Is Hardware More Secure than Software?

Lianying Zhao
Carleton University

David Lie
University of Toronto

***Abstract*—Computer hardware is usually perceived as more secure than software. However, recent trends lead us to reexamine this belief. We draw attention to the "firmwarization" of hardware and argue for revisiting the role of hardware and software in systems security.**

TRADITIONALLY, computer systems can be thought of as composed of two types of components: hardware and software. Hardware refers to the physical components that perform a fixed set of operations. Software on the other hand, defines logical components, instantiated as data and instructions, which can specify arbitrary sequences of hardware operations, as well as inputs to those operations. A function can be implemented using a various mixes of hardware and software components (or even entirely in hardware). The mix of hardware and software can have implications on certain properties of the function implementation (e.g., performance or security).

Hardware Security. Hardware has been deemed to be the safeguard for security-sensitive operations for decades. This is reflected in the longstanding use of hardware security modules (HSMs, such as the tamper-evident/resistant IBM Cryptocards) for cryptoprocessing, as well as the more recent interest in securing sensitive tasks with trusted computing technologies like Intel SGX and ARM TrustZone.

In academia, hardware security has also attracted significant attention from researchers. A quick search on Google Scholar returns at least 1,750 academic papers with the title containing both "hardware" and "security" as of the time of writing. While this is a conservative underapproximation of the actual number of such papers (as it only checks the title), and regardless of whether these papers propose a larger role of hardware in security, or seek to examine the vulnerability of hardware to security attacks, the sheer number informally illustrates that the security community has a strong interest in the relationship between computer hardware and security. Another recent trend is the implementation of hardware mechanisms to address well-known security problems. For instance, a survey [8] lists 21 hardware-based architectures to ensure control flow integrity (CFI, the correct flow of execution of a program). This implies that there is an underlying belief that hardware implementations correlate well with better security properties in computing computing systems.

The Perceived Security Benefits of Hardware

We begin by examining where this belief in the security benefits of hardware may have originated. Intuitively, one may assume that the feeling of security may naturally stem from the possession and physical existence of hardware.

The final version of this article will appear in the special issue Hardware-Assisted Security of IEEE Security & Privacy. This is the author's copy created on June 27th, 2020. DOI: 10.1109/MSEC.2020.2994827

Hardware switches provide visible assurance of operation integrity, e.g., a state is set and its state remains unambiguously obvious. Similarly, users can be confident that they possess and are in control of information and capabilities inherent in detachable devices like USB dongles and drives. Hardware components on first examination, appear simpler and more restricted in function than many software components. Components, like hard drives store data and little else, while keyboards provide a means to input information and commands. Indeed, the physical nature of hardware means that defects in hardware are not as easily fixed after they are shipped to customers, driving hardware designers to be more conservative and thoughtful in their designs. All these properties contribute to an overall feeling of better security and assurance in hardware over software.

However, if we set aside assumptions about the conscientiousness of hardware and software designers, as well as possible mental biases towards physical versus logical components, and instead focus on the functional properties that distinguish hardware and software implementations from a security perspective, we can contemplate two that appear to stand out: 1) **immutability** and 2) **privilege**.

We define immutability as the ability of a function's properties to resist being changed from those in their original intended design. Being physically implemented in circuits and transistors, hardware has an "intrinsic" immutability, to it—the functions implemented in hardware cannot be simply changed by altering some bits in memory. If an attacker wants to modify the function of hardware, they must physically change it, implying the need for physical access to the victim. This is comforting, as physical changes have a higher chance of being tamper-evident.

Such immutability can also be considered a benefit of the fact that hardware does not need to implement a "Turing machine". Hardware has an inherent bias towards the "Economy of Mechanism" [10]—because hardware has a physical existence, there is a penalty for extra or unused functionality, so there is a strong incentive to avoid it. Therefore, hardware logic often implements just enough functionality to execute prescribed tasks without enabling arbitrary operations. This naturally limits the damage that can be done if hardware is compromised or defective—a hardware switch in a circuit may be maliciously changed from off to on, but not reprogrammed to be something else substantially different. By contrast, software generally runs on a Turing-complete execution engine (e.g., the CPU), which can be programmed to do something far from the intended functionality of the original software. So if compromised or defective, software may intrinsically give an attacker access to the Turing-complete environment where arbitrary functions can be performed.

