To appear in ACM Computing Surveys (CSUR), accepted May 2024, author's version

A Survey of Hardware Improvements to Secure Program Execution

LIANYING ZHAO, Carleton University, Canada
HE SHUANG, SHENGJIE XU, WEI HUANG, RONGZHEN CUI, PUSHKAR BETTADPUR, and DAVID LIE, University of Toronto, Canada

Hardware has been constantly augmented for security considerations since the advent of computers. There is also a common perception among computer users that hardware does a relatively better job on security assurance compared to software. Yet, the community has long lacked a comprehensive study to answer questions such as how hardware security support contributes to security, what kind of improvements have been introduced to improve such support and what its advantages/disadvantages are.

By generalizing various security goals, we taxonomize hardware security features and their security properties that can aid in securing program execution, considered as three aspects, i.e., state correctness, runtime protection and input/output protection. Based on this taxonomy, the survey systematically examines 1) the roles: how hardware is applied to achieve security; and 2) the problems: how reported attacks have exploited certain defects in hardware. We see that hardware's unique advantages and problems co-exist and it highly depends on the desired security purpose as to which type to use. Among the survey findings are also that code as part of hardware (aka. firmware) should be treated differently to ensure security by design; and how research proposals have driven the advancement of commodity hardware features.

CCS Concepts: • **Security and privacy → Security in hardware**; **Software and application security**; **Systems security**.

Additional Key Words and Phrases: Hardware Security Support, Trusted Execution Environments

1 INTRODUCTION

For decades, software has been faced with advanced attacks that challenge its security, as exemplified by code reuse attacks [20, 135], side channels [108, 152], firmware/rootkit-level attacks [53, 70, 71, 76, 137] and physical memory attacks [68, 120, 178]. Traditionally, the protection of software against such attacks has been limited to software mechanisms. For example, software may deploy bounds checking [121, 183], stack cookies [41] or control-flow integrity [1, 28] enforcement. However, such defenses have been limited in deployment, partially due to their performance impact, e.g., while the overhead is reduced from SoftBound [121]'s 71% to WPBOUND [183]'s 45%, it is still significant.[1] More recently, security software tends to resort to hardware mechanisms to counter such advanced attacks, as reflected in the paradigm of trusted computing [100].

In general, implementations of a security mechanism can involve software, hardware, or both. Nonetheless, there exists a common belief in hardware's advantages [146] in security over software considering the following aspects: (1) Immutability. Hardware being implemented in physical silicon cannot be easily modified by a remote attacker. This makes hardware support useful under strong adversarial models, e.g., rootkit- [101] and hypervisor- [92] level threats; (2) Efficiency. Without the need to load and decode software instructions on general-purpose functional units, hardware can be specialized to a task thus offering better performance and energy efficiency; (3) Finally, hardware being the boundary/interface between the physical world and software makes it a natural

[1] In comparison, their hardware-based counterparts can achieve much lower performance overhead, e.g., HardBound [45] 5% to 9% and HardScope [125] 3.2%.

Authors' addresses: Lianying Zhao, lianying.zhao@carleton.ca, Carleton University, 1125 Colonel By Drive, Ottawa, Canada; He Shuang, he.shuang@mail.utoronto.ca; Shengjie Xu, shengjie.xu@mail.utoronto.ca; Wei Huang, wh.huang@mail.utoronto.ca; Rongzhen Cui, gavin.cui@mail.utoronto.ca; Pushkar Bettadpur, pushkar.bettadpur@mail.utoronto.ca; David Lie, david.lie@utoronto.ca, University of Toronto, 10 King's College Road, Toronto, Canada.

trust anchor, especially when it comes to securing user-machine interactions (see Section 5.2). For example, users know without ambiguity which piece of hardware they are interacting with, such as pressing a button or checking an LED security indicator.

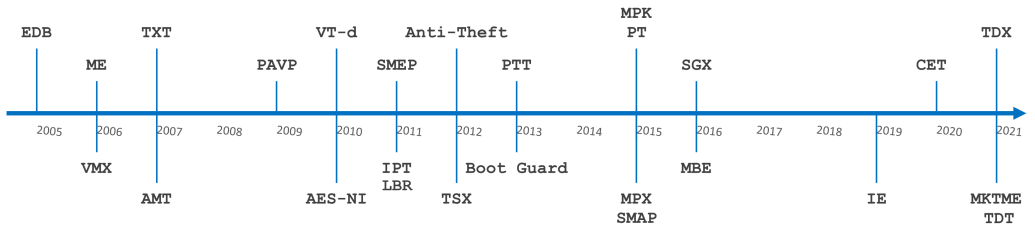

Fig. 1. Intel's hardware/firmware-based features and their years of introduction. Included are features: dedicated to security, as a foundation of security, or relied on by proposed security solutions. Note that this figure depicts the timeline but not necessarily the current availability of these features on all CPU/chipset models.

Over time, computer manufacturers have added various features in consideration of security. Figure 1 lists certain security-related hardware features introduced by Intel since 2005 (part of which will be discussed in this article). This phenomenon is analogous to adding more and more features to software, as pointed out by Baumann [16], which may make hardware share the behavior or even problems of software, e.g., memory corruption vulnerabilities like buffer overflows in hardware/firmware (e.g., CVE-2017-5705, CVE- 2017-5706). Therefore, we have a reason to wonder what roles the introduced hardware features play in improving security and how, through a systematic survey. This topic has been partially covered in the literature, yet none specifically achieves the exact purpose. For instance, Baumann [16]'s work was to briefly discuss the various implications of hardware being implemented more like software; Zhang and Zhang [189] focused on one special type of hardware features that support isolated execution; Dangwal et al. [42] chose an angle of software-hardware-security codesign to examine how feedback flows between the three aspects of software, hardware and security; Maene et al. [109] conducted a survey on hardware features for trusted computing (mainly isolation and attestation).

In this survey, we aim to study the hardware improvements to computing platforms that directly and indirectly benefit security (refer to Section 2 for detailed scoping), with a focus on *secure program execution*, and create a taxonomy according to the security properties they possess. We deem the secure execution of computer programs to be the foundation for individual security purposes, e.g., authentication or data encryption, which will all eventually rely on secure execution. We intend to study and answer the following research questions in response to the phenomenon reflected in Figure 1 through this survey:

- **RQ1:** How does hardware support contribute to achieving secure program execution, in particular, with what kind of security properties?
- **RQ2:** How do various security mechanisms/tools take advantage of hardware security support?
- **RQ3:** What problems have been identified with hardware security support in terms of attacks and other factors such as usability?

To answer RQ1, we propose to model where a program runs as an execution environment (EE), and break down the goal of secure execution into three aspects: state correctness, runtime protection and input/output protection (see Section 3.1). We then categorize hardware features into execution modes, extensions to the modes and co-processors (see Section 3.2), which collectively protect the EE, some taking care of state correctness with isolation from outside threats and some hardening the execution against common attack vectors internally, as shown in Table 1 for selected

commercially off-the-shelf (COTS) and academically proposed hardware features. This naturally forms a two-dimensional taxonomy: types of hardware support and types of security properties. As for the roles hardware plays in improving security (RQ2), we examine typical use cases as seen in existing academic/commercial security applications and make the connection with the advantages of hardware (see Section 5). Finally, despite the (relative) advantages over software-based mechanisms, hardware also has its attack vectors (RQ3). Therefore, we also review where hardware security support still needs improvements with regard to reported attacks, usability and adoptability. In particular, we attempt to examine the attacks systematically and discuss likely causes from multiple perspectives (see Section 6.1).

Contributions. We summarize our contributions in this survey as follows:

- To better understand hardware's role in securing program execution, we use a taxonomy to examine hardware security features based on their security properties with case studies. In particular, we discuss how execution modes, their extensions, and co-processors complement each other to ensure state correctness, runtime and I/O protection of program execution.
- We survey the state-of-the-art application of hardware security support, and see how the security properties enable hardware security features to play their role in various use cases of both research proposals and commercial solutions.
- We also discuss what can go wrong with hardware security support despite its advantages. The main focus is on attacks that have been identified and reported in the community with our analysis of likely causes.

2 SURVEY SCOPE

Hardware in this survey. In its narrow sense, the term "hardware" only refers to the immutable hardware, i.e., the physical silicon, as opposed to code. Nonetheless, it is worth noting that a common perception of hardware also implicitly includes firmware that runs on the hardware. The term *firmware* [156] denotes a type of software that resides in special storage (such as read-only memory and hidden sectors of a drive) and is not "soft" enough to be updated [150]. Firmware implements functionality that is logically part of the hardware providing abstractions (e.g., the instruction set architecture—ISA). Thus, although firmware is also code, like software, it is often treated as part of hardware. In this survey, where it concerns hardware security support, we also consider firmware as part of the broad-sense hardware.

Academically proposed features. As seen from the evolution paths of hardware security support, they often start from pure software designs, followed by hardware changes proposed by academia, before eventually being adopted by manufacturers as hardware features on commodity systems. For example, the timeline of buffer-overflow (related) attack defenses is: early compiler-assisted software-based checks represented by RAD [36] (2001) and SoftBound [121] (2009) were followed by later proposals gradually introducing hardware changes evaluated using simulation/emulation, such as SDMP [46] (2015) and Low-fat pointers [98] (2013); then, real hardware extensions were made available by manufacturers like Intel Memory Protection Extensions (MPX) [127] (circa 2015) and Intel Control-flow Enforcement Technology (CET) [82] (circa 2020). Another example is isolated execution environments. Research proposals like Iso-X [56], AEGIS [153], and XOM [105] paved the way for the advent of Intel SGX [39].

Based on this observation, the survey will cover both COTS features (to examine what users are exposed to — the status-quo) and proposed hardware changes in research (to see where such features originated), with the exception of Section 6.1 which exclusively focuses on documented attacks, not applicable to research proposals. The taxonomy (as shown in Table 1) will have a

slightly more focus on COTS features as we would like to examine how "hardware improvements" have been gradually added along the evolution that are available as building blocks to secure program execution. Research proposals are discussed in Sections 4.4 and 5.2.

Secure execution. We only survey hardware security support from the perspective of protecting the execution of computer programs (software serving a specific purpose). This is justified by the fact that other perspectives are built on top of or just rely on secure program execution. For example, secure communication needs secure execution of the protocol stack on individual hosts; authentication relies on the integrity of the checking logic (in addition to protecting the secrets). With regard to what "secure" execution entails (e.g., the security properties), refer to Section 3.1.

Execution environment (EE): we refer to where the computer program runs as its execution environment. An EE is determined by the combination of all hardware/software abstractions that underlies and supports the program. For a user-level process, its EE includes virtual memory (the address mapping and isolation from outside) supported by the hardware, e.g., the memory management unit (MMU), and software, including the OS kernel. In comparison, certain hypervisor code (underlying the OS) runs in an EE contributed by almost hardware alone, i.e., "bare-metal".

Varying threat models. It is important to note that various hardware security features assume different threats, not to mention their applications can have further different security assumptions. Therefore, we may not have a unified threat model throughout the survey. To facilitate the discussion, we consider several typical (but non-exhaustive) adversary types, which will be used in subsequent discussions, physical (ADV_PHY), privileged compared to victim code (ADV_PRIV, i.e., kernel-level or VMM-level) and unprivileged compared to victim code (ADV_UNP). Generally speaking, when the adversary is more privileged than the victim (ADV_PRIV), e.g., kernel-level for a regular user process, isolation or memory protection may become ineffective, and the victim code/data could thus be directly accessed by the adversary. Otherwise, in the case of ADV_UNP, the adversary may resort to exploiting vulnerabilities within the victim EE with crafted input without breaking isolation (see memory/type safety assurance in Section 5.2 and side channels in Section 6.1.2). In line with the surveyed works, by ADV_PHY, we refer to only simple lab efforts by a human attacker, such as adding/removing components, wiretapping, changing jumpers, excluding sophisticated capabilities like chip decapping/imaging, etc., or physical malicious devices introduced by the attacker, e.g., a tampered USB dongle. Note that availability is sometimes a non-goal especially when ADV_PHY is in-scope, i.e., excluding the DoS (denial-of-service) attacks.

3 HARDWARE SUPPORT FOR SECURE EXECUTION

This section starts with what secure program execution entails (i.e., the expected security properties), and then presents a taxonomy of hardware security support of three types, followed by a few examples of each type.

3.1 Aspects of Secure Execution

We consider the execution of a program to be secure when necessary security properties (e.g., integrity, confidentiality, freshness) corresponding to the execution are ensured. At a high level, these security properties can be classified into state correctness, runtime protection, and input/output protection. Each program has critical data that determines its execution state to be stored persistently/temporarily when it is not in execution, e.g., interrupted, exited, not started, or failed. Such state data includes but is not limited to initial program input/parameters, execution results, and internal data structures/metadata. State correctness needs to be ensured when the program starts/resumes execution. Runtime protection ensures the legitimacy of various memory accesses

when the program is in execution. Moreover, there is a third aspect: the program needs to communicate with outside of its EE at runtime. The communication could be with software outside the EE, peripheral devices (e.g., a keyboard), or a remote party, which will all fall in (runtime) input/output protection. Note that although all three aspects are required to achieve secure execution, specific hardware security support may only offer part of them (the design goal).

3.1.1 State Correctness. The program's state needs to be protected in two aspects: 1) *initial state* and 2) *state continuity*. If the initial state of a program is wrong, no subsequent correct execution can be expected. The program needs to start from a verifiably correct state according to certain specifications/policies, and its code being loaded for execution is integrity-protected. This can be done either statically through cryptographic checking once when deploying the code (denoted as Update in Table 1), or in certain cases every time the system is booted (denoted as Boot), or dynamically by re-verifying it at each launch-time (denoted as Launch). State continuity refers to the correct state when the program returns from exits/interrupts/failures. Although the situation is similar to the initial state, the resumed state carries information about previous execution with an additional requirement for preventing rollback attacks, where a stale/non-fresh state is provided to serve the attacker's purpose (e.g., so as to allow unlimited password guesses even with rate-limiting in place). There have been research proposals addressing this [110, 132, 151]. The typical adversary in such attacks is ADV_UNP (due to inability to compromise runtime state) as ADV_PRIV/ADV_PHY has other more powerful means to tamper with the EE.

