SoK: SGX.Fail: How Stuff Gets eXposed

Stephan van Schaik
University of Michigan
stephvs@umich.edu

Alex Seto
Purdue University
aseto@purdue.edu

Thomas Yurek
UIUC
yurek2@illinois.edu

Adam Batori
University of Michigan
aabatori@umich.edu

Bader AlBassam
Purdue University
balbassa@purdue.edu

Christina Garman
Purdue University
clg@cs.purdue.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Andrew Miller
UIUC
soc1024@illinois.edu

Eyal Ronen
Tel Aviv University
eyal.ronen@cs.tau.ac.il

Yuval Yarom
University of Adelaide
yval@cs.adelaide.edu.au

***Abstract*—Intel's Software Guard Extensions (SGX) promises an isolated execution environment, protected from all software running on the machine. As such, numerous works have sought to leverage SGX to provide confidentiality and integrity guarantees for code running in adversarial environments. In the past few years however, SGX has come under heavy fire, threatened by numerous hardware attacks. With Intel repeatedly patching SGX to regain security while consistently launching new (micro)architectures, it is increasingly difficult to track the applicability of various attacks techniques across the SGX design landscape.**

Thus, in this paper we set out to survey and categorize various SGX attacks, their applicability to different SGX architectures, as well as the information leaked by them. We then set out to explore the effectiveness of SGX's update mechanisms in preventing attacks on real-world deployments. Here, we study two commercial SGX applications. First, we investigate the SECRET network, an SGX-backed blockchain aiming to provide privacy preserving smart contracts. Next, we also consider PowerDVD, a UHD Blu-Ray Digital Rights Management (DRM) software licensed to play discs on PCs. We show that in both cases vendors are unable to meet security goals originally envisioned for their products, presumably due to SGX's long update timelines and the complexities of a manual update process. This in turn forces vendors into making difficult security/usability trade offs, resulting in security compromises.

1. Introduction

Trusted Execution Environments (TEEs) have long been the holy grail for security applications. Instead of enforcing isolation and access control by software mechanisms, TEEs aim to provide security via hardware, with the system's (micro)architecture enforcing protection. Indeed, with the promise of strong security with near-native performance, most hardware vendors offer TEEs, including ARM TrustZone [16, 96, 100], AMD SEV [72]) and Intel SGX [40].

Recently however, computer systems have encountered a new kind of threat. Starting from the origins of cryptographic key extraction [98, 99, 138], side-channel attacks have now become a threat to nearly all hardware-backed security primitives. In addition to breaking basic security primitives like user-kernel isolation [75, 85, 120, 123, 127], hardware attacks have been demonstrated against nearly every TEE deployment, including TrustZone [101, 105, 139], SEV [29, 82, 83, 84, 93, 134, 135] and SGX [24, 42, 49, 70, 90, 118, 120, 123, 125, 127, 131, 133].

Unlike SEV and TrustZone, SGX is unique in the TEE landscape in offering a way to mitigate the consequences of a compromise. Bolstered by remote attestation and provisioning mechanisms, microcode update options, and trusted computing base (TCB) recovery procedures, Intel can, in principle, recover SGX from compromise and update it after every successful hardware attack.

With Intel CPUs being the target of a wealth of side channel research and attacks, in this paper we aim to study and categorize SGX attack techniques and their data leakages, as well as ascertain the real world feasibility of SGX post-compromise recovery. Thus, in this paper we set out to study the following questions:

What techniques are available for attacking SGX enclaves and what information do these techniques recover? What mitigations exist, and how effective are SGX countermeasures and TCB recovery mechanisms at preventing compromises of SGX deployments?

1.1. Our Contribution

Given the wide variety of attacks on SGX enclaves, we start by studying and building a comprehensive categorization of publicly known hardware attacks on them. For each class of attacks, we detail what information can be leaked and what countermeasures are available for it. We then investigate the effectiveness of the SGX TCB recovery mechanism, presenting an overview of SGX update timelines. Finally, we examine two commercial SGX deployments, the SECRET network (an SGX-based blockchain) and PowerDVD (a UHD Blu-Ray software player).

By using SECRET and PowerDVD as case studies, we are able to ascertain how well real-world SGX deployments fare in the face of the SGX attack landscape. Unfortunately, we find that due to fundamental issues in the design of SGX,

it is difficult to build and securely deploy SGX applications that protect high-valued secrets. In particular, we argue that as soon as SGX is compromised, market forces place vendors with a difficult choice between significantly reducing their user base and foregoing the SGX security guarantees, allowing potential secret extraction.

Categorization of SGX Attacks. We begin by surveying publicly known SGX attacks, categorizing the information leakage via each attack technique (Section 3). We then proceed to describe countermeasures available, as well as if the countermeasures are applied by Intel, the operating system, or the enclave developer. Finally, we give an overview of the prominent SGX-enabled Intel CPU families, summarizing the attacks and mitigations applicable to each architecture.

Investigating SGX Update Cycles. With TCB updates being a prominent feature in SGX post-compromise recovery, we proceed to investigate the effectiveness of TCB updates in protecting SGX applications from publicly known mitigatable attacks. Here, we note that SGX's threat model assumes a malicious operating system, precluding the use of "regular" update mechanisms. Consequentially, Intel collaborates with motherboard vendors who distribute SGX microcode updates in BIOS updates. This is inefficient, as BIOS updates might damage the motherboard, and thus must be carefully vetted by each vendor for each separate product.

Measuring the update timelines, we perform a market study of the timeline of BIOS update postings across six motherboard vendors and six high-profile SGX vulnerabilities. As we show, SGX TCB updates can suffer from extremely long delays, with vendors achieving 50% update coverage of their product lines about 52 days *after* an SGX vulnerability publication. Finally, even when a BIOS update is available, its installation is often manual, and likely only performed by advanced users. While we are unable to remotely measure BIOS versions, we conjecture that many machines are not updated, thereby remaining vulnerable to well publicized attacks. Thus these machines cannot obtain a trusted attestation status.

The Developer's Dilemma. We observe that this long update cycle requires enclave developers to strike a difficult balance between security and usability. On the one hand, prioritizing security requires only using fully-updated machines, which, due to the long update cycles, are often not available in a timely manner for a large fraction of the user base. This in turn can cause developers to potentially lose a large part of their user base for often lengthy periods. On the other hand, prioritizing usability results in a potential security risk, as adversaries may exploit known vulnerabilities to breach the product's security mechanisms. Overall, we argue that the SGX update and recovery model can put enclave developers in a difficult dilemma. Either have secure products that require constant manual updates, with the possibility that a large fraction of machines remains permanently incompatible, or trust machines vulnerable to severe SGX compromises, where enclave contents and attestation keys can often be recovered reliably in seconds.

Emulating SGX in Software. To demonstrate the implications of allowing the use of vulnerable machines, in Appendix A we develop EGX (Emulated Guard eXtensions), an SGX emulator which runs enclaves (including production-quality ones) outside of SGX, using leaked attestation keys. Once the CPU's attestation key has been extracted, EGX allows executing enclaves on nearly any architecture and hardware, including AMD, ARM and Apple CPUs.

Breaching the SECRET Network. For the first commercial deployment we investigate as part of our study, we use our EGX framework to breach the privacy guarantees of the SECRET network [110]. SECRET is a privacy-preserving blockchain which leverages SGX to provide confidential execution of smart contracts. Since its launch in September 2020, SECRET has grown to a total market cap of $150 million, as of early October 2022.

As Intel did not address the xAPIC and MMIO issues [10, 11, 12] via TCB recovery, we were able to register a Rocket Lake server as a validator node. Despite not having sufficient funds to be trusted to actively validate transactions[1], SECRET's over promiscuous registration process nonetheless stores a copy of SECRET's global consensus seed inside our SGX enclave. Next, we extend the work of Borrello et al. [21] to Rocket Lake CPUs and extract the consensus seed of our SECRET node, as well as the its private EPID key. Using these keys we break SECRET's privacy-preserving features, decrypting the internal state of all smart contracts on the network including all digital assets embedded in them.

Breaching UHD Blu-ray DRM in PowerDVD. In our second real-world case study, we use our EGX framework to reverse engineer PowerDVD [2] and its DRM scheme for playing UHD Blu-ray discs. Being the only software licensed to playback UHD Blu-rays on PCs (as opposed to dedicated hardware players), PowerDVD uses SGX to ensure the integrity and confidentiality of the disc decryption keys. Remarkably, PowerDVD trusts unpatched machines with `GROUP_OUT_OF_DATE` attestation status, favoring usability over security. Consequently, we can extract attestation keys from such vulnerable devices using known techniques such as Foreshadow [120] and use them both in framework and to aid in our reverse engineering process.

Throughout this process, we also uncover previously undisclosed information about the Advanced Access Content System (AACS) 2 protocol, which is a closed-source proprietary protocol for UHD Blu-ray DRM. We present the first public specification of this protocol for the benefit of future researchers in Appendix C. Finally, we are also able to extract AACS2 decryption keys out of PowerDVD's SGX enclaves, allowing us to outline how one might completely remove the encryption from a UHD Blu-ray movie.

Inefficient Platform Revocation. Throughout our exploration, we also noticed that the remote attestation protocol in SGX is poorly equipped to deal with wide-scale key compromises, resulting in a potential denial of service (DoS) attack on the SGX ecosystem. More specifically, the complexity of the remote attestation protocol is linear in the

1. As of October 10, 2022, this requires having an account balance of 38,692 SCRT or $35,100 [111].

number of revoked platforms. As we show in Appendix B, by compromising about 720,000 machines in a given group ID and publishing their SGX keys, an attacker can mount a DoS attack on SGX attestation, forcing attestation on machines in the targeted group to take about an hour.

As prior works [120, 125, 127] show how to reliably extract SGX keys, we argue that mounting such an attack requires a budget of $36 million, which is within reach of wealthy individuals or large organizations.

Mitigations and Future Research. Finally, based on both our study of the different information leakages by known attacks, as well as how well TCB recovery can protect real world SGX deployments, we conclude with a discussion of mitigations and future research directions in the space. We provide concrete mitigations for the attacks presented here, as well as generic lessons that can be learned by future enclave developers from our two case studies. We also provide a number of general future design considerations and research directions for TEEs and TCB recovery strategies.

Summary of Contributions. In this paper we make the following contributions:

- We survey publicly known SGX attacks, categorize the information they expose, and document their applicability to prominent Intel architectures (Section 3).
- We document long delays and potential issues with the SGX microcode update model, and quantify them with a measurement study (Section 4).
- We present Emulated Guard eXtensions, an SGX virtualization framework capable of running commercial SGX enclaves on nearly any architecture (Appendix A).
- We breach the privacy guarantees of the SECRET network, allowing us to recover the internal state of SECRET's smart contracts and any digital assets in them (Section 5).
- We reverse-engineer PowerDVD and breach AACS2 DRM scheme, while providing the first public documentation of this mechanism (Section 6 and Appendix C).
- We analyze SGX's revocation protocol, presenting a denial of service attack against SGX attestation (Appendix B).
- We present concrete mitigations for the breaches discussed in this paper, as well as provide a broader discussion on recovery mechanisms and future research and TEE design directions based on our study (Section 7).

1.2. Disclosure and Ethics

Our research and disclosure were conducted ethically and responsibly in consultation with the Electronic Frontier Foundation (EFF) and with the aim of minimizing risk to all parties. We have disclosed our results to Intel, the SECRET network, and CyberLink (PowerDVD's vendor), and assisted both SECRET and CyberLink with handling these issues. Aiming to preserve the privacy of SECRET's users, all testing was only performed on our own transactions, with explicit consent of transacting parties.

1.3. Current Status

We have coordinated the public disclosure of this work with Intel, CyberLink and the SECRET network.

Intel. While an SGX TCB recovery addressing the xAPIC and MMIO issues [10, 11, 12, 21] was originally planned to happen no later than March 7, 2023 [67], 7 months after the public posting of these issues, this date has now been changed to November 29th, 2022. However, we note that this only partially addresses the issue, as TCB recovery for some popular SGX-enabled servers and desktops will only occur in January 2023 leaving these vulnerable until then [66].

SECRET Network. We have assisted the SECRET network in hardening their deployment against SGX attacks. In particular, SECRET no longer accepts nodes vulnerable to ÆPIC. However, as SECRET's current protocol design makes it quite difficult to change the network's consensus seed, it is currently impossible to guarantee privacy of past or future transactions performed on the SECRET network. Finally, while updating the consensus seed would provide protections for future transactions, there are unfortunately no guarantees that can ever be made about any transactions made under a leaked seed.

PowerDVD. Being perhaps the largest SGX deployment when counting the number of users that must have SGX-enabled machines, it is difficult for PowerDVD to expect these users to continuously update their BIOS as a condition to playing UHD Blu-Ray discs. As such, PowerDVD is likely to continue to trust unpatched machines with `GROUP_OUT_OF_DATE` attestation status, making it vulnerable to nearly all SGX attacks.

2. Background and Related Work

2.1. Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) [15, 89] is an extension of the x86_64 instruction set, supporting secure code execution in untrusted environments. SGX creates secure execution environments, called enclaves, which prevents inspection and modification of the code and data inside them. Additionally, SGX provides an ecosystem for remote attestation to ensure that these enclaves are running on genuine trustworthy Intel hardware, and not by a malicious simulator.

SGX Threat Model. SGX's threat model only trusts the processor's hardware and Intel-provided and Intel-signed architectural enclaves. Other than the architectural enclaves, SGX does not trust any software executing on the processor, including the operating system, the hypervisor, and the firmware (BIOS). The processor's microcode, however, is considered part of the processor and hence trusted.

Identifying Enclaves. For each enclave, SGX keeps an identity comprised of the enclave developer's identifier and a measurement representing the enclave's initial state. The developer's identifier, referred as `mrsigner` in SGX literature, is a cryptographic hash of the public RSA key the enclave developer used to sign the enclave's measurement. The measurement, representing the enclave's initial state, is a cryptographic hash of those parts of the enclave's contents (code and data) that its developer chose to measure. Following the SGX nomenclature, we refer to this measurement as `mrenclave`.

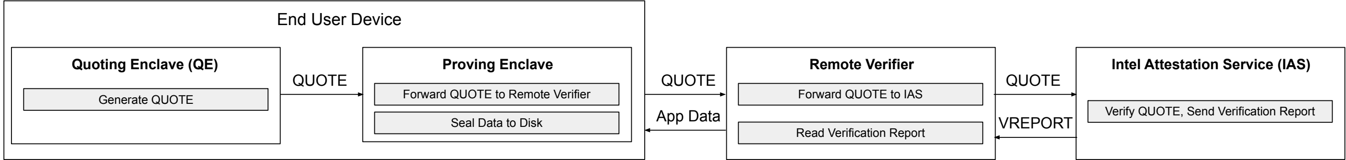

Figure 1: Remote Attestation Example

2.2. SGX's Attestation Mechanism

One of the most compelling properties that SGX provides is the ability of an enclave to attest to a remote verifying party that it is running on genuine and trustworthy Intel hardware, with confidentiality and security guarantees, as opposed to a malicious simulator. This allows the remote party to subsequently provision the enclave with secrets, while being assured that these secrets never leave the enclave's memory.

We now proceed with an overview of SGX's attestation process (see [71] for an extended discussion).

Local attestation. When an enclave wants to prove (or attest) to a remote verifier, it first needs to prove its identity to the Quoting Enclave (QE)—a special architectural enclave provided and signed by Intel—via a process referred to as *local attestation* [15, 60]. At a high level, this is done by having the proving enclave use the `ereport` instruction, which prepares a report containing the `mrenclave` and `mrsigner` values of the proving enclave. The report is also signed using a key that is only accessible to the QE. The proving enclave then passes the report to the Quoting Enclave, which proceeds with the remote attestation process.

Remote Attestation. Once local attestation is complete, the QE can generate and sign a "Quote" authenticating the proving enclave. Using the information from the Report, the Quote contains a code hash (`mrenclave`) of the proving enclave as well as its developer's identifier (`mrsigner`). The quote also contains information as to whether the proving enclave is running in production or debug mode. Next, each SGX-enabled CPU is provisioned with an attestation private key, which is obtained from Intel's attestation server during SGX initialization. The attestation key is then sealed with keys only available to the Quoting Enclave (QE).