We define privilege as the ability to observe and control the operations of another component. This definition has a subtle difference from the software definition of privilege (i.e. kernel vs user) as higher-privilege software generally is capable of a super-set of operations over lower-privilege software. However, the privilege of hardware over software is different in that hardware and software are not capable of the same set of operations. Yet, where it is comparable is that the additional operations that privileged software has usually include the ability to control (i.e. start, stop and interrupt), as well as observe (read and write) the execution state of unprivileged software. In the same way, hardware making up the execution engine of software, has the same ability to control and observe the state of any software it runs.

Modern computers all share the layered construction starting from the hardware/firmware, VM hypervisor, operating system (OS), to applications (see Figure 1). A hypervisor is secure against the OS kernel, which in turn is secure against applications. Hardware sits at the most privileged position in this stack, and as such is naturally secured against vulnerabilities in and attacks coming from the lower-privileged software layers.

Hardware, Firmware, and Software

So far we have discussed a high-level abstract model of computing system implementation where all components are all either entirely hardware or software. However, as the reader may have guessed, the view of computing systems promoted by this article is not so simple. There actually, exists a third, hybrid, type of implementation for computational functions called "firmware".

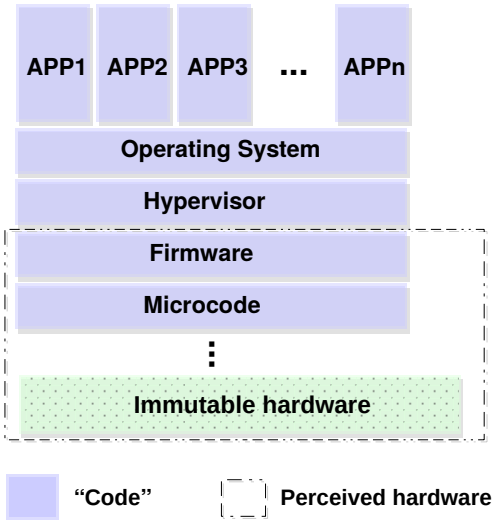

Figure 1. The layered construction of a computing device, and what is commonly considered as hardware.

The term firmware has existed almost as long as computer hardware has, first coined by Ascher Opler [4] in the 1960s. Firmware shares some properties with software, in that it is implemented as software instructions that execute on a general purpose, Turing-complete hardware processor.

To better understand hardware, firmware and software, we compare the three using 4 properties: immutability, privilege, efficiency and cost, summarized in Table 1.

Immutability: As defined earlier, immutability characterizes whether the functionality of a component can be altered. While the immutability of hardware is intrinsic, the immutability of firmware and software is not, and typically must be enforced by some other component (either hardware or software). For example, the address space of firmware typically is blocked from access by applications, and in many cases, even the OS by either a hardware reference monitor, or some combination of hardware and software reference monitors. From this, we argue that only hardware is truly immutable, while firmware and software can be either intentionally mutable by certain components, or mutable as a result of design or implementation flaws.

We exclude supply chain attacks in our discussion, because when products are compromised before acquired by the user, immutability no longer applies. Even in that case, the compromised hardware remains immutable, e.g., hardware Trojans inserted by the designer or manufacturer usually cannot be removed by firmware/software.

Privilege: Also previously defined, privilege indicates the ability of a higher-privileged component to observe and control the execution of a lower-privileged component. Because all software must execute on hardware, hardware inherently has higher-privilege than firmware or software. However, it is often the case that firmware exists on the hardware side of the hardware-software interface, and as such has privileges over software such as the OS kernel or applications. Thus, we order privileges with hardware at the top, firmware in the middle, and, finally, software at the bottom.

Efficiency: Efficiency defines how much performance an implementation provides as a function of some resource (electrical power for example). Hardware implementations are generally more power efficient than software, and provide better performance for similar power consumption. While both firmware and software are less efficient than pure hardware blocks, firmware often executes closer to the hardware, and may have access to special hardware functions or facilities that may make it more efficient than software. For example, CPU firmware (i.e. microcode) is not affected by context switches or OS scheduling like regular application software.

Cost: Hardware in general has a fixed cost with each system, while the incremental cost of deploying software for each system is nearly zero. As a result, systems that contain more specialized hardware generally cost more to produce, which translates into a higher cost for the end user to purchase. This creates an intuitive trade-off that nearly every computer engineer is aware of—faster systems that use hardware accelerators cost more.

Property	Hardware	Firmware	Software
Immutability	✓	✗	✗
Privilege	Highest	Middle	Low
Efficiency	High	Middle	Low
Cost	High	Low	Low

Table 1. Comparison of properties.

The Role of Firmware

Given the comparison in Table 1, one may wonder why firmware was introduced in the first place, as it seems to be superior to neither hardware nor software in any property. One strong reason for firmware, among others, is to maintain interoperability across a variety of computer systems interfaces.