3.1.2 Runtime Protection. Once started, the program may be attacked directly by code outside the EE or indirectly due to bugs inside the EE being exploited. We generalize such attacks to be violating the following types of memory accesses:

- Read/Write (R/W). This refers to data access control. Code outside the EE should not be able to read/write what is inside unless intended, as is often achieved by isolation. Meanwhile, code inside the EE should not have arbitrary read/write accesses spatially or temporarily (triggered by the attacker's input to serve a malicious purpose), which can be prevented by various fine-grained restrictions (see Extensions in Section 3.3 for details).
- Execute (X). To execute from a memory location is equivalent to loading the corresponding address into the program counter (e.g., the `IP` register of x86), either explicitly (e.g., a jump) or implicitly (e.g., a return). We generalize it to be the execute access to a memory address.

Such accesses can be further generalized to cover CPU/chipset registers (except for X as it is not applicable) and the R/W protection is often reflected in privileges, e.g., on x86, user-space code cannot modify the control register `CR3` which points to the page tables for virtual addressing.

3.1.3 Input/output Protection. A program needs to have output and, optionally, input to be useful to the user. If the input provided at run-time or the output generated by the execution were manipulated, input/output security would be undermined. We only consider the "path" between the party providing/receiving the input/output and the EE (assuming both to be trusted). The path usually involves multiple hops before reaching the EE in question, e.g., another device, physical peripheral, drivers in the OS, middleware, hence possibly faced with one or multiple of ADV_PHY, ADV_PRIV and even ADV_UNP. Therefore, the protection we discuss here applies between a specific hop and the EE. To protect runtime input/output, there could be two types of defenses:

- Software (SW)-Verified. By using cryptography or a tamper-evident equivalent, code (software) in/outside the EE can verify the integrity of the received data without trusting the communication path.

- Hardware (HW)-Enforceable. When the EE has control over the communication path (e.g., with a higher privilege), a secure channel can be established by hardware, explicitly protecting the exchanged data.

Figure 2 maps the three aspects of secure execution to an abstracted life cycle of a program.

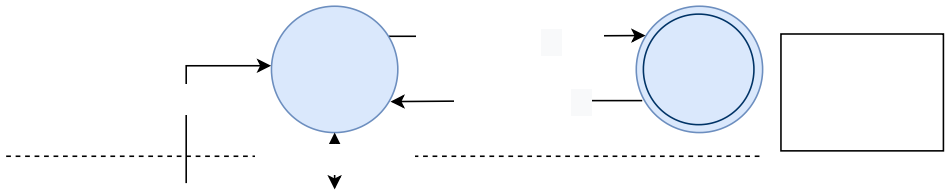

Fig. 2. An example Execution Environment (EE) states and the security properties involved in state transitions.

3.2 Taxonomy

Current hardware security support can be taxonomized based on their similarities in being able to achieve secure execution. We show in Table 1 that hardware security support can be categorized into three types: execution modes, extensions to execution modes and co-processors.

Type I: Execution Modes. An execution mode refers to the combination of processor hardware settings that influence program execution, often *entered/exited* with a register/instruction/event. In different modes, the processor behaves differently with different architectural features (e.g., new instructions). New execution modes introduced on a general-purpose processor (which is by default shipped with fundamental modes like x86's real/protected modes, and the latest 64-bit (sub) mode [81]) can directly provide security-enhancing capabilities to the execution environment where tasks run. In consideration of potential co-located ADV_UNP, we also distinguish whether an execution mode is based on the physical address space, hence exclusive, or virtual address spaces (e.g., user-level or VM-level), hence concurrent, supporting multiple instances (denoted with a † in Table 1)

Type II: Extensions. Hardware security features added to a processor may not create new execution modes, but instead augment existing execution modes, e.g., by placing more restrictions on what is allowed. Such hardware support can be considered an extension. Different from execution modes, an extension is not marked by any entry/exit, but more flexibly accessible, e.g., locally specific to memory references, objects, pages, code segments, etc.

Type III: Co-processors. Certain security support can also go off the main processor and exist in the form of a dedicated processor, commonly referred to as a co-processor (as opposed to the main processor). Note that, if the purpose is to support secure code execution on the main processor, the co-processor needs to be inherently supported by the main processor or its firmware, so that its software does not need privileged code (e.g., device drivers) to make use of the co-processor. Otherwise, software has to be part of the trusted computing base (TCB) for the co-processor to achieve its purpose. We refer to this property of a co-processor as software transparency. A counterexample is the commonly used graphics processing unit (GPU) on PCs and servers, which requires a driver to be installed in the OS/hypervisor, and may not protect against ADV_PRIV.

The rationale behind the taxonomy above is that hardware security features falling in the same category demonstrate similar security properties in achieving secure execution, as shown in the columns "Initial state" and "Runtime access", in Table 1. Nonetheless, this also naturally matches their technical implementation, e.g., co-processors are all stand-alone from the main processor, and execution modes are enter-able and exit-able.

	Feature	Arch	FW	Initial state	Runtime access	Input/output
I: Execution Modes	VMX†	x86	○	N/A	RWX (Isolation)	SW-Verified
	TXT/SVM	x86	○	Launch		HW-Enforceable
	SEV/TDX†	x86	○	Launch		SW-Verified
	PEF† [75]	POWER	○	Launch		SW-Verified
	SE†	System Z	○	Launch		SW-Verified
	SGX†	x86	○	Launch		SW-Verified
	SMM	x86	●	Update/Boot		HW-Enforceable
	TrustZone	ARM	⦿	N/A*		HW-Enforceable
	CCA Realm†	ARM	○	Launch		HW-Enforceable
	XuCode [85]	x86	●	Update/Boot		N/A
	Iso-X[56]	Academic	○	Launch		SW-Verified
	Sanctum [40]	Academic	○	Launch	Side channel ✗	SW-Verified
II: Extensions	MPX	x86	○	N/A	RWX (Range)	N/A
	MPK	x86	○		RWX (Key)	
	MPU/PMP	ARM/RISC-V	○		RWX (Range)	
	SMAP	x86	○		RW (Privilege)	
	CET	x86	○		X	
	PAC	ARM	○		RWX (Key)	
	SMEP/PXN	x86/ARM	○		X (Privilege)	
	NX	x86	○		X	
	MTE	ARM	○		RWX (Key)	
	TME/SME	x86	○		RX (Key)	
	CHERI	Academic	○		RWX (Capability)	
	IMIX [59]	Academic	○		RWX (Privilege)	
III: Co-processors	ME	x86	●	Launch+	SHM + SIG	HW-Enforceable
	IE [168]	x86	●	Undisclosed	Undisclosed	
	PSP	x86	●	Launch+	SHM + SIG	
	Baseband	N/A	●	Launch+	SHM + SIG	
	ACPI EC	x86	●	Static	SIG	
	T2	x86	●	Launch+	SIG	
	Titan (M2)	x86/ARM	●	Launch+	SIG	
	HyperCoffer	Academic	N/A	Launch+	Undisclosed	
	Vigilare	Academic	N/A	Undisclosed	SHM + SIG	
	H100 [124]	GPU (Hopper)	○	Launch+	SHM + SIG	

Table 1. Properties of selected hardware security features. We include several representative research proposals (denoted as "Academic" in the Arch column) among numerous other relevant ones not shown here. **FW**: whether it forms a firmware EE ● or not ○. The distinction between firmware and software on mobile platforms is not binary (e.g., they are just on different flash partitions) hence ⦿ for TrustZone. *: TrustZone does not directly protect initial state which is usually handled by the code running inside (TEE OS, Section 4.2). Launch+: These co-processors employ various mechanisms to check for integrity before execution with their functionality-rich OS (Section 4.1) hence designated as Launch+ covering both Update/Boot and Launch. The runtime access regulation can be based on"Range" (specified ranges), "Privilege" (the subject code's privilege), "Key" (certain secrets) or "Capability" (pointer metadata). "SHM" means the co-processor shares memory with the main processor (via memory bus) whereas "SIG" means the co-processor only uses signaling (I/O). †: the execution mode works on virtual address spaces, allowing multiple concurrent instances.

3.3 Example Hardware Security Support

To facilitate subsequent discussions, we select several representative COTS hardware features to match the categories.

Execution modes. Each architecture has several fundamental execution modes one of which the processor is always in. For instance, x86 has the 64-bit mode (a sub-mode of the long mode) as well real/protected modes. On top of such fundamental modes, other execution modes are introduced for various purposes. For example, to support x86 virtualization, Intel introduced the VMX root/non-root modes (one for the hypervisor and one for guest OSes). There exist also execution modes that are more security-oriented, as explained below (roughly in a chronological order for x86).

The system management mode (SMM) [50] has been part of the x86 architecture since the early 1990s, which handles low-level system configuration and run-time critical events such as hardware failures and power management. Hardware that detects these critical events raises system management interrupts (SMIs), which in turn cause SMI handlers—code running in SMM—to execute. SMIs can also be triggered by software. SMM can be considered to defend against ADV_UNP due to its own highly privileged nature.

Intel Trusted Execution Technology (TXT) [80], together with its AMD counterpart — Secure Virtual Machine (SVM), was among the early processor execution modes dedicated to security (introduced circa 2007). It aims to create an exclusive native execution environment where bare-metal code (e.g., OS/hypervisor) can run, by superseding any already started program on the same processor, hence also called a late launch. The programmer can specify a region of code known as the measured launch environment (MLE) [80] which will be protected in terms of both memory access and I/O access (against a weaker form of ADV_PHY; see Section 4.2). AMD SVM's protected region of code is referred to as the secure loader block (SLB) but limited to only 64KB in size.

Intel introduced Software Guard Extensions (SGX) [39] in 2015 for protecting user-space (hence concurrent by our definition) code from privileged code by running it in an *enclave*, which is part of a process. Unlike TXT, which can only have one instance for the entire system, there can be multiple SGX enclaves concurrently just like regular processes managed by the assumed untrusted OS. Enclave management relies on the potentially malicious OS (ADV_PRIV) assisted by the SGX platform software (PSW), but enclave protection is enforced by hardware and thus the untrusted OS, which is software, can be outside of the TCB [87].

AMD and Intel successively provided similar execution modes to protect guest VMs against other VMs or the hypervisor (ADV_UNP/ADV_PRIV), i.e., Secure Encrypted Virtualization (SEV) [115] and Trust Domain Extensions (TDX) [141]. For TDX, a trust domain (TD) is achieved partially relying on TXT for verified initial loading of the TDX module. IBM has also added hardware support for secure VMs on two of its main architectures, called Protected Execution Facility (PEF) [75] and Secure Execution (SE) [21], on POWER systems and System Z, respectively (cf. attestable cloud infrastructure in Section 5.2). PEF and SE both can create secure guest VMs as SEV and TDX do. The above-mentioned VM-protection execution modes are concurrent, allowing multiple instances.

On ARM Cortex-A, TrustZone [122] creates two parallel worlds, secure and normal, both allowing native execution, switchable through a privileged monitor mode. Memory/IO accesses are marked by an NS (non-secure) bit indicating whether they pertain to the normal/secure world. The two-world model can defend against both ADV_UNP and ADV_PRIV. TrustZone works in a similar way on Cortex-M. ARM also caught up on the VM-level protection (similar to TDX and SEV) by announcing in 2021 the Confidential Compute Architecture (CCA) [104], which enables the creation of VM-level "Realms" (comparable to TDs), supported by the Realm Management Extension (RME) hardware. This is orthogonal to the NS worlds. Note that as of this writing, there is still no COTS

ARM CCA hardware and most proposed security solutions based on CCA make use of ARM's official simulation models (e.g., [192]).

Extensions. More hardware features are not designed to be execution modes, but in various forms that extend and complement an existing EE, usually regulating run-time accesses (RWX) inside the EE in various ways. This notion of extension, aside from not creating an execution mode, means the new feature was extending an execution mode at the time of introduction, but thereafter may become part of that execution mode. For example, the long-established NX bit (also known as the Execute Disable Bit—EDB) can avoid executing "data", leading to security consequences. Today, the NX bit is commonly considered to be part of the baseline security protection.

The extensions can be compared in two aspects: *the involved access (RWX)* and *the way the extension enforces access control*. For example, on x86, with all three types of accesses considered, Intel MPX [127] enforces programmable bounds for pointer references to ensure that accesses always fall in the expected range. On non-x86 platforms, there is also a hardware extension called memory protection unit (MPU) [15], which allows to specify ranges of memory each having respective permissions for RWX. ARM Cortex-M (microcontrollers) makes use of the MPU to defend against unauthorized memory accesses. It is also referred to as xPU by Qualcomm. On RISC-V, this extension is called physical memory protection (PMP) which does roughly the same as the MPU. Such extensions are all designated to be `RWX (Range)` in Table 1 (all access types range-enforced).

The range protection can also be enforced using keys (not necessarily cryptographic, depending on the threats assumed). Intel Memory Protection Keys (MPK) [160] allows tagging memory pages with a 4-bit key so that they are accessible only to threads with a matching key. On ARM, Memory Tagging Extension (MTE) [10] uses 4-bit tags assigned per-allocation (i.e., `Key`) and prevents subsequent pointer accesses with an incorrect tag. ARM Pointer Authentication Code (PAC) [10] protects the integrity of the pointer (instead of tagging target memory) by signing and verifying the pointer value. As the signature has to match, we also designate PAC as `Key` (i.e., without the processor-specific key, tampered pointers cannot be re-signed). To enhance confidentiality (mainly defending against ADV_PHY [68]), several extensions were introduced dedicated to encryption. For instance, Intel Total Memory Encryption (TME) or its enriched multiple-key version MKTME and AMD Secure Memory Encryption (SME) both provide new CPU instructions for key configuration and encrypt memory at different levels of granularity and thus both are marked with `Key` with respect to `RX`.

Other extensions may also be access type-specific. For instance, the execute access (loading an address to the program counter register) concerns control flow transfers, so the aforementioned NX bit and the more recent Intel CET [82], protecting execution from violating control flow integrity using shadow stack, are both assigned an `X` in the table. There exist also extensions defending against confused deputy attacks [69] (i.e., privileged code tricked into abusing userspace code/data by ADV_UNP) for data access (`RW`) with Supervisor Mode Access Prevention (SMAP), and execute access (`X`) with Supervisor Mode Execution Protection (SMEP), of which ARM's counterpart is privileged execute-never (PXN).