For attestation, QE accesses the machine's private attestation keys and signs the proving enclave's quote. This quote is sent to the verifying party (e.g., service provider), which will in turn send it to Intel's Attestation Server (IAS) for verification. As Intel possesses the public keys corresponding to each SGX machine, a successful IAS response guarantees to the verifying party that the enclave is running in SGX and has not been tampered with. See Figure 1 for an outline of SGX attestation for an example program.

Trusted Compute Base. The trusted compute base (TCB) is the set of components that must be working correctly, and may not be malicious or compromised for SGX to operate securely. Among these components are the CPU itself, the microcode, the quoting and provisioning enclaves as well as the trusted runtime system (tRTS) from the Intel SGX SDK. Whenever a vulnerability is found that compromises the TCB, Intel has to release a microcode update to mitigate the issue and to restore trust in the TCB, a process known as TCB recovery.

Attestation Status. When the attestation report returns the `TRUSTED` status, then this indicates that any known compromises have been mitigated, thus that the platform is fully trusted. Other variants of the attestation status include that a configuration change is required (`CONFIGURATION_NEEDED`) or that the enclave developer has to provide software mitigations (`SW_HARDENING_NEEDED`) or a combination of these (`SW_HARDENING_AND_CONFIGURATION_NEEDED`). Finally, `GROUP_OUT_OF_DATE` indicates that the platform is affected by a known vulnerability, and that mitigations as well as a TCB recovery is required to restore trust.

Enhanced Privacy ID (EPID). Rather than using standard digital signatures, SGX attestation uses the Intel-designed EPID protocol [26], which is a type of group signature that allows a CPU to sign messages (using its private signing keys) without uniquely disclosing its identity. When executed in unlinkable mode, all that an external observer (e.g., Intel) can do is verify the signature without being able to link it to any specific Intel CPU or other previously signed quotes. This allows SGX providers to be convinced that their secrets are indeed stored in a genuine Intel enclave, without being able to identify the specific CPU in a given group.

3. Categorization of SGX Attacks, Consequences, and Mitigations

We now look at published SGX attacks and their impact from the point of view of an enclave developer as shown in Table 1. That is, our goal is to characterize what these attacks can leak and what impact they have from an SGX programmer's perspective, rather than focusing on the details of how to mount these attacks and how they work [106]. More specifically, these attacks can have an impact on confidentially, i.e. the attacker can infer sensitive information, and/or integrity, the attacker can tamper with the enclave's data. For each of the attacks we discuss this impact, the current mitigation strategies and possible improvements to mitigate such attacks in the future.

We first focus on vulnerabilities that rely on the attacker inferring sensitive data from the enclave's access patterns in Section 3.1 and locating and exploiting both memory corruption vulnerabilities and speculative execution gadgets within an enclave in Section 3.2 and Section 3.3. As these attacks rely on the enclave code, these generally require the enclave developer to mitigate them. It is also important to note that as the provisioning and quoting enclave as well as

Attack	Developer	Intel	Leakage	Mitigation
Branch predictors [45, 55, 80]	✓	✗	Branches (code)	Constant-time code
Caches [24, 34, 42, 51, 90, 107]	✓	✗	64B accesses (code + data)	Constant-time code
Memory dependencies [91]	✓	✗	4B accesses (code + data)	Constant-time code
Port contention [13]	✓	✗	μ-ops (code)	Constant-time code
Page faults [136]	✓	✗	4K accesses (code + data)	Constant-time code
A/D bit monitoring [119]	✓	✗	4K accesses (code + data)	Constant-time code
DRAM channel [131]	✓	✗	1K - 8K accesses (code + data)	Constant-time code
FLUSH+RELOAD on PTEs [119]	✓	✗	4K accesses (code + data)	Constant-time code
IA32 segmentation faults [50]	✗	✓	1B accesses (code + data)*	Mitigated
Interrupts [80, 90, 118, 121]	✓	✗	Instructions	Constant-time code
CopyCat [92]	✓	✗	Instructions	Constant-time code
MicroScope [114]	✗	✗	Instructions	Constant-time code
ROP gadgets [20, 79]	✓	✗	Gadget dependent†	Memory safety
SmashEx [41]	✓	✗	Gadget dependent†	Memory safety
Synchronization [128, 132]	✓	✗	Gadget dependent†	Thread safety
SgxPectre [32]	✓	✗	Gadget dependent†	`lfence` & retpoline
Load Value Injection [102, 122]	✓	✗	Gadget dependent†	`lfence`
Foreshadow [120]	✗	✓	Enclave memory, CPU registers	No Hyper-Threading
SA-00219 [63]	✗	✓	Enclave memory‡, CPU registers	No iGPU
MDS [31, 108, 123]	✗	✓	In-flight loads/stores, vector registers	No Hyper-Threading
CacheOut [127]	✗	✓	Enclave memory, CPU registers	
Crosstalk [103]	✗	✓	MSRs, `egetkey`, `rdrand`	
MMIO Stale Data [68]	✗	✓	MSRs, `egetkey`, `rdrand`	No Hyper-Threading
ÆPIC Leak [12, 21]	✗	✓	Enclave memory§, CPU registers	No Hyper-Threading
PlunderVolt/V0LTpwn [74, 94]	✗	✓	AES-NI keys	No voltage scaling MSRs
PLATYPUS [86]	✗	✓	AES-NI keys, control flow, etc.	No RAPL

TABLE 1: An overview of the SGX attacks and whether the developer and/or Intel is required to address the issue. Leakage indicates what can be leaked. Mitigation indicates what is required to reach trusted status where a microcode update is available. *: 1B accesses for enclaves $\leq$ 1 MiB, otherwise 4K accesses. †: (speculative) code execution inside the enclave which can lead to leaking enclave memory, CPU registers, keys, etc. ‡: 8B of every cache line. §: 75% of even cache lines.

the tRTS are part of the TCB, that Intel is also an enclave developer and that these may require software patches too.

Then we shift our focus towards vulnerabilities that externally affect the enclave, such as attacks that can leak enclave memory and thus extract sensitive information such as keys in Section 3.4, as well as fault attacks in Section 3.5. As these are outside of the enclave developer's control, these generally require Intel to provide mitigations through a microcode update and by performing a TCB recovery. However, ultimately, it is up to the enclave developer to decide what platforms and hardware configurations to trust, as it is sometimes possible to mitigate certain issues in software.

3.1. Inferring Access Patterns

Since CPU threads and cores competitively share microarchitectural resources such as branch predictors [45, 55, 80], caches [24, 34, 42, 51, 90, 107], dependency resolution [91], DRAM row buffers [131] and port contention [13], an attacker can rely on contention to infer the control flow and/or data access patterns of an SGX enclave at different granularities. These attacks can be used to recover ECDSA nonces [47, 137], attack RSA exponentation [80] as well as recover keys from S-box/T-table implementations of AES.

Xu et al. [136] showed that unmapping enclave pages can be used to track page accesses, as the enclave accessing that page causes a page fault. Van Bulck et al. [119] and Wang et al. [131] showed that the use of access and dirty bits can be used to monitor page activity as well. In addition, Wang et al. [131] demonstrated that a concurrently running SGX enclave can infer whether the victim is accessing the same DRAM bank and row through DRAM contention. Furthermore, Van Bulck et al. [119] explored mounting a FLUSH+RELOAD attack on the page tables to infer enclave page accesses. Gyselinck et al. [50] demonstrated another controlled-channel attack in 32-bit enclaves at a byte-level granularity in the first MiB of the enclave address space by using segmentation faults. However, a recent microcode update addresses this.

Another line of work [80, 90, 118, 121] focused on interrupting the enclave execution to sample side-channel measurements yielding a framework that allows an attacker to single-step the enclave execution. Nemesis [121] showed that different instructions have a different response time to service the interrupt. CopyCat [92] extended this work by counting the number of instructions executed to infer the control flow at a very fine-grained granularity, which can then be used to perform ECDSA key recovery from a single trace. Finally, MicroScope [114] showed that the attacker can speculatively replay a single page faulting instruction in the enclave, which leads to the amplification of other side-channel attacks, but more specifically to detect the input of certain instructions as well as infer branches.

Mitigation: The enclave developer needs to ensure that the

attacker cannot infer sensitive information from control flow or access patterns by avoiding secret-dependent branches and memory lookups [14, 17]. Bernstein and Yang [19] proposed a constant-time GCD algorithm that can be used for applications like modular inversion. Similarly, there are AES implementations using bit-slicing [87], vector instructions [53] and AES instructions on modern Intel CPUs [54].

However, such constant-time implementations are usually limited to specific cryptographic implementations, whereas generic algorithms require a different approach. Raccoon [104] is one such option that implements the oblivious RAM (ORAM) technique by always evaluating both paths of a conditional branch. Ohrimenko et al. [97] instead propose to eliminate all conditional branches by transforming them into a conditional move (`cmov`) instruction. However, their approach is limited to data-oblivious machine learning algorithms. Zigzagger [80] is a compiler-based mitigation against branch shadow attacks that obfuscates a set of conditional branches by merging them into a single indirect branch, which is harder to infer. As none of these tools adequately address the problem of inferring sensitive information from access patterns, as they either focus on constant-time cryptographic primitives or are otherwise limited by applicability or performance, this area is still open to future research.

Not only do some of the control channel attacks, the interrupt-driven attacks as well as MicroScope help with tracing the enclave execution, they also help with reaching a point where sensitive data is present in memory, whereupon it can then be leaked using another attack. Unfortunately, such use of these attacks cannot be addressed by the enclave developer, and Intel does not address these through microcode updates either. Thus, it would be interesting to see future improvements to SGX to address these issues. To prevent control channel attacks on the enclave's page tables, it would be interesting to shift the responsibility of managing these from the OS to microcode. Furthermore, another area of research is to explore the possibility of shifting the responsibility of handling interrupts triggered during enclave execution from the OS to the enclave, such that the OS can no longer arbitrarily interrupt the enclave. In addition, for timer-driven interrupts, it would be interesting to explore adding jitter to the interrupt delivery to defend against single-stepping as well as inferring what kind of instruction is being executed.

3.2. Memory Corruption Attacks

Memory corruption vulnerabilities such as buffer overflows, use-after-free, and return-oriented programming may also affect enclave code, which the attacker can exploit to achieve code execution inside the enclave resulting in the ability to read and/or modify enclave memory, extract keys, etc. While SGX aims to provide confidentiality and integrity guarantees, memory corruption bugs and other bugs introduced by the enclave developer fall outside of the SGX threat model, thus the enclave developer is fully responsible for addressing these. Even more so, the fact that SGX is often used for sensitive data, makes it a much more interesting target for an attacker to find such bugs and exploit them.

In fact, Lee et al. [79] first demonstrated the viability of return-oriented programming attacks against SGX enclaves. Biondo et al. [20] further improved this work by showing the practically of exploiting such gadgets from userspace without enclave crashes. Furthermore, SmashEx [41] shows how enclave SDKs that do not carefully handle re-entrancy in the exception handler can be exploited.

Weichbrodt et al. [132] and Vicarte et al. [128] exploit the page faulting mechanism to interrupt enclave execution, to consequently control the scheduling order in multi-threaded enclave applications, leading to the exploitation of TOCTOU and use-after-free vulnerabilities.

Mitigation: One way of mitigating memory corruption bugs is to consider writing enclaves in a memory safe programming language. For instance, both Apache Teaclave [129, 130] and Fortanix eDP [46] are SGX frameworks that allow the development of enclaves in Rust. Furthermore, Enarx [44] allows enclave developers to target a more restricted WASM environment running inside Intel SGX.

While the operating system does provide a system-wide implementation of Address Space Layout Randomization (ALSR), an attacker can decide to disable ASLR for SGX applications. Thus SGX-Shield [112] implements an ASLR scheme in LLVM that can be deployed in SGX enclaves to harden memory corruption bugs from exploitation without the attacker having control over it. SGXBOUNDS [77] instead relies on pointer tagging to implement bounds checking by encoding the upper bound in the upper part of the 64-bit pointers. In addition, SGXFuzz [36] is a coverage-guided fuzzer that can be used to find memory corruption bugs in SGX enclaves.

3.3. Speculative Execution Gadgets

SgxPectre [32] shows that various speculative execution gadgets can be found in SGX enclaves where the attacker can first train the branch predictor or perform branch target poisoning to have the processor mispredict branches and as a result have it speculatively execute arbitrarily chosen code in the enclave. Such speculative gadgets can then leave microarchitectural traces, e.g. in the cache, that allows an attacker to extract sensitive data, such as keys, from SGX enclaves. Furthermore, Load Value Injection (LVI) [122] and Floating-Point Value Injection [102] shows that the attacker can manipulate the value returned by load instructions in speculative execution gadgets by injecting data into the microarchitectural buffers. We provide a more elaborate overview of attacks that can lead to LVI in Section 3.4. More specifically, this affects speculative gadgets that consist of two memory dereferences, where the attacker uses LVI to poison the first with the target address of interest and where the second leaves a different microarchitectural trace that is dependent on the secret value loaded by the first. Such gadgets allow attackers to read arbitrary enclave memory, which can then lead to the extraction of keys.

Mitigation: These issues all rely on the fact that the attacker can poison resources shared with the enclave, such

as branch prediction and micro-architectural buffers, before executing the enclave. Intel provides *Single Thread Indirect Branch Predictors* (STIBP) to prevent the sibling thread from influencing the branch prediction, and *Indirect Branch Restricted Speculation* (IRBS) to prevent the attacker from poisoning branch prediction before executing the enclave. Finally, *Indirect Branch Predictor Barrier* (IBPB) provides a barrier to prevent branch poisoning across the barrier. To fully prevent exploitation of speculative execution gadgets, enclave developers should use a compiler that inserts the `lfence` instruction after direct branches as well as deploy the retpoline mitigation for indirect branches, and may use code analysis tools such as fuzzers to locate such gadgets. Intel ships gcc with these mitigations as part of their Intel SGX SDK [56], and additionally these mitigations are available as part of LLVM and thus Clang as well as other LLVM-based compilers.

While Intel also recommends this for LVI, a similar methodology to the one for branch poisoning can be followed by disabling Intel Hyper-Threading and flushing affected buffers before entering the enclave. Incidentally, the microcode updates to address issues such as MDS, may thus also indirectly (partially) address LVI as they impose requirements to disable Hyper-Threading and may flush affected buffers before entering the enclave. However, without an official statement from Intel, an enclave developer cannot rely on such assumptions.

3.4. Leaking Enclave Data

We now focus on attacks that leak enclave data, and thus have an impact on confidentiality and integrity as they can be used to read enclave memory, CPU register values and ultimately extract sensitive data such as sealing keys, used by enclave to encrypt data to disk, and Intel's attestation keys, used to sign attestation reports. Thus, access to these keys allows an attacker to decrypt sensitive data sealed by the enclave and forge attestation reports from non-trusted hardware, respectively. The latter is especially problematic as it allows an attacker to run any enclave code they wish, outside of SGX, thus violating all integrity guarantees. In general, these attacks can either (partially) read enclave memory, or sample in-flight data from specific instructions during their execution. Furthermore, these attacks tend to rely on a page swapping mechanism that SGX provides to the untrusted OS, that allows an attacker to target specific enclave data without having to execute the enclave, which means that there is nothing the enclave developer can do to mitigate these attacks.

Foreshadow. Foreshadow [120] allows an attacker to leak the data for any physical address as long as the corresponding data is cached, essentially allowing an attacker to read enclave memory, including sensitive data that is part of the enclave, as well as gain access to Intel's attestation keys, which can be used to forge attestation quotes from a non-trusted machine. Furthermore, the attacker can read the CPU registers, as they are stored in memory whenever the enclave is interrupted. Similarly, SA-00219 [63] allows the integrated GPU to access the first 8 bytes of every cache line on affected processors, essentially providing access to the first 64 bits of the key returned by the `egetkey` instruction.