In short, interoperability means that different implementations of components can work with each other because the components all adhere to a standard interface. However, implementing that interface using varying mixes of hardware and software allows manufacturers to create different products at different points in the cost-efficiency spectrum. For example, all network cards perform essentially the same function, but more expensive faster network cards may contain a variety of accelerators and off-load engines that allow faster packet processing with lower-power, and at the same time freeing up general purpose compute cycles on the main CPU. A similar example is specialized cryptographic function accelerators. In some cases, functions that are not performance-sensitive can always be implemented in firmware. As these variants do not affect the interface, they can be used interchangeably without changing software or other hardware components.

Yet another important advantage that firmware provides is the ability to update or patch hardware either to provide enhanced features or address defects. A variety of consumer devices from webcams, to thermostats, to home routers have the majority of their high-level functionality implemented in firmware so that they can be updated in the field and bring more value to the end consumer. In fact, a key selling point for some smartphone models is the length of time that the vendor will continue to patch security vulnerabilities in the smartphone firmware.

Hardware firmwarization

A key pillar of the security of hardware is its immutability to software, which holds most naturally when hardware is *actually implemented as hardware*. However, in recent years, what is traditionally thought of as hardware—the CPU, hard drives and various other peripherals, are increasingly taking on implementation characteristics normally associated with software. As hardware becomes increasingly complex, many of these complex features are actually implemented as firmware.

Baumann has stated that the hardware is the new software [7], emphasizing the rapid and iterative manner in which modern hardware features are being implemented. Many modern hardware "features" are actually implemented as firmware, and they are often rolled out iteratively, i.e., SGX, then SGX2. Rather than actually replacing the components, defects in hardware can be patched by changing the firmware. These characteristics increasingly give hardware both the advantages (i.e. flexibility, upgradeability) and disadvantages (complexity, mutability) of software.

It is not uncommon to consider hardware and firmware as a whole to be just hardware, as illustrated in Figure 1. For instance, a blanket statement such as "hardware-encrypted drives are more secure than software full-disk encryption" does not fully acknowledge nuances such as the fact that hardware encryption implementations often contain software (firmware) components. What is worse, devices traditionally perceived as "dumb" (i.e., not Turing-complete) when enriched with firmware (i.e., introducing a general-purpose processor) become Turing-complete, and if compromised, can perform functions never intended in the original design. One example is keyboards, as exemplified by an Apple keyboard turned into a keystroke logger [5] using merely software tools.

How Firmwarized Hardware can Go Wrong

When hardware functions are implemented using firmware, the immutability of hardware is impacted, weakening the security against attacks. Indeed, in this section, we will examine some hardware security failures from the past and see how a common thread has emerged where attackers unexpectedly modify functionalities implemented in firmware. We will also show how firmwarization and its security consequences are prevalent across different computer components. In our discussion we focus on two aspects: the vulnerability itself, which leads to firmware compromise for arbitrary code execution, and the harm that the compromise causes. We include

vulnerabilities that may reside in 1) the firmware code itself, e.g., buffer overflow, or 2) the environment where the firmware runs, which improperly enforces firmware privilege and enables lower-privilege software to compromise firmware. We group vulnerabilities by the type of component it affects, as shown in Figure 2, which illustrates how firmware vulnerabilities have existed in a plethora of system components, as well as the broad range of time over which those vulnerabilities have been found.

Input/output Devices

Computer peripherals, such as drives, keyboards, mice and printers are usually seen as too simple to be compromised and used for attack. However, these devices have been increasingly offering more and more functionality, often implemented in firmware, which has led to vulnerabilities.

Hard drives. Hard drives and solid-state drives (SSDs), normally viewed as simple block devices, actually contain quite a bit of firmware. A recent analysis of the (in)security of todays Self-encrypting SSDs [6] discovered a software-only firmware reflashing attack on Crucial MX100/MX200 using some undocumented vendor-specific commands (VSCs). This vulnerability could allow an attacker to remotely[1] and stealthily intercept any data to/from the disk without leaving traces on the host. Unlike regular malware, the malicious code resides in firmware, so even a full operating system re-installation would not remove such malware. In 2013, Zaddach et al. [3] already demonstrated that a stealth backdoor could be installed by software on an off-the-shelf mechanical hard drive. Jeroen Domburg (a.k.a. sprite_tm)" also found the same vulnerability independently in the same year.