Co-processors. Below, we list several typical co-processors with software transparency, i.e., inherently recognized by the main processor or its firmware. Such co-processors are usually either part of the chip package (e.g., multi-chip module, MCM) or at least part of the motherboard, e.g., as a microcontroller in the I/O subsystem.

Intel Management Engine (ME) [136] is a co-processor running on all Intel chipsets manufactured after circa 2008.[2] It is a stand-alone computer system with its own processor and memory, and runs critical system management tasks. The ME is micro-architecturally supported, i.e., the system

[2]The ME was later renamed to CSME (converged security and manageability engine) and for mobile/embedded platforms, it is called TXE (trusted execution engine) and SPS (server platform services) for servers. We refer to it just as ME hereafter.

works with ME, transparently to any software unless software explicitly calls it for service. Similar to the ME, Intel Innovation Engine (IE) [168] is also a co-processor but little information has been disclosed (reportedly available for deploying third-party trustlets; see below). AMD Platform Security Processor (PSP) [52] (with an ARM processor) is a counterpart of Intel ME. The baseband processor (BP) is a processor dedicated to telecommunication functionalities, e.g., protocol stack and signal processing for GSM/LTE, etc. BP is different from the application processor (AP), which could be of any architecture including ARM as mostly seen currently.

The Trusted Platform Module (TPM) is a security chip residing on the motherboard of many x86 systems and certain mobile platforms. The TPM provides numerous well-defined security and cryptographic functions. It contains volatile memory called platform configuration registers (PCRs) which can only be updated by a specific irreversible operation `extend` to hold measurement data, and its non-volatile storage can be accessed as "indices" with various forms of protection. Note that although TPMs are usually transparent to software[3], they are passive and not programmable (accepting commands for only a fixed set of operations) to run arbitrary code. But they play an important role in security and are used by various hardware security support, e.g., Intel TXT and AMD SVM.

Co-processors allowing deployable firmware, as opposed to fixed functionality, can host code to achieve various security purposes. In this case, such code is called *trustlets* in certain terminology [31], e.g., the Boot Guard and Platform Trust Technology (PTT) mentioned in Figure 1 running in Intel ME. These trustlets can serve as hardware-based security building blocks. We simply consider the trustlets as an application of the corresponding hardware security support (ME in this case) in Section 5.2.

Along the line of confidential computing (see Section 5.2), NVIDIA released the first GPU H100 [124] that supports trusted execution environments on GPUs in addition to CPUs, and even partitioned into multiple mutually untrusted isolated units called MIGs (Multi-Instance GPUs).

4 CASE STUDIES

Not all EEs are intended to provide all the aforementioned security properties (let alone security might not be a design goal). In this section, we examine two cases of EEs designed with explicit security considerations: Firmware EEs and trusted execution environment (TEE)s, and their security properties as shown in Table 1. Pertinent to the observation that certain features of commodity hardware are adopted from research proposals (in Section 2), a proposed EE (e.g., a combination of software and hardware) can simply be made a single execution mode by the manufacturer. Therefore, we will see in the following that the discussed commodity firmware EEs and TEEs are also themselves execution modes.

4.1 Firmware EEs

Firmware is part of the TCB of almost all software and hardware-assisted security solutions. This also includes SGX which Intel claimed to exclude firmware to minimize the TCB, because SGX actually relies on a portion of firmware in the Serial Peripheral Interface (SPI) flash chip for monotonic counters [110], and also the SGX functionalities are partially implemented in CPU microcode/XuCode (which can be considered a special form of firmware from a security perspective) [39].[4] Because of this, firmware execution is often protected by various mechanisms in an effort to make it more trustworthy. We classify firmware EEs into two types, based on where they reside, and examine their ability to satisfy the three aspects of secure execution.

[3]Host firmware (see Section 4.1) uses the TPM to measure/record the integrity of the loaded code upon power-on or restart, independent of and transparent to regular software.

[4]Microcode aligns with Opler's original definition of firmware [129] in 1967, although it does not run the same ISA as other firmware but instead contributing to the creation of the ISA.

Host firmware. This refers to firmware running on the main processor. In the case of x86, it mainly includes the BIOS/UEFI[5] firmware, which performs the early (but complex) initialization before bootloaders like GRUB [142] and the OS. Nevertheless, host firmware's influence extends beyond boot time. For instance, code running in the aforementioned SMM persists after system boot. Similarly, when a computer wakes up from sleep modes, its system and peripheral state remains uninitialized, the UEFI Boot Scripts as part of the host firmware get executed to reinitialize the system to the pre-sleep state. Host firmware can also include CPU microcode, which is used to implement various interfaces/instructions that may be called by software. A similar situation is for Intel's XuCode [85], which is on top of the microcode. Note that different from other host firmware, microcode and Xucode are not at the same abstraction level: while other host firmware runs the x86 ISA, microcode runs the micro-operations and XuCode runs the XuCode ISA (which eventually calls microcode), both creating the x86 ISA. Host firmware inevitably shares the same processor and, to a certain extent, memory with other untrusted code.

Co-processor firmware. Such firmware executes on a dedicated co-processor, which gives a natural edge over host firmware in terms of isolation (despite the remaining attack surface discussed in Section 6.1, e.g., caused by shared memory/storage with the main processor).

Firmware EEs on COTS systems are usually not open to developers (other than OEMs) to deploy new code. This can be seen from Table 1, wherever the column FW has a filled circle (●) the EE formed by the corresponding mode will need some hackish approaches for the defense code to be deployed (usually through exploiting a vulnerability and then having it patched [193]). ARM TrustZone is an exception (marked with a half-filled circle ⊙), as to end users it is a firmware EE (not open) but allows certain developer access in a vendor-specific manner. There are numerous firmware EEs with diverse properties, we discuss some notable firmware EEs with respect to the three secure execution aspects.

Initial state (FW EEs). One property common to firmware EEs is that they usually enforce update-time/boot-time state correctness. This is initially ensured during the firmware update process often through checking the cryptographic hash (for integrity) and signature (for authenticity) of the code before writing it to flash memory. For instance, the BIOS update [38] for PCs (which actually updates both host firmware (including CPU microcode) and co-processor firmware, e.g., that of Intel ME) takes a binary image file, verifies its signature and writes it to the onboard SPI flash. The Advanced Configuration and Power Interface (ACPI) Embedded Controller (EC) [159] is a microcontroller used to support OEM-specific implementations, whose firmware is also included in the BIOS update.

Furthermore, certain firmware is also measured on each boot, as a trade-off between the one-time update-time verification and more secure but frequent launch-time verification (with per-launch overhead). For example, three such mechanisms are static core root of trust for measurement (S-CRTM) [66], UEFI Secure Boot [169] and Intel Boot Guard [54], on x86. S-CRTM calculates the hash of the firmware components in a chained manner and stores them in the PCRs of a TPM, making them available for remote attestation (see Section 4.2). UEFI secure boot, on the other hand, verifies the binary image against a system-specific policy at boot time.

It is noteworthy that co-processor firmware usually provides launch-time integrity (Launch) in addition to Update/Boot. These co-processors have full-fledged functionalities even with their own OS, often enforcing more integrity verification other than just the initial firmware image checks. For instance, Apple's T2 [149] ARM-based security chip runs the bridgeOS providing a wide range of security functions including secure boot and the Secure Enclave. Likewise, both Intel's ME and AMD's PSP are functionality-rich enough [52, 148] to perform various launch integrity checks.

[5]The Unified Extensible Firmware Interface (UEFI) is a newer standard for platform firmware that is intended to replace the older BIOS specification.

Last, Secure Boot is also one of the main functionalities of the Google Titan chip (covering Titan's own firmware as well as the host's boot firmware). On mobile platforms, the BP's firmware goes through various checks including the verification by ARM TrustZone [157].

Runtime protection (FW EEs). Firmware EEs usually enforce runtime protection from external accesses (memory isolation), which varies with firmware types: as host firmware shares the same processor with the rest of the system, the focus is to use hardware/architectural mechanisms to *maximize isolation* (e.g., protecting SMM code in system management RAM (SMRAM) [136]); co-processor firmware is naturally isolated but due to resource constraints of the co-processor and functionality requirements, resource sharing is inevitable (e.g., the ME shares RAM with the CPU [55]), so the effort is to *minimize sharing/exposure*. Note that the co-location of co-processor firmware and host firmware, e.g., sharing the same flash storage, is addressed by initial state security assurance such as secure boot.

Most host firmware does not remain available to (coexist with) regular software once control is handed off, and thus runtime protections may not apply. As one exception, on x86, the SMI handlers in SMRAM are always there to be triggered by SMIs. They are well protected by the CPU and memory controller, e.g., regardless of whether the CPU is in the SMM mode, no external access (read/write/execute, hence RWX for SMM in Table 1) is allowed, even for accesses from the highly privileged ME [136] (see Section 6.1 for attacks). Another exception is the UEFI Runtime Services which, after the system has been initialized, still remain invocable by regular software providing services such as resetting/shutting down the system and time. The runtime protection is relatively weak as UEFI services are accessible by any ADV_PRIV (e.g., the OS), unless the system implements UEFI LockBox. The idea of a LockBox is to create a "container" to maintain the the integrity of firmware data (but not necessarily its confidentiality), which still remains a concept [181], to be potentially implemented by individual vendors.

For co-processor firmware, we consider two coarse types of attack vectors against minimal sharing/exposure: 1) signaling (SIG in Table 1) refers to communication with any protocols involving sending/receiving bytes that does not map into one another's address space. This is usually necessary as the co-processor needs to communicate with the main processor. 2) shared memory (SHM in Table 1) leads to better autonomy/performance but a larger attack surface at the same time. If memory mapping does not undergo sufficient checks, the co-processor firmware could regress to the same situation as host firmware has faced. Next, we examine individual cases (attacks will be discussed in Section 6.1).

As a typical co-processor firmware EE, Intel ME is inherently isolated. However, full-control out-of-band management requires bulk data transfer capability with the main processor. Therefore, Direct Memory Access (DMA) is constantly active via the Unified Memory Architecture (UMA) mechanism, which was initially used between the GPU and the CPU. Due to the limited memory space on the ME processor, it uses the UMA region (hence occupying part of the host memory) as its execution RAM [148]. Host Embedded Controller Interface (HECI) is used for signalling or transferring a small amount of data, and we would conjecture it is a similar case for Intel IE. AMD PSP also needs memory sharing (mapped) with the x86 processor [52] in addition to signaling, for a similar reason (PSP's memory is purportedly only a few hundred KB). Baseband processors are no exception on mobile platforms. BPs (also known as the "modem") were accessed merely via commands over a serial connection by the AP previously [145] (signalling), until in recent years, the complexity of mobile devices surpassed that of PCs. Today's BPs also share memory regions with the AP (e.g., in the case of Qualcomm [157]). By contrast, due to its simplicity, the ACPI EC is connected to SMBus [159] for lightweight communication with the CPU, hence can be considered to only have signaling. Although Apple's T2 (which is based on an A10 ARM processor)

also oversees the storage DMA path for encryption [149], it does not appear to expose shared memory with the CPU. Likewise, Google's Titan security chip only relies on the SPI interface [89] for various functionalities including secure boot. Note that signalling alone without shared memory does not necessarily mean better runtime protection but just a smaller attack surface.

State continuity (FW EEs). As most host firmware EEs only execute upon system initialization (with exceptions including SMM and XuCode) and co-processor firmware EEs are on a dedicated processor without exiting or being interrupted by host software, state continuity does not apply. Nonetheless, we still examine a special case of host firmware—SMM. It is by itself an execution mode which can be entered and exited, so the question is upon resuming/re-entering SMM whether the previous state's integrity can be ensured. The "state" of the SMM EE is stored in a memory region called the SMRAM, which is protected from outside accesses all the time even when the processor is not in SMM. This, in theory, can ensure state continuity as the SMRAM content cannot be modified, not to mention rolled back. Attacks will be discussed in Section 6.1.1 and Figure 3.

Input/output security (FW EEs). As host firmware is usually exclusive and privileged, when no ADV_PHY is considered, FW EE's input/output security can be directly enforceable by hardware (hence "HW-Enforceable" in Table 1). This also applies if the entity on the other end is merely software, e.g., as in Nighthawk [193] which uses Intel ME for OS introspection. Note that in both cases, as long as the path is protected, what has been received is only assumed authentic (e.g., still susceptible to confused deputy attacks [69]). Optionally, when the entity on the other end is capable of cryptographic computation (e.g., with a processor), the input/output can be verified ("SW-Verified") in the face of ADV_PHY, ensuring also authenticity.

4.2 Trusted Execution Environments

A TEE is a feature of the processor hardware which supports highly protected (compared to software-based protection) code execution through isolation and cryptography-backed measurement and attestation, hence achieving trusted computing [100]. The defining features that distinguish TEEs from other EEs are that TEEs provide Launch initial state correctness and attestation capabilities to prove their integrity to another party, aside from runtime isolation the execution mode provides. Today's processor architectures usually support one or multiple TEEs (even on microcontroller-like platforms, e.g., TrustZone profile M [123, 170]). They have different positionings suitable for various use cases. As introduced in Section 3.3, certain execution modes are per se also TEEs, inspired by or converted from previous (academic) prototypes, e.g., Intel TXT and AMD SVM are TEEs for native execution where an OS or hypervisor can be hosted; Intel SGX is a TEE targeting user-level/cloud applications with multiple instances; and AMD SEV(-ES), Intel TDX and ARM CCA work at the VM-level making each guest VM a protected EE. Also, ARM has TrustZone with the two-world model allowing native privileged execution.

Initial state (TEEs). TEEs provide launch state security via measurements of the code, inputs and environment upon loading of the code. The environment may include any software loaded up to the point of the protected program, as long as the TEE's integrity depends on it as per the threat model. Intel TXT achieves this by using a TPM as secure storage. The TPM can store long-term secrets, as well as perform cryptographic operations using those secrets. Measurements of previously loaded code are successively stored, in a chained manner in a PCR, where each measurement constitutes a cryptographic hash of code that was loaded, as well as its inputs. The complete set of measurements forms the launch environment up to that point. There are two ways that policies can be applied to these measurements. First, values pre-stored on the TPM can designate valid launch environments, and cause the launch to be aborted if the environment does not match one of these values. Second, a process called "remote attestation" allows code in the TEE to produce an attestation or "quote",

which is the measurement signed by the TPM that can be used to attest the identity and integrity of the TEE code and its environment to a remote party for record or later actions.

