Intel x86 considered harmful

Joanna Rutkowska

October 2015

Version: 1.0

Contents

Chapter 1

Introduction

Present-day computer and network security starts with the assumption that there is a domain that we can trust. For example: if we encrypt data for transport over the internet, we generally assume the computer that's doing the encrypting is not compromised and that there's some other "endpoint" at which it can be safely decrypted.

To trust what a program is doing assumes not only trust in that program itself, but also in the underlying operating system. The program's view of the world is limited by what the operating system tells it. It must trust the operating system to not show the memory contents of what it is working on to anyone else. The operating system in turn depends on the underlying hardware and firmware for its operation and view of the world.

So computer and network security in practice starts at the hardware and firmware underneath the endpoints. Their security provides an upper bound for the security of anything built on top. In this light, this article examines the security challenges facing us on modern off-the-shelf hardware, focusing on Intel x86-based notebooks. The question we will try to answer is: can modern Intel x86-based platforms be used as trustworthy computing platforms?

We will look at security problems arising from the x86's over-complex firmware design (BIOS, SMM, UEFI, etc.), discuss various Intel security technologies (such as VT-d, TXT, Boot Guard and others), consider how useful they might be in protecting against firmware-related security threats and other attacks, and finally move on to take a closer look at the Intel Management Engine (ME) infrastructure. In the latter part, we will discuss what threats it might present to various groups of users, and why it deserves special consideration. We will also briefly touch on the subject of (hypothetical) CPU backdoors and debate if

they might be indeed a problem in practice, especially compared to other threats discussed.

If you believe trustworthy clients systems are the fundamental building block for a modern healthy society, the conclusions at the end of this article may well be a depressing read. If the adversary is a state-level actor, giving up may seem like a sensible strategy.

Not all is lost. The author has joined up with a group of authors to write another, upcoming, article that will discuss reasonable solutions to many of the problems discussed here. But first we must understand the problems we are up against, with proper attention to technical detail.

Trusted, Trustworthy, Secure?

The word "trusted" is a sneaky and confusing term: many people get a warm fuzzy feeling when they read it, and it is treated as a good thing. In fact the opposite is true. Anything that is "trusted" is a potentially lethal enemy of any secure system [7]. Any component that we (are forced to) consider "trusted" is an ideal candidate to compromise the whole system, should this trusted component turn out to be buggy or backdoored. That property (i.e. the ability to destroy the system's security) is in fact the definition of the term "trusted".

The Operating System's kernel, drivers, networking- and storage-subsystems are typically considered trusted in most contemporary mainstream operating systems such as Windows, Mac OSX and Linux; with Qubes OS being a notable exception [50]. This means the architects of these systems assumed none of the these components could ever get compromised or else the security of the whole OS would be devastated. In other words: a successful exploit against one of the thousands of drivers, networking protocols and stacks, filesystem subsystems, graphics and windowing services, or any other OS-provided services, has been considered unlikely by the systems architects. Quite an assumption indeed!

As we have witnessed throughout the years, such assumptions have been proved to be wrong. Attackers have, time and again, succeeded in exploiting all these trusted components inside OS kernels. From spectacular remote attacks on WiFi drivers [12], [28], [9], exploits against all the various filesystems (especially dangerous in the context of pluggable storage media) [42] to attacks against many other (trusted) subsystems of our client OSes [29].

As a result, the level of security achievable by most computer systems has been very disappointing. If only all these subsystems were *not* considered trusted by

the systems' architects, the history of computer security might have been quite different. Of course, our applications, such as Web browsers and PDF readers, would likely still fall victims to the attackers, but at least our systems would be able to meaningfully isolate instances of compromised apps, so that opening the proverbial malicious attachment would not automatically render the whole system compromised and so our other data could be kept safe.

Moving now to the subject of this article: for years we have been, similarly, assuming the underlying hardware, together with all the firmware that runs on it, such as the BIOS/UEFI and the SMM, GPU/NIC/SATA/HDD/EC firmware, etc., is all... trusted.

But isn't that a rational assumption, after all?

Well, not quite: today we know it is rather unwise to assume all hardware and firmware is trusted. Various research from the last ten years, as discussed below, has provided enough evidence for that, in the author's opinion. We should thus revisit this assumption. And given what's at stake, the sooner we do this, the better.

This raises an interesting question: once we realize firmware, and (some) hardware, should be treated as untrusted, can we still build secure, trustworthy computer systems? This consideration will be the topic of the previously mentioned upcoming article.

Chapter 2

The BIOS and boot security

Let's start our review of an x86 platform from the first code that runs on the host CPU during boot[1], i.e. the BIOS.[2]

BIOS as the root of trust. For everything.

The BIOS, recently more often referred to as "UEFI firmware"[3] has traditionally been considered the root of trust for the OS that executes on the platform, because:

1. The BIOS is the first code that runs on the processor, and so it can (maliciously) modify the OS image that it is supposed to load later.
2. The BIOS has fully privileged access to all the hardware, so it can talk to all the devices at will and reprogram them as it likes: for instance to start shooting DMA write transactions, at some point in time, to a pre-defined memory location where the OS or hypervisor code is going to be loaded later.
3. The BIOS provides the code that executes in System Management Mode (or SMM, discussed later) during the whole lifespan of the platform, and so it can easily inject malicious SMM rootkits [19], [8], [32].