MDS. Microarchitectural Data Sampling (MDS) attacks such as RIDL, ZombieLoad and Fallout [31, 108, 123, 124, 126] target various micro-architectural buffers present in the CPU. Therefore these attacks can bypass most of the common security boundaries, including those between the application and SGX enclaves by leaking in-flight data from the SGX enclave through those buffers. In combination with the control flow attacks as outlined before, MDS attacks can be used to target in-flight data from specific instructions.

CacheOut. While MDS allows an attacker to target in-flight memory instructions [31, 108, 123] and vector registers [126], CacheOut [127] shows how to selectively evict data from the cache into these micro-architectural buffers to gain a primitive similar to Foreshadow.

CrossTalk. Furthermore, CrossTalk [103] shows that there are micro-architectural that can be sampled across cores the MDS attacks, and that enclaves can thus be attacked across cores. In particular, CrossTalk demonstrates how to extract an ECDSA private key from an enclave by sampling values from the `rdrand` instruction, but can also leak sealing keys from `egetkey` and indirectly Intel's attestation keys.

MMIO Stale Data. Finally, SA-00615 [68] presents another class of MDS attacks where misaligned or incorrectly sized accesses to device memory or memory-mapped I/O (MMIO) may cause stale data to be propagated from the uncore buffers to the fill buffers, or even become architecturally visible. Since this issue affects the `egetkey` and `rdrand` instructions, it provides similar capabilities as CrossTalk. Similarly, ÆPIC leak [21] shows how an attacker can read enclave memory using unaligned reads on the legacy APIC mapping to sample data from the superqueue, providing similar capabilities as Foreshadow.

Data at rest. To overcome the constraint of getting the enclave's data in the appropriate cache or micro-architectural buffer to leak it from, several works [21, 120, 127] rely on ability of the untrusted OS to use the `ewb` instruction to swap out enclave-owned pages and the `eldu` instruction to load them back in. An attacker can use this swapping mechanism with a variety of attacks to leak arbitrary enclave data, and as this mechanism is outside of the control of the enclave developer, there is nothing that can be done by the enclave developer to protect the enclave against these attacks, thus requiring Intel to provide mitigations.

Mitigation. Before we discuss mitigations, we provide an overview of the various attacks and the platforms they affect in Table 2 to give an idea of which attacks affect what platforms, and thus which platforms require what mitigations specifically. As we will see, most vulnerabilities share similar mitigation strategies, thus, even when a platform was previously unaffected, it eventually ends up with similar mitigations as new similar vulnerabilities get discovered and disclosed.

The first issue is that the attacker can run code simultaneously to the enclave executing on another CPU thread, CPU core or even the integrated GPU, requiring Hyper-Threading to be disabled for Foreshadow and MDS and

Attack Year		SKL 2015	KBL 2016	CFL 2017	CFL-R 2018	WHL 2019	CML 2020	ICL 2021	RKL 2021
Foreshadow [120]	2018	✓	✓	✓	✗	✗	✗	✗	✗
SA-00219 [63]	2019	✓	✓	✓	✓	✓	✗	✗	✗
MDS [31, 108, 123]	2019	✓	✓	✓	✓*	✓*	✗	✗	✗
CacheOut [127]	2020	✓	✓	✓	✓	✓	✗	✗	✗
Crosstalk [103]	2020	✓	✓	✓	✓	✓	✓	✗	✗
MMIO Stale Data [68]	2022	✓	✓	✓	✓	✓	✓	✓	✓
ÆPIC Leak [21]	2022	✗	✗	✗	✗	✗	✗	✓	✓
PlunderVolt / V0LTpwn [74, 94]	2019	✓	✓	✓	✓	✓	✓	✓	✗
Platypus [86]	2020	✓	✓	✓	✓	✓	✓	✓	✗

TABLE 2: An overview of the SGX attacks and the affected platforms. **SKL**: Skylake (e.g. Intel Core i7-6700K), **KBL**: Kaby Lake (e.g. Intel Core i7-7700K), **CFL**: Coffee Lake (e.g. Intel Core i7-8700K), **CFL-R**: Coffee Lake Refresh (e.g. Intel Core i9-9900K), **WHL**: Whiskey Lake (and Amber Lake) (e.g. Intel Core i7-8665U), **CML**: Comet Lake (e.g. Intel Core i9-10900K), **ICL**: Ice Lake (e.g. Intel Core i7-1165G7, **RKL**: Rocket Lake (e.g. Intel Xeon E-2334), *****: affected by Vector Register Sampling (VRS) and TSX Asynchronous Abort (TAA).

the integrated GPU for SA-00219. Second, whenever the enclave exits, sensitive data may be lingering around in the caches or micro-architectural buffer, whereupon the attacker can leak this data, thus requiring flushing the caches (e.g. Foreshadow) and micro-architectural buffers (e.g. MDS) upon enclave exit. Finally, some attacks, such as Crosstalk, affect micro-architectural buffers shared between multiple CPU cores, which requires serialized access to these buffers as well as flushing to ensure no data lingers around. Thus for each of these attacks Intel has to provide, and has provided, a microcode update implementing these mitigations.

As the SGX swapping mechanism is of interest to attackers, a future improvement would be to provide enclave developers to mark enclave pages as non-swappable, such that sensitive pages cannot be swapped out by an attacker. While this does not address the root cause of these attacks, it does severly limit the applicability of the attack. Another improvement would be to add an instruction to check that the arguments to enclave function calls strictly points to valid DRAM memory, which could help address the issue of an attacker providing pointers to memory-mapped I/O, as the enclave has no access to the physical addresses.

3.5. Fault Attacks

PlunderVolt [94] and V0LTpwn [74] abuse interfaces for dynamic voltage scaling on x86 CPUs to perform fault injection on SGX enclaves. More specifically, Plundervolt shows how to perform fault injection on the AES-NI instructions as well as on the `ereport` and `egetkey` instructions. The keys used for AES-NI can then be extracted through differential fault analysis. PLATYPUS [86] uses Intel RAPL to monitor the power consumption of instructions running inside an SGX enclave to infer sensitive data. This information can then be used to extract keys from the AES-NI instructions as well as be used to determine the control flow of branches among other attack scenarios.

Mitigation: Since access to Intel RAPL and the MSRs to control the CPU voltage is outside of the enclave developer's control, Intel released microcode updates that disables access to these interfaces. Access to these interfaces is required to be disabled in order to reach fully trusted status.

More broadly speaking, interfaces that can be used to monitor resource consumption (e.g. power, frequency, temperature, performance counters, etc.), and thus infer sensitive information from the enclave's execution, should not be accessible or should not be updated to reflect the enclave's execution. Similarly, interfaces that affect the enclave's execution, such as adjusting the voltage, should also not be accessible or SGX should maintain its own parameters while executing the enclave.

3.6. Summary & Discussion

To summarize, we have provided an overview of the various SGX attacks, whether the developer or Intel is responsible for their mitigation, what an attacker can achieve with these attacks, and what impact that translates to for an enclave developer.

Mitigations for Developers. First, we discussed attacks such as inferring access patterns, speculative execution attacks and memory corruption attacks, While these can be mitigated through compiler extensions and code analysis tools, there are also options such as the use of constant-time cryptographic primitives and SGX frameworks in Rust.

Mitigating poisoning & leakage. Next, attacks that exploit speculative execution and other CPU issues are not exclusively the developer's responsibility, and also require microcode updates that prevent the sibling thread from poisoning state shared. Furthermore, to mitigate attacks that read enclave memory, the microcode should flush any such shared state on enclave entry, on enclave exit and when swapping enclave pages. The microcode should also prevent poisoning or leaking state from concurrently running applications on the same hardware. Finally, it is up to the enclave developer whether their enclave is allowed to run with features such as Intel Hyper-Threading enabled or not.

Restricting Control. We also discussed mitigations that essentially restrict the control an attacker has, and in some cases even provide enclave developers with additional control over the enclave's execution environment. More specifically, we discussed solutions that move ASLR into the enclave, remove the OS responsibilities to manage page tables and interrupt handling for the enclave, limit the SGX

swapping mechanism, and limit access to interfaces that monitor or affect enclave execution.

4. Surveying SGX Update Timelines

Having provided an overview of the various SGX attacks and having discussed the various mitigations to deploy for these attacks, we now look at how long it takes for Intel's mitigations to reach the end-users of SGX. When a new SGX vulnerability is discovered, Intel typically issues a microcode update for most affected architectures. These updates are not persistent, rather they are re-applied every time the computer boots. Being unable to trust the (potentially malicious) operating system to keep SGX updated, Intel collaborates with motherboard vendors to distribute SGX updates through BIOS updates.

The Difficulty of SGX Updates. We argue that this BIOS-driven SGX update process presents two security issues. The first is that BIOS updates are manual, potentially dangerous, and generally only recommended if absolutely necessary. Next, compared to regular software updates, BIOS updates are often released very slowly, and in some cases not at all.

In this section, we aim to shed light on SGX update timelines, by quantifying the time duration between SGX vulnerability disclosure and microcode availability.

BIOS Scraping. We conducted a web scraping campaign in which we downloaded and analyzed every BIOS update we could find for six major manufacturers: ASRock, Dell, HP, Lenovo, MSI, and Gigabyte. As we found BIOS update documentation to be inconsistent and generally unhelpful, we opted instead to programmatically analyze each update to search for relevant microcode patches using both MC Extractor [88] and our own microcode header parsing tool. In all, we identified roughly 173,000 microcode updates, where about 5300 fixed a specific known attestation-breaking SGX vulnerability on a unique device for the first time.

Unfortunately for our analysis, BIOS update histories are not always well-kept: old updates are sometimes removed with no record of what they contained and we have to assume that the claimed upload times of different updates are accurate. Furthermore, BIOS packing methods vary greatly between manufacturers and even product lines, making it difficult to determine if all microcode updates have been extracted. Because of these limitations, we only count updates which include the specific microcode patch which fixes a vulnerability we consider, and do not make any claims about how many vulnerabilities are never patched.

SGX Update Lifecycle. We analyzed six common SGX vulnerability patches [4, 5, 6, 7, 8, 9] and cataloged the BIOS updates which applied them. Figure 2 presents a summary of our findings, plotting the percentage of products with available SGX patches against the elapsed days since public vulnerability disclosure, optimistically assuming that every patchable product is patched by the end of the survey.

Among the surveyed vendors, the median patch time ranged from 25 days (HP) to 125 days (Lenovo). Overall, the median vendor had a median patch time of 52 days or almost two months. Issue times varied by vulnerability[2] but we emphasize the large variance in responses: some product lines shipped patches before the vulnerability's disclosure, while others took many months (if they shipped at all).

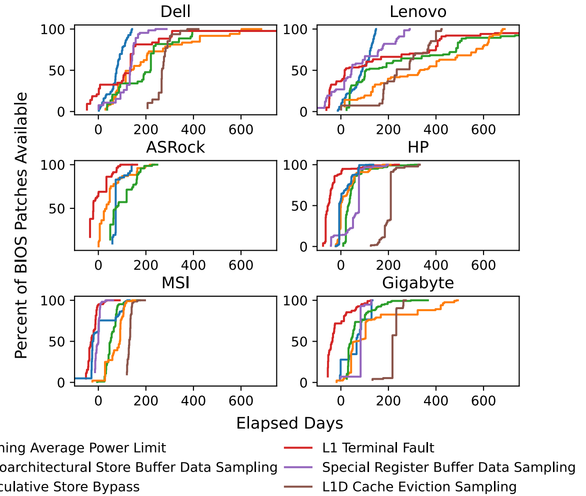

Figure 2: Time Taken To BIOS-Patch Major SGX Vulnerabilities After They Are Made Public

Hardware Update Lifecycle. For comparison purposes, we also discuss how SGX patching compares to other patching and update mechanisms, such as general microcode-to-BIOS propagation. Here, we survey the same six vendors but look at the time of propagation of all microcode updates to BIOS updates (security-critical or not) for both Intel and AMD. Figure 3 presents a summary of our findings, plotting the percentage of products with BIOS updates containing the most recent microcode against the elapsed days since that microcode patch was first seen in any BIOS update.

Among the vendors we survey, we notice a similar trend as before, with HP and MSI having the fastest median update time when releasing Intel microcode updates at 37 days, while Lenovo has the slowest at 329 days. For AMD microcode update propagation, MSI has the fastest time at just over 70 days, while Dell and Lenovo had the slowest at 477 days and 1043 days respectively[3]. Interestingly, we notice a distinct difference between the Intel and AMD microcode propagation times with median times of 61 days and 98 days respectively. We leave analysis of why this might be the case as an open problem for future work.

Comparison. We conclude that BIOS updates are generally slow, regardless of the motherboard manufacturer or processor vendor, even though security-critical microcode propagates quicker than general purpose microcode updates.

2. L1D Cache Eviction Sampling [8] is a notable outlier in our dataset, since in that instance Intel publicly released a fix three months after they first acknowledged the issue (though some Lenovo products received a patch months earlier than other vendors)

3. We note that, based on our survey, Lenovo shows exceptionally long update times for AMD microcodes. We do not know for sure why this is, but hypothesize that this is likely a function of our methodology, which assumes that the first available BIOS update to contain a specific microcode patch must be a microcode update.

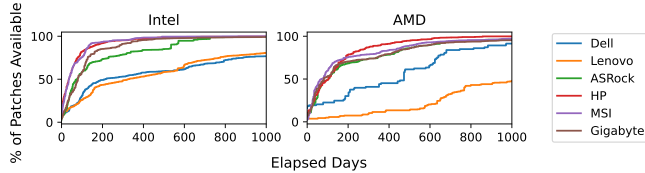

Figure 3: Time Between Microcode Releases and Their Integration into a BIOS Update

We contrast this with the update life-cycle and timeline for security-critical bugs in software. Li et al. provide a detailed study on exactly this topic, noting for example that for 78.8% of all CVEs, security fixes were released by public disclosure time, manifesting essentially a zero time difference [81]. This demonstrates the stark difference in patch propagation time in software versus hardware.

A Difficult Tradeoff. Even when Intel's microcode patches are available and even under the optimistic assumption that all devices eventually get patched, we are left with a troubling usability issue. With multiple SGX breaks in a year, combined with multiple months of patch delay per break, service providers are required to strike a fine balance between usability and security with respect to trusting SGX.

Ideally, vendors would require that products only run on fully updated machines, in `TRUSTED` status, which can presumably securely contain the vendor's secrets. However, the difficulty of installing BIOS updates means that achieving `TRUSTED` status is cumbersome for regular users, which limits the product's market share, and hurts user experience. Alternatively, vendors can prefer compatibility over security, storing sensitive information inside SGX enclaves running on unupdated machines with `GROUP_OUT_OF_DATE` attestation status. As we show in Section 6, this choice can have serious security consequences, up to and including the removal of all secrecy and integrity properties.

TCB Recovery. To further exemplify this trade off, we look at xAPIC and MMIO issues [10, 11, 12, 21] that were officially dislcosed on August 9, 2022. We found that, as of writing, platforms affected by these xAPIC and MMIO issues are still in `TRUSTED` status two months after this disclosure date, with TCB recovery originally planned to happen no later than March 7, 2023 [67] for platforms using Intel EPID attestation, jeopardizing any application of Intel SGX. Only following our disclosure, Intel has changed this timeline to November 29th, 2022 [66] for some platforms, with popular SGX-enabled servers and desktops being updated only in January 2023.

5. Unsealing The Secret Network

It is important to note that the majority of theoretical attacks that occur on TEEs (SGX in particular) happen within research labs. In reality, common attack vectors occur through implementation faults that leverage holes in protocol design.
-SCRT Network Graypaper

5.1. Secret Overview

In blockchain systems, *smart contracts* are stateful programs that users can interact with. They can make automated decisions regarding the transfer of assets, from ownership of virtual property such as Non-fungible Tokens (NFTs), as well as decentralized finance (DeFi) mechanisms such as auctions and automated market makers. Most blockchains are completely transparent by design and all smart contract state and transaction data can be reviewed by anyone. To regain privacy, the smart contracts community focused mostly on using zero knowledge proofs [22, 30, 48, 76]. However, as this incurs considerable performance overhead, several research projects have proposed an alternative TEE-based approach [23, 33, 43, 73, 113, 117], by moving the execution of smart contracts entirely into the enclave.