SGX provides similar capabilities of remote attestation and launch policies. However, the TPM's role is replaced by functionality in the CPU. Aside from enclave code developed by 3rd parties, SGX relies on a set of Intel-provided "architectural enclaves", which help provide functionality to SGX. One such enclave, called the Launch Enclave, implements launch policies that prevent an enclave from starting if it or its environment does not meet the requirements set by Intel. For example, currently, valid enclaves must be signed by a key certified by an Intel-run certificate authority (if not in the debug/simulation mode). Apart from remote attestation, SGX provides local attestation as well. Local attestation can be used to enable enclaves running on a system to attest their code identities to each other more efficiently than using remote attestation. Similar to SGX, Intel TDX, AMD SEV and ARM CCA ensure the correct initial state at launch time with attestation [24, 104, 141] in a similar manner.

TrustZone is an exception, in that it relies on the vendor-specific TEE OS (which could be thought of as certain firmware-equivalent running in the secure world) for TEE functionalities like measurements, attestation and sealing (see below). Therefore, it is designated as N/A in Table 1 for initial state security as the TEE does not achieve it on its own. Nonetheless, we do not consider it to be a limitation but just a different choice determined by the ARM business model.

State continuity (TEEs). As TEEs are designed for open developer access, maintaining application-specific data across TEE sessions has been in consideration so that the applications can be stateful.

For state data on persistent storage, another important feature TEEs provide is sealing, which allows TEE-protected data to be bound to the aforementioned measurements, allowing retrieval (unsealing) only within the same integrity-protected EE. Both TXT and SGX provide data sealing in similar ways. TXT's sealing capabilities are provided by the TPM (based on the PCR measurements) while SGX relies on the CPU keys [7], e.g., if data is sealed to the Enclave Identity it will be equivalent to TPM's `TPM_Seal` so that any alteration to the enclave/program will render unsealing impossible. For in-RAM state data, mainly applicable to SGX, a process forced out of the enclave mode by a fault will perform the asynchronous enclave exit (AEX) [39], which saves execution context securely in RAM, not on persistent storage, to be resumed later. State freshness for TEEs also needs to be ensured so as to prevent a stale state (intact but from a past point of time) from being presented as fresh. Monotonic counters (as provided by the TPM and SGX, despite the limitations [110]) and Merkle trees are commonly used in TEE solutions.

Runtime protection (TEEs). TEEs employ different mechanisms for isolating their runtime memory from outside as they face different threats. TXT is positioned to run a full operating system, a hypervisor or other native code, and designed to isolate itself from corrupted or maliciously configured peripherals because when entering a TXT session the current EE will be overridden, so that TXT's MLE runs exclusively on the CPU. TXT utilizes Intel's input–output memory management unit (IOMMU) technology VT-d [3], which enables the specification of access control policies for the memory ranges that peripherals may modify via DMA.[6] These policies are then enforced by the IOMMU hardware that is part of the CPU and motherboard chipset [80].[7]

By contrast, SGX is intended to run user-level processes and such has to contend with concurrently executing code at the same or higher ISA privilege level. Moreover, the operating system itself is considered untrusted and explicitly inside the SGX threat model. As such, memory is only available to code in an SGX enclave if it is allocated in the Enclave Page Cache (EPC) [39]. All memory in the EPC is protected from access by non-enclave code when cached on the CPU because

[6]Accomplished by specifying memory ranges in the DMA Protected Range (DPR) and Protected Memory Regions (PMRs).
[7]AMD SVM also has AMD's Device Exclusion Vector (DEV) support [6] for the same purpose.

it is part of the Processor Reserved Memory (PRM) range, which blocks access by the operating system, DMA, as well as SMM (mentioned earlier). Further, when EPC memory is evicted from the processor caches, it is encrypted and signed by the Memory Encryption Engine (MEE), thus cryptographically extending the protection beyond the CPU.

TEEs targeting to protect VMs are in between TXT/SVM and SGX. SEV (or its successors SEV-ES or SEV-SNP), the more recent TDX and the yet-to-come CCA defend against potentially malicious other VMs and the hypervisor (ADV_UNP and ADV_PRIV). To that end, SEV encrypts the VM's memory using AMD's SME feature in conjunction with the co-processor PSP for key management. Similarly, TDX makes use of the MKTME engine (multi-key total memory encryption, see Figure 1) to achieve memory encryption for the VM. ARM CCA also supports memory encryption, augmented by its dynamically configurable Granule Protection Table (GPT) [104] for access control. Note that such encryption alone also ensures memory integrity besides confidentiality as any corrupted encrypted memory will lead to message-authentication code (MAC) verification failure. This, in conjunction with the aforementioned freshness, naturally prevents physical memory attacks such as [68] caused by ADV_PHY. TrustZone marks anything pertaining to the secure world with the NS bit [122] unset hence distinguishing accesses. This propagates to the entire system, as opposed to just the processor address space, e.g., the system bridge (for peripheral communication), the cache controller and the DMA controller all look for this NS bit for runtime access control. By default, TrustZone does not encrypt the secure world's memory.

Input/output security (TEEs). Different from FW EEs, TEEs, because of their diverse privilege levels, have various types of entities on the other end. For the unprivileged SGX, due to no control over I/O channels, the only option is software-verified using cryptography (e.g., across the ECALL/OCALL [39] interface). The same applies to the privileged SEV and TDX, because their I/O is controlled by the potentially malicious hypervisor. By contrast, TXT and SVM are privileged and exclusively occupies the entire system, and TrustZone (although parallel with the normal world) has I/O partitioning support, i.e., TZPC and TZASC [122]. Therefore, their input/output integrity can be "HW-Enforceable" in addition to SW-Verified. More explanations can be found in Section 5.2.

4.3 Discussion

Security properties match design purposes. From the case studies above, we see that an EE's state, runtime access and input/output protection are largely determined by and match the EE's positioning. For example, firmware EEs all enforce strong (at least by design) update-time initial state protection, as the vendor/manufacturer is supposed to control what is allowed to run through the process of firmware updates so that firmware can (implicitly) serve as the RoT of any software. Meanwhile, TEEs are open to developers aiming to achieve late launch, i.e., loading security-sensitive code when untrusted code is already running, so they focus more on launch state and runtime protection, but still unavoidably relying on firmware EEs (e.g., microcode, Intel ME and AMD PSP).

Another observation is that while co-processor firmware EEs are inherently isolated from the main processor hence having more advantage in ensuring runtime security, due to the increasing complexity and functionality requirements, the traditional signaling-based interfaces are gradually widened up to memory sharing. For instance, the various BPs used to expose only a serial connection (UART, sometimes through internal USB) as the Radio Interface Layer (RIL) and this RIL interacts with telephony services (in the case of Android). Today's BPs usually share memory with the AP, as mentioned above. Another example is Intel ME, which had to use UMA in addition to the HECI interface. This larger attack surface will be reflected in the identified attacks (refer to Section 6.1).

EE's runtime protection is always relative. For example, TXT's memory protection does not prevent accesses from SMM code, meaning SMM must be part of the TCB of a TXT-based solution.

In comparison, SMM cannot access the memory of an SGX enclave, hence not part of its TCB. Furthermore, SMM's protection is effective against ME [136], although ME is deemed to have a higher privilege (-3) than SMM (-2). Last but not least, extensions like Intel VT-d (IOMMU) can also affect this, e.g., VT-d, when used by TXT, can protect the MLE's memory from being accessed by ME.

Memory encryption crucially protects enclave/VM memory from exposure to cold-boot attacks [68], a protection that TXT does not enjoy due to its lack of encryption (thus exposing its data to SMM and ME as well), hence vulnerable to both ADV_PRIV and ADV_PHY. In contrast, owing to its use of memory encryption and the PRM, SGX is only exposed to the CPU microcode/XuCode. TDX and SEV (with SME) also make use of memory encryption in a similar way. This observation reflects the usefulness of separately introduced memory encryption extensions, which can be potentially combined with an EE.

4.4 Hardware Features Proposed in Research

As reflected in our observed evolution paths of hardware security support (Section 2), there has been a continuous effort from academia that introduces new changes/improvements to COTS hardware for security purposes. Actually, this is an important driving factor for industry: numerous shadow-stack-based papers [29] paved the way for Intel CET. Before Intel MPX, there had been also various proposals [45, 183] enforcing "bounds" for allocated objects/pointers.

Another observation is that the majority of proposed hardware changes are new extensions to regulate memory accesses internally without creating an execution mode, corresponding to the various run-time RWX restrictions in Table 1. We conjecture this is because existing generic hardware features like TEEs can already be applied to enforce access control against external threats, while preventing vulnerability exploits internally involves more low-level metadata collection (a use case in Section 5.1). We observe that most proposed extensions defend against such internal vulnerability exploits, for memory safety assurance, among which we discuss a few below.

CHERI [126, 175] represents a series of research initiatives that introduce a capability-based architecture, known as Capability Hardware Enhanced RISC Instructions. This architecture aims to provide fine-grained memory safety and software compartmentalization. The capabilities in CHERI are akin to "fat pointers," augmented with permission metadata, enabling hardware to enforce read, write, and execute (RWX) permissions with rich semantics. Additionally, CHERI employs tagged memory and tag bits in registers, ensuring that capabilities are non-forgeable and maintaining their integrity. Such memory access regulation is denoted as `Capability` in Table 1. Recently, CHERI has undergone a community evaluation, which included its application in ARM environments [11].

IMIX [59] creates further fine-grained isolation within an EE (in-process isolated pages). Sensitive data can be moved with a special privileged instruction into and out of the isolated pages within the same address space and thus IMIX is designated as `Privilege`.

SDMP [46] proposes to use wide tagged memory as metadata storage for multi-purpose security policy enforcement. SDMP maps each register and machine word to a pointer-sized tag, and the tag propagates with the data along the processor pipeline. SDMP uses a rule cache to determine the output tag of computation results, and software will be responsible for handling misses in the rule cache. Using this approach, SDMP can enforce a variety of memory safety policies.

With the increasing prevalence of Internet of Things (IoT) devices (or embedded systems), there is a need for *out-of-band* memory safety assurance, due to the devices' resource constraints and distributed nature, which allows enforcement from a remote/central system. LiteHax [44] proposes hardware changes to send the control- and data-flow information to a remote verifier for analysis, presumably a desktop machine. HerQules [33] introduces a new IPC (inter-process communication) primitive, AppendWrite, that enables a monitored program to transmit execution events (low-level metadata) to another party for verification, e.g., for out-of-band control flow integrity enforcement.

In Table 1 we also list Iso-X [56] as an early example of execution mode from the research community, which is very similar to SGX to achieve concurrent isolated environments. Flicker [111] is also an early work of this type dated back to 2008 (not in the table as it was based on TXT/SVM, not hardware changes). A more recently proposed security mode for isolation is Sanctum [40] whose unique property is side channel resistance (see Section 6.1.2).

When it comes to introducing new co-processors, research proposals are usually for a specific purpose. For instance, HyperCoffer [176] introduces a secure processor to protect guest VMs against malicious hosting or even physical environments (ADV_PRIV/ADV_PHY). The secure processor's initial state is designed to be easily verified with a public key (while its connection method to the main processor is not disclosed). Vigilare [117] is an SoC (System-on-a-Chip, which runs Linux) designed to snoop the memory bus traffic of the host for kernel integrity.

5 ROLES OF HARDWARE IN ACHIEVING SECURITY

In this section, we survey industry solutions and research proposals that make use of hardware-based mechanisms for securing program execution, in an attempt to answer *RQ2* in Section 1.

5.1 Revisiting Hardware Advantages

To better understand the roles hardware plays and why hardware is more suitable than its software counterpart, we re-examine the perceived advantages of hardware discussed in Section 1 from a technical perspective.

Efficiency. First, dedicated hardware can perform computation in parallel with regular code execution, therefore gaining a performance advantage over software. For example, when an identical computation needs to be performed on a large set of data, a hardware implementation can simply replicate the necessary logic to process multiple data elements in each cycle, while a software implementation has to use loops to process them one at a time unless it can utilize the parallelism enabled by hardware (e.g., multithreading). Second, hardware has finer-grained control of the silicon asset and can better eliminate unnecessary latency introduced from the ISA abstractions. This means that even when no parallelism is available, hardware implementation can perform certain operations faster than software. Instruction fetching and decoding significantly contribute to execution overhead, which is the price of software's flexibility. For simple operations like bit manipulations and control flow transfer, while their execution needs little logic, software code needs to use instructions to wrap them for execution. This matches the perceived efficiency advantage and indeed hardware acceleration has been a well-known technique to speed up operations that are slow in software implementations. Since enforcing security policies can involve heavy-weight metadata maintenance and checking, hardware acceleration is a natural fit for implementing security extensions.

Immitability and lower layer of abstraction. In addition to relative immutability (i.e., not directly modifiable by software unless intended), hardware also sees what has been abstracted away from software. If efficiency is a benefit nice to have, the lower-layer advantage makes hardware security support indispensable. This is reflected in several typical use cases of hardware seucurity support.

Access control enforces a set of rules (i.e., policies) that specify what a subject can do with an object. Because the subject can come from untrusted sources and the enforcing mechanism must be as close as possible to the object (resources like memory, device I/O, etc.) for effective control, these rules must be enforced by components from a lower layer of the hardware/software stack than that of the subject. There exist also software-only solutions, e.g., sandboxing and compiler instrumentation (when source code is available) to check access to software constructs (e.g., files and the syscall interface). However, they cannot effectively enforce access control for hardware resources (e.g., memory) with a lower abstraction level, and more importantly software-based access control mechanisms need to also protect themselves from other software's tampering.

Furthermore, hardware support is the only choice for enforcing access control for hardware subjects (e.g., a potentially malicious PCIe/USB device — ADV_PHY). Pure-software solutions sit above the hardware layer have no opportunity to intercept and check such accesses. Therefore, hardware support is a natural choice as it sits below all software layers.

