2305-5.

[CFD17] Jia Chen, Yu Feng, and Isil Dillig. "Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic." In: *Conference on Computer and Communications Security – CCS'17*. ACM, 2017, pp. 875–890. ISBN: 978-1-4503-4946-8.

[CH01] Tzi-cker Chiueh and Fu-Hau Hsu. "RAD: A Compile-Time Solution to Buffer Overflow Attacks." In: *International Conference on Distributed Computing Systems – ICDCS'01*. IEEE Computer Society, 2001, pp. 409–417. ISBN: 0-7695-1077-9.

[Cha+17a] Somnath Chakrabarti, Rebekah Leslie-Hurd, Mona Vij, Frank McKeen, Carlos V. Rozas, Dror Caspi, Ilya Alexandrovich, and Ittai Anati. "Intel® Software Guard Extensions (Intel® SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment." In: *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 2017, 7:1–7:8. ISBN: 978-1-4503-5266-6.

[Cha+17b] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. "Quantifying the information leak in cache attacks via symbolic execution." In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*. ACM, 2017, pp. 25–35. ISBN: 978-1-4503-5093-8.

[Che+08] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'08*. ACM, 2008, pp. 2–13. ISBN: 978-1-59593-958-6.

[Che+09] Shimin Chen et al. "Flexible Hardware Acceleration for Instruction-Grain Lifeguards." In: *IEEE Micro* 29 (2009), pp. 62–72.

[Che+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns." In: *Conference on Computer and Communications Security – CCS'10*. ACM, 2010, pp. 559–572. ISBN: 978-1-4503-0245-6.

[Che+17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2017, pp. 7–18. ISBN: 978-1-4503-4944-4.

[Che+18] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races." In: *IEEE Symposium on Security and Privacy – S&P'18*. IEEE Computer Society, 2018, pp. 178–194. ISBN: 978-1-5386-4353-2.

[Che+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution." In: *IEEE European Symposium on Security and Privacy – EURO S&P'19*. IEEE, 2019, pp. 142–157. ISBN: 978-1-7281-1148-3.

[Chh+11] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. "SecureME: a hardware-software approach to full system security." In: *International Conference on Supercomputing – ICS'11*. ACM, 2011, pp. 108–119. ISBN: 978-1-4503-0102-2.

[CL10] David Champagne and Ruby B. Lee. "Scalable architectural support for trusted software." In: *High Performance Computer Architecture – HPCA'10*. IEEE Computer Society, 2010, pp. 1–12. ISBN: 978-1-4244-5659-8.

[CLD16] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation." In: *USENIX Security Symposium'16*. USENIX Association, 2016, pp. 857–874.

[Cod98] Coderpunks. *Rebutal to Schnorr's Patent Claims re DSA*. http://web.archive.org/web/20010829223720/http://www.privacy.nb.ca/cryptography/archives/coderpunks/new/1998-08/0009.html. (Accessed 2020/03/11). 1998.

[Coh+09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A Practical System for Verifying Concurrent C." In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Vol. 5674. LNCS. Springer, 2009, pp. 23–42. ISBN: 978-3-642-03358-2.

[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'09*. IEEE Computer Society, 2009, pp. 45–60. ISBN: 978-0-7695-3633-0.

[Cor18a] Jonathan Corbet. *Coscheduling: simultaneous scheduling in control groups*. https://lwn.net/Articles/764482/. (Accessed 2020/04/03). 2018.

[Cor18b] Jonathan Corbet. *The current state of kernel page-table isolation*. https://lwn.net/Articles/741878/. (Accessed 2020/03/31). 2018.

[Cos+17] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. "The Pyramid Scheme: Oblivious RAM for Trusted Processors." In: *CoRR* abs/1712.07882 (2017).

[Cow98] Crispan Cowan. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In: *USENIX Security Symposium'98*. USENIX Association, 1998.

[CP11] William D. Casper and Stephen M. Papa. "Trusted Boot." In: *Encyclopedia of Cryptography and Security, 2nd Ed.* 2011, pp. 1327–1328.

[Cra+15] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. "Readactor: Practical Code Randomization Resilient to Memory Disclosure." In: *IEEE Symposium on Security and Privacy – S&P'15*. IEEE Computer Society, 2015, pp. 763–780. ISBN: 978-1-4673-6949-7.

[CS13] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'13*. ACM, 2013, pp. 253–264. ISBN: 978-1-4503-1870-9.

[CTZ17] Ju Chen, Yuzhe Richard Tang, and Hao Zhou. "Strongly Secure and Efficient Data Shuffle on Hardware Enclaves." In: *System Software for Trusted Execution – SysTEX*. ACM, 2017, 1:1–1:6. ISBN: 978-1-4503-5097-6.

[CW14] Nicholas Carlini and David A. Wagner. "ROP is Still Dangerous: Breaking Modern Defenses." In: *USENIX Security Symposium'14*. USENIX Association, 2014, pp. 385–399.

[CWC06] Jedidiah R. Crandall, Shyhtsun Felix Wu, and Frederic T. Chong. "Minos: Architectural support for protecting control data." In: *TACO* 3 (2006), pp. 359–389.

[Dal+18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018 (2018), pp. 171–191.

[Dal19] Paul Dale. *Close side channels in DSA and ECDSA*. OpenSSL Pull Request #8906, https://github.com/openssl/openssl/pull/8906. (Accessed 2020/03/11). 2019.

[Dan18] Janis Danisevskis. *Android Protected Confirmation: Taking transaction security to the next level*. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html (Accessed 2020/03/18). 2018.

[Dev+08] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. "Hardbound: architectural support for spatial safety of the C programming language." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'08*. ACM, 2008, pp. 103–114. ISBN: 978-1-59593-958-6.

[DF14] Shaun Davenport and Richard Ford. *SGX: the good, the bad and the downright ugly*. https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly. (Accessed 2020/03/11). Jan. 2014.

[DFS20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments." In: *USENIX Security Symposium'20*. 2020.

[Dha+15] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and André DeHon. "Architectural Support for Software-Defined Metadata Processing." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'15*. ACM, 2015, pp. 487–502. ISBN: 978-1-4503-2835-7.

[Dha+17] Aritra Dhar, Der-Yeuan Yu, Kari Kostiainen, and Srdjan Capkun. "IntegriKey: End-to-End Integrity Protection of User Input." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 1245.

[Dha+20a] Aritra Dhar, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. "ProtectIOn: Root-of-Trust for IO in Compromised Platforms." In: *Network and Distributed System Security Symposium – NDSS'20*. 2020.

[Dha+20b] Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun. "ProximiTEE: Hardened SGX Attestation by Proximity Verification." In: *Conference on Data and Application Security and Privacy – CODASPY'20*. ACM, 2020, pp. 5–16. ISBN: 978-1-4503-7107-0.

[DK17] Goran Doychev and Boris Köpf. "Rigorous analysis of software countermeasures against cache attacks." In: *Programming Language Design and Implementation – PLDI'17*. ACM, 2017, pp. 406–421. ISBN: 978-1-4503-4988-8.

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Raksha: a flexible information flow architecture for software security." In: *International Symposium on Computer Architecture – ISCA'07*. ACM, 2007, pp. 482–493. ISBN: 978-1-59593-706-3.

[Dom+12] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks." In: *TACO* 8 (2012), 35:1–35:21.

[Dom18] Christopher Domas. "GOD MODE UNLOCKED - Hardware Backdoors in X86 CPUs." In: *BlackHat USA* (2018).

[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'13*. USENIX Association, 2013, pp. 431–446. ISBN: 978-1-931971-03-4.

[DS12] Daniel Y. Deng and G. Edward Suh. "High-performance parallel accelerator for flexible and efficient run-time monitoring." In: *Dependable Systems and Networks – DSN'12*. IEEE Computer Society, 2012, pp. 1–12. ISBN: 978-1-4673-1624-8.

[DXS19] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. "Analysis of Secure Caches Using a Three-Step Model for Timing-Based Attacks." In: *J. Hardware and Systems Security* 3 (2019), pp. 397–425.

[Eck85] Wim van Eck. "Electromagnetic radiation from video display units: An eavesdropping risk?" In: *Computers & Security* 4.4 (1985), pp. 269–286.

[Edg13] Jake Edge. *Kernel address space layout randomization*. https://lwn.net/Articles/569635/. (Accessed 2020/03/11). 2013.

[Edg15] Jake Edge. *Android Verified Boot*. https://lwn.net/Articles/638627/. (Accessed 2020/03/11). 2015.