[1] Or so one might think...

[2] Technically speaking: the boot firmware, although on x86 it is customary to call it: BIOS.

[3] Although UEFI is just one possible implementation of a BIOS, one that adheres to the UEFI specification which dictates how modern BIOSes should be written, what features and services they should expose to the OS, etc. Nevertheless other BIOS implementations exist, such as coreboot [57].

Bad SMM vs. Tails

Perhaps the most spectacular demonstration of how security sensitive the BIOS really is, has been provided in the previously mentioned paper [32] through a proof-of-concept SMM rootkit named "LightEater".

LightEater, which executes inside an SMM, is capable of stealing keys from the software running on the host by periodically scanning memory pages for patterns that resemble GPG keys. The authors of the mentioned paper have chosen to demonstrate LightEater attack against Tails OS [59].

Tails has long been (falsely) advertised as being capable of providing security even on a previously compromised laptop[4], as long as the adversary has not been allowed to tamper with the hardware [58]. LightEater has demonstrated in a spectacular way what had been known to system-level experts for years: namely that this assumption has been a fallacy.

An attacker who compromised the BIOS might have chosen to compromise Tails (or any other OS) in a number of other ways also, as discussed above, such as e.g. by subverting the OS code that has been loaded at boot time, even if it was read from a read-only medium, such as a DVD-R.

The key point in the attacks considered here is that the BIOS (and the SMM) might become compromised not only by an attacker having physical access to the system (as Tails developers used to think), but in a number of other ways, which involve only a remote *software* attack, as we discuss below.

How can the BIOS become malicious?

Generally speaking, a BIOS can become malicious in one of two ways:

1. Due to being maliciously written by the vendor, i.e. a backdoored BIOS, or
2. Due to somebody being able to later modify the original (benign) BIOS with a rogue one, either due to:

a. Lack of proper reflashing protection implemented by the original BIOS [51],

b. In case of a BIOS that does apply proper reflashing protection: by exploiting subtle flaws in the original BIOS and getting code execution before the reflashing or SMM locks are applied [71], [11], [30], [74], [32],

[4] E.g. a laptop which used to run e.g. Windows OS that got subsequently compromised.

c. If we include physical attacks in our threat model: by an attacker who is able to connect an SPI programmer to the SPI chip and replace the firmware content stored there.

Lots of effort has been put into eliminating all these latter possibilities of compromising the BIOS, which included: (1) chipset-enforced protections of the flash memory where the firmware is stored, paired with the BIOS requiring digital signature on its own updates, (2) the use of hardware-aided measurement of what firmware really executed on the platform, accomplished with the help of TPM and (optionally) Intel Trusted Execution Technology (TXT), and finally, more recently (3): via Intel Boot Guard technology, which brings a hardware enforced (but in reality ME-enforced) way of whitelisting the firmware so the CPU won't execute any other firmware, even if the attacker somehow managed to smuggle it onto the SPI flash chip. We review all these approaches below.

One should note, however, that no serious effort has been made so far by the industry to deal with the threat of a backdoored BIOS. In other words, no viable solution has been offered by the industry to allow architects to design their systems so that the BIOS could be considered *untrusted*. While Intel TXT could be thought of as an attempt to achieve just that, we will see later that in practice it fails terribly on this promise.

Write-Protecting the flash chip

The most straightforward approach to protect the BIOS from getting persistently compromised, as mentioned above, is to prevent reflashing of its firmware by software that runs on the platform.

Simple as it might sound, the challenge here is that there are legitimate cases where we would like to allow reflashing from the host OS, such as for updating the BIOS, or for storing persistent configuration of the platform (e.g. which device to boot from).

In order to solve this dilemma (protecting against reflashing by malware on the one hand, allowing to reflash for legitimate updates on the other), x86 platforms provide special locking mechanisms (often referred to just as "locks") that the BIOS is supposed to properly set in such a way as to allow only itself access to the flash chip.

Unfortunately this line of defense has been demonstrated not to be foolproof several times (again, see: [71], [11], [30], [74], [32]). Typically attackers would

look for a BIOS vulnerability in the early code that executes before the BIOS locks down the chipset registers.

But even if we had a perfectly secure BIOS (i.e. not containing any exploitable flaws that would allow for code execution during the early stage) an attacker with physical presence could still always reflash the BIOS by opening up the case, attaching an SPI programmer and flashing arbitrary content onto the chip on the motherboard. The approaches discussed below attempt to remedy such attacks too...

Measuring the firmware: TPM and Static Root of Trust

An alternative approach was to invent a mechanism for reliably measuring[5] what firmware and system software *had executed* on the platform, together with an unspoofable way of reporting that to either the user, or to a 3rd party, such as a remote server.[6]

It's important to understand the differences between the approach mentioned previously, i.e. to ensure no unauthorized firmware flash modifications, which really is a white-listing approach, vs. the measuring approach, which does *not* prohibit untrusted code from being executed on the machine. It only provides ways to realize whether what executed was what the user, vendor, or service provider, intended or not, *post-factum*.

To implement such measurement of firmware, some help from hardware is needed for storing and reporting of the measurements reliably, as otherwise malicious software could have forged all the results. Typically this is implemented by the TPM [61], a passive device usually connected to the southbridge via LPC or a similar bus. A TPM device plays the role of a Root of Trust for such measurements[7]. It offers an API to send measurements to it (implemented by so called "PCR Extend" operation) and then provides ways to either reliably report these (e.g. via the "Quote" operation, which involves signing of the measurement

[5] Measurement is a fancy name for calculating a hash in Trusted Computing Group parlance.