The SECRET Network. The first TEE-based blockchain to reach significant adoption is SECRET network, which launched its smart contracts feature in September 2020, and has since grown to a total market cap of 150M USD as of early October 2022. DeFi apps in use today on SECRET include a Uniswap-like automated market maker and a Compound-like automated margin lending system. Another notable application is Private NFTs that can attach encrypted payloads that only the owner can access.

Overview of the SECRET Architecture. SECRET consists of two components: a consensus protocol based on Tendermint [28] to commit transactions and to serve as a bulletin board, and an SGX-based smart contract execution layer. Tendermint consensus does not use enclaves, rather it uses Proof of Stake, requiring significant security deposits to propose blocks. In SECRET, block proposers must be among the top 80 nodes by stake [109] (a minimum stake of 38,692 SCRT or $35,100 at the time of writing [111]).

SGX-based Smart Contract Execution Setup. Secret's smart contract execution framework is derived from Cosmos, but adapted to run within an enclave. To send a message to a smart contract, users derive an encryption key from a master public key, `consensus_io_exchange_pubkey`, and include the encrypted message in a transaction. The corresponding secret key, derived from the *consensus seed*, is replicated throughout the network as files sealed by SGX enclaves. A seperate `consensus_state_ikm` key, also derived from the consensus seed, encrypts the database of the current state, e.g., account balances.

Performing Transactions. Once a transaction is committed to the network, the enclaves decrypt the message and execute the contract, updating the encrypted state. The consensus seed is persistent and has not changed over the lifetime of the blockchain, allowing all validator nodes to inspect the blockchain's state at any time.

New Node Registration. To add new enclave nodes to the network, SECRET implements a registration process based on remote attestation. First the new node invokes `ecall_keygen()` to create an ephemeral keypair, for future use to seal the new node's local copy of the consensus seed. The corresponding public key, along with a verified attestation report from IAS, is packaged within a transaction and published on SECRET's blockchain.

Observing the transactions published by the joining node, existing nodes invoke `ecall_authenticate_`

`new_node` in their own enclave, passing in the verified attestation report. The ecall verifies the IAS response, checks that the joining node's *mrenclave* value matches the one used by existing nodes, and verifies that the node's hardware platform is secure against known SGX vulnerabilities. If all checks pass, existing nodes encrypt their consensus seed using the joining node's public key, sharing the resulting ciphertext on the blockchain. Finally, the joining node observes this ciphertext on the blockchain, and passes it to its enclave. Upon decryption, the enclave stores the consensus seed locally using SGX protected FS, avoiding the need to continuously re-attest upon reboots and platform upgrades.

SECRET's Integrity and Privacy Guarantees. While SECRET is designed so that an SGX breach cannot affect the integrity of the blockchain or allow for theft of funds or freely issuing new tokens, it nonetheless can eliminate its privacy guarantees, essentially downgrading SECRET to an ordinary transparent blockchain and allowing attackers to read the internal state of all smart contracts.

5.2. Extracting the Consensus Seed

Hardware Setup. We begin our attack by setting up an SGX-capable CPU which is vulnerable to ÆPIC leak [21]. SECRET's documentation states[4] that nodes are required to use SGX with Intel's Server Platform Services (SPS), leading its community to believe that AEPIC leak did not affect the network, as no architectures attacked directly in the paper supported SPS. We thus investigated Rocket Lake CPUs, as this is the only Xeon architecture which supports EPID-based attestation and is sufficiently new to be potentially vulnerable to AEPIC leak. Thus, we acquired an HPE ProLiant DL20 server equipped with an Intel Xeon E-2334 (Rocket Lake) CPU and installed Ubuntu 20.04 LTS with Linux kernel 5.4.0. Finally, despite being reported on August 2022 Intel did not perform TCB recovery at the time of writing, allowing our machine to run microcode version 0x53 while still being considered trusted by the IAS.

Node Setup. Next, to obtain a copy of the consensus seed inside our node's SGX enclave, we registered our hardware onto the network as a validator node. While joining SECRET's active block proposer pool requires a substantial investment, merely running a non-proposing validator only requires passing attestation. This is a deliberate design decision made by SECRET, as block explorers, developer-friendly API endpoints and other services benefit from the ability to make queries against the encrypted state.

Attacking the Enclave. To guarantee confidentiality, the enclave relies on the Intel Protected File System Library to seal and store the seed and key to disk. More specifically, the consensus seed is sealed using 128-bit AES-GCM and stored as `consensus_seed.sealed` in `/opt/secret/.sgx_secrets`. The SECRET source code reveals that `ecall_init_node()` (in `librust_cosmwasm_enclave.signed.so`), retrieves and expands the AES sealing key to then unseal the consensus seed. Moreover, the enclave's symbol table shows the offset of `k0_aes_DecKeyExpansion_NI()`. Thus, we simply target `ecall_init_node()` through the `init_node()` function in the Go bindings (`libgo_cosmwasm.so`), use a control channel attack [136] to stop the enclave after the key expansion, and then proceed to recover the AES key.

4. During our disclosure process, we discovered that SECRET made this assertion in error. In particular, SECRET also allows for validator nodes using non-server machines, thus increasing their exposure to attacks using other architectures such as Ice-Lake based laptops.

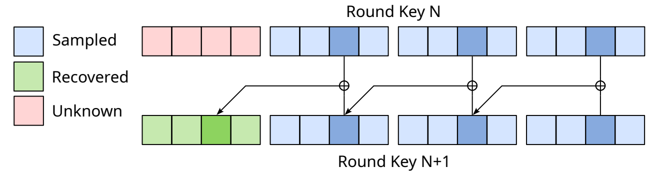

Figure 4: A simplified 128-bit AES key schedule for two consecutive round keys showing the XOR relationship between byte triplets, with the sampled (blue), recovered (green) and unknown (red) bytes.

AES Key Recovery. Since 128-bit AES performs 10 encryption rounds, the key scheduling algorithms expands the AES key into 10 round keys, consisting of four 32-bit words each. As ÆPIC leak can only target data from even cache lines, and more so only the last three 32-bit words of every round key, we exploit the redundancy of the AES key schedule [52] to fully recover one of the round keys, and thus the AES key. Figure 4 shows the bytes we can sample from round key N and $N+1$ using ÆPIC leak in blue. To compute any word k in round key $N+1$, except for the first, we simply XOR word $k-1$ in round key $N+1$ with word k of round key N. Thus, as XOR is symmetric, we can compute word $k-1$ of round key $N+1$ given word k in both round keys. This allows us to identify the AES key schedule, as well as recover the first word of round key $N+1$ as shown in Figure 4 in green.

Once we recover one of the round keys, we can reverse the key schedule to recover the initial AES key. However, unaware of which round key we recovered, we must bruteforce the AES key for each index, and try to decrypt the sealed data using each of the 10 recovered keys. As authenticated encryption (128-bit AES-GCM) is used, decryption only succeeds if the authentication tag matches, allowing us to identify the correct key.

Unsealing the Consensus Seed. After successfully recovering the AES sealing key, we proceeded to decrypt the sealed files used by SECRET. Since the consensus seed is only 32 bytes in size, the sealed file only consists of a single 4 KiB block that serves as the root block. Thus we decrypt this block using the extracted key to obtain the consensus seed.

Extracting EPID keys. Note that the aforementioned AES key extraction process is not limited to the AES sealing key used by SGX protected FS, but is also applicable to the sealing key protecting the machine's private EPID attestation key [21, 120, 125, 127]. Demonstrating this empirically, we were able to extract our server's EPID key using a similar methodology. We conjecture that once an attacker extracts the EPID key from a trusted platform, the attacker

can bootstrap an entire SECRET node outside of an SGX enclave. More specifically, the attacker would generate a key pair, sign their own quote containing that public key and the appropriate enclave measurement, and then retrieve a valid IAS certificate from Intel, whereupon the other node validates the certificate and encrypts the consensus seed using the provided public key.

5.3. Decrypting Transactions

As mentioned above, one of the main applications of the SECRET network is privacy preserving transactions, allowing parties to privately transfer control over digital assets. While all transactions, including encrypted ones, can be viewed via common block explorers (e.g., secretnodes.io), using keys derived from the consensus seed we are able to directly decrypt any transaction, completely breaching SECRET's confidentiality guarantees. To demonstrate this, we decrypted our own mainnet transaction and obtained its JSON description.


```
{"transfer":{"recipient":"secret1...","amount":"1333370"}}
```


Decrypting Private NFTs. Besides private payments, another use case for SECRET has been the creation of private NFTs, such as a collection released by famous movie director Quentin Tarantino [78]. Through the use of viewing keys, the NFT's current owner can query the exclusive private metadata and view the contents, hiding it from other network users. To exemplify the danger of our compromise to such assets, we created our own private NFT with a hidden image. We then subsequently breached our NFT's confidentiality on SECRET's mainnet using the extracted consensus seed. See Figure 5.

Figure 5: Example NFT that we created and then decrypted.

Deanonymizing SECRET. Arguably the most concerning application of our attack, however, is bulk financial surveillance of SECRET users. With the consensus seed, we could reconstruct all the confidential account balances and transfer histories of SNIP-20 fungible tokens, which include popular bridged assets like Ethereum and the USDC stablecoin. While users of SECRET desire and expect privacy, the compromise of the consensus seed threatens them with comprehensive retroactive surveillance.

6. CyberLink PowerDVD

For our second case study, we investigate the usage of SGX in CyberLink PowerDVD 20, a popular software application to play UHD Blu-rays on computers. PowerDVD provides us with an interesting case study, as its user base is very different than SECRET's and consists of a large and potentially non-technical set of users. As PowerDVD is closed-source, this involved a significant reverse-engineering effort to understand its usage of SGX, which we now describe. Afterwards, we describe our attack on PowerDVD and AACS2 key exraction. To help with our efforts, we leveraged EGX to make PowerDVD's SGX enclave pass attestation while in debugging mode, thus allowing for introspection as the enclave runs. This is typically impossible to do with a production-level enclave, which are properly attested to have SGX's security guarantees.

6.1. Reversing PowerDVD

While the PowerDVD application contains numerous files and binaries, there are only a key few that relate directly to AACS2 and SGX. We list and briefly describe them here:

- `CLTA.dll`: CyberLink's Trusted Agent containing PowerDVD's AACS2 implementation. This DLL is hardened with Themida [3], a commercial software obfuscator.
- `CLTA_SW.dll`: Similar to `CLTA.dll` except not obfuscated, and all SGX-related functions are stubbed.
- `CLKDE.dll`: Provides the CyberLink Key Downloader Enclave, a production enclave to provision AACS2 keys.
- `CLTE.dll`: Provides the CyberLink Trusted Enclave, a production enclave for AACS2 algorithms. This DLL is encrypted via SGX's PCL (Protected Code Loader) [57].

Playing UHD-BDs. Upon clicking "Play", PowerDVD begins the disc loading process, referencing `CLTA.dll` (and AACS2 code), to call the first SGX-related function.

Initializing SGX. PowerDVD first checks if the SGX driver is properly initialized before loading and calling into `CLTA.dll`. If the verification fails, playing should be aborted as per the AACS protection requirements [39]. Curiously, PowerDVD still tries to play the movie, loading `CLTA_SW.dll` instead of `CLTA.dll`. As all high level SGX-related functions in `CLTA_SW.dll` are unimplemented stubs, playback eventually fails. We cannot explain the presence of `CLTA_SW.dll`, and hypothesize that this is an internal debug module that should not have been shipped.

`CLTA.dll` Patching. With SGX successfully initialized, PowerDVD proceeds to call `CLTA.dll`. To analyze its contents, we de-obfuscated `CLTA.dll`'s Themida protection, allowing for further reverse-engineering. This task was assisted by the presence of the unprotected and very similar `CLTA_SW.dll`, which contains much of the same code, as well as useful debugging print statements and log messages.

Additional patching of the main PowerDVD executable was needed to allow it to run with a debugger attached, up to the point that control flow enters an SGX enclave.

Checking For Existing Keys. We can now explore PowerDVD's core component involving SGX: the key provisioning process. At this point, `CLTA.dll` verifies that the AACS2 device keys exist on the device before continuing. If `CLTA.dll` cannot find an encrypted key file at this time, it assumes that the AACS2 keys need to be updated and begins the key downloading process from CyberLink's provisioning servers. Note that unlike legacy AACS 1.X software players, PowerDVD does not ship any AACS2 keys, requiring at least one key download to play AACS2 protected content.

AACS2 Key Provisioning. To download AACS2 keys `CLTA.dll` loads `CLKDE.dll`, the CyberLink Key Downloader Enclave, to handle remote attestation and AACS2 key provisioning. Reverse-engineering `CLKDE.dll`, we found CyberLink's implementation of attestation to be largely standard, behaving like Intel's reference implementation [59], see Section 2.2 for protocol details and Appendix E.1 for a list of ECALLS and functionality.

Should Remote Attestation succeed, CyberLink's provisioning server returns a Base64-encoded blob to the CyberLink Key Downloader Enclave. We discovered that the blob is encrypted with AES-GCM using a static, hardcoded key and IV, and in fact contains all necessary cryptographic material required for playing UHD Blu-ray discs.

Blob Sealing. Upon decrypting the blob, the CyberLink Key Downloader Enclave uses CyberLink's `mrsigner` to seal it for access by any CyberLink signed enclave.

6.2. Attacking PowerDVD

Having reversed-engineered PowerDVD's SGX use, we now present an end-to-end attack on Blu-ray DRM, allowing us to extract AACS2 key material. With these keys, we can playback UHD-BD movies on any hardware, regardless of SGX status or availability, as well as clone UHD-BD disks.

Step 1: Obtaining Private Attestation Keys. We observe that PowerDVD agrees to play AACS2-protected movies on machines with a `GROUP_OUT_OF_DATE` attestation status. Thus, we begin by extracting the SGX attestation keys from such a machine, by mounting the Foreshadow attack [120] on an unpatched Skylake i7-6820HQ CPU.

Step 2: Constructing a Rogue Quoting Enclave. Having obtained private attestation keys, we now craft a Rogue Quoting Enclave (QE) which subverts SGX's attestation process. In our case, the rogue QE ignores actual measurements of the enclave for which the Quote is to be generated, sets the measurement data to some desired values, and follows the remaining logic of an unsubverted QE in order to sign the resulting Quote with the keys obtained in Step 1. Finally, note that as the attestation keys used are provided externally, the Rogue QE need not be an actual SGX enclave, but in our case is a mere Python script implementing the same logic.

Step 3: Extracting CLKDE. We now extract CyberLink's Key Downloader Enclave (CLKDE) from inside SGX, running it as a normal binary. We then have our extracted CLKDE call the Rogue QE from Step 2 to produce a signed Quote. The Rogue QE in turn signs the extracted CLKDE using CyberLink's original `mrsigner` and `mrenclave` values for CLKDE, via the keys provided in Step 1.

Step 4: Obtaining AACS2 Keys. With the extracted CLKDE receiving a signed Quote with correct `mrsigner` and `mrenclave` values, we use the extracted CLKDE to contact CyberLink's AACS2 provisioning service. Being unable to distinguish our extracted CLKDE logic from a genuine CLKDE enclave, CyberLink proceeds to provision our extracted CLKDE application with secret key material despite it being executed entirely outside of SGX. At this point, we are able to receive production AACS2 device keys and host certificates, without ever using SGX hardware.

Step 5: Decrypting Blu-ray Discs. The possession of AACS2 keys can also be used to entitle software players other than PowerDVD to play UHD Blu-ray discs. To demonstrate this, we modified the open-source `libaacs` plugin for the VideoLAN VLC video player software to support the new AACS2 specifications and algorithms we discovered. When supplied with the keys extracted from the CyberLink server's provisioning payload, we were able to playback an unmodified UHD Blu-ray from a licensed AACS2 disc drive using VLC, on a Linux machine running without any SGX support. This constitutes a complete bypass of AACS2 DRM, as PowerDVD requires both Windows and SGX to operate, thus formerly limiting UHD-BD playback to only SGX-enabled Windows platforms.