[EEM] EEMBC. *CoreMark*. https://www.eembc.org/coremark/. (Accessed 2020/03/11).

[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'12*. The Internet Society, 2012.

[Esk+19] Saba Eskandarian et al. "Fidelius: Protecting User Secrets from Compromised Browsers." In: *IEEE Symposium on Security and Privacy – S&P'19*. IEEE, 2019, pp. 264–280. ISBN: 978-1-5386-6660-9.

[Evt+14] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V. Ponomarev, Nael B. Abu-Ghazaleh, and Ryan Riley. "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution." In: *Symposium on Microarchitecture – MICRO'14*. IEEE Computer Society, 2014, pp. 190–202. ISBN: 978-1-4799-6998-2.

[Evt+18] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'18*. ACM, 2018, pp. 693–707.

[FDD12] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs." In: *Workshop on Scalable Trusted Computing – STC'12*. 2012, 3–8.

[Fer+14] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. "TIVOs: Trusted Visual I/O Paths for Android." In: *University of Michigan CSE Technical Report CSE-TR-586-14* (2014).

[Fer+17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. "Komodo: Using verification to disentangle secure-enclave hardware from software." In: *Symposium on Operating Systems Principles – SOSP'17*. ACM, 2017, pp. 287–305. ISBN: 978-1-4503-5085-3.

[Feu72] Edward A. Feustel. "The Rice research computer: a tagged architecture." In: *American Federation of Information Processing Societies – AFIPS*. Vol. 40. AFIPS Conference Proceedings. AFIPS, 1972, pp. 369–377.

[FGR12] Jean-Charles Faugère, Christopher Goyet, and Guénaël Renault. "Attacking (EC)DSA Given Only an Implicit Hint." In: *Selected Areas in Cryptography – SAC'12*. Vol. 7707. LNCS. Springer, 2012, pp. 252–274. ISBN: 978-3-642-35998-9.

[FGS09] Thomas Finke, Max Gebhardt, and Werner Schindler. "A New Side-Channel Attack on RSA Prime Generation." In: *Cryptographic Hardware and Embedded Systems – CHES'09*. Vol. 5747. LNCS. Springer, 2009, pp. 141–155. ISBN: 978-3-642-04137-2.

[FH05] Norman Feske and Christian Helmuth. "A Nitpicker's guide to a minimal-complexity secure GUI." In: *Annual Computer Security Applications Conference – ACSAC'05*. IEEE Computer Society, 2005, pp. 85–94. ISBN: 0-7695-2461-3.

[Fil+11] Atanas Filyanov, Jonathan M. McCune, Ahmad-Reza Sadeghi, and Marcel Winandy. "Uni-directional trusted path: Transaction confirmation on just one device." In: *Dependable Systems and Networks – DSN'11*. IEEE Compute Society, 2011, pp. 1–12. ISBN: 978-1-4244-9233-6.

[FS86] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems." In: *Advances in Cryptology – CRYPTO'86*. Vol. 263. LNCS. Springer, 1986, pp. 186–194.

[Fu+17] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. "Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults." In: *Recent Advances in Intrusion Detection – RAID'17*. Vol. 10453. LNCS. Springer, 2017, pp. 357–380. ISBN: 978-3-319-66331-9.

[FWC16] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. "Attacking OpenSSL Implementation of ECDSA with a Few Signatures." In: *Conference on Computer and Communications Security – CCS'16*. ACM, 2016, pp. 1505–1515. ISBN: 978-1-4503-4139-4.

[Gam84] Taher El Gamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." In: *Advances in Cryptology – CRYPTO'84*. Vol. 196. LNCS. Springer, 1984, pp. 10–18. ISBN: 3-540-15658-5.

[Gao+20] Si Gao, Ben Marshall, Dan Page, and Thinh Pham. "FENL: an ISE to mitigate analogue micro-architectural leakage." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020 (2020), pp. 73–98.

[Gar+03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. "Terra: a virtual machine-based platform for trusted computing." In: *Symposium on Operating Systems Principles – SOSP'03*. ACM, 2003, pp. 193–206. ISBN: 1-58113-757-5.

[Gaw] *The GNU Awk User's Guide*. Edition 4.2. https://www.gnu.org/software/gawk/manual/gawk.html. (Accessed 2020/03/11).

[GB17] Cesar Pereida García and Billy Bob Brumley. "Constant-Time Callees with Variable-Time Callers." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 83–98.

[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'11*. IEEE Computer Society, 2011, pp. 490–505. ISBN: 978-1-4577-0147-4.

[GBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “”Make Sure DSA Signing Exponentiations Really are Constant-Time”.” In: *Conference on Computer and Communications Security – CCS'16*. ACM, 2016, pp. 1639–1650. ISBN: 978-1-4503-4139-4.

[Ge+18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware.” In: *J. Cryptographic Engineering* 8 (2018), pp. 1–27.

[Ge+19] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. “Time Protection: The Missing OS Abstraction.” In: *European Conference on Computer Systems – EUROSYS'19*. ACM, 2019, 1:1–1:17. ISBN: 978-1-4503-6281-8.

[GEL16] Jason Gionta, William Enck, and Per Larsen. “Preventing kernel code-reuse attacks through disclosure resistant code diversification.” In: *Communications and Network Security – CNS'16*. IEEE, 2016, pp. 189–197. ISBN: 978-1-5090-3065-1.

[Gen+18] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. “K-Miner: Uncovering Memory Corruption in Linux.” In: *Network and Distributed System Security Symposium – NDSS'18*. The Internet Society, 2018.

[Geo+04] Loukas Georgiadis, Renato Fonseca F. Werneck, Robert Endre Tarjan, Spyridon Triantafyllis, and David I. August. “Finding Dominators in Practice.” In: *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*. Vol. 3221. LNCS. Springer, 2004, pp. 677–688. ISBN: 3-540-23025-4.

[GHG18] Daniel Gruss, Dave Hansen, and Brendan Gregg. “Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer.” In: *;login:* 43 (2018).

[Gje+17] Anders T. Gjerdrum, Robert Pettersen, Håvard D. Johansen, and Dag Johansen. “Performance of Trusted Computing in Cloud Infrastructures with Intel SGX.” In: *Conference on Cloud Computing and Services Science – CLOSER'17*. SciTePress, 2017, pp. 668–675. ISBN: 978-989-758-243-1.

[GLQ98] Tanguy Gilmont, Jean-Didier Legat, and Jean-Jacques Quisquater. “An architecture of Security Management Unit for Safe Hosting of Multiple Agents.” In: *International Workshop on Intelligent Communication and Multimedia Terminals*. 1998, pp. 79–82.

[GLQ99] Tanguy Gilmont, Jean-Didier Legat, and Jean-Jacques Quisquater. “Hardware security for software privacy support.” In: *Electronic Letters*. Vol. 35. 24. 1999, pp. 2096–2098.

[GO96] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs.” In: *J. ACM* 43 (1996), pp. 431–473.

[Gop+09] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. "Fast and Constant-Time Implementation of Modular Exponentiation." In: *Embedded Systems and Communications Security – ECSC 2009*. 2009.

[Gra+19] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. "IskiOS: Lightweight Defense Against Kernel-Level Code-Reuse Attacks." In: *CoRR* abs/1903.04654 (2019).

[Gri+19] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. "Triggerflow: Regression Testing by Advanced Execution Path Inspection." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'19*. Vol. 11543. LNCS. Springer, 2019, pp. 330–350. ISBN: 978-3-030-22037-2.

[Gro+09] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. "Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications." In: *Information Security and Cryptology – ICISC'09*. Vol. 5984. LNCS. Springer, 2009, pp. 176–192. ISBN: 978-3-642-14422-6.

[Gru+16] 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'16*. ACM, 2016, pp. 368–379. ISBN: 978-1-4503-4139-4.

[Gru+17] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 217–233.

[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'18*. IEEE Computer Society, 2018, pp. 245–261. ISBN: 978-1-5386-4353-2.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium'15*. USENIX Association, 2015, pp. 897–912.

[Gu+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels." In: *Operating Systems Design and Implementation – OSDI'16*. USENIX Association, 2016, pp. 653–669.

[Gue10] Shay Gueron. *White Paper: Intel Advanced Encryption Standard (AES) Instructions Set*. https://software.intel.com/file/24917. (Accessed 2020/03/11). 2010.