Execution metadata collection. Security mechanisms sometimes need software execution metadata, e.g., control flow and cache accesses. However, commodity hardware only provides execution metadata limited to *summary* information that can be correctly and efficiently read by software. When more detailed information needs to be collected, although it is possible to use software-only approaches like compiler instrumentation or binary rewriting to transform the program and record the information, it may not be practical. The first problem of a software-only information collector is due to its residing in the same layer as attacker software, which can compromise it and forge/erase the data to evade detection. The second problem has to do with the low efficiency unlike hardware. For high-bandwidth metadata (e.g., control flow traces), a software-only tracer can incur unacceptable performance overhead. Fortunately, hardware support can perform well in these scenarios, which can collect information in parallel with code execution, avoiding the performance overhead.

5.2 Application

We next survey use cases of hardware security features in various individual security solutions from both industry and academia.

Generic isolated environments. Hardware security support as an execution mode is often directly used to achieve or enhance isolation. This is especially necessary when a desired level of TEE is absent. For example, on the x86 platform, SMM has been a popular mechanism used as a TEE equivalent or to extend TEE capabilities. Before the introduction of SGX, SICE [14] used SMM to create and manage ICEs (isolated computing environments), which are isolated from the OS/VMM similar to modern SGX enclaves. TrustZone has also been used to implement ICEs for ARM, where no equivalent to SGX currently exists. Sun et al. proposed TrustICE [154], which supports ICEs in the normal world, managed by a trusted domain controller (TDC) in the secure world.

A major challenge for such application-level ICEs is that the code inside them does not have access to trustworthy OS services (blindly using OS services outside an ICE can open the door to Iago attacks [32]). Solutions include adding more OS functions in the ICEs, like library OS Haven [17], or performing checks [144] at the system call interface to the untrusted OS.

Attestable remote execution. Users of cloud resources need a way to remotely ensure that their workload is running securely. This situation differs from when a PC/smartphone user or a server admin can physically see and control their system. Here, what is unique is that the owner of the workload is remote in the face of ADV_PHY/ADV_PRIV. TEEs supporting remote attestation fit well here. In general, three sets of typical hardware features used here are TXT/SVM with the TPM (privileged, to protect the infrastructure resources), SEV/TDX (VM-level), and SGX (unprivileged, to protect user application instances).

There has been a recent trend of confidential computing [63, 75, 114], where remote attestable execution is essential. For example, Openstack allows to set up compute nodes in trusted compute pools [128] backed up by TXT which can be attested to by a preconfigured attestation server. Similarly, VMware's vSphere ESXi [86] (its enterprise-class bare-metal hypervisor) also supports using TXT for host integrity verification and trusted compute pools. Similar support is found in XenServer [37] as well. On the instance protection end, major cloud service providers usually make use of SGX, e.g., Alibaba Cloud ECS Bare Metal Instance [5], SEV, e.g., Google Cloud [63], or both SEV and SGX, e.g., Microsoft Azure [114], for confidential computing VMs. In all the cases of TXT/SGX/SEV/TDX, isolation and attestation capabilities are the security properties involved.

NVIDIA introduced its own confidential computing hardware support on the GPU H100 [124] (see Table 1). Remote attestable execution is also needed in the other direction, where client devices attest to a remote server (or another client in multi-party computation), e.g., on embedded systems [2, 31] (in addition to SGX-capable PCs).

Bootstrapping and protecting monitor code. In addition to creating generic ICEs, hardware security support is more often used to secure sensitive code for both initial state and runtime access.

Hypervisor (VMM) integrity. HyperSafe [167] makes use of TXT to bootstrap a solution for hypervisor integrity protection. TXT as a TEE does launch-time measurement, ensuring the correct initial state. It then implements memory lockdown for hypervisor pages by first protecting the pages with W⊕X (on x86 this relies on the NX bit), and then trapping any writes to page tables using the MMU functionality. HyperGuard [138] and HyperCheck [166] are two other examples of using SMM for hypervisor integrity. The difference is that HyperCheck outsources the core logic to a remote machine, with the network card driver also in SMRAM; whereas HyperGuard collaborates with a chipset-based mechanism DeepWatch [27] (a trustlet in Intel ME). Both assume proper firmware protection (e.g., initial SMRAM integrity).

Guest VM integrity. CloudVisor [186] protects the integrity of VMs under the threat of compromised hypervisors (ADV_PRIV). It also relies on TXT to ensure clean initialization. Then, it implements an integrity monitor using nested virtualization to enforce isolation and protection of VM guests' resources. Sensitive operations such as page faults, critical instructions, and I/O are trapped and examined by the monitor. The drawback of nested virtualization is that it increases overheads from the inspection of exceptions by the monitor.

OS kernel integrity. There are also works on kernel integrity protection (against ADV_PRIV) using hardware security support, mainly TEEs. While conceptually similar to switching between other execution modes, ARM TrustZone's implementation has lower switching overhead between its "normal" and "secure" worlds. This enables solutions such as TZ-RKP [12, 13] and SPROBES [61] to protect normal-world kernel integrity by handling critical kernel events in the secure world. In both systems, kernel binary rewriting is needed to replace sensitive instructions with an invocation to the secure-world monitor.

Secure user-machine interaction. When a human user interacts with a computer, it is possible for ADV_PRIV to intercept and tamper with the traffic between the I/O devices and the application. In this case, hardware can be used to set up a trusted path between them. On the other hand, as the third advantage mentioned in Section 1, the user can be assured of what she is interacting with by visually paying attention to hardware as the interface between the physical world and software, such as an LED indicator.

Here it matters if peripherals are aware of whether they are being accessed by trusted or untrusted code. ARM TrustZone's advantage lies in its use of a bus-level bit that signals peripherals whether the processor is in the secure or normal world, necessitating separate discussions for x86 and ARM.

x86-based solutions: Since the x86 platform does not employ a bus-level bit, trusted I/O with peripherals requires that the TEE takes exclusive control over both the system and the peripheral, the latter often by re-initializing it to put it in a known good state. A number of solutions employ TXT or SVM to do this. UTP [58] hooks pre-configured user data entry events/transactions, such as confirming an online purchase. It redirects the user to a TEE session (TXT/SVM reusing the Flicker [111] framework) and confirms the transaction with the user over trusted I/O. Similarly, Bumpy [112] also makes use of TXT/SVM with Flicker to protect user keyboard/mouse inputs. It involves more components for better usability and security: a USB interposer (an ARM board that could be later integrated into the keyboard/mouse) and a Trusted Monitor (a smartphone). A

keystroke is encrypted by the USB interposer, processed and verified by the Flicker session, user-confirmed on the Trusted Monitor, and sent to the server. SMM can also enhance the trustworthiness of user interactions. TrustLogin [187] employs SMM to secure password-based login by directly transferring inputs from the keyboard to the network.

ARM TrustZone-based solutions utilize a bus-level bit to segregate external peripherals into trusted and untrusted zones through its TrustZone Protection Controller (TZPC) and TrustZone Address Space Controller (TZASC). This capability underpins research proposals like TruZ-Droid and TrustUI, which enhance user interaction security by moving sensitive inputs to the secure world, indicated by a dedicated LED, and employing input and display randomization for a trusted user-device path, respectively, both leveraging the TZPC.

TrustZone has also been applied to secure interactions with external sensors. SeCloak [103] can securely and verifiably place an external device in a user-approved state (on/off). It leverages both TZASC (as secure memory of the s-kernel) and CSU (Central Security Unit, a custom TZPC).

Commodity application of firmware EEs (trustlets). As we consider the trustlets defined in Section 3.3 as an application of hardware security support (firmware EEs), we briefly discuss a few such cases. Taking Intel ME and AMD PSP together as an example, based on disclosed information [148], currently hosted trustlets include: Intel Active Management Technology (AMT) for out-of-band remote management, such as powering on/off, booting from a disk image and viewing video output (even including the BIOS screen); the firmware-based TPM (fTPM, and in Intel's case PTT), Intel Protected Audio-Video Path (PAVP) [136], Intel Threat Detection Technology (TDT) [84], Intel Boot Guard [54], etc. fTPM/PTT is similar to a discrete TPM but implemented as firmware in ME. Intel PAVP is a DRM trustlet that ensures secure high-definition video playback from a supported content provider by directly accessing the GPU for decoding, so that a local user cannot pirate content from such providers. Its successor is called Intel Insider [133]. All these trustlets run as an "application" in the ME/PSP firmware EE.

Memory/Type safety assurance. In type-unsafe languages like C and C++, programmers must manage memory and use features like pointer arithmetic and type casts safely. Failure to do so can lead to "undefined behaviors" such as crashes, data loss, or vulnerabilities exploitable by attackers. This is known as *memory corruption*, which breaches memory object properties (e.g., range), and its mitigation toward memory safety involves various aspects like spatial/temporal safety and integrity checks. However, ensuring the absence of memory corruption in large-scale software is challenging due to human error and the scalability limits of current program analysis techniques.

Enforcing memory safety imposes internal restrictions on code behaviour for run-time accesses (RWX). In contrast to traditional memory protection mechanisms (e.g., page-based virtual memory) that achieve isolation, memory safety solutions typically enforce a more strict policy that incorporates programming language semantics like objects and functions, as memory corruption happens because low-level details are not sufficiently abstracted away by the language. Hence, most such defences also involve compiler instrumentation. The added semantics effectively introduces more constraints to a successful exploit. On the other hand, the semantic-rich policy causes heavier performance overhead because it requires collection/maintenance of low-level *metadata* that represent language semantics and more complex checks, calling for the information collection role of hardware (mostly column "Runtime access" of Extensions in Table 1). As these extensions are already for specific purposes (unlike the generic TEEs), below we map a few examples of them to typical memory safety solutions.

Pointer-based protection [45]. By logically extending each pointer to keep track of the lower and upper bounds of the permitted address range, spatial memory safety (e.g., no over/underflow) can be

ensured. Intel MPX [127] can catch out-of-bounds memory accesses, e.g., due to bad pointer arithmetic. It stores the pointer bounds in bounds registers and a set of bound tables, a distinct memory range with two-level address translation (similar to that of regular virtual memory), and associates each virtual address to a bounds entry so that the bounds for any pointer can be found and enforced.

Tagged memory [88]. ARM MTE can cover both spatial memory safety and temporal memory safety (e.g., use-after-free) by assigning a random value (which is referred to as the tag) to the memory storing an object and storing the same tag value in the unused high address bits of the pointer to the object. When the pointer is later dereferenced, MTE will check whether the tag of the pointer matches the tag of the memory. A mismatch indicates memory corruption.

Control flow integrity (CFI) [28, 43] with shadow stack. CFI can be considered a weaker form of memory safety that avoids branch destinations that should not occur in correct execution. Intel CET [82] maintains a *shadow stack* [29] in a protected memory region for each thread to keep a copy of the correct return addresses, and when a return instruction is executed, it checks the used return address against the shadow copy to detect corruption on function returns (which is referred to as backward-edge CFI). In addition, Intel CET also inserts a new instruction at the beginning of each function to explicitly label the function entry point, and verify that each indirect jump lands on a labelled entry point to detect corruption of function pointers (i.e., forward-edge CFI).

Points-to authentication. Directly protecting the integrity and authenticity of pointers can also prevent memory corruption from a different angle. ARM Pointer Authentication (PAC) [10] ensures pointer integrity by storing a cryptographic hash to the unused high address bits of a pointer which will lead to an invalid pointer in case of tampering. It provides instructions to "sign" (using a key hidden in the process context) and "unpack" (restore) the pointer. ARM PAC can also serve as a building block for better security guarantees [106], e.g., PTAuth [57] that enforces the points-to validity using PAC when the pointer is being dereferenced.

Secure language environments. The aforementioned memory safety solutions in practice usually assume isolation from malicious parties, thus assuming ADV_UNP only, where hardware-based isolated environments can be applied. On the other hand, when possible, using type-safe programming languages such as Rust in such environments also directly addresses memory corruption. Therefore, secure/isolated language environments for type-safe languages have become a new research direction. For example, Rust-SGX [165] enables the development of SGX enclave code in Rust by creating a binding layer between Rust and the SGX SDK (C/C++). RusTEE [163] employs a similar approach to adapt Rust to support ARM TrustZone by addressing challenges like securely invoking high-privileged system services and securely communicating with the untrusted normal world. Also, SGXPy [185] supports unmodified Python code to run in Intel SGX relying on an adapted library OS. Numerous other secure language environments have also been proposed [140, 164]. From a different perspective, supporting languages programmers are familiar with also improves usability/adoptability of hardware security support (see Section 6.2).

5.3 Repurposing Existing Hardware Support

It is not uncommon to repurpose hardware support designed for another but specific purpose (unlike the generic TEEs) for security. Intel Transactional Synchronization eXtensions (TSX) [81] is an extension that can ensure atomicity of code in a critical section by monitoring access conflicts from other threads without relying on explicit and frequent software locks, hence improving the performance of multi-threaded programs. When no conflict or interrupt is detected, the code can complete the critical section, or it will be rolled back with an exception. T-SGX [143] repurposes TSX as a side-channel defence to suppress the page fault notifications to the adversarial OS (ADV_PRIV), which exploits them to infer enclave memory accesses. As another example, Mimosa's use of

TSX [67] is closer to its original positioning: to abort upon any external read accesses to the secrets of the protected thread, preventing memory disclosure attacks. We also note that certain hardware features are (ab)used to weaken security, e.g., Meltdown [107] uses TSX to improve attack efficiency.

Repurposing for memory safety. Intel Processor Trace (PT) [81] can record control flow traces (including branching decisions of conditional branches and branch targets of indirect branches) of a program and encode the traces into compact packets. The traces can be used to reconstruct control flow. GRIFFIN [60] uses Intel PT to enforce both forward-edge and backward-edge control-flow integrity. μCFI[73] uses Intel PT to improve the protection accuracy of forward-edge CFI, by narrowing the possible set of destinations of each indirect call to a single function. Aside from PT, kBouncer [131] uses Intel Last Branch Record (LBR, a facility recording recent branches taken in registers) to detect abnormal control transfers. CFIMon [177] relies on Intel Branch Trace Store (BTS, similar to LBR but using memory as storage allowing for more records) to detect CFI violations. In all such cases, the involved hardware feature was originally designed for performance tracing/debugging.

6 PROBLEMS

In this section, we discuss where hardware security support still needs improvements.

6.1 Attacks

Despite the advantages hardware possesses in achieving security, hardware security features have been successfully attacked. Generally speaking, the attacks break the various aspects of secure execution as discussed in Section 3.1. Most runtime access or initial state vulnerabilities can be patched in a straightforward manner, e.g., by adding corresponding checks. One exception is side channels, which are more inherent, and developing a patch may not be straightforward.