[6] E.g. a corporate gateway which would then allow access only to clients who run a proper system, or to a movie streaming service which would stream content only to those clients which run a "blessed" software stack.

[7] Often referred to as *Static* Root of Trust for Measurement (SRTM), to differentiate this scheme from a *dynamic* scheme as implemented e.g. by Intel TXT SENTER instruction, discussed later.

with the known-to-the-tpm-only private key) or conditional release of some secrets if the measurements match a predefined value (the "Seal"/"Unseal" operations).

In order to make the measuring approach meaningful in practice, one needs to somehow condition the platform normal boot process on the measurement of the firmware that executed (i.e. only unlock some functionality of the platform, system, or apps if the hash of the firmware that executed is correct). In practice, the most common way of doing this is to combine a secret that becomes available ("unsealed" in TPM parlance) only if the measurements are "correct", with a user-provided passphrase to obtain the disk decryption key. This then assures that the user will get access to his or her system only if the firmware and the OS that loaded was one the user intended.

Among the first software that made use of this scheme was MS Bitlocker [63], however imperfect in practice as demonstrated by a variation of an Evil Maid Attack [48] presented in [62].

The problem exploited by Evil Maid attacks has been addressed later by Anti Evil Maid [40], [20].

The Evil Maid attacks clearly demonstrated the need to authenticate the machine to the user, in addition to the commonly used authentication of a user to the machine.

A forgotten element: an immutable CRTM

Besides the Evil Maid attacks mentioned above (which haven't really demonstrated problems specific to TPM-based trusted boot, there have been two other problems demonstrated, inherent to this scheme, which is often referred to as Static Trusted Boot:

1. The problem of maintaining a long chain of trust (discussed further down),
2. The need to anchor the chain at some trusted piece of code, somewhere at the very beginning of the platform life cycle. This piece is usually referred to as the Core Root of Trust for Measurement (CRTM).

For the BIOS-enforced (Static) Trusted Boot to make sense, the CRTM would have to be stored in some immutable ROM-like memory. In fact this is exactly what the TCG spec has required for years [61]:

```
The Core Root of Trust for Measurement (CRTM) MUST be an
immutable portion of the Host Platform's initialization
code that executes upon a Host Platform Reset
```

This way, even if the attacker somehow managed to reflash the BIOS code, the (original) trusted CRTM code would still run first and measure the (modified) code in the flash, and send this hash to the TPM, before the malicious code could forge the measurements.[8]

Interestingly, up until recently the CRTM was implemented by the BIOS using ... the normal SPI flash memory. The same flash memory the attacker can usually modify after successfully attacked the BIOS. This has only changed with the latest Intel processors which implement the so called Boot Guard technology.

Intel Boot Guard

Intel has recently introduced a new technology called Boot Guard [37], which can be used in one of two modes of operation: 1) "measured boot", or 2) "verified boot", plus a third option that combines the two. In both cases a special, processor-provided and ROM-based, trusted piece of code [9] executes first and plays the role of a CRTM discussed above. This CRTM then measures the next block of code (read from the flash), called the Initial Boot Block (IBB).

Depending on the mode of operation (determined by blowing fuses by the OEM during platform manufacturing[10]) the Boot Guard's CRTM will either: 1) passively extend the TPM's PCRs with the hash of the measured boot block, or 2) will check if the boot block is correctly signed with a key which has been encoded in the processor fuses by the OEM. In this latter case the Boot Guard thus implements a form of whitelisting, only allowing to boot into a vendor-blessed initial boot block, which in turn is expected to allow only vendor-approved BIOS, which itself is expected to allow only vendor-approved OS, and so on.

It's worth stressing that Intel Boot Guard allows for trivial, yet likely very hard to detect, backdoors to be introduced by Intel. Namely it would be just enough for the

[8] It's worth stressing that the TPM Extend operation works much like a one-way hashing function (in fact it is based on SHA1), in that once we screw up one of the measurements (i.e. send a wrong hash), then there is no way any later code could recover from it. This property allows to always catch the bad code, provided we assure trusted code executes first.

[9] This is the processor boot ROM, and later Intel's signed Authenticated Code Modules (ACM)

[10] This is said to be an irreversible process [37].

processor's boot ROM code, and/or the associated ACM module, to implement a simple conditional: `if (IBB[SOME_OFFSET] == BACKDOOR_MAGIC)` then ignore the OEM-fused public key and proceed unconditionally to execute whatever IBB is currently there.

If such a backdoor was to be implemented in the processor internal boot ROM it would only be possible to find it if we were able to read this boot ROM, which given the bleeding edge processor manufacturing process doesn't seem possible anytime soon. Yet, Intel might also choose to plant such a conditional in the ACM blob which is executed in between the processor boot ROM and the OEM-provided Initial Boot Block [37], this way ensuring no persistent modifications to the processor are added ever. In fact the backdoored ACM might not even be created or distributed by Intel, but rather Intel might only hand the signing key (for the ACM) to the blessed backdoor operator (e.g. a nation-state's signals intelligence bureaucracy). This way the malicious ACM would only ever be used on the target's laptop, making it essentially impossible to discover.