[Gue12] Shay Gueron. "Efficient software implementations of modular exponentiation." In: *J. Cryptographic Engineering* 2 (2012), pp. 31–43.

[Gue16] Shay Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 204.

[GYH18] Qian Ge, Yuval Yarom, and Gernot Heiser. "No Security Without Time Protection: We Need a New Hardware-Software Contract." In: *Asia-Pacific Workshop on Systems – APSys'18*. ACM, 2018, 1:1–1:9.

[Gys+18] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution." In: *Engineering Secure Software and Systems – ESSoS'18*. Vol. 10953. LNCS. Springer, 2018, pp. 44–60. ISBN: 978-3-319-94495-1.

[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'14*. IEEE Computer Society, 2014, pp. 575–589. ISBN: 978-1-4799-4686-0.

[Göt+15] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix C. Freiling, and Ingrid Verbauwhede. "Soteria: Offline Software Protection within Low-cost Embedded Devices." In: *Annual Computer Security Applications Conference – ACSAC'15*. ACM, 2015, pp. 241–250. ISBN: 978-1-4503-3682-6.

[Göt+17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX." In: *European Workshop on System Security – EUROSEC'17*. ACM, 2017, 2:1–2:6. ISBN: 978-1-4503-4935-2.

[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'08*. USENIX Association, 2008, pp. 45–60. ISBN: 978-1-931971-60-7.

[Har88] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)." In: *Operating Systems Review* 22 (1988), pp. 36–38.

[Hay19] Richard Hayton. *The Benefits of Trusted User Interface (TUI)*. https://www.trustonic.com/news/blog/benefits-trusted-user-interface/ (Accessed 2020/03/18). 2019.

[HCP17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems." In: *USENIX Annual Technical Conference – USENIX ATC'17*. USENIX Association, 2017, pp. 299–312.

[He+18] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. "SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution." In: *International Conference on Computer Design – ICCD'18*. IEEE Computer Society, 2018, pp. 108–114. ISBN: 978-1-5386-8477-1.

[Hei18] Gernot Heiser. "For Safety's Sake: We Need a New Hardware-Software Contract!" In: *IEEE Des. Test* 35 (2018), pp. 27–30.

[Hel17] Helpnetsecurity. *The cost of IoT hacks: Up to 13% of revenue for smaller firms*. https://www.helpnetsecurity.com/2017/06/05/iot-hacks-cost/. (Accessed 2020/03/11). 2017.

[Hoe+13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. "Using innovative instructions to create trustworthy software solutions." In: *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 2013, p. 11. ISBN: 978-1-4503-2118-1.

[Hof+13] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. "InkTag: secure applications on an untrusted operating system." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'13*. ACM, 2013, pp. 265–278. ISBN: 978-1-4503-1870-9.

[HR06] Martin Hlavác and Tomás Rosa. "Extended Hidden Number Problem and Its Cryptanalytic Applications." In: *Selected Areas in Cryptography – SAC'06*. Vol. 4356. LNCS. Springer, 2006, pp. 114–133. ISBN: 978-3-540-74461-0.

[HS01] Nick Howgrave-Graham and Nigel P. Smart. "Lattice Attacks on Digital Signature Schemes." In: *Des. Codes Cryptography* 23 (2001), pp. 283–290.

[HS05] Daniel Hedin and David Sands. "Timing Aware Information Flow Security for a JavaCard-like Bytecode." In: *Electron. Notes Theor. Comput. Sci.* 141 (2005), pp. 163–182.

[Hun+16] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data." In: *Operating Systems Design and Implementation – OSDI'16*. USENIX Association, 2016, pp. 533–549.

[Huo+20] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020 (2020), pp. 321–347.

[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'13*. IEEE Computer Society, 2013, pp. 191–205. ISBN: 978-1-4673-6166-8.

[Iga15] Benjamin Igal. *Bits, Please! Exploring Qualcomm's TrustZone implementation*. http://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html. (Accessed 2020/03/11). 2015.

[Ins98] American National Standards Institute. *Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)*. 1998.

[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: *CoRR* abs/1709.01552 (2017).

[Isp+18] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks." In: *Conference on Computer and Communications Security – CCS'18*. ACM, 2018, pp. 1868–1882. ISBN: 978-1-4503-5693-0.

[Jac+15] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. "Lightweight Capability Domains: Towards Decomposing the Linux Kernel." In: *Operating Systems Review* 49 (2015), pp. 44–50.

[Jan17] Yeongjin Jang. "Building trust in the user I/O in computer systems." PhD thesis. Georgia Institute of Technology, 2017.

[Jeo+19] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. "Razzer: Finding Kernel Race Bugs through Fuzzing." In: *IEEE Symposium on Security and Privacy – S&P'19*. IEEE, 2019, pp. 754–768. ISBN: 978-1-5386-6660-9.

[JL17] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. https://tools.ietf.org/html/rfc8032. (Accessed 2020/03/11). Request for Comments: 8032. 2017.

[Joa+17] Alexandre Joannou et al. "Efficient Tagged Memory." In: *International Conference on Computer Design – ICCD'17*. IEEE Computer Society, 2017, pp. 641–648. ISBN: 978-1-5386-2254-4.

[Joh+11] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. "Differential Slicing: Identifying Causal Execution Differences for Security Applications." In: *IEEE Symposium on Security and Privacy – S&P'11*. IEEE Computer Society, 2011, pp. 347–362. ISBN: 978-1-4577-0147-4.

[Joh18] Simon Johnson. *An update on 3rd Party Attestation*. https://software.intel.com/en-us/blogs/2018/12/09/an-update-on-3rd-party-attestation. (Accessed 2020/03/11). 2018.

[JP06] Marc Joye and Pascal Paillier. "Fast Generation of Prime Numbers on Portable Devices: An Update." In: *Cryptographic Hardware and Embedded Systems – CHES'06*. Vol. 4249. LNCS. Springer, 2006, pp. 160–173. ISBN: 3-540-46559-6.

[JZD16] Simon Johnson, Dan Zimmerman, and B. Derek. *Intel SGX: Debug, Production, Pre-Release What's the Difference?* https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference. (Accessed 2020/03/11). Jan. 2016.

[Kal98] Burton S. Kaliski. *PKCS #1: RSA Encryption Version 1.5*. https://tools.ietf.org/html/rfc2313. (Accessed 2020/03/11). Request for Comments: 2313. 1998.

[KDK09] Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor." In: *Dependable Systems and Networks – DSN'09*. IEEE Computer Society, 2009, pp. 105–114. ISBN: 978-1-4244-4422-9.

[KG13] Cameron F Kerry and Patrick D Gallagher. "Digital Signature Standard (DSS); FIPS PUB 186-4." In: *Information Technology Laboratory, National Institute of Standards and Technology: Gaithersburg, MD, USA* (2013).

[Kha+20] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. "COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'20*. ACM, 2020, pp. 971–985. ISBN: 978-1-4503-7102-5.

[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'14*. IEEE Computer Society, 2014, pp. 361–372. ISBN: 978-1-4799-4396-8.

[Kim+19] Deokjin Kim, DaeHee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent ByungHoon Kang. "SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure." In: *Comput. Secur.* 82 (2019), pp. 118–139.

[Kin+06] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. "SubVirt: Implementing malware with virtual machines." In: *IEEE Symposium on Security and Privacy – S&P'06*. IEEE Computer Society, 2006, pp. 314–327. ISBN: 0-7695-2574-1.

[Kir19] Felix Kirchengast. *Secure Network Interface with SGX*. Graz University of Technology. Master thesis. https://github.com/fkirc/secure-network-interface-with-sgx/raw/master/thesis.pdf. (Accessed 2020/03/16). 2019.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Advances in Cryptology – CRYPTO'99*. Vol. 1666. LNCS. Springer, 1999, pp. 388–397. ISBN: 3-540-66347-9.

[Kle+09] Gerwin Klein et al. "seL4: formal verification of an OS kernel." In: *Symposium on Operating Systems Principles – SOSP'09*. ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3.

[Kle+10] Gerwin Klein et al. "seL4: formal verification of an operating-system kernel." In: *Commun. ACM* 53 (2010), pp. 107–115.

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive formal verification of an OS microkernel." In: *ACM Trans. Comput. Syst.* 32 (2014), 2:1–2:70.

[Kle+18] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby C. Murray, and Gernot Heiser. "Formally verified software in the real world." In: *Commun. ACM* 61 (2018), pp. 68–77.