6.3. Extracting The AACS2 Protocol Code

Unlike for AACS 1.0, AACS-LA (the consortium which maintains AACS standards) decided to not publish any publicly-available specifications for AACS2 and typically mandates that any AACS code is obfuscated and/or encrypted in some form [39]. For the case of PowerDVD, the CLTE enclave containing the AACS2 algorithm is encrypted on disk, using an SGX feature called Intel SGX PCL (Protected Code Loader) [57]. To decrypt the code and data sections of the application, the PCL unseals a symmetric AES-GCM key, and decrypts the sections in-place.

AACS2 Code Extraction. Inspecting the key blob obtained in Step 2 of Section 6.2, we discovered that it contains the AES-GCM key required for decrypting the code implementing AACS2. This in turn allowed us to analyze this protocol, presenting its implementation in Appendix C.

The Curious Case of `CLTA_SW.dll`. Finally, we further analyzed the contents of `CLTA_SW.dll`. Remarkably, we discovered that `CLTA_SW.dll` contains nearly identical code, strings, and cryptographic constants as the `CLTE.dll` enclave. In particular, `CLTA_SW.dll` contained the code for the AACS2 algorithm, without any protection. The latter is significant, as it directly violates AACS-LA's requirement to not publish AACS algorithms without obfuscation and/or encryption. We thus reaffirm our hypothesis from Section 6.1, that `CLTA_SW.dll` is an internal debugging module that should not have been shipped.

6.4. The AACS2 Protocol

As part of this study, we are also able to provide the first public presentation and deployment analysis of the Advanced Access Content System (AACS) 2.0 and 2.1 protocols, as well as do a deep dive into how AACS actually

manages keys and revocations in practice, which, while not directly related to the security of SGX deployments, may be of independent interest. To summarize briefly, we note that much of the AACS 2.0 and 2.1 protocols are similar to the publicly released 1.0 specification [38], with notable exceptions being an improvement to the traitor-tracing scheme, an altered Media Key derivation process, and an update to more modern cryptographic primitives. The AACS2 protocol seems technically well designed, but is nonetheless defeated because of reliance on SGX's security.

We describe the full details in Appendix C for the benefit of future researchers.

7. Mitigations, Discussion, and Conclusion

In this paper we studied SGX attack techniques, categorized the impact and information leaked by them, as well as discussed what countermeasures are available for enclave developers. We also showed that wide scale deployments of SGX-based applications can be hindered by the slow update cycle of CPU microcode. This forces software vendors into difficult choices between security and usability, which are often very hard to balance correctly. We now describe mitigations for SECRET and PowerDVD, as well as general directions for future TEE designs based on these case studies and the insights from our study.

SECRET Mitigations. Working with SECRET, the first action was to revoke SECRET's developer keys, which prevented the creation of new attestation reports, thereby blocking the registration of new nodes. While this reduced the attack surface, it alone is not sufficient, as vulnerable machines with existing attestation reports might still be provisioned with the network's consensus seed which can be subsequently extracted.

The root cause of our attack is that Intel's IAS server still reported machines vulnerable to xAPIC and MMIO issues [10, 11, 12, 21] as trusted for multiple month after these vulnerabilities were made public. Luckily, in addition to the machine's security status, attestation reports also contain the `group-id` of the attesting machine, which uniquely identify its CPU architecture and microcode version. While Intel does not publish this mapping, we have worked with SECRET and Intel to make sure that `group-id`'s representing machines vulnerable to xAPIC [12] issues are not allowed to be registered on the SECRET network.

However, SECRET still allows machines affected by MMIO issues [10, 11] to be registered as validator nodes. With TCB recovery for some popular SGX-enabled platforms scheduled for January 2023 [66], further attacks on the SECRET network might be possible.

Hard Fork and New Consensus Seed. As Intel's xAPIC and MMIO issues [10, 11, 12, 21] have been public since August 2022, it is impossible to ensure the confidentiality of the network's current consensus seed. This is concerning, as it allows attackers to decrypt the network's entire state.

While SECRET's current protocol makes it quite difficult to update the consensus seed, SECRET does plan to implement a hard fork in its current blockchain, moving the new chain to a freshly generated seed. While ad-hoc measures such as deliberate erasure of the existing seed was performed by node operators, it is the unfortunate reality that the privacy of all transactions present in the current chain should assume to be compromised.

PowerDVD Mitigations. The deprecation of SGX on client-oriented hardware, together with PowerDVD's likely audience of non-technical users, poses a significant challenge in obtaining a secure client-oriented deployment. In particular, having to support older hardware which cannot obtain trusted status due to numerous SGX side channel vulnerabilities [21, 31, 61, 62, 63, 64, 65, 68, 94, 103, 108, 120, 122, 123, 127] puts PowerDVD at risk of compromise. We thus recommend that PowerDVD divides its user base into small groups, provisions each group with different AACS keys, employs the AACS 2.1 traitor tracing mechanism. In case a 4K copy of a Blu-ray disk is discovered, PowerDVD can quickly revoke the compromised key, preventing it from further compromising future disk releases.

Future Designs of Trusted Execution Environments. Following from our overview of attacks and our discussion of the mitigations in Section 3, future research should look to generalize the idea of constant-time code for attacks where inferring access patterns can be used to extract sensitive data. We also looked at the current mitigation strategies for attacks that leak enclave data, which involve flushing various affected caches and buffers on enclave entry and exit, and ensuring other CPU threads and cores cannot poison these caches and buffers or infer sensitive data from them. As we expect to see more similar vulnerabilities in the future, we expect them to require a TCB recovery as well as similar mitigations. Finally, we discussed several improvements to consider for the SGX swapping mechanism, the page table management and interrupt handling to harden Intel SGX against these attacks.

Faster and Seamless TCB Recovery. From a usability perspective, PowerDVD cannot expect users to continuously install BIOS updates to keep their system in `TRUSTED` status. Thus, PowerDVD allows SGX platforms to be in `GROUP_OUT_OF_DATE` status to playback AACS2 content. This choice means that an attacker can use a number of attacks outlined in Section 3 to extract Intel's attestations keys and then use these to extract AACS2 keys. Instead of having motherboard vendors integrate microcode updates as part of BIOS updates that users of SGX would then have to install, one possible solution would be to shift this responsibility to the Intel Management Engine (ME), such that Intel can transparently deploy SGX-related microcode updates and perform the TCB recovery through the Intel ME independently of the motherboard vendors. Of course, as moving this responsibility to the Intel ME comes with many unique challenges, we leave this open to future research and discussion.

Planning TCB Recovery. As discussed in Section 4, it can take quite a while from the disclosure of an SGX vulnerability to actually deploying BIOS updates containing the appropriate microcode update to perform the TCB recovery. With the original TCB recovery data planned to be March 7,

2023 for the xAPIC and MMIO issues, the SECRET network would have been vulnerable for seven months after the disclosure date. Thus, for enclave developers it is paramount to push Intel and partners to move such TCB recovery dates to be as early as possible to minimize the risk and impact from these vulnerabilities. Furthermore, given the history of SGX attacks, as outlined in Section 3, enclave developers should expect more vulnerabilities to be discovered and disclosed. As such, enclave developers must understand the implications and potential data leakages of different types of attacks and what needs to be done to protect against them. It is also critically important for enclave developers to carefully lay out a TCB recovery plan, in which they prevent machines from running their enclave on affected machines the moment a vulnerability is known to prevent these machines from acquiring sensitive data to defend against attackers that are tipped off by the public disclosure. Finally, they need to carefully think about how to invalidate sensitive data on platforms that already executed the enclave and acquired sensitive data, while the platform was trusted, as an attacker can now extract this data after the fact.

Acknowledgments

This work was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award DE200101577; an ARC Discovery Project number DP210102670; the Blavatnik ICRC at Tel-Aviv University; the National Science Foundation under grant CNS-1954712, CNS-2047991, CNS-2112726 and CNS-1943499; and gifts from Intel and Qualcomm.

Parts of this work were undertaken while Yuval Yarom was affiliated with Data61, CSIRO.

References

[1] Intel Product Specification Advanced Search. Intel. https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html. 23

[2] PowerDVD. CyberLink. https://www.cyberlink.com/products/powerdvd-ultra/features_en_US.html. 2

[3] Themida. Oreans. https://www.oreans.com/Themida.php. 12

[4] SA-00115. Intel, May 2018. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html. 9

[5] SA-00161. Intel, August 2018. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. 9

[6] SA-00233. Intel, May 2019. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html. 9

[7] SA-00320. Intel, June 2020. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00320.html. 9

[8] SA-00329. Intel, January 2020. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00329.html. 9

[9] SA-00389. Intel, November 2020. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html. 9

[10] SA-00615. Intel, June 2022. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00615.html. 2, 3, 10, 14

[11] SA-00645. Intel, June 2022. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00645.html. 2, 3, 10, 14

[12] SA-00657. Intel, August 2022. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00657.html. 2, 3, 5, 10, 14

[13] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, 2019. 5

[14] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, pages 53–70, 2016. 6

[15] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013. 3, 4

[16] Arm. Arm TrustZone technology. https://developer.arm.com/ip-products/security-ip/trustzone. 1

[17] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014. 6

[18] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, 2005. 20

[19] Daniel J Bernstein and Bo-Yin Yang. Fast constant-time GCD computation and modular inversion. *TCHES*, 2019. 6

[20] 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*, 2018. 5, 6

[21] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security*, 2022. 2, 3, 5, 7, 8, 10, 11, 14

[22] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *IEEE SP*, 2020. 10

[23] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private data objects: an overview. *arXiv preprint arXiv:1807.05686*, 2018. 10

[24] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and

Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *WOOT*, 2017. 1, 5, 20

[25] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing. Cryptology ePrint Archive, Report 2009/095, 2009. https://ia.cr/2009/095. 22

[26] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2*, 1(1):3–33, 2011. 4

[27] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering ..., 2004. 20

[28] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016. 10

[29] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization. In *CCS*, 2021. 1

[30] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020. 10

[31] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019. 5, 7, 8, 14

[32] 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 *Euro S&P*, 2019. 5, 6

[33] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *EuroS&P*, 2019. 10

[34] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the Kalyna key expansion. In *CT-RSA*, pages 272–296, 2022. 5

[35] Tobias Cloosters, Michael Rodler, and Lucas Davi. TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. USENIX Security, 2020. 22

[36] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. SGXFuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing. In *USENIX Security*, 2022. 6

[37] AACS-LA Consortium. Advanced Access Content System (AACS). https://aacsla.com/. 23

[38] AACS-LA Consortium. Advanced access content system (aacs): Introduction and common cryptographic elements. https://aacsla.com/wp-content/uploads/2019/02/AACS_Spec_Common_0.91.pdf, 2006. 14, 23

[39] AACS-LA Consortium. Advanced access content system ("aacs"): Adopter agreement. https://aacsla.com/wp-content/uploads/2021/05/AACS-Adopter-Agreement_20121115_review-only.pdf, 2009. 12, 13

[40] Victor Costan and Srinivas Devadas. Intel SGX explained. IACR Cryptology ePrint Archive 2016/086, 2016. 1

[41] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. SmashEx: Smashing SGX enclaves using exceptions. In *CCS*, 2021. 5, 6

[42] 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. *TCHES*, 2018. 1, 5, 20

[43] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. FastKitten: Practical smart contracts on bitcoin. In *USENIX Security*, 2019. 10

[44] Enarx. Enarx: WebAssembly + confidential computing. https://enarx.dev, 2022. 6

[45] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018. 5

[46] Fortanix Inc. Fortanix runtime encryption platform. https://www.fortanix.com/assets/Fortanix_RTE_Platform_Whitepaper.pdf, 2019. 6

[47] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Security*, 2017. 5

[48] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. URL https://eprint.iacr.org/2021/1063. https://eprint.iacr.org/2021/1063. 10

[49] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *EuroSec*, 2017. 1, 20

[50] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, 2018. 5

[51] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017. 5

[52] 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. *Communications of the ACM*, 52 (5):91–98, 2009. 11

[53] Mike Hamburg. Accelerating AES with vector permute instructions. In *CHES*, 2009. 6
[54] Gael Hofemeier and Robert Chesebrough. Introduction to Intel AES-NI and Intel secure key instructions. *Intel, White Paper*, 62, 2012. 6
[55] 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. 2020. 5
[56] *Intel Software Guard Extensions for Linux OS*. Intel, . https://github.com/01org/linux-sgx. 7
[57] Intel. Intel Software Guard Extensions (SGX) Protected Code Loader for Linux OS. https://github.com/intel/linux-sgx-pcl, . 12, 13
[58] *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Intel, . https://cdrdv2.intel.com/v1/dl/getContent/671200. 20
[59] Intel. Code sample: Intel software guard extensions remote attestation end-to-end example, . URL https://software.intel.com/content/www/us/en/develop/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.html. 13
[60] *Intel Software Guard Extensions SDK for Linux OS*. Intel, 2016. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf. 4
[61] Intel. Deep dive: Intel analysis of microarchitectural data sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling, May 2019. 14
[62] Intel. Deep dive: Intel transactional synchronization extensions (Intel TSX) asynchronous abort. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort, Nov 2019. 14
[63] Intel. 2019.2 IPU – Intel SGX with Intel processor graphics update advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00219.html, November 2019. 5, 7, 8, 14
[64] Intel. Deep dive: Load value injection. =https://software.intel.com/security-software-guidance/insights/deep-dive-load-value-injection, Mar 2020. 14
[65] Intel. L1D eviction sampling. https://software.intel.com/security-software-guidance/software-guidance/l1d-eviction-sampling, Jan 2020. 14
[66] Intel. Intel® software guard extensions (Intel® SGX) trusted computing base (TCB) recovery plans for stale data read from legacy xAPIC. http://archive.today/2022.11.28-035641/https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/q4-2022-intel-sgx-tcb-recovery-guidance.html, 2022. 3, 10, 14
[67] Intel. Intel® software guard extensions (Intel® SGX) trusted computing base (TCB) recovery plans for stale data read from legacy xAPIC. https://web.archive.org/web/20221020202933/https://www.intel.cn/content/www/cn/zh/developer/articles/technical/software-security-guidance/resources/intel-sgx-software-and-tcb-recovery-guidance.html, 2022. 3, 10
[68] Intel. Intel processors MMIO stale data advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00615.html, jun 2022. 5, 7, 8, 14
[69] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An open platform for SGX research. In *NDSS*, 2016. 20, 21
[70] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *SysTEX*, 2017. 1, 20
[71] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: EPID provisioning and attestation services. White paper, 2016. 4
[72] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016. 1
[73] Gabriel Kaptchuk, Ian Miers, and Matthew Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. *Cryptology ePrint Archive*, 2017. 10
[74] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 processor integrity from software. In *USENIX Security*, 2020. 5, 8
[75] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2019. 1
[76] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE SP*, 2016. 10
[77] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *EuroSys*, 2017. 6
[78] SCRT Labs. Tarantino nfts. https://tarantinonfts.com/, 2022. 12
[79] Jaehyuk Lee, Jinsoo 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*, 2017. 5, 6
[80] 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*, 2017. 5, 6

[81] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *CCS*, 2017. 10

[82] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CrossLine: Breaking "security-by-crash" based memory isolation in AMD SEV. In *CCS*, 2021. 1

[83] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB poisoning attacks on AMD secure encrypted virtualization. In *ACSAC*, 2021. 1

[84] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CipherLeaks: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security*, 2021. 1

[85] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018. 1

[86] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based power side-channel attacks on x86. In *IEEE SP*, 2021. 5, 8

[87] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In *CHES*, 2007. 6

[88] Plato Mavropoulos. MC Extractor. https://github.com/platomav/MCExtractor. 9

[89] 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 *HASP@ ISCA*, page 10, 2013. 3