Problems maintaining long chains of trust

But even with a solid CRTM implementation (which we apparently can finally have today, modulo the hypothetical, impossible-to-find Intel-blessed backdoors, as discussed above), there is yet another problem inherent to the Static Trusted Boot: the need to maintain a very long and complex chain of trust. This chain reaches from the first instruction that executes at the reset vector (part of the Initial Boot Block when Boot Guard is used), through initialization of all the devices (which is done by the BIOS/UEFI), through execution of the OS chooser (e.g. GRUB) and the OS loader, and the OS kernel device initialization code. During this whole period a single flaw in any of the software mentioned, such as in one of the UEFI device or filesystem drivers (exploited e.g. via malicious USB stick parsing, or a DMA from a malicious WiFi device, or an arriving network packet triggering a bug in the BIOS' own networking stack, or an embedded image with the OEM's logo having a maliciously malformed header) all could allow the attacker to execute his or her code and thus break the chain and render the whole Trusted Boot scheme useless [71][16].

Additionally, this chain of trust requires us to trust many different vendors in addition to the processor vendor: the BIOS, the driver-providing OEMs, the peripherals' vendors, the OS vendor. To trust, as in: that they didn't put a backdoor there, nor make a mistake that introduced an implementation flaw such as one in [71].

An attempt to solve this problem has been Intel TXT technology, which we discuss further down, but first let's finish with the topic of Static RTM-enforced boot schemes by looking at the recently popular UEFI Secure Boot.

UEFI Secure Boot?

First: what is referred to as "UEFI Secure Boot" is not really a new hardware technology, but rather merely a way of writing the BIOS in such a way that it hands down execution only to code, typically OS loader or kernel, which meets certain requirements. These requirements are checked by validating if the hash of this next block of code is signed with a select key, itself signed with a key provided by the OEM (called a Platform Key).

Now, a BIOS implementing UEFI Secure Boot still faces all the same problems we just discussed in the previous paragraph, related to processing various untrusted inputs from devices and what not, as well as potential DMA attacks and the need to trust all these OEMs to be well intentioned and competent. See also [10].

Additionally, the PKI-like scheme as used for the Secure Boot scheme is not only complex, but also subject to easy abuse. Imagine the BIOS vendor was asked to provide a backdoor to local law enforcement. In that case all the vendor would have to do would be to sign an additional certificate, thus allowing "alternative" code to be booted on the platform. Such "additionally" authorized code could then be immediately used to implement perhaps an Evil Maid attack against the user's laptop [40].

A way to protect against such backdooring would be to combine Secure Boot with measurements stored in a TPM, as discussed previously. But then it becomes questionable if we really need this whole complex UEFI secure boot scheme in the first place? Except, perhaps, to limit the user's freedom of choice with regards to the OS she wants to install on her laptop...

More practical information on UEFI Secure Boot, from the user and administrator point of view, can be found in [60].

Intel TXT to the rescue!

As noted above, the primary problem with the static-based trusted boot is related to the overly long and complex chain of trust, which includes the whole BIOS,

with all its drivers, other subsystems, and also the OS loaders. We also mentioned Intel TXT has been an attempt to resolve this problem.

Additionally, for all those years where CRTM could not be reliably implemented, because there was no Boot Guard feature on Intel processors until recently, Intel TXT was also supposed to provide a reliable alternative to the CRTM.

Intel TXT's two main selling points have been thus:

1. to implement a separate root of trust for platform measurement, independent from the BIOS-provided (or OEM-selected in case of Boot Guard on recent platforms), and
2. to shorten the chain of trust, by excluding the BIOS, boot- and OS loader from the TCB.

Intel TXT is a very complex technology and it's impossible to clearly explain all that it does in just a few sentences. An in-depth introduction to Intel TXT can be found in [22] and [23], and all the other technical details are provided in [25] and [26].

In short the idea is to perform a platform restart ... without really restarting the platform, yet make the restart deep enough so that after the TXT launch the environment we end up with is clear of all the potential malware that might have been plaguing the system before the magic TXT launch instruction was executed.[11]

If that looks tricky, it's because it is! And this is no good news for the TXT's security, as we shall see below.

The broken promise of Intel TXT

As indicated above, Intel TXT is a tricky technology and often security and trickery do not combine very well. In fact, over the years, several fatal attacks have been demonstrated against it.

The first attack, and (still) the most problematic one [66] is that Intel, while designing the TXT technology, apparently overlooked one important piece of code that, as it turned out, survives the TXT launch. This piece of code is able to

[11] The magic instruction is called SENTER, and belongs to the SMX instruction set. It's a very complex instruction...

compromise the TXT-loaded hypervisor or OS. It is called SMM[12] and we briefly mentioned it already, and will be looking at it more below. Suffice it to say: the SMM is provided by the BIOS, and so if the BIOS gets compromised it can load arbitrary SMM. This means that the Intel TXT launch could be compromised by a malicious BIOS, ergo the whole point of Intel TXT (to get rid of trusting BIOS) is negated.

Additionally, due to the high complexity of the technology involved, other surprising attacks have been demonstrated against Intel TXT. One exploited the complex process of configuring VT-d protections, which had to be jointly done by the TXT's SENTER (and more specifically by the Intel-provided SINIT module) and the system software that is being launched [70]. The attack presented allowed an attacker to fool the TXT's SINIT into mis-configuring VT-d protections, thus leaving the subsequently launched code open to DMA attacks triggered by malicious code running before the TXT launch.