[KMO12] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. "Automatic Quantification of Cache Side-Channels." In: *Computer Aided Verification – CAV'12*. Vol. 7358. LNCS. Springer, 2012, pp. 564–580. ISBN: 978-3-642-31423-0.

[Knu11] Nick Knupffer. *Intel Insider – What Is It? (Is It DRM? And Yes It Delivers Top Quality Movies to Your PC)*. https://blogs.intel.com/technology/2011/01/intel_insider_-_what_is_it_no/. (Accessed 2020/03/11). Jan. 2011.

[Koc+19] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *IEEE Symposium on Security and Privacy – S&P'19*. IEEE, 2019, pp. 1–19. ISBN: 978-1-5386-6660-9.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Advances in Cryptology – CRYPTO'96*. Vol. 1109. LNCS. Springer, 1996, pp. 104–113. ISBN: 3-540-61512-1.

[Koe+14] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. "TrustLite: a security architecture for tiny embedded devices." In: *European Conference on Computer Systems – EUROSYS'14*. ACM, 2014, 10:1–10:14. ISBN: 978-1-4503-2704-6.

[Kon+08] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. "Deconstructing new cache designs for thwarting software cache-based side channel attacks." In: *Computer Security Architecture Workshop – CSAW*. ACM, 2008, pp. 25–34. ISBN: 978-1-60558-300-6.

[Kor+18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer." In: *Workshop on Offensive Technologies – WOOT'18*. USENIX Association, 2018.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*. White paper. 2016.

[Kra91] David W. Kravitz. "Digital signature algorithm." Pat. U.S. Patent 5231668A. 1991.

[KS09] Emilia Käsper and Peter Schwabe. "Faster and Timing-Attack Resistant AES-GCM." In: *Cryptographic Hardware and Embedded Systems – CHES'09*. Vol. 5747. LNCS. Springer, 2009, pp. 1–17. ISBN: 978-3-642-04137-2.

[Kuv+17] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory Safety for Shielded Execution." In: *European Conference on Computer Systems – EUROSYS'17*. ACM, 2017, pp. 205–221. ISBN: 978-1-4503-4938-3.

[Kuz+14] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity." In: *Operating Systems Design and Implementation – OSDI'14*. USENIX Association, 2014, pp. 147–163.

[Kön08] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES." In: *Topics in Cryptology – CT-RSA'08*. Vol. 4964. LNCS. Springer, 2008, pp. 187–202. ISBN: 978-3-540-79262-8.

[Lan+15] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. "Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses." In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 2015, pp. 190–197. ISBN: 978-1-4673-7952-6.

[Lan10] Adam Langley. *ctgrind: Checking that Functions are Constant Time with Valgrind*. https://github.com/agl/ctgrind. (Accessed 2020/03/11). 2010.

[Lar17] Selena Larson. *FDA confirms that St. Jude's cardiac devices can be hacked*. https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack/. (Accessed 2020/03/11). 2017.

[Lee+17a] 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'17*. USENIX Association, 2017, pp. 523–539.

[Lee+17b] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 557–574.

[Lee+19] Sangyub Lee, Sung Min Cho, Heeseok Kim, and Seokhie Hong. "A Practical Collision-Based Power Analysis on RSA Prime Generation and Its Countermeasure." In: *IEEE Access* 7 (2019), pp. 47582–47592.

[Lee+20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. "Keystone: An Open Framework for Architecting TEEs." In: *European Conference on Computer Systems – EuroSys'20*. 2020.

[Li+14] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. "Building trusted path on untrusted device drivers for mobile devices." In: *Asia-Pacific Workshop on Systems – APSys'14*. ACM, 2014, 8:1–8:7. ISBN: 978-1-4503-3024-4.

[Li+18] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. "VButton: Practical Attestation of User-driven Operations in Mobile Apps." In: *Mobile Systems – MobiSys'18*. ACM, 2018, pp. 28–40.

[Lia+17] Xueping Liang, Sachin Shetty, Deepak K. Tosh, Charles A. Kamhoua, Kevin A. Kwiat, and Laurent Njilla. "ProvChain: A Blockchain-based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability." In: *International Symposium on Cluster, Cloud and Grid Computing – CCGRID'17*. IEEE Computer Society / ACM, 2017, pp. 468–477. ISBN: 978-1-5090-6610-0.

[Lia+18] Weixin Liang, Kai Bu, Ke Li, Jinhong Li, and Arya Tavakoli. "MemCloak: Practical Access Obfuscation for Untrusted Memory." In: *Annual Computer Security Applications Conference – ACSAC'18*. ACM, 2018, pp. 187–197. ISBN: 978-1-4503-6569-7.

[Lie+00] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. "Architectural Support for Copy and Tamper Resistant Software." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'00*. ACM Press, 2000, pp. 168–177. ISBN: 1-58113-317-0.

[Lin+16] Joshua Lind, Ittay Eyal, Peter R. Pietzuch, and Emin Gün Sirer. "Teechan: Payment Channels Using Trusted Execution Environments." In: *CoRR* abs/1612.07766 (2016).

[Lip+18] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium'18*. USENIX Association, 2018, pp. 973–990.

[Liu+13] Dongtao Liu, Eduardo Cuervo, Valentin Pistol, Ryan Scudellari, and Landon P. Cox. "ScreenPass: secure password entry on touchscreen devices." In: *Mobile Systems – MobiSys'13*. ACM, 2013, pp. 291–304. ISBN: 978-1-4503-1672-9.

[Liu+15a] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'15*. ACM, 2015, pp. 87–101. ISBN: 978-1-4503-2835-7.

[Liu+15b] 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'15*. IEEE Computer Society, 2015, pp. 605–622. ISBN: 978-1-4673-6949-7.

[Liu+16] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. "CATalyst: Defeating last-level cache side channel attacks in cloud computing." In: *High Performance Computer Architecture – HPCA'16*. IEEE Computer Society, 2016, pp. 406–418. ISBN: 978-1-4673-9211-2.

[LL13] Matthias Lange and Steffen Liebergeld. "Crossover: secure and usable user interface for mobile devices with multiple isolated OS personalities." In: *Annual Computer Security Applications Conference – ACSAC'13*. ACM, 2013, pp. 249–257. ISBN: 978-1-4503-2015-3.

[LL14] Fangfei Liu and Ruby B. Lee. "Random Fill Cache Architecture." In: *Symposium on Microarchitecture – MICRO'14*. IEEE Computer Society, 2014, pp. 203–215. ISBN: 978-1-4799-6998-2.

[LLL82] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. "Factoring polynomials with rational coefficients." In: *Mathematische Annalen* 261.4 (1982), pp. 515–534.

[LPW19a] Kangjie Lu, Aditya Pakki, and Qiushi Wu. "Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs." In: *European Symposium on Research in Computer Security – ESORICS'19*. Vol. 11736. LNCS. Springer, 2019, pp. 3–25. ISBN: 978-3-030-29961-3.

[LPW19b] Kangjie Lu, Aditya Pakki, and Qiushi Wu. "Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences." In: *USENIX Security Symposium'19*. USENIX Association, 2019, pp. 1769–1786.

[LS09] Dirk Leinenbach and Thomas Santen. "Verifying the Microsoft Hyper-V Hypervisor with VCC." In: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. Vol. 5850. LNCS. Springer, 2009, pp. 806–809. ISBN: 978-3-642-05088-6.

[Luk+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation." In: *Programming Language Design and Implementation – PLDI'05*. ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6.

[Maa+13] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. "PHANTOM: practical oblivious computation in a secure processor." In: *Conference on Computer and Communications Security – CCS'13*. ACM, 2013, pp. 311–324. ISBN: 978-1-4503-2477-9.

[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.

[Mai+13] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. "Verifying security invariants in ExpressOS." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'13*. ACM, 2013, pp. 293–304. ISBN: 978-1-4503-1870-9.

[MAP18] Spyridon Mastorakis, Tahrina Ahmed, and Jayaprakash Pisharath. "ISA-Based Trusted Network Functions And Server Applications In The Untrusted Cloud." In: *CoRR* abs/1802.06970 (2018).

[Mar+16] Claudio Marforio, Ramya Jayaram Masti, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. "Evaluation of Personalized Security Indicators as an Anti-Phishing Mechanism for Smartphone Applications." In: *Conference on Human Factors in Computing Systems – CHI'16*. ACM, 2016, pp. 540–551. ISBN: 978-1-4503-3362-7.