[90] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017. 1, 5, 20

[91] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47 (4):538–570, 2019. 5

[92] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled instruction-level attacks on enclaves. In *USENIX Security*, 2020. 5

[93] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In *EuroSec*, 2018. 1

[94] 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 SP*, 2020. 5, 8, 14

[95] Dalit Naor, Moni Naor, and Jeff Lotspiech. Revocation and tracing schemes for stateless receivers. Cryptology ePrint Archive, Report 2001/059, 2001. https://ia.cr/2001/059. 24

[96] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone explained: Architectural features and use cases. In *CIC*, pages 445–451, 2016. 1

[97] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party machine learning on trusted processors. In *USENIX Security*, 2016. 6

[98] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006. 1

[99] Colin Percival. Cache missing for fun and profit, 2005. 1

[100] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A comprehensive survey. *ACM CSUR*, 51 (6):1–36, 2019. 1

[101] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *CCS*, 2019. 1

[102] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021. 5, 6

[103] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CROSSTALK: Speculative data leaks across cores are real. In *IEEE SP*, 2021. 5, 7, 8, 14

[104] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015. 6

[105] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone. In *CCS*, 2019. 1

[106] Michael Schwarz and Daniel Gruss. How trusted execution environments fuel research on microarchitectural attacks. 2020. 4

[107] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017. 5

[108] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019. 5, 7, 8, 14

[109] SCRT. Staking secrets: A live guide to staking and delegating scrt. https://scrt.network/blog/staking-secrets-guide-to-staking-delegating-scrt, 2020. 10

[110] SCRT. Secret network overview - private smart contracts on the blockchain. https://scrt.network/about/about-secret-network/, 2022. 2

[111] SCRT. Keplr dashboard. https://wallet.keplr.app/chains/secret-network, 2022. 2, 10

[112] 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 *NDSS*, 2017. 6

[113] Rohit Sinha, Sivanarayana Gaddam, and Ranjit Kumaresan. LucidiTEE: A TEE-blockchain system for policy-compliant multiparty computation with fairness. *Cryptology ePrint Archive*, 2019. 10

[114] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, 2019. 5

[115] Jiayuan Sui and Douglas R Stinson. A critical analysis and improvement of AACS drive-host authentication. In *ACISP*, pages 37–52, 2008. 23

[116] Shih-Wei Sun, Chun-Shien Lu, and Pao-Chi Chang. AACS-compatible multimedia joint encryption and fingerprinting: Security issues and some solutions. *Signal Processing: Image Communication*, 23(3): 179–193, 2008. 23

[117] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. Obscuro: A bitcoin mixer using trusted execution environments. In *ACSAC*, 2018. 10

[118] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017. 1, 5, 20

[119] 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*, 2017. 5

[120] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018. 1, 2, 3, 5, 7, 8, 11, 13, 14, 20, 22, 23

[121] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *CCS*, 2018. 5

[122] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, 2020. 5, 6, 14

[123] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Rogue in-flight data load. In *IEEE SP*, 2019. 1, 5, 7, 8, 14, 20

[124] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Addendum a to RIDL: Rogue in-flight data load. 2019. 7

[125] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf, 2020. 1, 3, 11, 20, 22, 23

[126] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Addendum 2 to RIDL: Rogue in-flight data load. 2020. 7

[127] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE SP*, 2021. 1, 3, 5, 7, 8, 11, 14, 20, 22, 23

[128] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W Fletcher. Game of threads: Enabling asynchronous poisoning attacks. In *ASPLOS*, 2020. 5, 6

[129] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with Rust-SGX. In *CCS*, 2019. 6

[130] Pei Wang, Yu Ding, Mingshen Sun, Huibo Wang, Tongxin Li, Rundong Zhou, Zhaofeng Chen, and Yiming Jing. Building and maintaining a third-party library supply chain for productive and secure SGX enclave development. In *ICSE*, 2020. 6

[131] 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 *CCS*, 2017. 1, 5, 20

[132] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS*, 2016. 5, 6

[133] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, 2018. 1, 20

[134] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *CCS*, 2019. 1

[135] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In *IEEE SP*, 2020. 1

[136] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, 2015. 5, 11

[137] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *Cryptology ePrint Archive*, 2014. 5

[138] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014. 1

[139] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. IACR Cryptology ePrint Archive

2016/980, 2016. 1

Appendix A. Emulated Guard eXtensions

Unfortunately, despite numerous SGX breaches [24, 42, 49, 70, 90, 118, 120, 123, 125, 127, 131, 133], there does not appear to be any tooling primarily focused on reverse-engineering enclaves or running unmodified, production-quality enclaves in a variety of different ecosystems, including ones without actual SGX hardware.

In this section we tackle this problem and support the reverse-engineering efforts in Section 5 and Section 6 by presenting Emulated Guard eXtensions (EGX), a framework that runs arbitrary SGX enclaves without actual SGX hardware (albeit without any of the typical hardware security properties). We note that building EGX involves unique challenges not considered by other emulators (e.g., OpenSGX [69]), including compatibility with existing (sometimes obfuscated) binaries and supporting attestation using extracted keys.

To emulate arbitrary enclaves, we must support the SGX instruction set, loading enclaves into memory, and application-enclave or enclave-CPU interactions. We use the publicly available Intel 64 and IA-32 Architectures Software Developer's Manual [58] to develop a framework featuring two different methods that achieve our goal: a full-system emulation mode virtualizing SGX hardware in QEMU [18], and an instrumentation mode modifying code at runtime using DynamoRIO [27]. This two-method approach offers the best of both worlds for SGX emulation, as QEMU targets more architectures, while DynamoRIO offers performance on x86.

We now discuss emulating the SGX instruction set, how both approaches address loading enclaves, and handling transitions from the application to and from the enclave.

A.1. The SGX ISA and Supporting Software

The SGX Instruction Set introduces the `enclv`, `encls` and the `enclu` instructions to interact with SGX from a virtual environment, the operating system, and the SGX application respectively. More specifically, the operating system uses `encls` to manage enclave while the application uses `enclu` to interact with enclaves. As the `rax` register determines which leaf function to call, our framework inspects `rax` and calls the corresponding helper function for the leaf as well.

Of particular interest are the following categories of instructions: enclave creation (`epa`, `ecreate`, `eadd`, `eextend`, `einit`), debugging (`edbgrd`, `edbgwr`), enclave control flow (`eenter`, `eexit`, `eresume`), cryptographic functionality (`ereport`, `egetkey`). EGX implements these instructions as a series of helper functions that emulate the behavior of the actual instructions. The helper functions are the same for both modes, but differ in how enclave memory is handled. While instrumentation mode can directly reference memory, as the enclave lives in the application's address space, full-system emulation handles memory accesses through a helper function, as the enclave lives in guest physical memory.

Intel also provides a number of software components that make use of the SGX instruction set: the SGX driver, to manage SGX enclaves; the SGX Platform SoftWare (PSW), to provide remote attestation services; the Trusted RunTime System (tRTS), to interact with the outside world; and the Untrusted RunTime System (uRTS), to interact with enclaves. The full-system emulation mode can simply run all of these software components, whereas the instrumentation mode replaces some functionality of the SGX uRTS, and only needs to handle enclave interactions via the `enclu` instruction.

A.2. Enclave Loading

While we can piggyback on the existing implementation of SGX uRTS in the full-system emulation mode, the instrumentation mode has to replace certain functions provided by uRTS responsible for enclave management and interaction. More specifically, it replaces the `sgx_create_enclave` and `sgx_destroy_enclave` functions with simulated versions that are responsible for loading and managing SGX enclaves, and cleaning up SGX enclaves respectively.

File Formats. Enclaves are distributed in the ELF (Linux) and PE (Microsoft Windows) format respectively, with a special metadata section (`.note.sgxmetadata` or `sgxmeta`) to provide information to the loader. Our implementation of `sgx_create_enclave` first parses the metadata to determine the enclave size to then allocate sufficient memory.

Program Segments. To ensure that debugging symbols are available to DynamoRIO, our loader first maps in the entire enclave file. Then, as is conventional, it iterates over the program header to map in the program segments. Finally, the loader parses the patch entries that are unique to Linux enclaves to patch the enclave memory with the correct data.

Memory Layout. Next the loader determines the exact memory layout of the SGX enclave required to set up per-thread data, stacks and the heap. On Linux, the enclave file contains layout entries that describe the memory layout needed, whereas on Windows the metadata describes the stack size, the heap size and the number of threads to support.

Memory Protection. Once the enclave memory has been set up, our loader applies the correct memory protections according to the program headers and the memory layout. To keep track of where the enclave resides in memory, as well as various bits of information such as the enclave and signer measurement, the loader maintains a mapping of enclave IDs, such that later calls to functions like `sgx_destroy_enclave` can operate in the context of the appropriate enclave by simply looking up the bookkeeping data for the given enclave ID.

Enclave Measurement. To verify that an enclave has not been tampered with prior to execution, `mrenclave` is computed over the entire contents as described in Section 2. Specifically, the `mrenclave` value is a SHA-256

sum computed with inputs from `ecreate`, `eadd`, and `eextend`. In full-system emulation mode, we implement these instructions and have full control over how the value is calculated, allowing us to load a modified enclave that still has its original `mrenclave` value. In instrumentation mode, EGX simply assumes that the enclave measurement provided by the enclave is correct and does not validate it, though it does compute the `mrsigner` field by calculating the SHA-256 hash of the modulus.

A.3. Enclave Interactions

After setup, we must support the various application-enclave and enclave-CPU interactions. More specifically: transitions between application and enclave (ECALLs/OCALLs), access to per-thread structures (segmentation), asynchronous exits interrupting the enclave execution (AEX), information regarding the SGX status of the system (`cpuid` and MSR).

ECALLs/OCALLs. The application can perform ECALLs to enclave functions by setting the appropriate registers and issuing `eenter`. Similarly, SGX enclaves can issue OCALLS to invoke untrusted code outside SGX. As the uRTS `sgx_ecall` function is used to perform ECALLs, the instrumentation mode replaces the `sgx_ecall` function while the full-system emulation mode handles them transparently by emulating `eenter`. To handle OCALLs, the instrumentation mode instruments `eexit` to execute the function specified by the `rdi` register, sets the `rdi` register to indicate completion and jumps back into the enclave. As before, the full-system emulation mode handles OCALLS transparently by emulating `eexit`.

Asynchronous EXits (AEX). SGX allows enclave execution to be interrupted in the form of Asynchronous EXit(s), and does so by preserving the enclave state in the EPC and `CREG`s and jumping to the Asynchronous Exit Point (AEP) provided when entering the enclave. Upon handling the interrupt, the asynchronous exit handler can then invoke `eresume` to resume execution of the enclave. The instrumentation mode simply relies on the OS's interrupt handler to suspend and resume enclave execution. To provide support for AEX, the full-system emulation mode saves state when an AEX occurs and implements the `eresume` leaf to resume execution.

Segmentation, `cpuid`, and MSRs. SGX enclaves may access per-thread data through instructions that reference the segment registers with a byte offset into the per-thread data. EGX calculates the memory address to access in instrumentation mode, whereas EGX sets the segment registers directly through the guest's MSRs in full-system emulation mode.

Additionally, EGX alters the behavior of `cpuid` and certain MSRs to report SGX's presence. For example, `cpuid` reports the emulated EPC size as well as available SGX features.

Debugging. While the actual `edbgrd` and `edbgwr` instructions will prevent us from debugging a non-debug enclave on SGX hardware, as these instructions check the enclave control structure to determine if the enclave is in debug or production mode, our software can simply ignore this check.

Figure 6: A successful run of an Intel-provided sample SGX enclave using our full-emulation mode on an ARM Mac.

A.4. Running Enclaves

EGX can successfully run a number of given debug and non-debug hardware enclaves. We tested the full-system emulation mode of EGX on two systems with Ubuntu 18.04 as guests: one with an Intel Core i9-9980HK and one with an Apple M1. Additionally, we tested the instrumentation mode on a system with an AMD Ryzen 5 PRO 5650G and a system with an Intel Xeon Platinum 8352Y, both running Ubuntu 20.04 LTS. To run SGX on any of these systems in either mode, we must first build the Linux SGX SDK and PSW package and run the installers, as one would do on any typical SGX-capable platform (in emulation mode this is performed on the guest). We then successfully used EGX to run a modified version of Intel's sample enclave that prints "Hello, world!" to the screen using an OCALL in each of the aforementioned configurations. We demonstrate a successful run of our full-system emulation mode on an Apple M1 processor in Figure 6.

Benchmarks. Being a framework aimed at reverse-engineering production enclaves, EGX is optimized for strong enclave compatibility with many hardware platforms, rather than performance. Nonetheless, to evaluate the performance of our implementation, we implemented a number of microbenchmarks measuring the cost of enclave creation (avg. 10 runs), ECALLs, OCALLs, performing 1,000,000 AES-256 encryptions and calculating the 10,000th Fibonnaci number using the ibig and num-bigint libraries (avg. over 100 runs). The results are shown in Table 3.

A.5. Related Work

Prior work has sought to emulate SGX before, with the most notable being that of OpenSGX [69]. OpenSGX is early work which predates Intel's SGX SDK and was intended to kick-start research as processors with SGX were just becoming commonplace. It was originally designed for user-mode emulation around their proprietary 'sgxlib' interface and was partially reverse-engineered from the SGX specification. Unfortunately, OpenSGX has a number of limitations which makes it incompatible with what we wish to achieve and use an emulation framework for in this work. In particular, as OpenSGX does not support the official SGX stack and is written around a custom interface, it cannot

Platform (Mode)	Creation	ECALL	OCALL	AES-256	Fib (ibig)	Fib (num-bigint)
Intel Xeon Platinum 8352Y (N)	69.053ms	0.006ms	0.012ms	0.010ms	0.017ms	0.018ms
Intel Xeon Platinum 8352Y (I)	18.390ms	0.570ms	0.590ms	0.591ms	0.532ms	0.424ms
AMD Ryzen 5 PRO 5650G (I)	9.404ms	0.093ms	0.100ms	0.363ms	0.109ms	0.105ms
Apple M1 (E)	10385.082ms	0.197ms	N/A	0.613ms	0.636ms	0.619ms
Intel Core i9-9980HK (N)	61.155ms	0.013ms	0.015ms	0.027ms	0.035ms	0.032ms
Intel Core i9-9980HK (I)	8.030ms	0.123ms	0.120ms	0.252ms	0.115ms	0.135ms
Intel Core i9-9980HK (E)	7407.324ms	0.453ms	N/A	0.669ms	0.663ms	0.713ms

TABLE 3: The results of running the microbenchmarks measuring the cost of enclave creation (10 runs), ECALLs, OCALLs, 1,000,000 AES-256 encryptions and calculating the 10,000th Fibonacci number (100 runs). Modes: **N**ative, **I**nstrumentation, **E**mulation.

emulate unmodified SGX-enabled executables, thus making it incompatible with modern enclave software. Additionally, OpenSGX did not consider making existing (commercial) software pass SGX attestation using authentic (leaked) attestation keys. This is challenging, as one would need to essentially re-implement the services provided by Intel's SGX platform software (PSW) framework and hook the software's attestation calls. While TeeRex [35] is not a true full emulator, it does build a framework (with partial emulation) that is designed for static vulnerability analysis of unencrypted enclaves, but not to run or debug them, or provide full-featured support, like is our goal.

Appendix B. SGX Revocation Analysis

To complete our exploration of the real world implications of SGX design decisions and their potential impact on security guarantees, we study how the key revocation mechanisms leave the SGX ecosystem open to a potentially unexpected form of attack and discuss how, for certain classes of attackers, this denial of service attack is feasible in practice.

Intel's Enhanced Privacy Identification [25] (EPID) scheme is used in the remote attestation process to allow for anonymous authentication. A set of devices is assigned to a group and each is issued a unique private key which verifies successfully against the group's public key without identifying the signer. However, sometimes it is necessary to revoke the ability of some signers to create valid signatures and this revocation process itself could potentially be abused to make the attestation process less usable.