Mechanisms and Policies. Most hardware security features, due to the lack of high-level software semantics (as explained in Section 5.1), only provide a *mechanism* (the low-level atomic technique) and need to rely on a certain form of configuration data as the *policy* (when and how the technique should be applied). Taking Intel MPX as an example, hardware only enforces the ranges as specified by the BNDx registers for memory accesses but software needs to properly set the values in those registers. The same applies to the NX bit (on which page to set this bit), the MPU/PMP (which ranges to protect), etc. When applicable, we try to reflect this aspect in the subsequent discussions.

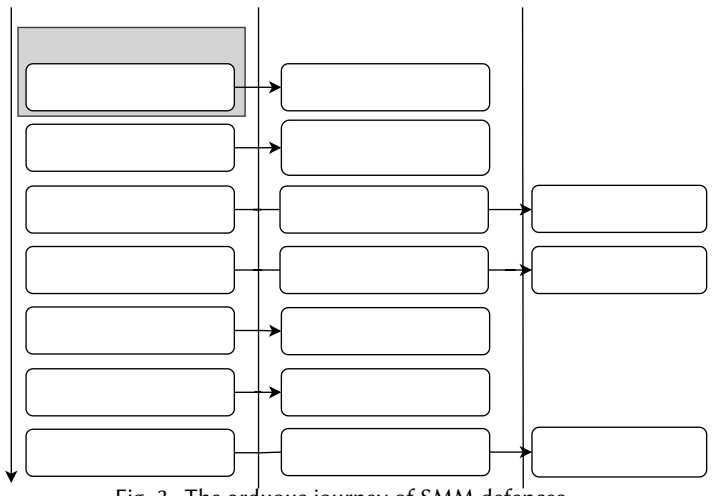

Fig. 3. The arduous journey of SMM defences.

6.1.1 Breaking run-time access protection – Firmware memory not treated differently. To attack hardware security support, although one cannot alter the immutable silicon (even for ADV_PHY in common threat models), they can target the firmware (which is also code) that the hardware depends on. Note that this applies to both firmware EEs and TEEs as the latter also rely on firmware. Therefore, if firmware memory is not treated differently at the architectural or micro-architectural level, problems may occur. Here, we enumerate a few attack vectors.

For example, x86 CPUs use a memory reclaiming mechanism that allows DRAM whose physical addresses conflict with memory-mapped I/O (MMIO) ranges to be transparently remapped to another physical address range by the operating system, thus "reclaiming" that DRAM and making it available again to the OS. In a sense, like MMIO, this remapping abstracts away the use of physical addresses as a means to access both peripherals and DRAM, thus making such conflicts invisible to the rest of the OS. Unfortunately, access control was not considered properly, actually enabling malicious code, such as a compromised driver, to use the reclaiming mechanism to circumvent existing firmware EE's memory access protections.

In the case of SMM, the SMRAM where SMI handlers reside is protected by the CPU at all times, which if fails, would undermine the runtime security of the EE. This can happen either when SMM is in execution or when it is waiting for SMIs at other times, as the SMRAM is just part of RAM. Access to SMRAM undergoes a series of checks in the memory controller hub (MCH). However, since the SMM as a mode is limited to only the CPU, the location information of the SMRAM region does not cross the interface between the CPU and the MCH, and thus the MCH cannot include a check against remapping the SMM range in its reference monitor. Historically, the remapping mechanism was used to remap the protected SMRAM to a location that malicious OS code can access [138]. Subsequent implementations patched this hole and Intel TXT also disables reclaiming [77].

As for Intel ME, the same problem existed. As the MCH was intended to provide out-of-band management transparent to the OS, which requires bulk data transfer between it and the main processor and the MCH processor has very limited memory, sharing from the main memory is needed. As mentioned earlier, Intel MCH uses the UMA region to store execution state [148]. Unfortunately, the effort to minimize this interface did not include sufficient access control, and the security was purely enforced by obscurity. Once the use of this region was revealed, the UMA region could be remapped and accessed by malicious OS code [55, 158]. This weakness was eventually closed when Intel added UMA protection using encryption [55]. For both SMM and ME, the attacks caused by remapping can be considered to be a problem of the mechanism, as opposed to the policy, as software/firmware does not have a way to configure the remapping behavior.

Caching is another example of a case in which firmware memory was not treated differently, resulting in vulnerabilities. Although access to the SMRAM region is enforced by a reference monitor in the MCH, it is only aware of what regions can be cached, which code running on the CPU specifies by setting the memory-type range registers (MTRRs), and whether the CPU is running an SMI handler in SMM mode or not. Since the SMI handler benefits from using the cache, the MCH permits memory access to the SMRAM region while the CPU is executing the SMI handler and denies access at all other times. However, this excessively narrow interface has resulted in problems. For example, a cache poisoning attack proposed by Duflot et al. [49] and again by Wojtczuk and Rutkowska [173], found that the SMI handler remained in the processor cache after it was done executing, and thus could be modified without going through the MCH reference monitor[8]. On the next invocation, the modified SMI handler, still in the cache will run. Duflot et al. also discussed a more efficient scheme to make the attack persistent and not be confined by the size

[8]We see some indication [47, 62] that Intel was also likely aware to some extent of these vulnerabilities before their public disclosure.

of the cache. This problem also pertains to the mechanism as caching firmware memory or not was determined by the MCH hardware. It was eventually fixed by widening the interface and adding a system-management range register SMRR [81], which extends the reference monitor onto the CPU, preventing access to cache lines in the SMRAM range unless the CPU is executing the SMI handler.

If we examine how SMM attacks and defences have evolved over the past two decades (Figure 3), we see that the SMM interface and memory protection has been continually refined as vulnerabilities were found. In every instance, there was an attack affecting the run-time protection (in this case, protections from external accesses), except the 2010 reflash attack, which will be discussed in Section 6.1.3. Early attacks resulted purely from implementation errors and oversights [90, 139, 174]), e.g., if the `D_LCK` bit in the SMRAM Control Register (`SMRAMC`) was not set at boot by the motherboard, any ADV_PRIV would be able to manipulate SMRAM. This was more a policy problem (as opposed to mechanism) as the firmware should have set this bit. Later vulnerabilities after 2009 or so took on a markedly different tone. In addition to the remapping and cache poisoning attacks, SMM callout also needed to be prevented (SMI handler branches outside of SMRAM running arbitrary code), which was fixable with `SMM_Code_Chk_En`. Also, arguments passing to SMI handlers could trick the SMI handler into overwriting SMRAM. The seemingly ultimate approach to address potential SMM compromises was the SMI Transfer Monitor (STM) [78], which limits the trust that other components must have in the SMI handlers by hosting them in a VM. While STM is not ideal, far from adopted and still requires trust in other firmware EEs, it at least reduces the attack surface for the SMM EE.

Pertaining to the host firmware EE, the UEFI defines an interface between operating systems and firmware. Similar to the SMI handlers, UEFI leaves the S3 boot script available at run-time, which defines privileged code that runs when a system wakes up from S3 sleep[9]. A failure to properly set the UEFI variables, a set of mutable values over which the OS and firmware communicate in UEFI, allows ADV_PRIV to maliciously modify a pointer to the S3 boot script, thus allowing redirection to attacker-specified code [171]. Another attack on UEFI Secure Boot [91] was based on a similar approach (i.e., modifying an UEFI variable storing the boot policy). In summary, we can see that there has been additional care needed to treat firmware EE memory differently (which was not present).

6.1.2 Breaking run-time protection – Side channels. Although leading to similar memory access violations (read/write), different from the aforementioned attacks, side channels are more intrinsic and cannot be mitigated by simply adding more checks. Hardware security features (e.g., TEEs) also suffer from such attacks. Side channels have been a long-standing problem for computer security and cryptography [94], which steals data from unintended channels such as timing, power, electromagnetic, and acoustic channels. As far as hardware security support (especially execution modes) is concerned, this is about not directly attacking any components (e.g., by ADV_UNP) but still revealing the secret information (read access). Note that when generalized, side channels can also refer to fault attacks where malicious changes are made (write access), as exemplified by Plundervolt [119] that manipulates execution by undervolting the CPU power, and Rowhammer [178] that can flip bits in memory without accessing them. In this section, we focus on traditional side channels not involving write access. Side channels are usualy deemed to be an inherent problem of the mechanism (let alone policies) as the design goals between functionality and defenses are often in conflict, e.g., side channel defenses favor constant latency while economically minimal latency is desired.

Microarchitectural side channels. The hardware-software interface is one of the most important and heavily abstracted interfaces in computing. The ISA exposes software instructions as an abstraction to CPU hardware, often referred to as the CPU *microarchitecture*. An instruction hides behind it, enormously complex microarchitectural machinery designed to improve performance and

[9]The x86 platform defines several levels of system sleep, S3 is one of them.

efficiency, such as memory caches, out-of-order execution and speculation. Issues not considered in the ISA abstraction have given rise to *microarchitectural side channels* [155] over which information that is meant to be hidden by the ISA can flow to unauthorized principals.

One of the properties of the ISA abstraction is that it does not specify timing, because, for efficiency, instructions should appear to execute as fast as possible. Thus, the execution speed of instructions always reveals information about the microarchitectural state of the hardware involved in the execution of the instruction. Timing side channels account for the majority of microarchitectural side channels. Such side channels can be used to leak both memory content and execution traces. To access memory content, cache-based side channels that exploit timing differences between cache hits and misses have been demonstrated to perform unauthorized access to memory, as detailed in attacks such as Prime+Probe [130] and Flush+Reload [182]. In addition, the TLB (translation lookaside buffer), which caches virtual to physical page mappings, has also been shown to be prone to microarchitectural side-channel attacks [64], due to timing differences. Different types of instructions, when scheduled to run on the CPU's execution engine, may go to different execution units (referred to as "ports"), leading to port contentions for instructions of the same type. This can leak what instructions are being executed by measuring contention over hardware execution ports using timing, as done in attacks such as PortSmash [4] and SMoTherSpectre [19]. Moreover, Nemesis [161] can infer the instruction-granular execution state from the protected task by measuring interrupt handling latency.

Aside from directly leaking secrets (due to the unspecified timing), a pre-attack stage can also be involved to infer or configure microarchitectural state with the objective of *controlling* what is leaked during a side-channel attack. For example, vulnerabilities like Meltdown [107] and Spectre [93] (with multiple variants) exploit abstraction issues in speculative or out-of-order execution to control what information is leaked over cache-based channels like the ones described above. In recent years, more such channel control attacks have been identified, as exemplified by microarchitectural data sampling (MDS) (e.g., RIDL [162]) and gather data sampling (MDS) (e.g., Downfall [116])

All the micro-architectural side channels discussed above, when applied to an EE, can effectively circumvent the runtime protection. For example, it has been well-known that Intel SGX is subject to side-channel attacks, which can be used to leak control-flow information [102] and cache contents [22], as well as an SGX variant of Spectre [34], and in particular, the one that got the most attention (and perhaps the most effective) is Foreshadow, which allowed researchers to directly extract cryptographic keys from SGX's architectural enclaves [25].

Although less reported, side channels have also been shown to be effective against ARM TrustZone. TruSpy [190, 191] exploits the cache contention between normal world and secure world to extract secure-world secrets from an application and the OS respectively in the normal world. Unsurprisingly, AMD SEV is no exception in face of side channels. SEVered [118] allows a malicious hypervisor to extract the full contents of main memory in plaintext from SEV-encrypted VMs. Later, stating that SEVered is inefficient for having to extract large amounts of data in search of a targeted secret, Morbitzer et al. proposed an approach to first observe the guest VM activities in order to infer memory regions likely containing the secrets and hence can extract them more efficiently. Side-channel attacks tend not to be as stealthy as one may imagine as they require fairly complex measurement techniques, meaning that it may be possible to detect and mitigate them [35, 143]. Furthermore, Green et al. also identified AutoLock [65] as a performance enhancement in the ARM processor's cache that adversely affects the cache-based side-channel attacks.

Information leakage via micro-architectural side channels can also be abused to facilitate attacking memory safety hardware extensions, e.g., the PACMAN [134] attack can brute force the correct PAC value of ARM PAC through speculative execution (like Meltdown) without causing crashes.

Controlled channel attacks. For ADV_PRIV and ADV_PHY, non-micro-architectural side channels are also an option, because of their higher privilege. In a controlled channel attack [179], a malicious/compromised OS can observe the patterns of page faults triggered by the protected victim code to extract sensitive information, and thus defeat isolation (e.g., provided by TEEs). T-SGX [143] was proposed to mitigate such attacks for SGX.

The fact that the EEs affected by microarchitectural side channels are mostly TEEs is because firmware EEs are often exclusive, i.e., they do not have other untrusted code concurrently sharing the same processor, leaving mainly the possibility of physical side channels by ADV_PHY (e.g., acoustic or power). For example, SGX has the untrusted OS and all applications in parallel; TrustZone has the untrusted normal world; and SEV has the untrusted hypervisor and other VMs. By contrast, Intel TXT and AMD SVM preempt anything running on the processor and occupy the whole platform, similar to the exclusiveness of firmware EEs.

6.1.3 Breaking initial state protection – Firmware update issues. Two characteristics are common to firmware EEs: first, they are all highly privileged, as shown by the fact that it is considered part of hardware which all software relies on; second, they tend to employ update/boot-time integrity checking (see Table 1), meaning that any corruption of their code after installation or introduced during improper updates will go undetected. As a result, such weak initial state protection, once compromised, could lead to disastrous problems.

On x86, both host firmware and co-processor firmware are loaded from the SPI flash chip on the motherboard, whose only access protection is enforced by the CPU and chipset.[10] This is why if it is desoldered from the motherboard, ADV_PHY can easily read its content with a programmer. What the SPI flash contains includes but is not limited to: the CPU microcode, Intel ME firmware, all trustlets hosted on the ME, certain SGX data (e.g., for the monotonic counter), SMM code (SMI handlers), etc. The BIOS update process is responsible for securely writing the image to the SPI flash and relies on signature verification in conjunction with additional mechanisms like the mentioned CRTM and UEFI Secure Boot, which have been successfully attacked [90, 139, 174], e.g., using oversized boot splash logo to cause a buffer overflow. As part of the arms race, there are also defense techniques [30, 97]. All in all, sharing of the same microchip largely makes the exploits homogeneous and simplifies attacks.