[Mar18] Marion Marschalek. "The Wolf In SGX Clothing." In: *Bluehat IL* (Jan. 2018). https://www.youtube.com/watch?v=zVakbneMYbY. (Accessed 2020/03/11).

[Mas+15] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. "CCFI: Cryptographically Enforced Control Flow Integrity." In: *Conference on Computer and Communications Security – CCS'15*. ACM, 2015, pp. 941–951. ISBN: 978-1-4503-3832-5.

[Mat+17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. "ROTE: Rollback Protection for Trusted Execution." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 1289–1306.

[McC+08] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. "Flicker: an execution infrastructure for tcb minimization." In: *European Conference on Computer Systems – EUROSYS'08*. ACM, 2008, pp. 315–328. ISBN: 978-1-60558-013-5.

[McC+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation." In: *IEEE Symposium on Security and Privacy – S&P'10*. IEEE Computer Society, 2010, pp. 143–158. ISBN: 978-0-7695-4035-1.

[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: *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 2013, p. 10. ISBN: 978-1-4503-2118-1.

[Mer14] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment." In: *Linux Journal* (2014).

[MES18] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX." In: *Topics in Cryptology – CT-RSA'18*. Vol. 10808. LNCS. Springer, 2018, pp. 21–44. ISBN: 978-3-319-76952-3.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks." In: *Cryptographic Hardware and Embedded Systems – CHES'17*. Vol. 10529. LNCS. Springer, 2017, pp. 69–90. ISBN: 978-3-319-66786-7.

[ML89] John M. Mellor-Crummey and Thomas J. LeBlanc. "A Software Instruction Counter." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'89*. ACM Press, 1989, pp. 78–86. ISBN: 0-89791-300-0.

[Mog+20] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction." In: *CoRR* abs/2002.08437 (2020).

[Mol+05] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks." In: *Information Security and Cryptology – ICISC'05*. Vol. 3935. LNCS. Springer, 2005, pp. 156–168. ISBN: 3-540-33354-1.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[Mor+18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption." In: *European Workshop on System Security – EUROSEC*. ACM, 2018, 1:1–1:6.

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MPR06] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. "Bump in the Ether: A Framework for Securing Sensitive User Input." In: *USENIX Annual Technical Conference – USENIX ATC'06*. USENIX, 2006, pp. 185–198.

[MPR09] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. "Safe Passage for Passwords and Other Sensitive Data." In: *Network and Distributed System Security Symposium – NDSS'09*. The Internet Society, 2009.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[Mul+13] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. "Using Bleichenbacher"s Solution to the Hidden Number Problem to Attack Nonce Leaks in 384-Bit ECDSA." In: *Cryptographic Hardware and Embedded Systems – CHES'13*. Vol. 8086. LNCS. Springer, 2013, pp. 435–452. ISBN: 978-3-642-40348-4.

[Mul+14] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. "Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version." In: *J. Cryptographic Engineering* 4 (2014), pp. 33–45.

[Mur+13] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. "seL4: From General Purpose to a Proof of Information Flow Enforcement." In: *IEEE Symposium on Security and Privacy – S&P'13*. IEEE Computer Society, 2013, pp. 415–429. ISBN: 978-1-4673-6166-8.

[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'20*. 2020.

[MV15] Charlie Miller and Chris Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. http://illmatics.com/Remote%20Car%20Hacking.pdf. (Accessed 2020/03/11). 2015.

[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'17*. Vol. 10379. LNCS. Springer, 2017, pp. 213–230. ISBN: 978-3-319-62104-3.

[MY07] Michael Myers and Stephen Youndt. "An introduction to hardware-assisted virtual machine (hvm) rootkits." In: *Mega Security* (2007).

[Nac+05] David Naccache, Phong Q. Nguyen, Michael Tunstall, and Claire Whelan. "Experimenting with Faults, Lattices and the DSA." In: *Public Key Cryptography – PKC'05*. Vol. 3386. LNCS. Springer, 2005, pp. 16–28. ISBN: 3-540-24454-9.

[Nay+17] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya V. Lokam, Elaine Shi, and Vipul Goyal. "HOP: Hardware makes Obfuscation Practical." In: *Network and Distributed System Security Symposium – NDSS'17*. The Internet Society, 2017.

[Nel+17] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel." In: *Symposium on Operating Systems Principles – SOSP'17*. ACM, 2017, pp. 252–269. ISBN: 978-1-4503-5085-3.

[NIS] NIST. *National Vulnerability Database*. https://nvd.nist.gov. (Accessed 2020/03/11).

[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'13*. USENIX Association, 2013, pp. 479–494. ISBN: 978-1-931971-03-4.

[Noo+17] Job Noorman et al. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices." In: *ACM Trans. Priv. Secur.* 20 (2017), 7:1–7:33.

[NS00] Phong Q. Nguyen and Jacques Stern. "Lattice Reduction in Cryptology: An Update." In: *International Algorithmic Number Theory Symposium – ANTS'00*. Vol. 1838. LNCS. Springer, 2000, pp. 85–112. ISBN: 3-540-67695-3.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES." In: *Topics in Cryptology – CT-RSA'06*. Vol. 3860. LNCS. Springer, 2006, pp. 1–20. ISBN: 3-540-31033-9.

[Owu+13] Emmanuel Owusu, Jorge Guajardo, Jonathan M. McCune, James Newsome, Adrian Perrig, and Amit Vasudevan. "OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms." In: *Conference on Computer and Communications Security – CCS'13*. ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2477-9.

[Pag05] Dan Page. "Partitioned Cache Architecture as a Side-Channel Defence Mechanism." In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 280.

[Par08] Bryan Parno. "Bootstrapping Trust in a "Trusted" Platform." In: *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association, 2008.

[Per05] Colin Percival. *Cache Missing for Fun and Profit*. Technical report. http://www.daemonology.net/hyperthreading-considered-harmful/. (Accessed 2020/03/11). 2005.

[Pes+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium'16*. USENIX Association, 2016, pp. 565–581.

[Pet+18] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. "BASTION-SGX: bluetooth and architectural support for trusted I/O on SGX." In: *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 2018, 3:1–3:9.

[PG08] Dan R. K. Ports and Tal Garfinkel. "Towards Application Security on Untrusted Operating Systems." In: *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association, 2008.

[Pom+19] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. "Kernel Protection Against Just-In-Time Code Reuse." In: *ACM Trans. Priv. Secur.* 22 (2019), 5:1–5:28.

[Por13] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. https://tools.ietf.org/html/rfc6979. (Accessed 2020/03/11). Request for Comments: 6979. 2013.

[PPK12] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization." In: *IEEE Symposium on Security and Privacy – S&P'12*. IEEE Computer Society, 2012, pp. 601–615. ISBN: 978-0-7695-4681-0.

[PPM16] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. "Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT." In: *Computer Security Foundations – CSF'16*. IEEE Computer Society, 2016, pp. 387–400. ISBN: 978-1-5090-2607-4.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption." In: *Cryptographic Hardware and Embedded Systems – CHES'17*. Vol. 10529. LNCS. Springer, 2017, pp. 513–533. ISBN: 978-3-319-66786-7.

[PSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Just a Little Bit More." In: *Topics in Cryptology – CT-RSA'15*. Vol. 9048. LNCS. Springer, 2015, pp. 3–21. ISBN: 978-3-319-16714-5.

[PVC18] Christian Priebe, Kapil Vaswani, and Manuel Costa. "EnclaveDB: A Secure Database Using SGX." In: *IEEE Symposium on Security and Privacy – S&P'18*. IEEE Computer Society, 2018, pp. 264–278. ISBN: 978-1-5386-4353-2.

[Qur18] Moinuddin K. Qureshi. "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping." In: *Symposium on Microarchitecture – MICRO'18*. IEEE Computer Society, 2018, pp. 775–787. ISBN: 978-1-5386-6240-3.

[Raj+09] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. "Resource management for isolation enhanced cloud services." In: *Cloud Computing Security Workshop – CCSW*. ACM, 2009, pp. 77–84. ISBN: 978-1-60558-784-4.

[Ram+19] Kartik Ramkrishnan, Antonia Zhai, Stephen McCamant, and Pen-Chung Yew. "New Attacks and Defenses for Randomized Caches." In: *CoRR* abs/1909.12302 (2019).

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "Dude, is my code constant time?" In: *Design, Automation & Test in Europe – DATE'17*. IEEE, 2017, pp. 1697–1702. ISBN: 978-3-9815370-8-6.