SGX Revocation Process. Intel maintains three types of revocation lists: group-based (GroupRL), private key-based (PrivRL), and signature-based (SigRL). Of these, group revocation is the most extreme, as it represents a scenario where all device keys belonging to a specific group are revoked, as if a whole family of products is compromised.

When a device private key is leaked and discovered in its entirety, it is added to the PrivRL of its respective group and a verifier will need to sign all future messages under it in order to check for a collision with the signature it is verifying. Alternatively, if a signature is known to come from a compromised device, but that device's private key is unknown, a part of the revoked signature is added to the group's SigRL. To facilitate signature-based revocation, every EPID signature includes two group elements (B, K)

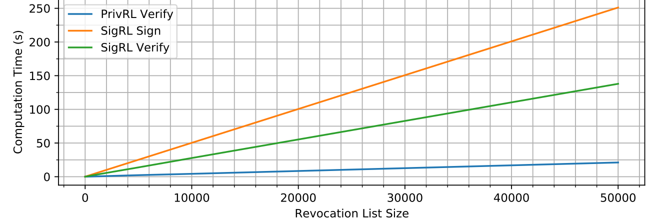

Figure 7: EPID Performance versus Revocation List Size

where B is randomly sampled from a cyclic group in which the Decisional Diffie-Hellman problem is hard and $K = B^f$, where f is a part of the prover's secret key. A zero-knowledge proof is used to ensure the correct value for f is used to calculate (B, K).

Proving Non-Revocation. We note the linear complexity for both PrivRL and SigRL revocation. While the PrivRL only affects the verifier's computation, the SigRL length affects both the prover and verifier computation: For every (B, K) in the SigRL, the prover needs to create a zero-knowledge proof of discrete logarithm inequality and the verifier, of course, needs to check these proofs.

A Potential Denial of Service Attack. SGX's linear time revocation scheme, combined with large scale key compromises via microarchitectural side channel attacks [120, 125, 127] leads us to investigate the possibility of using key revocations to mount a denial of service attacks on SGX's attestation process. To that aim, we wrote a tool to call Intel's EPID library functions (from the Linux 2.10 Open Source Gold Release) from outside of SGX. We generated our own custom PrivRLs and SigRLs which we used to instantiate the member and verifier contexts in the SDK. While varying the sizes of the revocation lists, we measured the performance of the `EpidSign` and `EpidVerify` functions at least ten times on an Intel NUC with an i7-10710U CPU running Ubuntu 18.04.

Attestation Runtimes. Figure 7 shows the running time of the EPID signature and verification algorithms, as a function of revocation list size. We remark that computational costs related to handling revocation lists dominate very quickly, showing a clear linear pattern: Each entry in the SigRL represents roughly a 5ms and 3ms increase in proving and verification time respectively, while each PrivRL entry adds .4ms to the verification time. We thus propose signature-based revocation as the better target for an attack on EPID performance.

The Cost of Delaying Attestation. Under the assumption that the utility of SGX will be considerably weakened if it takes an hour to perform a remote attestation, this translates to a requirement for roughly 720,000 entries in a SigRL corresponding to a given group ID. With microarchitectural attacks like Foreshadow, CacheOut and SGAxe [120, 125, 127] reliably extracting SGX keys, assuming a cost of $50 per CPU[5], this results in a $36 million budget for significantly delaying attestation for all CPUs belonging to the targeted group. Next, while $36 million is well within reach of wealthy individuals and large organizations, this cost can be further reduced if one considers a virus that extracts SGX keys, or a scheme which pays volunteers a small fee for running a key-extraction program on their hardware.

Mapping CPU Architectures to Groups. We note, however, that this is the cost of significantly slowing down attestation for all CPUs belonging to a specific, targeted, EPID group. While Intel does not publish how many groups are used in practice, at the time of writing there are only 589 valid group IDs on Intel's SGX development network.

Next, at time of writing, there are 470 unique Intel processors with SGX capabilities [1]. Given that group assignments change upon a microcode update, this suggests that at any given point there are at least a few different processor models per group in practice, leading to millions of individual machines belonging to each group. Thus, a denial of service attack on even a single group is likely to impact millions of SGX machines, considerably slowing down their attestation process. Finally, while Intel could fix slow attestation by changing the group ID of affected CPUs, this patch would need to propogate via BIOS updates, which takes months in practice per our survey in Section 4.

Mitigations? While such a DoS attack is not fundamental to all revocation mechanisms and schemes of this type, we observe that this specific revocation mechanism is inherent to SGX's current attestation scheme, and was designed to handle only a small number of compromised machines. While Intel could, for example, monitor the number of leaked keys to observe that such a DoS attack might be occurring, their only recourse is to use the revocation mechanism described above. To improve scaling, SGX would need a completely different mechanism, which is unlikely to be backward compatible.

Appendix C. The AACS2 Protocol

As part of our reverse-engineering efforts, we are able to provide the first public presentation and deployment analysis of the Advanced Access Content System (AACS) 2.0 and 2.1 protocols, which we describe here for the benefit of future researchers.

C.1. Overview

The Advanced Access Content System (AACS) is a standard for content distribution and digital rights management (DRM), maintained by a cross-industry consortium called The AACS Licensing Administrator (AACS-LA) [37]. While the first version of the protocol, AACS 1.0, has an official protocol and cryptographic specification that has been publicly released [38] and studied [115, 116], newer versions have been kept closed-source and proprietary. These most recent versions, AACS 2.0 and above, allegedly break significantly from prior versions, containing protocol updates and requiring the use of SGX for software players to play 4k UltraHD Blu-ray format discs, though to date there is no publicly published specification for these newer versions.

In AACS, one party (i.e., a Blu-ray manufacturer) encrypts the disc content and many users must be able to decrypt said contents. The users in this scheme are either dedicated hardware players or software Blu-ray players, such as PowerDVD. Additionally, some of these users might be considered "revoked" and thus content must be encrypted in such a way that all but these revoked users can access it.

We describe the high level AACS2[6] protocol here, and, to allow for a clear and concise overview, refer the interested reader to Appendix D for a detailed explanation of the primitives and components. AACS has three core requirements a player must satisfy to be able to decrypt a disk and play a movie: it must be (1) licensed by AACS-LA (checked through the Host Certificate), (2) not revoked (achieved through being able to derive the Media Key Block (MKB)), and (3) an original pressed disc (checked through the Volume ID (VID)). We describe how these requirements are achieved cryptographically in the following sections.

AACS Cryptographic Primitives. On any AACS-protected disc, the root cryptographic key used to encrypt the content is known as the Title Key and is generated at random by the licensed replicator. For practical purposes, rather than encrypting the full disc content (e.g. a 2-hour movie) with the Title Key directly, the media content is split into 6144-byte chunks called "Aligned Units", which are each encrypted with a per-chunk block key that is derived from the Title Key and a per-chunk random seed.

For most cryptographic operations, AACS uses AES in Cipher Block Chaining (CBC) mode with a 128-bit block size (denoted AES-128E(k, d) and AES-128D(k, d)) and defines a default CBC initialization vector[7]. AACS also defines AES-G, a cryptographic one-way function based on the AES cipher, and an extended version AES-G3, which repeats the AES-G operation three times on a 128-bit input to produce 384 bits of output. For processing data in calculations involving keys, AACS defines a cryptographic hashing function AES-H(x). For verifying the authenticity

5. There are several low-end SGX-enabled CPUs for under $50: https://ark.intel.com/content/www/us/en/ark/products/129487/intel-celeron-g4900-processor-2m-cache-3-10-ghz.html.

6. We will use this notation to cover both recent versions of the protocol (2.0 and 2.1), as all the information presented here (except where noted otherwise) applies to both protocols.

7. $0BA0F8DDFEA61FB3D8DF9F566A050F78_{16}$

and integrity of non-key content on a disc, AACS uses ECDSA-SHA1, upgraded to ECDSA-SHA256 for AACS2.

We refer the reader to Appendix D.1 for more details about each of these primitives and their exact constructions.

C.2. AACS Key Derivation

We now detail how AACS cryptographically realizes and enforces the aforementioned three core requirements.

Checking the Host Certificate. In order for an authorized software player such as PowerDVD (the "Host") to be able to decrypt and play an AACS-protected disc, it must complete a mutual authentication with an AACS compatible disc drive, followed by a series of key derivations to compute the Unit Key(s) required to decrypt the media content. The software player must contain an AACS Host Certificate, consisting of an ECC Host Public Key and a unique Host/Node ID, and the corresponding private key. For a Host Certificate to be valid, it must be digitally signed by AACS-LA and verifiable with the built-in AACS-LA public key. Similarly, an AACS-licensed disc drive will contain a Drive Certificate, also signed by the same AACS-LA root private key.

We provide a detailed description of the AACS2 Drive-Host mutual authentication scheme in Appendix D.2. At a high level it uses a similar procedure as AACS 1.0, except that the use of SHA1 and 160-bit ECC have been replaced by SHA256 and 256-bit ECC respectively. This procedure successfully ensures that both the host and the drive are licensed by AACS-LA, as their public keys are contained within digitally signed certificates. The corresponding private keys are required to be stored in protected areas of the host/drive software, which in the case of software players like PowerDVD utilizes SGX.

Obtaining the VID. If the mutual authentication is successful, the host will request that the drive return the Volume ID (VID) of the disc[8]. The VID will only be read by an AACS-licensed drive and returned to the host on a successful authentication. To ensure the integrity of the VID, the drive also protects it with `AES-CMAC`.

Obtaining the MKB. Once the host has obtained and verified the VID, it must then obtain a Media Key K_m by processing the Media Key Block (MKB). The MKB contains a Media Key encrypted with many processing keys to cover all non-revoked licensed players. The MKB also contains the necessary information for a licensed player to be able to derive a Processing Key K_p using the subset-difference algorithm and the player's set of built-in device keys (see below for more information on this derivation process).

Decrypting the disc. If the host can successfully arrive at a Processing Key, it knows it has not been revoked. It will then reference the Processing Key's index (or node) to read the corresponding encrypted Media Key data C from the Media Key Data Record in the MKB. The disc's Media Key can then be decrypted as: $K_m = \text{AES-128D}(K_p, C) \oplus uv$ where uv is the subset-difference node number of the Processing Key. The final key in the derivation sequence is the Volume Unique Key (VUK) K_{vu}. The VUK is used to encrypt the Title Key(s) and is generated from the Media Key K_m and the VID.

8. This is unique to each item of content, but not unique to each individual instance of a media.

At this stage, the player has now satisfied all three of the AACS core requirements, and it can calculate the VUK for a given disc as: $K_{vu} = \text{AES-G}(K_m, \text{VID})$. This allows it to decrypt and play the disc.

AACS 2.0 vs. AACS 2.1. The main difference between AACS 2.0 and 2.1 is the extension of the Media Key derivation process described above to include a Media Key Variant (MKV). Up to the calculation of the Processing Key K_p, both versions are identical (and follow closely with the known AACS 1.0 specification). Unlike for AACS 2.0, however, AACS 2.1 uses sequence keying with multiple Media Keys in the MKB that correspond to certain device nodes which can access them, as an improvement to the traitor-tracing scheme. We omit the full details here for brevity but document them completely in Appendix D.2.1, providing the first formal description of this new portion of the AACS 2.1 derivation.

Handling AACS Key Derivation and Revocation in Practice. Naively, the aforementioned key derivation and revocation checking process can be accomplished by assigning each user a unique key and then encrypting the title key for each non-revoked user. However, this scales linearly with the number of non-revoked users. This can be accomplished more efficiently using the Subset-Cover framework presented in NNL [95], which is used by AACS in practice. NNL utilizes a binary structure where each user (or device) is viewed as a leaf in this binary tree. The Blu-ray manufacturer finds a subset cover that encompasses all non-revoked users, encrypts the disk contents under a random key, then encrypts that key for all subsets in that subset cover. For a summary of how NNL works, we refer the interested reader to Appendix D.3.

Since Blu-ray discs are an offline physical medium, the keys associated with them cannot be updated. If the AACS-LA were to revoke a device or set of devices they could not do so retroactively, but only for future releases. To accomplish this, they would need to calculate a new subset-difference tree (i.e., MKB), which is associated with a new MKB version number. Hereafter, AACS-LA would need to use this new MKB for all future Blu-rays (until the need arises for another revocation).

C.3. Insights on AACS MKB Usage in Practice

As part of our exploration, we were able to delve more deeply into how AACS actually manages keys in practice. We omit the exact details of how PowerDVD specifically leverages NNL to perform MKB key derivation for brevity, and include only the insights that we deemed interesting here. However, for more information (and additional insights) we refer the interested reader to Appendix D.4.

We found that the PowerDVD software provisions 253 keys to an end user (effectively a device). This user corresponds to a leaf node in the NNL subset-difference tree, and the 253 keys correspond to subset-differences of nodes

Figure 8: A visualization of AACS 2.1 MKB Version 70 revocations. The small underlined portion near 0x20000000 indicates the region that corresponds to the PowerDVD keys we found.

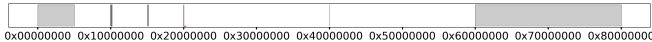

Figure 9: A visualization of AACS 2.1 MKB Version 70 revocations with periodic revocations removed.

between the user leaf and root, and siblings of said nodes. See Appendix D.5 for a concrete example of how this works. At first, this might seem to indicate that the height of the tree is only 23 (and thus the size of keyspace is 2^{22}), but an analysis of a series of MKBs (obtained through our reverse-engineering efforts) seems to indicate that the full tree is actually of height 32. We thus hypothesize that PowerDVD has only assigned a subset of keys corresponding to their allocation.

We also were able to gain further insight into the key revocation system by extracting the MKBs from various Blu-rays and building a visual representation of the revoked keyspace ranges, shown in Figure 8 for "Zombieland" with MKB version 70. With these visualizations, we were able to notice additional interesting characteristics related to the practical deployment of AACS.

All of the MKBs we plotted produce nearly identical plots: the first sixteenth of the keyspace is entirely revoked, the second sixteenth is entirely unrevoked, and the final fourth is entirely revoked. The rest of the keyspace alternates between large unrevoked regions and small, sporadic revoked regions. We hypothesize that the large revoked regions are not actually revoked, but rather not-not-revoked; i.e., some of this key space may be reserved for future usage or allocation.

Revocations from the middle portion of the MKB keyspace seemed to be somewhat periodic: for this region, the MKBs included a revocation every 2^{22} keys. Figure 9 illustrates the same keyspace as before, but with the periodic revocations removed. These are almost certainly not real revocations (i.e., they do not correspond to a real user), but instead seem to serve as a sort of pseudo-revocation that has a useful result of partitioning the keyspace into equal subregions. Practically speaking, these smaller subregions mean that any possible revocation can be expressed by $\{T_i \setminus T_j\}$ where no i has height greater than 23. As such, revocations in one subregion will have no effect on other subregions. Were it not for this partitioning, any revocation would require a full recomputation of a subset cover; instead, only a partial recomputation is needed. We note that this is not a requirement of the NNL algorithm but seems to be an interesting optimization for practical deployment. See Appendix D.5 for a concrete example of how this might work.

Existing AACS Revocations. To conclude our analysis of AACS, we analyzed a series of MKBs to try to determine how many actual device revocations have occurred in practice. We found that this appears to have happened a variety of times already: between MKB versions 61 and 70, 9 ranges are revoked, each comprised of either 4, 5, or 6 device keys, and totaling to 45 keys revoked; and between MKB version 70 and 72 a single range is revoked, spanning 1000 device keys. Of note, three of the ranges in the first set of revocations actually occurred within the known PowerDVD allocation. We found two releases with the same MKB version number, but different MKBs. Two Blu-ray discs of "Zombieland" both contain an MKB of the same type and with version number 70, yet one version contains an additional revocation. Further, there are no additional device revocations in the MKB between versions 72 and 76 (even though the MKB version has been updated). From this observation, we conclude that the MKB version number does not directly correlate with the number of subset-difference revocations.