Yet another attack [68] took advantage of a memory corruption vulnerability found inside TXT's SINIT module, again due to its high complexity... [13]

The last two bugs have been promptly patched by Intel after the details have been made available to the processor vendor, but the first attack, exploiting the architectural problem with the SMM code surviving the TXT launch has never really been correctly addressed in practice. We discuss this more in the next chapter.

Also, Intel TXT does not protect against other attacks that try to modify the pre-SENTER environment in such a way that once the securely-loaded hypervisor (MLE) starts executing, it might get compromised if not careful enough with processing some of the residual state of the pre-launch world. Great examples of this type of attacks are maliciously modified ACPI tables as discussed in [15]. While one might argue that the TXT-launched hypervisor should be prepared for such attacks, and thus e.g. perform ACPI tables processing in an isolated, untrusted container, this is really not the case in practice due to the large complexity of hardware management.

[12] More technically correct: the SMI Handler, which lives in the so called SMRAM region of specially protected memory, not accessible (officially) even to a hypervisor running on the host.

[13] SINIT modules are conceptually similar to processor microcode, execute with special level of privileges, e.g. having access to the SMRAM, yet they are written using traditional x86 code. They are written and digitally signed by Intel and the processor checks this signature before loading and executing the SINIT code.

Rescuing TXT: SMM sandboxing with STM

Intel's response to the problem with SMM attacks against Intel TXT has been twofold: 1) they decided to harden SMM protections, so that it was harder for the attackers (who don't control the BIOS) to get into SMM and subsequently bypass TXT, and 2) They came up with a notion of a dedicated hypervisor for... SMM sandboxing, so that even if the attacker managed to compromise it, the damage could be confined, especially the TXT-loaded code could not be compromised.[14]

It should be clear that the attempts to harden SMM protections do not really solve the main problem here: Intel TXT must still trust the BIOS to be non-malicious because the BIOS can always - by definition - provide whatever SMM handler it wants.

Additionally, there have been a number of SMM attacks presented over the following years (e.g. [66], [67], [18], [68][15],[74], [3]), all allowing to compromise the SMM, yet not requiring the attacker to control or compromise the BIOS persistently.

It seems fair to say that it has been proven, beyond any argument, that SMM on general purpose x86 platforms *cannot* be made secure enough, and thus should be considered untrusted.

Let's now look closer at the idea of SMM sandboxing, which seems like the only potential way for Intel to not fully surrender the TXT idea... [16]

Intel's idea to rescue the honour of TXT was to introduce a construct of a special additional hypervisor, dedicated to sandboxing of SMM, called SMM Transfer Monitor, or STM.

This sandboxing of the SMM is possible thanks to yet-more-special-case architectural features introduced by Intel on top of the x86 CPU architecture[17], namely the so-called Dual Monitor Mode [26].

[14] As one of the reviewers pointed out, arguably Intel has been planning this SMM-sandboxing technology for quite some time, years before the first attack against TXT was demonstrated publically, as evidenced by the introduction of the Dual Monitor Mode into the processor ISA. Yet, no single mention of an STM has been seen in the SDM or any other official Intel spec known to the author. It thus remains a true mystery what Intel's thinking was and what threat model they considered for TXT.

[15] This attack, for a change, allowed for SMM access as a result of another attack allowing TXT compromise, so could be thought as reverted scenario compared to the original TXT attack which was possible due to SMM compromise which had to be done first:)

[16] Arguably another approach would be to modify SENTER to disable SMI and never let the MLE enable it again. For some reason Intel has never admitted this is a viable solution though.

[17] Yes, yet another convoluted construction Intel has decided to introduce into its complex ISA...

The broken promise of an STM?

There are two main reasons why STM might not be an effective solution in protecting the Intel TXT from malicious SMM handlers. Below we discuss these reasons.

First STM is provided by the BIOS vendor or the OEM. While the BIOS only provides the STM code image, itself not being able to actually start it, and it's the SENTER that measures and loads the STM[18], it still means we need to trust the BIOS vendor that the STM they provided is secure and non-malicious.

Admittedly if the BIOS vendor provided a backdoored STM, they could get caught because the MLE (i.e. the hypervisor loaded by TXT) gets a hash value of the STM, and so could compare it with the value of a known-good STM to see if the BIOS indeed provided a benign STM. The only problem is... there are no such known-to-be-good STMs anywhere out there (at least not to the author's knowledge) which doesn't leave MLE with many options to compare the reported hash to anything meaningful.

Ideally, we should get one (or very few) well known STM implementations, which would be open for anybody to audit, so ideally with their source code open. The source code should then deterministically (reproducibly) build into the same binary. Only then would the hash reported by SENTER be meaningful.

The Intel STM specs [27] seem to support this line of reasoning:

```
"Unlike the BIOS SMI handler and the MLE, however,
frequent changes to the STM are not envisioned. Because
of the infrequency of updates, is expected that using
well known STM hashes, public verification keys, and
certificates are feasible. This will facilitate a
reasonable 'opt-in' policy with regard to the STM from
both the BIOS and the MLE's point of view."
```

This seems to also suggest Intel believes that STM could be made safer than the SMM in terms of resistance to various privilege escalation attacks. Because the author has not seen any real implementation of an STM yet, not to mention a "well known" implementation, and given it has already been about 6 years since the attack has been presented to Intel, the author reserves the right to remain

[18] This is good, because it means the SENTER can reliably compute a hash of the STM. Otherwise, if the STM were to be loaded and started by the BIOS, this would not be possible, of course.

sceptical about the above claims for the time being and hopes Intel addresses this scepticism sometime this decade.