NICs. Aside from hard disks, network interface cards (NICs) are another front for firmware compromise. Back in 2006/2007, in the so-called Project Maux Mk.I, Triulzi demonstrated a partial proof-of-concept takeover of a Broadcom NIC by modifying the firmware. In 2010, Duflot and Perez then expanded this attack to take full control of a computer by exploiting a vulnerability in a Broadcom NetXtreme NIC. The vulnerability was located in the firmware handling the ASF protocol, an obscure protocol intended for remote administration.

Video cards. Video cards also contain firmware in a component called the Video BIOS (VBIOS) which is loaded and executed on the CPU much the same way that the regular system BIOS is. There have been attempts to tweak the VBIOS such as nvresolution [12] which added support for better resolutions with framebuffer drivers.

Video cards can be co-opted to execute malicious code that can evade traditional malware detection. The Graphics Processing Unit (GPU) and video memory (VRAM) provide a full execution environment for malware, and the shared memory with the CPU can allow for attacks. For example, in 2013 researchers implemented a GPU-based keylogger that peaked into the keyboard buffer that is normally managed by the CPU.

Other devices. A variety of devices interface with systems over platform buses such as PCI and USB. Devices for these buses implement generic, but complex functionality, such as automatic device discovery and configuration and arbitration between different devices. This functionality, also usually implemented in firmware, has also been the source of a number of vulnerabilities. For example, in 2006, Heasman explored the special role of PCI expansion ROM (e.g., the VBIOS), a piece of program as part of the device firmware that can be loaded by the system OS during initialization. Heasman discussed [13] how a PCI card's expansion ROM can be reflashed by malicious software, due to lack of firmware signature verification. The card can then be used to mount various pre-boot attacks including subverting the OS kernel when plugged into another computer.

Similarly, the plethora of USB devices exists partially because the USB interface offers rich and flexible functionality. This is a double-edge sword as almost any type of devices can be emulated from a USB device, as opposed to only storage in the case of SATA (before the OS is compromised). Nohl et al. demonstrated the concept of BadUSB [14], where USB devices can be reprogrammed to attack the host they are plugged into. As an example, they were able

[1] We use remotely here to mean that they do not need physical access, but may still need to perform a remote privilege escalation on the local OS first.

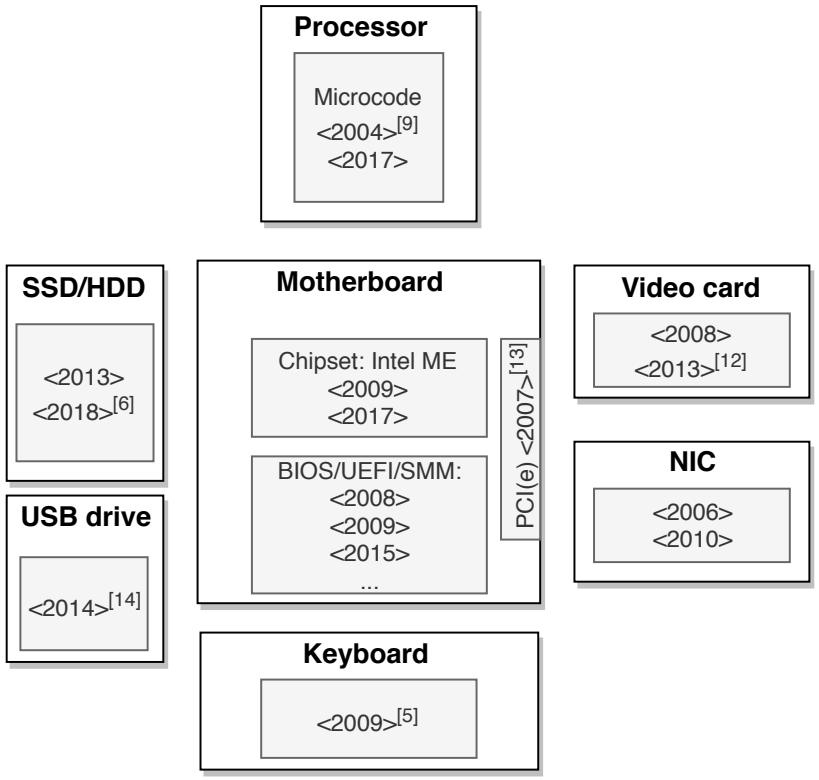

Figure 2. Distribution of firmware across computer components. Grey area designates presence of firmware and years indicate when a software-only compromise was reported.

to reprogram a USB device to emulate another one, such as a keyboard, which when plugged-in could send malicious keystrokes to the victim machine. These vulnerabilities stem from the lack of signature verification for USB firmware. This means that malicious software on one machine can use USB devices as a vector for infecting other machines. For example, malware on one machine can corrupt the firmware of a webcam or USB drive, which can then compromise the next machine the device is plugged into. The lack of firmware verification is widespread—Nohl et al. find that out of 52 chip families and 33 actual devices only one chip family implemented any form of defense.