Therefore, although SMRAM is protected at run-time, SMM code loaded from the SPI flash may have been already compromised, which corresponds to the BIOS reflash attack in Figure 3. When it comes to Intel ME, aside from being a victim of the BIOS reflash, due to itself running a full-fledged OS, multiple buffer overflows in the ME firmware kernel (CVE-2017-5705,6,7) were identified leading to arbitrary code execution and its file system (as part of the BIOS image) is also being revealed [147], which may allow more targeted alteration to individual components. Last but not least, the CPU microcode updates, while also loaded from the same image, are different in that they are performed at each boot.[11] The process is initiated by writing to MSRs (model-specific registers) and accepted after cryptographic verification. An early documented attempt was found in an anonymous report [8] which showed an example of inadequate access control enabling the partial update of the adversary's choice. Also, the group of Koppe and Kollenda have used reverse-engineered microcode to change the behavior of earlier CPU models (specifically AMD K8 and K10) [95, 96].

Taking the discussion further on firmware updates, if we consider peripheral devices as the host system (i.e., their processor comparable to the host CPU), the same initial state of the firmware EE can be examined, which also affects the secure execution of their code (e.g., data encryption on a self-encrypting drive – SED). Their firmware update also relies on similar signature verification (if

[10] BIOS write protection is controlled by the `BIOSWE` and `BLE` bits in the `BIOS_CNTL` register of the chipset.
[11] Microcode patches are not persistent and are reloaded during the early boot process from the SPI flash.

any), and is further worsened by the less protected interface (e.g., SATA). For instance, an analysis of the (in)security of today's SEDs identified several vulnerabilities [113]. Some of the attacks can even be mounted using undocumented vendor-specific commands (VSCs)[12] that can be performed by any ADV_PRIV, where the firmware fails to perform the appropriate checks. Similar vulnerabilities due to such check failures exist on other peripherals such as hard drives [184] and network cards [51]. Some work on defences has been proposed, such as Zhang et al. who use SMM on the host to monitor and verify the integrity of the firmware [188], and Hendricks and van Doorn [72] who describe how device firmware can be verified in a trustworthy manner. These failures show that the initial state of firmware EEs is widely affected by the update process.

6.2 Usability and Adoptability Issues

In addition to security issues, there might also be other factors that affect hardware security support being adopted and applied by users. While attacks/vulnerabilities directly cause a hardware feature to fail its purpose, if users are reluctant to make use of a new hardware security feature it may not even get the chance to function, hence losing popularity or even being removed. In this section, we review some common factors affecting adoptability and usability that people from either academia or industry have discussed or expressed opinions about.

Cost. The cost involved in adopting a new hardware security feature is always a key problem, which could be either monetary or in other forms. For example, the error correction code (ECC) memory that can detect and correct data corruption can effectively prevent certain types of memory attacks that can exploit physical memory corruption errors (e.g., Rowhammer [178]). However, adoption of ECC requires more budget not only for the more expensive ECC memory sticks but also for corresponding compatible CPUs and motherboards.

Porting/manufacturer barriers. Adoption of new hardware security features needs to deal with technical incompatibility and manufacturers' business models. For instance, most of the TEEs require code changes/rewrite and certain research proposals aim to ease porting effort, e.g., Panoply [144] can help partition an existing application for SGX. On the other hand, Intel requires commercial SGX users to be licensed [79] for using Intel's attestation services which cannot be implemented by a third party (unsupported by the CPU hardware). Similarly, ARM TrustZone also faces adoptability challenges. For developers to benefit from TrustZone's protection, they must be granted access by device manufacturers to use the secure world TEE OS, which is usually not possible for individual developers, not to mention the TrustZone TEE OS is rather vendor-specific and closed. Although there are proposals to allow TrustZone to be open for individual application developers [74], such proposals have not seen adoption by commercial manufacturers.

Efficiency. As with software, added hardware/firmware features often also come with performance overhead. Intel MPX was first available as an extension in the Skylake series of CPUs, while unfortunately it was not widely adopted due to significant overheads (50% on average [127]) in the second-level table for storing pointer boundary addresses, and was eventually removed by common development tool-chains like GCC 9 [99].

Correctness. The implementation of new hardware security features can also be buggy. After Intel TSX was shipped as a hardware extension to Broadwell and Haswell CPUs, a bug was found in the hardware that caused TSX instructions to result in unpredictable system behavior [83], and a later microcode update patch from the Intel directly disabled TSX. Such incidents may affect users' overall confidence in hardware security support in general.

[12]These VSCs are device commands sent through the SATA or NVMe interface.

7 DISCUSSION AND CONCLUSION

Although there has been a trend to turn to hardware support for security purposes that can be dated back to the early days of computers, we have not seen a systematic study of how such support contributed to improving security and its advantages/disadvantages, which is the goal of this survey. By coarsely categorizing hardware support into execution modes, extensions to the modes and co-processors, and modeling the protection target as execution environments (EEs) with initial state correctness, runtime protection and input/output protection, we examined the following as three research questions: how each category of hardware support achieves secure execution, with firmware EEs and TEEs discussed in detail; how hardware support is applied in various security solutions; and what attack vectors hardware still suffers from, as well as other usability/adoptability issues.

We could see that while hardware possesses obvious advantages, caution should be exercised when designing or applying hardware security features. For example, almost all hardware features rely on firmware at least implicitly, while very often, either firmware memory protection is not treated differently from software, or intrinsic attack vectors like side channels also apply to firmware, rendering the reliant hardware feature vulnerable. Moreover, each type of hardware support (e.g., modes vs. co-processors) has its advantages and disadvantages, and thus, it highly depends on the desired security purpose as to which type to use, usually in combination, as opposed to considering hardware to be one universal security solution. It was also observed that research proposals are a driving factor for commodity hardware features as seen from such proposals paving the way for later introduction of the features. We hope this survey could cast some light on the relationship between securing program execution and introducing hardware support to achieve it, and where improvements can be made, further to the increasing quantity.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*.

[3] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel Virtualization Technology for Directed I/O. *Intel technology journal* 10, 3 (2006), 179–192.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. *IACR Cryptology ePrint Archive* 2018 (2018), 1060.

[5] Alibaba Group. 2022. ECS Bare Metal Instance. https://www.alibabacloud.com/product/ebm [Accessed May 8, 2024].

[6] AMD. 2018. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*.

[7] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13.

[8] Anonymous. 2004. Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates. https://web.archive.org/web/20191103220248/https://securiteam.com/securityreviews/5FP0M1PDFO/ [Accessed May 8, 2024].

[9] Anonymous. 2009. Numerous System Management Mode (SMM) privilege escalation vulnerabilities in ASUS motherboards including Eee PC series. https://dl.packetstormsecurity.net/0908-advisories/smm-escalate.txt [Accessed May 8, 2024].

[10] ARM. 2021. *ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile*.

[11] ARM. 2022. Arm Morello Program. https://www.arm.com/architecture/cpu/morello [Accessed May 8, 2024].

[12] Ahmad Atamli-Reineh, Ravishankar Borgaonkar, Ranjbar A. Balisane, Giuseppe Petracca, and Andrew Martin. 2016. Analysis of Trusted Execution Environment Usage in Samsung KNOX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX '16)*.

[13] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*.

[14] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*.
[15] Ying Bai. 2016. ARM® Memory Protection Unit (MPU).
[16] Andrew Baumann. 2017. Hardware is the New Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*.
[17] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–26.
[18] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, and Mickey Shkatov. 2015. A new class of vulnerabilities in SMI handlers. In *The 15th annual CanSecWest conference*.
[19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. arXiv preprint arXiv:1903.01843.
[20] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*.
[21] C. Bornträger, J. D. Bradbury, R. Bündgen, F. Busaba, L. C. Heller, and V. Mihajlovski. 2020. Secure your cloud workloads with IBM Secure Execution for Linux on IBM z15 and LinuxONE III. *IBM Journal of Research and Development* 64, 5/6 (2020), 2:1–2:11.
[22] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT'17)*.
[23] BSDaemon, coideloko, and D0nand0n. 2008. System Management Mode Hacks: Using SMM for 'Other Purposes'. http://phrack.org/issues/65/7.html [Accessed May 8, 2024].
[24] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. 2019. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*.
[25] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.
[26] Yuriy Bulygin, John Loucaides, Andrew Furtak, Oleksandr Bazhaniuk, and Alexander Matrosov. 2014. Summary of attacks against BIOS and secure boot. In *Defcon*. http://www.c7zero.info/stuff/DEFCON22-BIOSAttacks.pdf [Accessed May 8, 2024].
[27] Yuriy Bulygin and David Samyde. 2008. Chipset based approach to detect virtualization malware. In *BlackHat Briefings USA*. http://me.bios.io/images/2/23/DeepWatch.pdf [Accessed May 8, 2024].
[28] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.
[29] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*.
[30] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. 2013. Bios chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM conference on Computer & communications security*.
[31] G Cabodi, P Camurati, C Loiacono, G Pipitone, F Savarese, and D Vendraminetto. 2015. Formal verification of embedded systems for remote attestation. *WSEAS Transactions on Computers* 14 (2015), 760–769.
[32] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.
[33] Daming D. Chen, Wen Shih Lim, Mohammad Bakhshalipour, Phillip B. Gibbons, James C. Hoe, and Bryan Parno. 2021. HerQules: Securing Programs via Hardware-Enforced Message Queues. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[34] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SGXpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy*.
[35] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*.
[36] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*.
[37] Citrix. 2015. Foundational Security with Intel TXT and Citrix XenServer. https://stg-xyz.sky.citrix.com/content/dam/citrix/en_us/documents/partner-documents/foundational-security-with-intel-txt-and-citrix-xenserver.pdf [Accessed Feb 26, 2024].

[38] David Cooper, William Polk, Andrew Regenscheid, Murugiah Souppaya, et al. 2011. BIOS protection guidelines. *NIST Special Publication* 800 (2011), 147.
[39] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained.* Technical Report. Cryptology ePrint Archive.
[40] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*.
[41] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, Vol. 98.
[42] Deeksha Dangwal, Meghan Cowan, Armin Alaghi, Vincent T. Lee, Brandon Reagen, and Caronline Trippel. 2020. SoK: Opportunities for Software-Hardware-Security Codesign for Next Generation Secure Computing. In *Hardware and Architectural Support for Security and Privacy (HASP'20)*. Article 8, 9 pages.
[43] Ruan de Clercq and Ingrid Verbauwhede. 2017. A survey of hardware-based control flow integrity (CFI). arXiv preprint arXiv:1706.07257.
[44] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. 2018. LiteHAX: lightweight hardware-assisted attestation of program execution. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, Iris Bahar (Ed.).
[45] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*.
[46] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.
[47] Martin G Dixon, David A Koufaty, Camron B Rust, Hermann W Gartler, and Frank Binns. 2014. Steering system management code region accesses. US Patent 8,683,158. Filed in 2005.
[48] Loıc Duflot, Daniel Etiemble, and Olivier Grumelard. 2006. Using CPU system management mode to circumvent operating system security functions. In *CanSecWest*. https://www.ssi.gouv.fr/en/publication/using-cpu-system-management-mode-to-circumvent-operating-system-security-functions/
[49] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. 2009. Getting into the SMRAM: SMM Reloaded. In *CanSecWest*. https://www.ssi.gouv.fr/uploads/IMG/pdf/Cansec_final.pdf
[50] Loıc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. 2010. System management mode design and security issues. IT Defense. https://cyber.gouv.fr/sites/default/files/IMG/pdf/IT_Defense_2010_final.pdf [Accessed May 8, 2024].
[51] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. 2011. What if You Can't Trust Your Network Card?. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*.
[52] Alexander Eichner and Robert Buhren. 2020. All you ever wanted to know about the AMD Platform Security Processor and were afraid to emulate. In *BlackHat*.
[53] Shawn Embleton, Sherri Sparks, and Cliff C Zou. 2010. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks* 6, 12 (2010), 1590–1605.
[54] Alexander Ermolov. 2016. Safeguarding Rootkits: Intel Boot Guard. Zeronights. https://papers.put.as/papers/firmware/2016/Intel_BootGuard_final.pdf [Accessed May 8, 2024].
[55] Mark Ermolov and Maxim Goryachy. 2017. How to hack a turned-off computer, or running unsigned code in Intel management engine. Blackhat Europe 2017.
[56] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. 2014. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*.
[57] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*.
[58] Atanas Filyanov, Jonathan M. McCune, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. Uni-directional trusted path: Transaction confirmation on just one device. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*.
[59] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX:In-process Memory Isolation eXtension. In *27th USENIX Security Symposium (USENIX Security 18)*.
[60] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*.
[61] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. arXiv preprint arXiv:1410.7747.

[62] Sergiu D Ghetie. 2010. Protecting system management mode (SMM) spaces against cache attacks. US Patent 7,698,507. Filed in 2007.
[63] Google Cloud. 2022. Confidential Computing concepts. https://cloud.google.com/compute/confidential-vm/docs/about-cvm [Accessed May 8, 2024].
[64] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*.
[65] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *26th USENIX Security Symposium*.
[66] Trusted Computing Group. 2017. *TCG Glossary*.
[67] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory. In *2015 IEEE Symposium on Security and Privacy*.
[68] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Commun. ACM* 52, 5 (may 2009), 91–98.
[69] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.
[70] John Heasman. 2006. Implementing and detecting a PCI rootkit.
[71] John Heasman. 2006. Implementing and detecting an ACPI BIOS rootkit. In *BlackHat Federal*.
[72] James Hendricks and Leendert van Doorn. 2004. Secure Bootstrap is Not Enough: Shoring Up the Trusted Computing Base. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop (EW 11)*.
[73] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.
[74] Wei Huang, Vasily Rudchenko, He Shuang, Zhen Huang, and David Lie. 2018. Pearl-TEE: supporting untrusted applications in TrustZone. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX 2018)*.
[75] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. 2021. Confidential Computing for OpenPOWER. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*.
[76] Duha Ibdah, Nada Lachtar, Abdulrahman Abu Elkhail, Anys Bacha, and Hafiz Malik. 2020. Dark Firmware: A Systematic Approach to Exploring Application Security Risks in the Presence of Untrusted Firmware. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*.
[77] Intel. 2015. *Intel Xeon Processor E3-1200v4 Product Family Datasheet – Volume 2 of 2*.
[78] Intel. 2015. SMI Transfer Monitor (STM). https://www.intel.com/content/www/us/en/developer/articles/tool/smi-transfer-monitor-stm.html [Accessed May 8, 2024].
[79] Intel. 2017. *Intel Software Guard Extensions Developer Guide*.
[80] Intel. 2017. *Intel Trusted Execution Technology: Software Development Guide*.
[81] Intel. 2018. *Intel 64 and IA-32 Architectures Software Developer's Manual*.
[82] Intel. 2019. *Control-flow enforcement technology preview*.
[83] Intel. 2020. Intel Xeon E3-1200 v3 Processor Family Specification Update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf
[84] Intel. 2021. Product brief: Hardware-enhanced threat detection. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/tdt-product-brief.pdf [Accessed May 8, 2024].
[85] Intel. 2021. XuCode: An Innovative Technology for Implementing Complex Instruction Flows. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html [Accessed May 8, 2024].
[86] Intel and VMware. 2013. Embrace Cloud Computing with Intel and VMware Security Solutions. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/intel-vmware-security-solution-brief.pdf [Accessed Feb 26, 2024].
[87] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2020. Trusted Execution Environments: Properties, Applications, and Challenges. *IEEE Security & Privacy* 18, 2 (2020), 56–60. https://doi.org/10.1109/MSEC.2019.2947124
[88] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*.