The second potential problem about STM is the so called resource negation between the SMM and the STM. As it turns out the STM cannot really sandbox SMM the way it wants (i.e. thinks is reasonable) – rather it's given a list of resources the SMM wants access to, and must obey this list. To be fair: the MLE is also given a chance to pass a list of protected resources to the STM, and the role of the STM is to: 1) check if the requirements of the two parties are not in conflict (and if they are the MLE might choose not to continue), and 2) keep enforcing all the protections throughout the MLE lifespan.

It's really unclear how, in practice, the MLE should establish a complete list of protected resources that would guarantee the system would not be compromised by the SMM given its access to its granted resources. This might be really tricky given the SMM might want access to various obscure chipset registers, and, as it has been demonstrated a few times in the past, access to some of the chipset registers might allow to catastrophic compromises (see [49], [67], [72]).

Ultimately, the author thinks the Intel way of "solving" the SMM problem has not been really well thought out. The author is of an opinion that Intel, while designing Intel TXT, has never really considered an SMM as a truly non-trusted element. Indeed, while there was some early research about SMM subversion [14], this only targeted unprotected (i.e. unlocked) SMMs. It wasn't until 2008 and 2009 that the first software-based attacks against locked SMMs were presented [49], [66], [67], [18]. But this was already years after TXT was designed and its implementation committed into processors' blueprints.

Intel SGX: a next generation TXT?

In much respect Intel SGX could be thought as of "next-generation TXT". It goes a step further than TXT in that it promises to put not just the BIOS and firmware outside of the TCB, but actually also... most of the OS, including its kernel! This is to be achieved by so called *enclaves* which are specially protected modes of operation implemented by the processor. Code and data of the processes running inside SGX enclaves is inaccessible even to the kernel, and any DRAM pages used by such a process are automatically encrypted by the CPU[19].

The author has more throughly discussed this new technology in [43] and [44], and so here we will only provide the most relevant outcomes:

[19] Actually [37] says that Intel ME is heavily involved in SGX implementation.

1. The SGX cannot really replace a secure boot technology, as it is still not possible to use it for elimination of significant parts of the TCB on desktop systems (see [43]).
2. The SGX doesn't seem capable of protecting the user apps from ME eavesdropping (discussed later), for it seems like SGX itself is implemented with the help of Intel ME [37].
3. Intel SGX might allow for creation of software impossible to reverse engineer, building on SGX-provided DRAM encryption, memory protection, and remote attestation anchored in the processor.
4. For SGX Intel has given up on using a discrete TPM chip in favour of an integrated TPM (also implemented by ME [37]), which opens up possibility for almost perfectly deniable way for Intel to hand over private keys to 3rd parties (e.g. select government organizations) enabling them to bypass any security based on remote attestation for the SGX enclaves without risk of ever being caught, even if others could analyze every detail of the processor, and all the events on the platform (signals on the buses, etc), as described in [44]. With a discrete TPM this might not be the case provided the TPM vendor was considered not part of the conspiracy.

It thus seems like Intel SGX, however exciting and interesting a technology, can neither be used in place of traditional boot security technologies, nor be used to build a more trustworthy environment for special applications which need protection from Intel ME (discussed later).

Summary of x86 boot (in)security

The conclusion from this chapter is that, despite all the various complex hardware-enforced technologies available on x86 platform, such as Boot Guard, TXT, TPM, and others, it's very challenging to implement a satisfactory secure boot scheme today...

About the best approach we can embrace today seems the following:

1. Enable Boot Guard in measured boot mode (not in "verified" mode!),
2. Use a custom BIOS optimized for security (small, processing only minimal amount of untrusted input, embracing DMA protection before DRAM gets initialized),

3. Employ additional protection of the SPI flash chip (e.g. a variant of the work presented in [56])

. . . but this still presents the following problems:

1. Intel Boot Guard might be easily backdoored (as discussed above) without any chance of discovering this in practice,
2. The author is not aware of any BIOS implementation satisfying the requirements mentioned above, especially with regards to the DMA protection,
3. Even if one were to ground the SPI flash chip's Write-Protection signal (as discussed in [56]) this is merely "asking" the chip to enforce the read-only logic, but not forcing it. Admittedly though, by carefully selecting the manufacturer of the flash chip (which thankfully is a discrete element, still) we can distribute trust among the CPU vendor and the flash vendors.

The work required for having the reasonably secure BIOS surely presents a challenge here. It should be clear, of course, that the mere fact that the BIOS we might decide to use was to be open source, does not solve any problem by itself. An open source firmware might be just as insecure as a proprietary one. Of course having such a secure BIOS available as open source, should significantly increase chances of making it actually secure, no question about it. It seems like the coreboot [57] should be the reasonable starting point here.[20]

[20]Sadly even coreboot is not 100% open source, as present platforms require an Intel-provided binary blob, called Firmware Support Package (FSP), which is used to initialize the silicon and DRAM. From a security point of view, FSP doesn't change much in the picture here, as there are also other binary blobs running before any BIOS will get to execute anyway, such as the previously discussed internal boot ROM of the processor and Intel-provided ACM if one uses Boot Guard. Also, one of the reviewers pointed out there is an ongoing work to create an open source equivalent of the FSP.