These problems are further complicated by the fact that users in general pay less attention to patching and securing their peripherals than traditional software systems such as the main system OS and services. While most mainstream OS and application software offer automatic updates that keep these systems patched against known vulnerabilities, firmware updates for peripherals such as hard drives and Bluetooth/Wifi adapters are both less frequent and user-driven, and the overriding advice for such updates is to apply only when there is an apparent need to, resulting in a lower application rate of firmware updates.

CPU and Chipset

Chipset firmware. Excluding peripherals, older PCs could be thought of as a CPU surrounded by non-programmable parts on the motherboard. The only vulnerable software running on such a system would be the code running on the CPU(s). However, this simplistic view of a computer system no longer holds. For example, the Intel Management Engine (ME) is a stand-alone "computer" with its own OS, implemented in the chipset of client PCs. The Intel ME powers the Intel Active Management Technology (AMT) suite of tools. When security vulnerabilities in AMT and the ME came to light in 2017 (CVE-2017-5689), it caused much alarm in the security community, as it simultaneously became known that both the ME and its associated security

weaknesses might have been present since as early as 2008.

Part of Intel AMT is an application (trustlet) running in the ME environment, which enables remote management of computers. This highly-sensitive functionality was implemented in firmware, which could be overwritten and subverted. An attacker who does this could gain control over a system remotely, regardless of what software safeguards are in place in the operating system. Moreover, once compromised, being in a hardware-like position in the system, the Intel ME enjoyed unchecked privileges over the rest of the system, making it difficult to be sure that any malicious code is removed. Security researchers have been actively studying the Intel ME since 2009. Tereshkin and Wojtczuk coined the term *ring -3 rootkits* referring to the code they injected to the ME with a privilege higher than any software and other firmware of the computer. The injection attack exploited a weakness in the memory reclaiming mechanism in Intel CPUs that allowed an improper memory remapping that should have been disallowed. The memory reclaiming that enabled the attack was intended to allow system software to remap DRAM that collides with physical address ranges mapped to I/O devices. The original design of the memory reclaiming mechanism neglected to take into account proper access control, such as checking for memory used by the Intel ME.

Although Intel later fixed this remapping issue and encrypted the ME memory in an attempt to prevent further compromise of the ME isolation, further vulnerabilities in the ME firmware kernel (e.g., CVE-2017-5705,6,7) still lead to firmware compromise that allowed arbitrary code execution. Similar to the Intel ME, the Intel IE (Innovation Engine) was introduced to the Lewisburg chipset in 2017. Unlike ME that is restricted to only run firmware deployed by Intel, IE is intended for OEM uses, i.e., potentially opening up the door for more chipset firmware vulnerabilities.

For brevity, we do not discuss other types of chipset firmware, such as the Baseboard Management Controller (BMC, similar to the Intel ME, part of server motherboards for decades). What these numerous examples demonstrate is that chipsets, far from being simple non-programmable parts on a motherboard, contain complex functionality and are a host of mysterious firmware that most users only have the faintest awareness of.

Host firmware. We refer to firmware that runs on the CPU (as opposed to other chips) as host firmware. Compared to chipset firmware, the host firmware, known as the BIOS/UEFI, is also more than just something that shows the bootsplash screen and initializes computer hardware. Even after system boot, an essential part of it still continues to run in the System Management Mode (SMM), which runs custom and highly privileged software that provides critical low-level system functions, such as power management and system hardware control. Because SMM code runs with privileges higher than even the OS, it is a perennial favorite for attacks. On early motherboards (pre-2006), SMM code could be simply corrupted by malicious kernel code because the BIOS failed to hide the SMRAM (where SMM code executes) from regular/system software. In the SMRAM Control Register (SMRAMC), the `D_OPEN` bit controls whether the SMRAM is visible, and the `D_LOCK` bit locks up the whole SMRAMC and `D_OPEN` until the next reboot. `D_LOCK` was not set by some motherboards.

While this was easily fixed on subsequent motherboards, it was not the end of weaknesses in the SMM firmware. SMM exploits have been continually identified since 2008. Later compromises consisted of failures to properly secure SMM from various other seemingly unrelated mechanisms. For example, the memory reclaiming flaw used to compromise the Intel ME was also exploited to attack SMM. This allowed a malicious operating system to once again gain access to SMRAM by using the reclaiming mechanism to remap the SMRAM region into an accessible region of memory.