Appendix D. AACS Extended Details

D.1. AACS Cryptographic Primitives

AES-G is built upon the AES-CBC decryption operation, where the 128-bit result from two 128-bit inputs x_1 and x_2 is computed as

$$\text{AES-G}(x_1, x_2) = \text{AES-128D}(x_1, x_2) \oplus x_2$$

In place of the x_2 input of AES-G, AES-G3 maintains an internal 128-bit "seed register" s, which is incremented by 1 for each of the three iterations. The seed register is initialized to s_0 defined as the constant:

$$7B103C5DCB08C4E51A27B01799053BD9_{16}$$

AES-H is based on the AES-G one-way function, and returns a 128-bit hash value from an arbitrary-length input. The input data to be hashed, x, is first padded to a multiple of 128 bits using the standard SHA-1 padding method, i.e., the input is padded to a multiple of 128 bits. The padded data x' is then split into 128-bit blocks $x'_1, x'_2, \ldots, x'_n$, which are used to calculate the 128-bit hash h_i as:

$$h_i = \text{AES-G}(x'_i, h_{i-1})$$

The initial 128-bit hash value h_0 is defined by AACS as:

$$2DC2DF39420321D0CEF1FE2374029D95_{16}$$

The result of the AES-H function is the final hash value h_n, thus AES-H$(x) = h_n$.

D.2. AACS Key Derivation

The AACS2 Drive-Host mutual authentication scheme is preformed in the following procedure:

1) The host randomly generates the 256-bit Host Nonce H_n
2) The host sends the nonce H_n and the Host Certificate H_{cert} to the drive
3) The drive verifies that the Host Certificate is of the correct AACS2 type, and supports bus encryption

4) The drive verifies the signature of the Host Certificate using the AACS-LA Public Key AACS_LA_{pub}
5) The drive checks the Host Revocation List to ensure the provided Host ID is not revoked
6) The drive randomly generates the 256-bit Drive Nonce D_n
7) The drive sends the nonce D_n and the Drive Certificate D_{cert} to the host
8) The host performs identical type, signature, and revocation list checks on the Drive Certificate
9) The host sends a request for a point on the elliptic curve D_v and its associated signature
10) The drive randomly generates a 256-bit ephemeral private key D_k
11) The drive computes the point D_v as

$$D_v = D_k G$$

where G is the base point of the elliptic curve
12) The drive creates a digital signature of the concatenation of the Host Nonce and the point D_v as

$$D_{sig} = \text{AACS_Sign}(D_{priv}, H_n || D_v)$$

13) The drive sends the point D_v and associated signature D_{sig} to the host
14) The host verifies the signature of $H_n || D_v$ with the Drive Public Key D_{pub} contained in the Drive Certificate

$$\text{AACS_Verify}(D_{pub}, D_{sig}, H_n || D_v)$$

15) The host randomly generates a 256-bit ephemeral private key H_k
16) The host computes the point H_v as

$$H_v = H_k G$$

where G is the base point of the elliptic curve
17) The host creates a digital signature of the concatenation of the Drive Nonce and the point H_v as

$$H_{sig} = \text{AACS_Sign}(H_{priv}, D_n || H_v)$$

18) The host sends the point H_v and associated signature H_{sig} to the drive
19) The drive verifies the signature of $D_n || H_v$ with the Host Public Key H_{pub} contained in the Host Certificate

$$\text{AACS_Verify}(H_{pub}, H_{sig}, D_n || H_v)$$

20) The drive calculates the Bus Key BK from the point on the elliptic curve

$$BK = \text{x-coordinate}(D_k H_v)_{lsb_128}$$

where BK is the least significant 128-bits of the x-coordinate
21) The host calculates the Bus Key BK from the point on the elliptic curve

$$BK = \text{x-coordinate}(H_k D_v)_{lsb_128}$$

where BK is the least significant 128-bits of the x-coordinate

D.2.1. AACS 2.1 Media Key Variants

We describe the full Media Key derivation process for AACS 2.1 here, which includes the addition of the Media Key Variants.

We start by looking up C from the Encrypted Media Key Variant Data record entry in the MKB. Instead of K_m, we can only get the Media Key Precursor K_{mp} as: $K_{mp} = \text{AES-128D}(K_p, C) \oplus uv$ where uv is the subset-difference node number of the Processing Key. The Media Key Precursor is then combined with a per-manufacturer key (called the Key Correction Data (or KCD)) to aid in traitor tracing. The Media Key Precursor and KCD key is combined to create the new processing key as: $K_{pnew} = K_{mp} \oplus \text{KCD}$.

The next step in the AACS2.1 process is to compute the variant number K_{vn} for the Processing Key node (and hence device group) which we have arrived at as: $K_{vn} = \text{AES-G}(K_p, \text{Nonce}) \& \text{0xFFFF}$ where Nonce is loaded from the MKB Variant Number record. Additionally, AACS 2.1 uses 16-bit variant numbers, where AACS 1.X uses 10-bit variant numbers. Next, the correct Variant Key Data (VKD) entry has to be selected from the Variant Key Data record in the MKB. The index of the VKD is computed as: $VKD_{idx} = K_{vn} \oplus \text{VARIANTS[uv]}$ where VARIANTS[uv] is the entry in the variant number data record corresponding to the index of the P_k uv node in the MKB Explicit Subset-Difference record. The appropriate VKD for our device node can then be looked up in the correct record, using this index (one of 0xFFFF possible entries).

Finally, with both K_{pnew} and VKD corresponding to our device node and player KCD, we can compute the final disc Media Key K_m as: $K_m = \text{AES-128D}(K_{pnew}, \text{VKD}) \oplus uv$.

In the case of CyberLink's implementation of AACS, we make a few interesting observations. PowerDVD checks the 2nd LSB (& 0x4) of K_{mp} to determine whether to use the hardcoded CyberLink KCD key (found through our reversing efforts in `CLTA_SW.dll` and `CLTE.dll`) if the bit is zero, or some other key if the bit is one. This other key is potentially a network-updateable SoftKCD value, as code and URLs were found to support this in `CLTE.dll` and `CLTA_SW.dll`, but currently no products seem to utilize this functionality and the system might not be fully implemented yet as a result. In our testing, all AACS 2.1 MKBs, when processed using PowerDVD device keys, used the hardcoded KCD value.

D.3. Overview of NNL

Consider a full binary tree of size $N = 2^h$. Each user is viewed as a leaf in this binary tree. For a node i, denote T_i as the subtree rooted at i. Each node i is assigned a label $LABEL_i$. Denote $S_{i,j}$ as the subset difference of $T_i \setminus T_j$, in other words, all nodes with i as an ancestor, but not j. Denote G as a cryptographic pseudorandom sequence generator that provides an output 3 times the length of the input. Denote $G_L(K), G_M(K)$, and $G_R(K)$ as the left, middle, and right thirds of the output of $G(K)$.

Now consider a subtree T_i. For a node in this subtree with some label K, its left and right children will be given

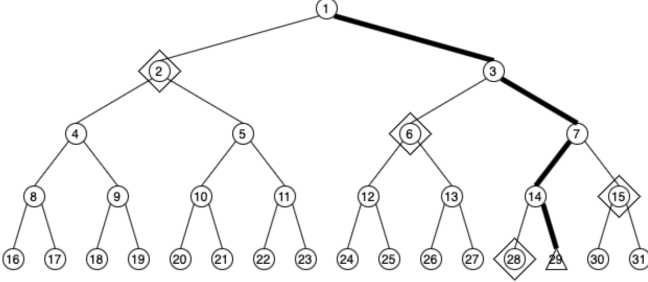

Figure 10: Subset-Difference Tree example

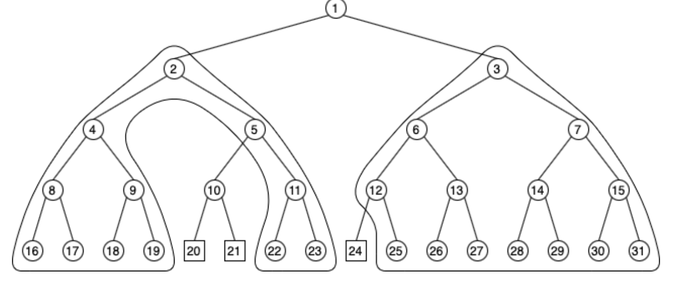

Figure 11: Subset-Difference Tree Revocation Example Starting State

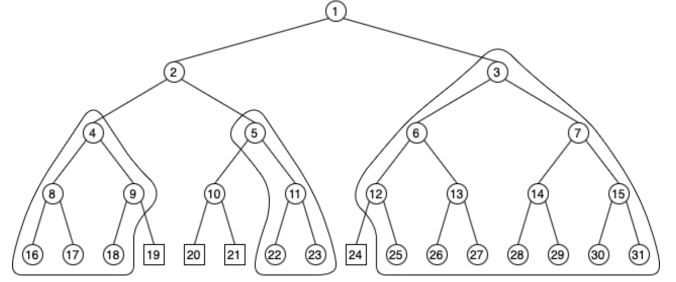

Figure 12: Subset-Difference Tree Revocation Example After Revoking 19

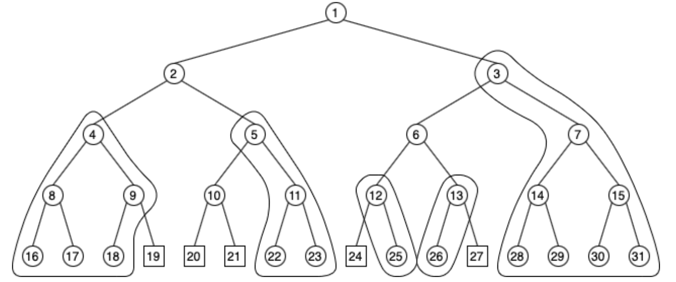

Figure 13: Subset-Difference Tree Revocation Example After Revoking 27

labels $G_L(K)$ and $G_R(K)$ respectively. Let $LABEL_{i,j}$ denote the label of node j derived from $LABEL_i$ in subtree T_i. For example, in subtree T_i the children of i will be $2i$ and $2i+1$, and will be given labels (in that subtree) $LABEL_{i,2i}$ and $LABEL_{i,2i+1}$. It's important to note that each node i will have a label $LABEL_i$, and labels $LABEL_{k,i}$ for each ancestor k. Finally, let $L_{i,j}$ denote $G_M(LABEL_{i,j})$, this will be the key assigned to subset $S_{i,j}$. Note that given $LABEL_i$, $L_{i,j}$ can be computed with at most $\log N$ invocations of G.

A subset cover $S = \{S_{i_1,j_1}, \ldots, S_{i_k,j_k}\}$ is calculated such that it contains all non-revoked users. The $L_{i_1,j_1}, \ldots, L_{i_k,j_k}$ corresponding to the subsets in the cover are used for encryption. For each subtree T_i of which leaf u is a descendant, u will store the labels of all the siblings of nodes on the path from u to i. For example in Figure 10, $u = 29$ would store $\{T_1 \setminus T_2, T_1 \setminus T_6, T_3 \setminus T_6, T_1 \setminus T_{15}, T_3 \setminus T_{15}, T_7 \setminus T_{15}, T_1 \setminus T_{28}, T_3 \setminus T_{28}, T_7 \setminus T_{28}, T_{14} \setminus T_{28}\}$.

Per NNL, this means a user u will store $1+\sum_{k=1}^{\log N+1} k-1$ labels. Each tree T_i containing u will contribute $k-1$ keys, plus 1 for the case with no revocations. (Note that in practice, the extra key for the case with no revocations is omitted, see Appendix D.5 for details.) If a user is not revoked, then it will be in at least one subset of the subset cover, and will be able to access the encrypted media.

Figures 11, 12, and 13 show an example NNL tree, the same tree after revoking 19, and the tree after revoking 27. Each encircled region represents a subset difference, and would correspond to a key. For example, in Figure 13 the region containing 13 and 26 corresponds to $T_13 \setminus T_27$ and $LABEL_{13,27}$.

D.4. MKB Key Derivation in PowerDVD

To compute G_L, G_M, and G_R PowerDVD uses the AES-128 decryption function and fixed constant: s_0 = $7B103C5DCB08C4E51A27B01799053BD9_{16}$.

1) $G_L(K) = \text{AES-128D}(K, s_0) \oplus s_0$
2) $G_M(K) = \text{AES-128D}(K, s_0+1) \oplus (s_0+1)$
3) $G_R(K) = \text{AES-128D}(K, s_0+2) \oplus (s_0+2)$

Rather than storing each of the 253 keys with corresponding i, j values, the keys are stored with 3 corresponding values: a path, and two masks, a u mask and a v mask.

Since j is a descendant of i, a path starting at the root of the tree and between both can be expressed using a sequence of 0s and 1s, where a 0 denotes going left and a 1 denotes going right while traversing the tree downwards. The u mask specifies what position along the path corresponds to the T_i tree and the v mask what position corresponds to the T_j tree.

The AACS specification uses the term "device key" to mean $LABEL_{i,j}$ in NNL, and "processing key" to mean $L_{i,j}$. AACS additionally uses "media keys" and a "title key." The Title Key is used for the final media decryption, and the Media Key is used to encrypt the Title Key. The Media Key is obtained by processing the MKB. The MKB is a set of keys encrypted under each $L_{i,j}$ corresponding to entries in the subset cover of non-revoked users.

D.5. Additional Insights From Real World MKB Usage

PowerDVD and NNL. We provide a concrete example of PowerDVD's usage of NNL to help better explain the process. In Figure 10, the user corresponds to node 29 and

the sibling nodes are all demarcated by squares. For this toy example, the keys provided to the user would correspond to $\{T_1 \setminus T_2, T_1 \setminus T_6, T_3 \setminus T_6, T_1 \setminus T_{15}, T_3 \setminus T_{15}, T_7 \setminus T_{15}, T_1 \setminus T_{28}, T_3 \setminus T_{28}, T_7 \setminus T_{28}, T_{14} \setminus T_{28}\}$.

Partitioning the MKB Keyspace. We observe that the MKB keyspace seems to have a fixed set of periodic revocations, essentially partitioning the larger keyspace into smaller subregions. This optimization allows for only partial recomputation upon revocation in one region. Figure 14 demonstrates what this would look like in a toy example; each partitioned subregion can be treated as its own tree. Note that this technique means there would never be a case with no revocations.

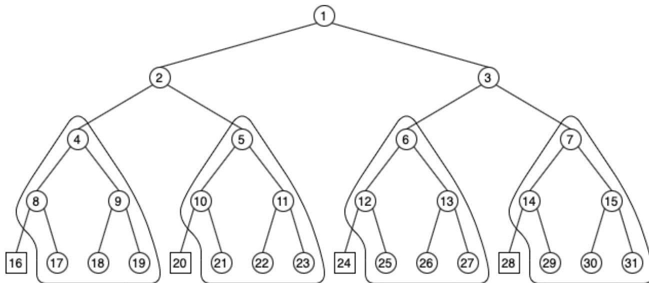

Figure 14: Subset-Difference Partitioning Example

Appendix E. PowerDVD Extended Details

E.1. `CLKDE.dll`

This is the detailed list of the ECALLs and functionality provided by the `CLKDE.dll` enclave.

1) Wrapper for `sgx_calc_sealed_data_size`
2) Wrapper for `sgx_ra_get_ga`
3) Wrapper for `sgx_ra_proc_msg2_trusted`
4) Wrapper for `sgx_ra_get_msg3_trusted`
5) Call `sgx_create_pse_session` and initialized Remote Attestation session context
6) Decrypts data using AES-GCM with a hardcoded key/IV pair
7) Decrypts blob using Remote Attestation session key, and seals data to disk
8) Cleanup session
9) Unused/return hardcoded error code

This enclave maintains a PSE session and implements the secure portion of the SGX remote attestation process. This is the mechanism that allows AACS2 keys to be securely provisioned to software players, as the authenticity of the enclave is verified by Intel prior to keys being released. A variation of the Elliptic-Curve Diffie-Hellman handshake is utilized to establish an ephemeral session key between the CyberLink server and CyberLink CLKDE enclave that is used to encrypt the key material in transit. Interestingly, we observed that CyberLink uses an additional layer of AES-GCM encryption for each key (Step 6 above), although the security benefit is questionable at best as the key and IV are hardcoded.