Chapter 3

The peripherals

The flash chip[1], which stores the BIOS image, typically holds firmware also for other devices, especially the integrated devices as used on most laptops, such as e.g. the networking devices, the Intel ME, and potentially also the GPU.

As this firmware does not run on the main CPU (unlike the BIOS firmware discussed above), it's often substantially easier to put outside of the TCB by sandboxing the whole device. This allows us, BTW, to bake three cakes at the same time, i.e. to 1) put the hardware (e.g. a WiFi device), 2) its firmware, as well as 3) the corresponding OS drivers and stacks, all outside of the TCB!

Networking devices & subsystem as attack vectors

First are, of course, those pesky networking cards: the Ethernet and wireless devices such as the WiFi, Bluetooth, and 3G/LTE modems. They, of course, represent a potential attack vector as these devices, as well as the associated drivers and stacks running on the host, perform continuous processing of a lot of untrusted input. The untrusted input in this case is all the incoming packets. And given the wireless nature of WiFi, Bluetooth, and 3G/LTE modems, this input is really as untrusted as one might only imagine.

Multiple attacks have been presented attacking Wireless networking: targeting hardware [16], drivers [12][9], and networking stacks [28].

[1] Often referred to as "SPI flash", although on some systems other buses might be used to connect the flash to the chipset, such as e.g. LPC. What is important is that these flash chips are still discrete elements, and it doesn't seem like this is going to change anytime soon, due to silicon manufacturing processes, apparently.

Networking devices as leaking apparatus

Another problem related to networking devices, especially the wireless ones, is that they could be used to leak sensitive information stolen by malware already running on the user's laptop. In this case the attack might be delivered by some other means (e.g. by opening a maliciously malformed document), often not requiring an active networking connection, and would thus be applicable even in case of "air-gapped"[2] off-line systems.

Especially worrying might be a scenario where a maliciously modified firmware on the WiFi card cooperates with malware running somewhere in the ME (discussed below) or SMM (mentioned above). In that case the malware, after it stole some crucial user data, such as a few private keys it might have found by asynchronously scanning host memory pages[3], might passively wait for a magic trigger in the form of a special WiFi broadcast packet to appear, and only then send out the accumulated data in one quick, encrypted burst to its "mother ship". This seems like an especially practical approach for exfiltration of data from attendees at larger gatherings, such as conferences. In that case the magic trigger could be e.g. the broadcast 802.11 packets for a particular SSID, e.g. "BlackHat" :-) Potentially combined with some additional condition, such as presence of traffic packets with a particular magic MAC address, which would indicate proximity of the "mother ship" laptop.

Sandboxing the networking devices

Historically it's been difficult to properly sandbox drivers and their corresponding devices on x86 platforms. Even though the processors have been offering 4 isolated rings of execution (ring 0..3) since the introduction of the 80386 model back in mid-80s[4], this mechanism was not really enough for drivers de-privileging, because of the problem known as "DMA attacks" (see e.g. [65] for discussion and proof-of-concept attacks in the context of the Xen hypervisor).

Only about 20 years later, i.e. mid-2000s, did we get the technology that could be used to meaningfully de-privilege drivers and devices on x86 platforms. This new technology[5] is called IOMMU, and Intel's implementation goes by the product

[2] The reader should not be too picky about the use of the term "air-gapped", used here in a very loose meaning.

[3] Like the previously discussed LightEater malware [32]

[4] The author was stuck with an old 80286 still during the early 90s...

[5] "New" for the x86 world, at least

name "VT-d"[6]. Intel VT-d is a technology implemented in the processor.[7]

With Intel VT-d one is capable of fully de-privileging drivers and devices [49], [50]. Interestingly one does not need to use the additional rings (i.e. ring 1 and 2) for that.

Keeping networking devices outside of the TCB

Sandboxing a component is one thing, but making it effectively not-trusted in the system's design is another. E.g. the mere sandboxing of the networking devices and subsystem would not buy us much if the attacker, who managed to compromise e.g. the networking stacks, was able to extract sensitive information (such user passwords) or modify the packets that pass though this subsystem (such as those carrying binary executables with system updates) or inject some malicious ones in order to perform efficient spoofing attacks. This would be the case if the apps running on the system were to (for example) not use encryption and integrity protection for their network communications (e.g. the OS would download updates and install them without checking signatures on the actual binaries).

Fortunately, in case of networking, a lot of effort has been put into securing transmissions going over unreliable and insecure infrastructure (things such as SSL, SSH, digital signatures on binary blobs, etc.) and most mature applications use it today, so this should not be a problem, in theory, at least.[8]

More discussion on this topic can be found in [41] and [46].

Preventing networking from leaking out data

Mere sandboxing of the networking devices and subsystem using VT-d cannot prevent malware which has compromised the host OS from using the hardware to

[6]Intel VT-d should not be confused with Intel VT-x, which is a CPU virtualization technology. Many consumer systems have Intel VT-x but lack VT-d. The opposite is not true though: if a platform implements VT-d, then it also should have VT-x.

[7]On older platforms: in the MCH, but for a few years now the MCH has been integrated into the same physical package as the CPU.