About a year later, another SMM compromise was found via a cache poisoning attack. Normally, system software can configure the Memory-Type Range Registers (MTRRs) to control which memory regions can be cached and how. But SMRAM did not get treated differently. Then the attacker could just modify the “spilled” SMM code (copied to cache) without accessing SMRAM directly, which would get executed in

SMM the next time and manipulate the actual copy in SMRAM. To fix this issue, a new register System Management Range Register (SMRR) was introduced, accessible only from SMM, that can configure SMRAM caching properties.

Unfortunately, the war in SMM did not end there. Later in 2009, researchers found that improperly written SMM code could invoke functions outside SMRAM, leading to potential arbitrary code execution in SMM. To prevent such callouts, another new control was added, i.e., the `SMM_Code_Chk_En` bit in the `MSR_SMM_FEATURE_CONTROL` register, so that whether non-SMRAM code can run in SMM can be configured by the initial SMM code. This merely bought a few years of peace until 2015, when yet another SMM vulnerability was discovered, this time because it needed to take external arguments. If the pointer passed in is used without checking, SMM code can be tricked into writing to its own SMRAM (pointed to by the pointer) in a way that facilitates the attack.

All in all, these SMM issues have driven Intel to create a high-level solution, the SMM-Transfer Monitor (STM), first available in 2015. Rather than remove all vulnerabilities in SMM, STM seeks to reduce opportunities for privilege escalation via SMM by reducing the privilege of SMM code by wrapping it with a light-weight monitor. By enforcing necessary checks in the STM monitor, SMM code can be sanitized and behave as expected. However, STM has been far from usable as multiple parties are involved, e.g., who will write best-practice/universal monitor code that can verify all model-specific SMM code (there are a lot of vendors/models), and how to coordinate the OS developer, BIOS developer and hardware vendor, as STM checks would only proceed when all components are in agreement.

After the various complications of firmware have been discussed, motherboard technologies like Secure Boot (by BIOS/UEFI) and Intel Boot Guard (by the Intel ME) may sound less trustworthy as a result of the many firmware vulnerabilities found in these components.

The CPU. Last but not least, if we look further into the CPU, it has never been a purely hardware execution engine. The microcode, which can be thought of as another layer of hardware-level instructions, is used to implement more and more complex operations to support new hardware features. An anonymous report [9] reverse-engineered the microcode update mechanism. It highlighted that even if only vendor-verified updates are allowed, an attacker in control of this process can still choose to patch the microcode in a way that facilitates his attack. What is fortunate is that no recent CPUs are known to be vulnerable to such attacks [1].

Analysis of Firmware Vulnerabilities

Examination of the reported attacks shows that the danger of hardware firmwarization can be analyzed in two aspects: static and dynamic. The former concerns the persistent storage of firmware and the latter concerns the runtime security of firmware in execution.

As with software, one of the key advantages of firmware is that it is updatable. However, like software, it also needs to run in volatile memory, and thus needs isolation from other software. Such factors may be neglected when firmware is considered to be part of hardware.

Static: Update Mechanisms

As long as the firmware exposes an update interface that is accessible by lower-privileged components, including the system OS, the potential for corruption of that firmware exists. While many such interfaces only allow firmware updates whose integrity has been cryptographically verified, there are various ways that verification can go awry, leading to compromise. These ways fall in 1) Undocumented/unscrutinized interface, such as the VSCs of SSDs. This might be residual from factory testing or intended for easier maintenance. 2) Improperly-protected persistent storage. One example is the SPI flash chip on the motherboard where numerous types of firmware (ME, BIOS/UEFI, SMM, certain SGX secrets, etc.) are stored. For example, while many motherboards only trust SMM with SPI write access [2], previous sections have shown that SMM itself is far from secure. 3) Defective verification. This is mainly caused by bugs in the verification logic. Microcode attacks to the old CPUs [1] belong to this category.

Dynamic: Memory Corruption

Since firmware itself is really just software, it inherits vulnerabilities that can be common to all types of software. Memory corruption refers to a wide range of attacks where program defects may allow an attacker to alter the memory of a program in a way not intended by the original programmer. It happens because certain firmware needs to be loaded into volatile memory for execution, even if it is stored in an immutable location. Memory safety remains an open research problem for regular software, let alone the less formalized firmware development. The latest Intel ME vulnerabilities were a typical example: multiple buffer overflows in the kernel of the ME firmware.