[Rei18] Charlie Reis. *Mitigating Spectre with Site Isolation in Chrome*. https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html. (Accessed 2020/03/11). 2018.

[Ren+19] Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. "Design and Implementation of the Ascend Secure Processor." In: *IEEE Trans. Dependable Sec. Comput.* 16 (2019), pp. 204–216.

[Ris] *RISC-V: The Free and Open RISC Instruction Set Architecture*. https://riscv.org/. (Accessed 2020/03/11).

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution." In: *USENIX Security Symposium'15*. USENIX Association, 2015, pp. 431–446.

[Ron+17] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. "IoT Goes Nuclear: Creating a ZigBee Chain Reaction." In: *IEEE Symposium on Security and Privacy – S&P'17*. IEEE Computer Society, 2017, pp. 195–212. ISBN: 978-1-5090-5533-3.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21 (1978), pp. 120–126.

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. "Bitslice Implementation of AES." In: *Cryptology and Network Security – CANS'06*. Vol. 4301. LNCS. Springer, 2006, pp. 203–212. ISBN: 3-540-49462-6.

[Rua14a] Xiaoyu Ruan. "Boot with Integrity, or Don't Boot." In: *Platform Embedded Security Technology Revealed*. Berkeley, CA: Apress, 2014, pp. 143–163. ISBN: 978-1-4302-6572-6. DOI: 10.1007/978-1-4302-6572-6_6.

[Rua14b] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed. Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. ApressOpen, 2014. ISBN: 978-1-4302-6572-6.

[Rus] *Rust Programming Language*. https://www.rust-lang.org/. (Accessed 2020/03/11).

[Rut13] Joanna Rutkowska. *Thoughts on Intel's upcoming Software Guard Extensions (Part 2)*. http://theinvisiblethings.blogspot.co.at/2013/09/thoughts-on-intels-upcoming-software.html. (Accessed 2020/03/11). Sept. 2013.

[Rya19] Keegan Ryan. "Return of the Hidden Number Problem. A Widespread and Novel Key Extraction Attack on ECDSA and DSA." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019 (2019), pp. 146–168.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)." In: *IEEE Symposium on Security and Privacy – S&P'10*. IEEE Computer Society, 2010, pp. 317–331. ISBN: 978-0-7695-4035-1.

[San17] Ardalan Amiri Sani. "SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications." In: *Mobile Systems – MobiSys'17*. ACM, 2017, pp. 197–210. ISBN: 978-1-4503-4928-4.

[Sat18] Michael Satran. *Measured Boot*. https://docs.microsoft.com/en-us/windows/win32/w8cookbook/measured-boot. (Accessed 2020/03/11). 2018.

[Sca+18] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives*. https://software.intel.com/sites/default/files/managed/f1/b8/intel-sgx-support-for-third-party-attestation.pdf. (Accessed 10/03/2020). 2018.

[SCA18] Laurent Simon, David Chisnall, and Ross J. Anderson. "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers." In: *IEEE European Symposium on Security and Privacy – EURO S&P'18*. IEEE, 2018, pp. 1–15. ISBN: 978-1-5386-4228-3.

[Sch+15] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX." In: *IEEE Symposium on Security and Privacy – S&P'15*. IEEE Computer Society, 2015, pp. 38–54. ISBN: 978-1-4673-6949-7.

[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'17*. 2017, pp. 3–24.

[Sch+18a] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2018, pp. 587–600.

[Sch+18b] 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'18*. 2018.

[Sch+19a] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-Flight Data Load." In: *IEEE Symposium on Security and Privacy – S&P'19*. IEEE, 2019, pp. 88–105. ISBN: 978-1-5386-6660-9.

[Sch+19b] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *Conference on Computer and Communications Security – CCS'19*. ACM, 2019, pp. 753–768. ISBN: 978-1-4503-6747-9.

[Sch+20a] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *CacheOut: Leaking Data on Intel CPUs via Cache Evictions*. https://cacheoutattack.com/. (Accessed 2020/04/03). 2020.

[Sch+20b] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. "Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86." In: *USENIX Security'20*. Accepted for publication with minor revision at the time of writing. 2020.

[Sch15] Bruce Schneier. *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W. W. Norton & Company, 2015.

[Sch89] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards." In: *Advances in Cryptology – CRYPTO'89*. Vol. 435. LNCS. Springer, 1989, pp. 239–252. ISBN: 3-540-97317-6.

[Sel16] Surenthar Selvaraj. *Overview of Intel Protected File System Library Using Software Guard Extensions*. https://software.intel.com/en-us/articles/overview-of-intel-protected-file-system-library-using-software-guard-extensions. (Accessed 2020/03/11). 2016.

[Seo+17] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs." In: *Network and Distributed System Security Symposium – NDSS'17*. The Internet Society, 2017.

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *USENIX Annual Technical Conference – USENIX ATC'12*. USENIX Association, 2012, pp. 309–318.

[SGF18] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. "ZeroTrace : Oblivious Memory Primitives from Intel SGX." In: *Network and Distributed System Security Symposium – NDSS'18*. The Internet Society, 2018.

[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'07*. ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2.

[Shi+11] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring." In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011), Hong Kong, China, June 27-30, 2011*. IEEE Computer Society, 2011, pp. 194–199. ISBN: 978-1-4577-0374-4.

[Shi+16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. "Preventing Page Faults from Telling Your Secrets." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2016, pp. 317–328. ISBN: 978-1-4503-4233-9.

[Shi+17a] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In: *Network and Distributed System Security Symposium – NDSS'17*. The Internet Society, 2017.

[Shi+17b] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB Linux Applications With SGX Enclaves." In: *Network and Distributed System Security Symposium – NDSS'17*. The Internet Society, 2017.

[Shi19] Mingwei Shih. "Securing Intel SGX against Side-channel Attacks via Load-time Synthesis." PhD thesis. Georgia Institute of Technology, 2019.

[Sno19] Edward Snowden. *Live Q&A with Edward Snowden: Thursday 23rd January, 8pm GMT, 3pm EST*. https://www.freesnowden.is/asksnowden/. (Accessed 2020/04/30). 2019.

[Son+06] JunHyuk Song, Radha. Poovendran, Jicheol Lee, and Tetsu Iwata. *The AES-CMAC Algorithm*. https://tools.ietf.org/html/rfc4493. (Accessed 2020/03/11). Request for Comments: 4493. 2006.

[Son+08] Dawn Xiaodong Song et al. "BitBlaze: A New Approach to Computer Security via Binary Analysis." In: *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*. Vol. 5352. LNCS. Springer, 2008, pp. 1–25. ISBN: 978-3-540-89861-0.

[Son+16] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. "HDFI: Hardware-Assisted Data-Flow Isolation." In: *IEEE Symposium on Security and Privacy – S&P'16*. IEEE Computer Society, 2016, pp. 1–17. ISBN: 978-1-5090-0824-7.

[Son+19] Dokyung Song et al. "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary." In: *Network and Distributed System Security Symposium – NDSS'19*. The Internet Society, 2019. ISBN: 1-891562-55-X.

[SP16] Raoul Strackx and Frank Piessens. "Ariadne: A Minimal Approach to State Continuity." In: *USENIX Security Symposium'16*. USENIX Association, 2016, pp. 875–892.

[SP17] Raoul Strackx and Frank Piessens. "The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks." In: *CoRR* abs/1712.08519 (2017).

[Sri+19] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard E. Shrobe, and Mathias Payer. "FirmFuzz: Automated IoT Firmware Introspection and Analysis." In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019*. ACM, 2019, pp. 15–21. ISBN: 978-1-4503-6838-4.

[SRS17] Rohit Sinha, Sriram K. Rajamani, and Sanjit A. Seshia. "A compiler and verifier for page access oblivious computation." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 649–660. ISBN: 978-1-4503-5105-8.

[SS15] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++." In: *Symposium on Code Generation and Optimization – CGO'15*. IEEE Computer Society, 2015, pp. 46–55. ISBN: 978-1-4799-8161-8.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. "The protection of information in computer systems." In: *Proceedings of the IEEE* 63 (1975), pp. 1278–1308.

[Suh+03] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. "AEGIS: architecture for tamper-evident and tamper-resistant processing." In: *International Conference on Supercomputing – ICS'03*. ACM, 2003, pp. 160–171. ISBN: 1-58113-733-8.

[Suh+04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. "Secure program execution via dynamic information flow tracking." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'04*. ACM, 2004, pp. 85–96. ISBN: 1-58113-804-0.