[89] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. 2018. Titan: enabling a transparent silicon root of trust for Cloud. In *Hot Chips: A Symposium on High Performance Chips*, Vol. 194.
[90] Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. 2014. *Defeating signed BIOS enforcement.* Technical Report. The MITRE Corporation. https://apps.dtic.mil/sti/citations/trecms/AD1107701
[91] Corey Kallenberg, Sam Cornwell, Xeno Kovah, and John Butterworth. 2014. Setup for failure: defeating secure boot. In *The Symposium on Security for Asia Network (SyScan)(April 2014)*.
[92] Samuel T King and Peter M Chen. 2006. SubVirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*.
[93] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. arXiv preprint arXiv:1801.01203.
[94] Paul C Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'96)*.
[95] Benjamin Kollenda, Philipp Koppe, Marc Fyrbiak, Christian Kison, Christof Paar, and Thorsten Holz. 2018. An Exploratory Analysis of Microcode As a Building Block for System Defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.
[96] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. 2017. Reverse Engineering x86 Processor Microcode. In *26th USENIX Security Symposium*.
[97] Xeno Kovah, John Butterworth, Corey Kallenberg, and Sam Cornwell. 2014. Copernicus 2: SENTER the dragon.
[98] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proceedings of the 2013 ACM Conference on Computer & Communications Security (CCS '13)*.
[99] Michael Larabel. 2018. Intel MPX Support Removed From GCC 9. https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9 [Accessed May 8, 2024].
[100] Donald C Latham. 1986. Department of defense trusted computer system evaluation criteria. Department of Defense.
[101] Hojoon Lee, HyunGon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. 2013. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*.
[102] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium*.
[103] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. Secloak: ARM TrustZone-based mobile peripheral control. In *16th Annual International Conference on Mobile Systems, Applications, and Services*.
[104] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, Gareth Stockwell, Mark Knight, and Charles Garcia-Tobin. 2023. Enabling Realms with the Arm Confidential Compute Architecture.
[105] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices* 35, 11 (2000), 168–177.
[106] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. 2021. PACStack: an Authenticated Call Stack. In *30th USENIX Security Symposium*.
[107] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*.
[108] Yangdi Lyu and Prabhat Mishra. 2018. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security* 2, 1 (01 Mar 2018), 33–50.
[109] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2018. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Comput.* 67, 3 (March 2018), 361–374.
[110] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium*.
[111] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys'08*.
[112] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. 2009. Safe passage for passwords and other sensitive data. In *Proceedings of the Network and Distributed System Security Symposium, 2009*.
[113] Carlo Meijer and Bernard Van Gastel. 2019. Self-encrypting deception: weaknesses in the encryption of solid state drives. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*.
[114] Microsoft.com. 2022. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute [Accessed May 8, 2024].
[115] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. 2018. A Comparison Study of Intel SGX and AMD Memory Encryption Technology. In *7th International Workshop on Hardware and Architectural Support for Security*

and Privacy.
[116] Daniel Moghimi. 2023. Downfall: Exploiting Speculative Data Gathering. In *32nd USENIX Security Symposium*.
[117] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *ACM Conference on Computer and Communications Security*.
[118] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *11th European Workshop on Systems Security, April 23, 2018*.
[119] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*.
[120] Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2020), 1555–1571. https://doi.org/10.1109/TCAD.2019.2915318
[121] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*.
[122] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. In *2nd IEEE International Conference on Collaboration and Internet Computing, CIC 2016, Pittsburgh, PA, USA, November 1-3, 2016*.
[123] Nuvoton.com. 2018. NuMicro M2351 Series – a TrustZone empowered microcontroller series focusing on IoT security. https://www.nuvoton.com/products/microcontrollers/arm-cortex-m23-mcus/m2351-series/ [Accessed May 8, 2024].
[124] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core [Accessed May 8, 2024].
[125] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. 2017. Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement. arXiv preprint arXiv:1705.10295.
[126] University of Cambridge. 2019. Capability Hardware Enhanced RISC Instructions (CHERI). https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/ [Accessed May 8, 2024].
[127] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2 (2018), 28:1–28:30.
[128] Openstack.org. 2021. Trusted compute pools. https://docs.openstack.org/nova/pike/admin/security.html [Accessed Feb 26, 2024].
[129] Ascher Opier. 1967. Fourth generation software, hardware. Datamation. https://archive.org/details/TNM_4th_generation_software_hardware_-_Datamation_20171010_0125
[130] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*.
[131] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium*.
[132] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. 2011. Memoir: Practical state continuity for protected modules. In *2011 IEEE Symposium on Security and Privacy*.
[133] PCMag Staff. 2011. Is Intel Insider Code For Drm In Sandy Bridge? https://www.pcmag.com/archive/is-intel-insider-code-for-drm-in-sandy-bridge-258868 [Accessed May 8, 2024].
[134] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *49th Annual International Symposium on Computer Architecture*.
[135] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages.
[136] Xiaoyu Ruan. 2014. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress Open, New York, NY.
[137] Ethan M. Rudd, Andras Rozsa, Manuel Günther, and Terrance E. Boult. 2017. A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions. *IEEE Communications Surveys Tutorials* 19, 2 (2017), 1145–1172. https://doi.org/10.1109/COMST.2016.2636078
[138] Joanna Rutkowska and Rafał Wojtczuk. 2008. Preventing and detecting Xen hypervisor subversions. In *Blackhat*. https://invisiblethingslab.com/resources/bh08/part2-full.pdf [Accessed May 8, 2024].
[139] Anibal L Sacco and Alfredo A Ortega. 2009. Persistent BIOS infection. In *CanSecWest Applied Security Conference*. https://www.coresecurity.com/sites/default/files/private-files/publications/2016/05/Persistent-BIOS-Infection.pdf [Accessed May 8, 2024].
[140] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[141] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. 2021. Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification. *IEEE Access* 9 (2021), 83067–83079.
[142] Rebecca Shapiro. 2018. *Types for the chain of trust: No (loader) write left behind.* Ph. D. Dissertation. Dartmouth College.
[143] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Anal Network and Distributed System Security Symposium.*
[144] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *Proceedings of the 24st Annual Network and Distributed System Security Symposium, (NDSS).*
[145] AJ Singh and Akshay Bhardwaj. 2014. Android internals and telephony. *Int. J. Emerg. Technol. Adv. Eng* 4 (2014), 51–59.
[146] N Sklavos, K Touliou, and C Efstathiou. 2006. Exploiting cryptographic architectures over hardware vs. software implementations: advantages and trade-offs. *Memory* 13 (2006), 18.
[147] Dmitry Sklyarov. 2017. Intel ME: flash file system explained. In *Black Hat Europe.* https://www.blackhat.com/docs/eu-17/materials/eu-17-Sklyarov-Intel-ME-Flash-File-System-Explained.pdf [Accessed May 8, 2024].
[148] Igor Skochinsky. 2014. Intel ME Secrets. https://papers.put.as/papers/firmware/2014/Recon2014Skochinsky.pdf [Accessed May 8, 2024].
[149] D. Sladović, D. Topolčić, and D. Delija. 2020. Overview of Mac system security and its impact on digital forensics process. In *43rd International Convention on Information, Communication and Electronic Technology.*
[150] Mark Smotherman. 2009. *A brief history of microprogramming.* Technical Report. School Computing, Clemson University Clemson, SC, USA. https://ed-thelen.org/comp-hist/MicroprogrammingABriefHistoryOf.pdf
[151] Raoul Strackx and Frank Piessens. 2016. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security 16).*
[152] Chao Su and Qingkai Zeng. 2021. Survey of CPU cache-based side-channel attacks: systematic analysis, security models, and countermeasures. *Security and Communication Networks* 2021 (June 2021), 5559552.
[153] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing.*
[154] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *45th IEEE/IFIP International Conference on Dependable Systems and Networks.*
[155] Jakub Szefer. 2019. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security* 3, 3 (2019), 219–234.
[156] Chu Jay Tan, Junita Mohamad-Saleh, Khairu Anuar Mohamed Zain, and Zulfiqar Ali Abd. Aziz. 2017. Review on Firmware. In *International Conference on Imaging, Signal Processing and Communication.*
[157] Tencent Blade Team. 2018. Exploring Qualcomm Baseband via ModKit. CanSecWest 2018.
[158] Alexander Tereshkin and Rafal Wojtczuk. 2009. Introducing Ring -3 Rootkits. Invisible Things Lab. https://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf [Accessed May 8, 2024].
[159] UEFI Forum Inc. 2020. *Advanced Configuration and Power Interface (ACPI) Specification.* Technical Report. UEFI Forum Inc. https://uefi.org/sites/default/files/resources/ACPI_Spec_6_3_A_Oct_6_2020.pdf
[160] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19).*
[161] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *ACM SIGSAC Conference on Computer and Communications Security.*
[162] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy.*
[163] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. 2020. RusTEE: developing memory-safe ARM TrustZone applications. In *Annual Computer Security Applications Conference.*
[164] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng, and Yinqian Zhang. 2019. Running language interpreters inside SGX: A lightweight, legacy-compatible script code hardening approach. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security.*
[165] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.*
[166] Jiang Wang, Angelos Stavrou, and Anup Ghosh. 2010. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Recent Advances in Intrusion Detection*, Somesh Jha, Robin Sommer, and Christian Kreibich (Eds.).
[167] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE symposium on security and privacy.*

[168] WikiChip.org. 2022. Innovation Engine (IE) - Intel. https://en.wikichip.org/wiki/intel/innovation_engine [Accessed May 8, 2024].
[169] Richard Wilkins and Brian Richardson. 2013. *UEFI secure boot in modern computer security solutions.* Technical Report. UEFI.org. https://uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf
[170] Ally Winning. 2018. First look at Nordic's "cellular made easy" nRF91 low-power solution. https://www.eenewseurope.com/en/first-look-at-nordics-cellular-made-easy-nrf91-low-power-solution/ [Accessed May 8, 2024].
[171] Rafal Wojtczuk and Corey Kallenberg. 2015. Attacking UEFI Boot Script. https://bromiumlabs.files.wordpress.com/2015/01/venamis_whitepaper.pdf [Accessed May 8, 2024].
[172] Rafal Wojtczuk and Corey Kallenberg. 2015. Attacks on UEFI security. In *Proc. 15th Annu. CanSecWest Conf.(CanSecWest).* https://repo.zenk-security.com/Techniques%20d.attaques%20%20.%20%20Failles/Attacks-on-UEFI-security.pdf [Accessed May 8, 2024].
[173] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM Memory via Intel CPU Cache Poisoning. https://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf [Accessed May 8, 2024].
[174] Rafal Wojtczuk and Alexander Tereshkin. 2009. Attacking Intel BIOS. In *BlackHat USA.* https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf [Accessed May 8, 2024].
[175] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA).*
[176] Yubin Xia, Yutao Liu, and Haibo Chen. 2013. Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks. In *19th IEEE International Symposium on High Performance Computer Architecture.*
[177] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012).*
[178] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *25th USENIX Security Symposium.*
[179] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy.*
[180] Jiewen Yao and Vincent J Zimmer. 2015. *White Paper A Tour Beyond BIOS Launching a STM to Monitor SMM in EFI Developer Kit II.* Technical Report. Intel Corporation.
[181] Jiewen Yao, Vincent J Zimmer, and Star Zeng. 2014. *White Paper A Tour Beyond BIOS Implementing S3 Resume with EDKII.* Technical Report. Intel Corporation.
[182] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14).*
[183] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *2014 IEEE 25th International Symposium on Software Reliability Engineering.*
[184] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. 2013. Implementation and Implications of a Stealth Hard-drive Backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC '13).*
[185] Denghui Zhang, Guosai Wang, Wei Xu, and Kevin Gao. 2019. SGXPy: Protecting Integrity of Python Applications with Intel SGX. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC).*
[186] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *23rd ACM Symposium on Operating Systems Principles.*
[187] Fengwei Zhang, Kevin Leach, Haining Wang, and Angelos Stavrou. 2015. Trustlogin: Securing password-login on commodity operating systems. In *10th ACM Symposium on Information, Computer and Communications Security.*
[188] Fengwei Zhang, Haining Wang, Kevin Leach, and Angelos Stavrou. 2014. A framework to secure peripherals at runtime. In *European Symposium on Research in Computer Security.*
[189] Fengwei Zhang and Hongwei Zhang. 2016. SoK: A study of using hardware-assisted isolated execution environments for security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016.*
[190] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2016. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive* 2016 (2016), 980.
[191] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. 2018. TruSense: Information Leakage from TrustZone. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA.*
[192] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. SHELTER: Extending Arm CCA with Isolation in User Space. In *32nd USENIX Security Symposium.*
[193] Lei Zhou, Jidong Xiao, Kevin Leach, Westley Weimer, Fengwei Zhang, and Guojun Wang. 2019. Nighthawk: Transparent System Introspection from Ring-3. In *European Symposium on Research in Computer Security.*