Dynamic: Shared Address Space

In addition, firmware may not have physical isolation, but only logical isolation from regular software running on the same system, which may be prone to flaws and vulnerabilities. This is in contrast with non-Turing-complete hardware that does not share resources with software or is physically isolated.

- Insufficient separation. Firmware memory is often visible or can be configured to be visible to regular software for convenience, performance or other reasons. For example, shared cache memories, memory-mapped I/O and memory reclaiming can all cause unintended access. The past exploits of both SMM and ME were related to such insufficient separation, e.g., the ME processor needs to "steal" a region from the main memory due to its own limited SRAM.
- DMA (Direct Memory Access). This applies to all high-bandwidth device firmware. To reduce the main CPU's intervention as a performance bottleneck, the CPU's memory controller allows devices and the system software to set up memory ranges that can be accessed by both the device processor and the CPU. Then, the device processor can move data autonomously. Enabled by its controllers, the DMA opens a hole in the already-separate spaces, i.e., exposing memory to USB/SATA storage/network devices.

The Closed Nature of Firmware

When firmwarization already creates some concerns, what worsens the situation is that the design and implementation of most firmware remains proprietary and largely undocumented. The security community mainly relies on reverse-engineering to unveil (partially) the details. Such opaqueness hides potential problems from the public while real attacks do not necessarily depend on public information. It has been well documented that security-through-obscurity does not work, and rather, can lead to serious vulnerabilities going unnoticed as security researchers/professionals must spend additional effort to get basic information before they can discover and disclose vulnerabilities.

Lessons-learned and the Way Forward

All the aforementioned firmware attacks show that compromise no longer needs physical access, i.e., the desired hardware property of modifications requiring physical access is void. This completely contradicts the common perception of hardware immutability.

Moreover, the security of software depends on the underlying hardware. If the threat model of a computer system assumes that hardware is to be trusted, then compromised hardware undermines all security guarantees. Therefore, firmwarized hardware can cause both the hardware and the software to fail. We examine some possible approaches to better securing firmwarized hardware.

Can Attacks Be Mitigated by Avoiding Firmwarization?

A large number of firmware attacks target "reflashing" the persistent storage and thus are static, affecting the immutability. If firmarization did not exist, there would be no update mechanisms, so such attacks would no longer be possible. Other attacks target the runtime protection of firmware and are thus dynamic. Similarly, if there were no firmarization, such attacks would fail, as hardware does not need a shared address space, nor is it susceptible to memory corruption, and thus this attack vector would also be removed.

It would then appear that an easy security solution to firmwarization is to do away with it. However, as explained earlier, there are many non-security reasons to use firmware. Firmware

allows the possibility of different implementations that trade-off efficiency and cost, a feature that is very important for a broad and healthy computing industry. In addition, having hardware be partially field-updatable has many important benefits that cannot be ignored.

Finally, it should be noted that hardware with no (updatable) firmware can still contain bugs. Indeed, one of the variants of the Spectre vulnerability cannot be patched via a firmware (i.e. microcode) update as it is embedded directly in the hardware logic. Thus, being able to patch the hardware when a vulnerability is discovered is also a benefit of firmware, from a security perspective, as opposed to leaving doors open for attackers with completely immutable hardware. For these reasons, it would seem that the benefits of firmwarization may outweigh its supposed threats to security, and thus we should instead search for methods to make firmwarized hardware as secure as pure hardware implementations.

Then, should firmware be treated the same way as software?

As firmwarization is inevitable, it is important to admit that the vast quantities of code fulfilling hardware functions deserve at least the same (or more) degree of attention as regular software. Yet, these highly trusted, low-level software components are often opaque in their functionality. Their proprietary nature prevents open auditing and validation of their security. Their low-level privileges often encourage complex and unrelated functionality to be packed into their implementation.

Experience has shown time and again that the notion of hardware-assisted security approaches may end up relying on firmwarized hardware, in one way or another. Therefore, a change of mindset is necessary for hardware vendors and the research community: we must treat firmware the same way as software.

Firmware security engineering. Security design principles widely accepted for software, such as Saltzer and Schroeder's design principles [10] can also be applied to firmware, in particular:

Openness. Public scrutiny helps minimize straightforward issues caused by opaqueness. Some open frameworks can serve as references for secure designs of firmware, e.g., coreboot and OpenWrt.

Least privilege. When vulnerabilities do appear, reducing the damage they can cause is as important as preventing them. Firmware's privileges may be managed by a lower minimal layer for fine-grained access control.

To address the insufficient separation issue, the least privilege approach can be taken to further consider an improved firmware model that abstracts away more from software, instead of tolerating sharing with access control.