[Sum+10] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. "Precise calling context encoding." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 525–534. ISBN: 978-1-60558-719-6.

[SW99] Sean W. Smith and Steve H. Weingart. "Building a high-performance, programmable secure coprocessor." In: *Comput. Networks* 31 (1999), pp. 831–860.

[Swa17] Yogesh Swami. "SGX Remote Attestation is not Sufficient." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 736.

[SWG19] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA'19*. Ed. by Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren. Vol. 11543. Lecture Notes in Computer Science. Springer, 2019, pp. 177–196.

[SWT01] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH." In: *USENIX Security Symposium'01*. USENIX, 2001.

[SZ10] William N. Sumner and Xiangyu Zhang. "Memory indexing: canonicalizing addresses across executions." In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 217–226. ISBN: 978-1-60558-791-2.

[SZ13] William N. Sumner and Xiangyu Zhang. "Identifying execution points for dynamic analyses." In: *International Conference on Automated Software Engineering – ASE'13*. IEEE, 2013, pp. 81–91.

[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'13*. IEEE Computer Society, 2013, pp. 48–62. ISBN: 978-1-4673-6166-8.

[Sze+14] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. "Eternal War in Memory." In: *IEEE Security & Privacy* 12 (2014), pp. 45–53.

[Sze19] Jakub Szefer. "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses." In: *J. Hardware and Systems Security* 3 (2019), pp. 219–234.

[Tan+20] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. "PhantomCache: Obfuscating Cache Conflicts with Localized Randomization." In: *Network and Distributed System Security Symposium – NDSS'20*. 2020.

[TE13] Tianhao Tong and David Evans. "Guardroid: A Trusted Path for Password Entry." In: *Mobile Security Technologies*. MoST'13 (2013).

[Tia+17] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. "SGXKernel: A Library Operating System Optimized for Intel SGX." In: *Computing Frontiers Conference – CF'17*. ACM, 2017, pp. 35–44. ISBN: 978-1-4503-4487-6.

[TLL06] Richard Ta-Min, Lionel Litty, and David Lie. "Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable." In: *Operating Systems Design and Implementation – OSDI'06*. USENIX Association, 2006, pp. 279–292. ISBN: 1-931971-47-1.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures." In: *J. Cryptology* 23 (2010), pp. 37–71.

[TPV17] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX." In: *USENIX Annual Technical Conference – USENIX ATC'17*. USENIX Association, 2017, pp. 645–658.

[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'18*. ACM, 2018, 98:1–98:6. ISBN: 978-1-5386-4114-9.

[Tsa+20] Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sánchez. "Safecracker: Leaking Secrets through Compressed Caches." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'20*. ACM, 2020, pp. 1125–1140. ISBN: 978-1-4503-7102-5.

[TSS15] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. "Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads." In: *Conference on Computer and Communications Security – CCS'15*. ACM, 2015, pp. 256–267. ISBN: 978-1-4503-3832-5.

[Tur18] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. https://support.google.com/faqs/answer/7625886. (Accessed 2020/03/31). 2018.

[Val96] Antti Valmari. "The State Explosion Problem." In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*. Vol. 1491. LNCS. Springer, 1996, pp. 429–528. ISBN: 3-540-65306-6.

[Vas+12] Amit Vasudevan, Jonathan M. McCune, James Newsome, Adrian Perrig, and Leendert van Doorn. "CARMA: a hardware tamper-resistant isolated execution environment on commodity x86 platforms." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2012, pp. 48–49. ISBN: 978-1-4503-1648-4.

[Vas+13] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan M. McCune, James Newsome, and Anupam Datta. "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework." In: *IEEE Symposium on Security and Privacy – S&P'13*. IEEE Computer Society, 2013, pp. 430–444. ISBN: 978-1-4673-6166-8.

[Vas+16] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. "überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor." In: *USENIX Security Symposium'16*. USENIX Association, 2016, pp. 87–104.

[Ven+08] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. "FlexiTaint: A programmable accelerator for dynamic taint propagation." In: *High Performance Computer Architecture – HPCA'08*. IEEE Computer Society, 2008, pp. 173–184. ISBN: 978-1-4244-2070-4.

[Ver] *Verified by Visa*. https://www.visaeurope.com/making-payments/verified-by-visa/. (Accessed 2020/03/11).

[VEW12] Camille Vuillaume, Takashi Endo, and Paul Wooderson. "RSA Key Generation: New Attacks." In: *Constructive Side-Channel Analysis and Secure Design – COSADE'12*. Vol. 7275. LNCS. Springer, 2012, pp. 105–119. ISBN: 978-3-642-29911-7.

[WA17a] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Tech. rep. SiFive Inc., University of California, Berkeley, 2017.

[WA17b] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. Tech. rep. SiFive Inc., University of California, Berkeley, 2017.

[Wan+17a] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. "CacheD: Identifying Cache-Based Timing Channels in Production Software." In: *USENIX Security Symposium'17*. USENIX Association, 2017, pp. 235–252.

[Wan+17b] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX." In: *Conference on Computer and Communications Security – CCS'17*. ACM, 2017, pp. 2421–2434. ISBN: 978-1-4503-4946-8.

[Wan+18] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. "A survey of the double-fetch vulnerabilities." In: *Concurrency and Computation: Practice and Experience* 30 (2018).

[Wan+19] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. "Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation." In: *USENIX Security Symposium'19*. USENIX Association, 2019, pp. 657–674.

[Wat+15] Robert N. M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization." In: *IEEE Symposium on Security and Privacy – S&P'15*. IEEE Computer Society, 2015, pp. 20–37. ISBN: 978-1-4673-6949-7.

[WB11] Peter Williams and Rick Boivie. "CPU Support for Secure Executables." In: *Trust and Trustworthy Computing – TRUST'11*. Vol. 6740. LNCS. Springer, 2011, pp. 172–187. ISBN: 978-3-642-21598-8.

[WBA17] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves." In: *International Symposium on Computer Architecture – ISCA'17*. ACM, 2017, pp. 81–93. ISBN: 978-1-4503-4892-8.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanovic. "Mondrian memory protection." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'02*. ACM Press, 2002, pp. 304–316. ISBN: 1-58113-574-2.

[Wei+16] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves." In: *European Symposium on Research in Computer Security – ESORICS'16*. Vol. 9878. LNCS. Springer, 2016, pp. 440–457. ISBN: 978-3-319-45743-7.

[Wei+18b] 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'18*. USENIX Association, 2018, pp. 603–620.

[Wei16] Samuel Weiser. *Secure I/O with Intel SGX*. Graz University of Technology. Master thesis. https://graz.pure.elsevier.com/files/7516934/2016_Weiser_Thesis_SecureIO_SGX.pdf. (Accessed 2020/03/11). 2016.

[Wer+16] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. "No-Execute-After-Read: Preventing Code Disclosure in Commodity Software." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2016, pp. 35–46. ISBN: 978-1-4503-4233-9.

[Wer+19] 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'19*. USENIX Association, 2019, pp. 675–692.

[Wic+18] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries." In: *Annual Computer Security Applications Conference – ACSAC'18*. ACM, 2018, pp. 161–173. ISBN: 978-1-4503-6569-7.

[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'07*. ACM, 2007, pp. 494–505. ISBN: 978-1-59593-706-3.

[WL08] Zhenghong Wang and Ruby B. Lee. "A novel cache architecture with enhanced performance and security." In: *Symposium on Microarchitecture – MICRO'08*. IEEE Computer Society, 2008, pp. 83–93. ISBN: 978-1-4244-2836-6.

[WLY18] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. "Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels." In: *Conference on Computer and Communications Security – CCS'18*. ACM, 2018, pp. 1899–1913. ISBN: 978-1-4503-5693-0.

[Won15] David Wong. "Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really?" In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 839.

[WSB18b] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. "Single Trace Attack Against RSA Key Generation in Intel SGX SSL." In: *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 2018, pp. 575–586.

[WW17b] Samuel Weiser and Mario Werner. "SGXIO: Generic Trusted I/O Path for Intel SGX." In: *CoRR* abs/1701.01061 (2017).

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *IEEE Symposium on Security and Privacy – S&P'15*. IEEE Computer Society, 2015, pp. 640–656. ISBN: 978-1-4673-6949-7.

[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'17*. ACM, 2017, pp. 859–874. ISBN: 978-1-4503-4946-8.