Economy of mechanism. As with regular software, complexity is a major contributor to vulnerabilities. Traditionally, hardware being difficult to patch, has had economy of mechanism "baked in" by necessity. With increasing use of firmware, this traditional bias towards simplicity and conservative design has been eroded. We advocate a return to those original design principles due to the highly privileged and trusted nature of hardware.

Related work in academia. Fortunately, some of the discussed issues have already attracted attention in the research community and led to progress in firmware security research. Zhang et al. proposes IOCheck[11] to verify the firmware integrity of various peripherals by directly reading peripheral memory. One major concern is that IOCheck relies on SMM as the trust anchor to perform checks. If SMM is no more secure than other firmware, then tools depending on it cannot be relied upon. Also, the heterogeneity of peripherals can make the firmware retrieval very difficult, not to mention maintaining all the correct checksum/signatures. At least, IOCheck sheds some light on the assurance of overall firmware integrity, not specific to a device type.

To address a subset of the BadUSB problem, USBCheckIn [15] uses human user's physical interaction with HID devices (e.g., keyboards and mice) to detect misbehavior. This proposal points to a new direction of firmware integrity: checking the behavior of a device against a set of specifications before trusting it.

Through this article, we hope to waken the community's awareness of the phenomenon that an increasingly greater part of modern hardware is implemented in "software". In light of this, the phrase "software is eating the world" seems to apply to more than just startup companies. We

worry that this trend has implications for the security of our future computer systems, prompting us to encourage the reader to consider: *Is the next hardware-assisted security feature any better than a software-assisted security mechanism?*

REFERENCES

1. P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar and T. Holz, "Reverse Engineering x86 Processor Microcode," In *USENIX Security'17*, Vancouver, BC, Canada, 1163–1180, 2017. (Conference proceedings)
2. C. Kallenberg, J. Butterworth, X. Kovah and S. Cornwell, "Defeating signed BIOS enforcement," The MITRE Corporation, 2014. Available: https://www.mitre.org/sites/default/files/publications/defeating-signed-bios-enforcement.pdf (Tech. Rep.)
3. J. Zaddach, A. Kurmus, D. Balzarotti, E-O. Blass, A. Francillon, T. Goodspeed, M. Gupta and I. Koltsidas , "Implementation and Implications of a Stealth Hard-Drive Backdoor," In *ACSAC'13*, New Orleans, LA, USA, 2013. (Conference proceedings)
4. A. Opler, "Fourth generation software," *Datamation.*, 13(1), 22-24, Jan. 1967. (Journal)
5. K. Chen, "Reversing and exploiting an Apple firmware update," Black Hat 2009. Available: https://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf (Tech. Rep.)
6. C. Meijer and B. van Gastel, "Self-encrypting deception: weaknesses in the encryption of solid state drives (SSDs)," Radboud University and Open University of the Netherlands, 2018. Available: https://www.ru.nl/publish/pages/909282/draft-paper.pdf (Tech. Rep.)
7. A. Baumann, "Hardware is the new software," *HotOS'17*, Whistler, BC, Canada, pp. 132-137, ACM, 2017. (Conference proceedings)
8. R. de Clercq and I. Verbauwhede, "A survey of Hardware-based Control Flow Integrity (CFI)," KU Leuven, 2017. Available: https://arxiv.org/ftp/arxiv/papers/1706/1706.07257.pdf (Preprint)
9. Anonymous, "Opteron exposed: Reverse engineering AMD K8 microcode updates," 2004 [Online]. Available: https://securiteam.com/securityreviews/5FP0M1PDFO/ (URL)
10. J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, 63(9), 1278-1308, 1975. (Conference proceedings)
11. F. Zhang, H. Wang, K. Leach and A. Stavrou, "A framework to secure peripherals at runtime," *ESORICS'14*, Wroclaw, Poland, pp. 219-238, 2014. (Conference proceedings)
12. Tom Li, NVIDIA Video BIOS Hack, 2014 [Online]. Available: https://notabug.org/niconiconi/nvresolution (URL)
13. J. Heasman, "Implementing and detecting a PCI rootkit," Black Hat 2007. Available: https://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf (Tech. Rep.)
14. K. Nohl, S. Krißler and J. Lell, "BadUSB – On Accessories that Turn Evil," *PacSec'14*, Tokyo, Japan, 2014, Available: https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf (Conference proceedings)
15. F. Griscioli, M. Pizzonia and M. Sacchetti, "USBCheckIn: Preventing BadUSB attacks by forcing human-device interaction," *PST'16*, Auckland, New Zealand, 2016, pp. 493-496. (Conference proceedings)