[XSL16] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. "Intel® Software Guard Extensions (Intel® SGX) Software Support for Dynamic Memory Allocation inside an Enclave." In: *Hardware and Architectural Support for Security and Privacy – HASP*. ACM, 2016, 11:1–11:9. ISBN: 978-1-4503-4769-3.

[XSZ08] Bin Xin, William N. Sumner, and Xiangyu Zhang. "Efficient program execution indexing." In: *Programming Language Design and Implementation – PLDI'08*. ACM, 2008, pp. 238–248. ISBN: 978-1-59593-860-2.

[Xu+16] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. "A Practical Verification Framework for Preemptive OS Kernels." In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Vol. 9780. LNCS. Springer, 2016, pp. 59–79. ISBN: 978-3-319-41539-0.

[Yav19] Tuba Yavuz. "Detecting Callback Related Deep Vulnerabilities in Linux Device Drivers." In: *Cybersecurity Development, SecDev'19*. IEEE, 2019, pp. 62–75. ISBN: 978-1-5386-7289-1.

[YB14] Yuval Yarom and Naomi Benger. "Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack." In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 140.

[YF14] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium'14*. USENIX Association, 2014, pp. 719–732.

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA." In: *Cryptographic Hardware and Embedded Systems – CHES'16*. Vol. 9813. LNCS. Springer, 2016, pp. 346–367. ISBN: 978-3-662-53139-6.

[YGZ15] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. "Trusted Display on Untrusted Commodity Platforms." In: *Conference on Computer and Communications Security – CCS'15*. ACM, 2015, pp. 989–1003. ISBN: 978-1-4503-3832-5.

[YH10] Jean Yang and Chris Hawblitzel. "Safe to the last instruction: automated verification of a type-safe operating system." In: *Programming Language Design and Implementation – PLDI'10*. ACM, 2010, pp. 99–110. ISBN: 978-1-4503-0019-3.

[YS08] Jisoo Yang and Kang G. Shin. "Using hypervisor to provide data secrecy for user applications on a per-page basis." In: *International Conference on Virtual Execution Environments – VEE'08*. ACM, 2008, pp. 71–80. ISBN: 978-1-59593-796-4.

[Yu+19] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing." In: *Network and Distributed System Security Symposium – NDSS'19*. The Internet Society, 2019. ISBN: 1-891562-55-X.

[Zel+08] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Hardware Enforcement of Application Security Policies Using Tagged Memory." In: *Operating Systems Design and Implementation – OSDI'08*. USENIX Association, 2008, pp. 225–240. ISBN: 978-1-931971-65-2.

[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'12*. ACM, 2012, pp. 305–316. ISBN: 978-1-4503-1651-4.

[Zha+19] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. "PeX: A Permission Check Analysis Framework for Linux Kernel." In: *USENIX Security Symposium'19*. USENIX Association, 2019, pp. 1205–1220.

[Zha+20] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. "Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'20*. ACM, 2020, pp. 1263–1276. ISBN: 978-1-4503-7102-5.

[Zha19] Mingwei Zhang. *XOM-Switch*. https://github.com/intel/xom-switch. (Accessed 2020/03/11). 2019.

[Zho+12] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. "Building Verifiable Trusted Path on Commodity x86 Computers." In: *IEEE Symposium on Security and Privacy – S&P'12*. IEEE Computer Society, 2012, pp. 616–630. ISBN: 978-0-7695-4681-0.

[Zho14] Zongwei Zhou. "On-Demand Isolated I/O for Security-Sensitive Applications on Commodity Platforms." PhD thesis. Carnegie Mellon University, 2014.

[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'16*. Vol. 10146. LNCS. Springer, 2016, pp. 228–244. ISBN: 978-3-319-54668-1.

[ZRZ16] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches." In: *Conference on Computer and Communications Security – CCS'16*. ACM, 2016, pp. 871–882. ISBN: 978-1-4503-4139-4.

[ZYG14] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. "Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O." In: *IEEE Symposium on Security and Privacy – S&P'14*. IEEE Computer Society, 2014, pp. 308–323. ISBN: 978-1-4799-4686-0.

[ZZP04] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. "HIDE: an infrastructure for efficiently protecting information leakage on the address bus." In: *Architectural Support for Programming Languages and Operating Systems – ASPLOS'04*. ACM, 2004, pp. 72–84. ISBN: 1-58113-804-0.

[AMD20a] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. White paper. 2020.

[AMD20b] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Ref. no. 24593, revision 3.33. 2020.

[Arma] Arm Limited. *Arm Cortex-A Series Processors*. https://developer.arm.com/ip-products/processors/cortex-a. (Accessed 2020/03/11).

[Armb] Arm Limited. *Arm Cortex-M Series Processors*. https://developer.arm.com/ip-products/processors/cortex-m. (Accessed 2020/03/11).

[Arm09] Arm Limited. *ARM Security Technology: Building a Secure System Using TrustZone Technology*. Ref. no. PRD29-GENC-009492C. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. (Accessed 2020/03/11). 2009.

[Arm17] Arm Limited. *TrustZone Technology for ARMv8-M Architecture*. Ref. no. 100690_0200_00_en. https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf. (Accessed 2020/03/11). 2017.

[Arm19] Arm Limited. *Armv8.5-A Memory Tagging Extension*. White paper. 2019.

[Dep85] Department of Defense. *Department of Defense Trusted Computer System Evaluatino Criteria*. DOD 5200.28-STD. http://www.iwar.org.uk/comsec/resources/standards/rainbow/5200.28-STD.html. (Accessed 2020/03/11). 1985.

[Hex] Hex Five. *MultiZone*. https://hex-five.com. (Accessed 2020/03/11).

[Int] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. https://software.intel.com/en-us/articles/pintool/. (Accessed 2020/03/11).

[Int15] Intel Corporation. *Intel Trusted Execution Technology (Intel TXT), Software Development Guide*. Reference no. 315168-012. July 2015.

[Int16a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Reference no. 325462-061US. Dec. 2016.

[Int16b] Intel Corporation. *Intel Software Guard Extensions SDK for Linux OS. Developer Reference*. Rev. 1.5. 2016.

[Int17a] Intel Corporation. *Control-flow Enforcement Technology Preview*. Revision 2.0. June 2017.

[Int17b] Intel Corporation. *Intel Software Guard Extensions Developer Guide*. https://software.intel.com/en-us/sgx-sdk/documentation. (Accessed 2020/03/11). 2017.

[Int19] Intel Corporation. *Intel SgxSSL Library User Guide*. Rev. 1.2.5. https://software.intel.com/sites/default/files/managed/3b/05/Intel-SgxSSL-Library-User-Guide.pdf. (Accessed 2020/03/11). 2019.

[Lia+20] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. "Establishing Trusted I/O Paths for SGX Client Systems With Aurora." In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 1589–1600. ISSN: 1556-6021.

[Lin17] Linux kernel. *SECure COMPuting with filters*. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. (Accessed 2020/03/11). 2017.

[NJC16] NJCCIC. *Mirai Botnet*. https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet. (Accessed 2020/03/11). 2016.

[OWA19] OWASP. *OWASP Application Security Verification Standard 4.0*. 2019.

[Ope] OpenSSL Software Foundation. *OpenSSL – Cryptography and SSL/TLS Toolkit*. https://www.openssl.org/. (Accessed 2020/03/11).

[Ope19] OpenSSL. *Security policy*. https://www.openssl.org/policies/secpolicy.html. (Accessed 2020/03/11). 2019.

[PaX03] PaX Team. *Address space layout randomization (ASLR)*. http://pax.grsecurity.net/docs/aslr.txt. (Accessed 2020/03/11). 2003.

[PaX15] PaX Team. "Rap: Rip rop." In: *Hackers to Hackers Conference* (2015).

[Ste67] J. Stein. "Computational Problems Associated with Racah Algebra." In: *Journal of Computational Physics* 1 (1967), pp. 397–405.

[TCG14] TCG. *Trusted Platform Module Library. Part 1: Architecture. Family 2.0*. Revision 01.16. Oct. 2014.

[Tru12] Trusted Computing Group. *Glossary*. www.trustedcomputinggroup.org/developers/glossary. (Accessed 2020/03/11). 2012.

[UEF19] UEFI Forum. *Unified Extensible Firmware Interface (UEFI) Specification*. Version 2.8. 2019.

[seL20] seL4 Foundation. *seL4 Docs*. https://docs.sel4.systems. (Accessed 2020/03/11). 2020.