[8]In particular, we can say that an attacker who compromised a properly sandboxed networking compartment, gains nothing more than he or she would gain by sitting next to the user in an airport lounge, while the user was using a WiFi hotspot. Or if the user's ISP was assumed to be not trusted, which is always a wise assumption.

leak sensitive data stolen from the user.[9]

In order to prevent this type of abuse of networking devices we need hardware kill switches to disable the devices. Many laptops allow to switch off wireless devices, but they often exhibit the following problems:

1. The switches are implemented in a form of software-controlled buttons (e.g. Fn-Fx), in which case we still trust the code or firmware handling these to be non-malicious (typically this would be the Embedded Controller (EC), discussed below, or the SMM handler, part of the BIOS, mentioned above).
2. Even if the switches are implemented as physical switches, they might not physically control power to the device(s), but rather ground specific signals on the device, merely "asking it" to disable itself (e.g. this is the case for the networking kill switch discussed in [35]).

Of course, if we could assure that the OS would never get compromised, or that any parts of the OS that might get compromised (e.g. specific apps or VMs) would not get access to the networking hardware, this might be a good enough solution without using a hardware kill switch (see e.g. [46]).

Unfortunately, we can do little against a potentially backdoored ME or an SMM. While the physical kill switch would be an effective solution here, it presents one significant disadvantage (in addition to the problem that nearly no laptop really implements it, that is): namely it prevents user from using (wireless) networking...

The USB as an attack vector

The mere term "USB" is a bit ambiguous, as it might refer to different things:

1. The USB controllers, which themselves are actually PCI Express (PCIe) type of devices, capable of being bus-masters and so requiring proper sandboxing using VT-d, as discussed for the networking devices above.
2. The actual USB devices which are connected to USB controllers, such as portable storage "sticks", USB keyboards & mice, cameras, and many others.

[9] It's convenient to think about IOMMU and VT-d as a "diode"-kind of protection, i.e. protecting the OS from malicious devices, but not preventing the OS from communicating with, or using the devices.

> These devices are not PCIe type of devices, and thus cannot be individually sandboxed by VT-d. But even if they could, the type of problems they present to the OS can not always be effectively solved by mere sandboxing (see [42]). Also, some of these devices, such as e.g. cameras, might be presenting additional vectors, somehow orthogonal to the problems inherent to the USB architecture, as discussed below.

Additionally the border between the USB subsystem and the rest of the OS and even some of the applications is much less sharply defined than is the case with the networking subsystem.

E.g. plugging of a USB mass storage device triggers a number of actions inside the OS, seemingly not related to the USB subsystem, such as parsing of the device's partition table and filesystem meta-data. Sometimes, the device (or filesystem) might be encrypted and this might additionally start software that attempts to obtain a passphrase from the user and perform decryption. On some OSes, such as Microsoft Windows, the system will try to download matching drivers for whatever device is being plugged in, which also presents a number of security challenges.[10]

All the above makes proper sandboxing of USB devices somehow less trivial than it was the case with networking devices (because we can only isolate USB devices in chunks defined by which USB controller are they connected to, without having any finer grained controls), and also even more challenging to actually keep these devices, as well as the USB subsystem, outside of the system's TCB. In fact in some cases the latter might not be possible.

One problematic case is when USB is used for input devices, such as keyboard and mouse. In this case, even though the USB controller might be assigned to an unprivileged domain (e.g. a VM), still this brings little gain for the overall system security, as the keyboard represents a very security-sensitive element of the system. After all, it is the keyboard and mouse that conveys the user's will to the system, so whoever controls the keyboard is able to impersonate the user, or, at the very least, sniff on the user: e.g. capture the keystrokes used to compose a confidential email, or the login/screenlocker/disk-decryption passphrase.[11]

Fortunately on most laptops, with Apple Macs being a notable exception, the integrated keyboard and touchpad are almost always connected through LPC or

[10] Perhaps the driver for an exotic device that is being downloaded and installed automatically, while not directly malicious, because it was "verified" and digitally signed by MS-approved certificate, might be exposing vulnerabilities that could allow access to the kernel [47].

[11] We don't assume it might be able to sniff any other passwords, as it's assumed the user is reasonable enough and uses a password manager, potentially combined with 2FA, rather than trying to memorize his or her passwords and enter them by hand!

some other bus, not through USB.[12] Fortunately.

It might seem surprising that the mere use of an LPC-connected keyboard/mouse over a USB-connected one provides such a huge security benefit. What's so special about this LPC bus then, that the USB architects have missed, one might ask? Perhaps the whole world should switch from using USB to LPC then?

Such thinking is wrong. The main advantage of an LPC-connected keyboard over that connected via USB is the static nature of the former.[13] This, in turn, allows the OS to skip all the actions necessary to 1) (dynamically) discover the devices when they are plugged in, 2) retrieve and parse their (untrusted) configuration, 3) search and load (potentially untrusted) drivers, and 4) let these drivers talk to the device, often performing very complex actions, etc.

Of course, while the static bus works well for the integrated keyboard and touchpad, it would not work so well for other devices, for which the USB bus has been designed.

Another problem with de-privileging of USB subsystems might be faced when the OS is booting from USB storage, which is connected to the same controller we want to make untrusted. Of course, typically, we would like to make all USB controllers untrusted, by VT-d-assigning them all to some VM or VMs. In that case a reliable de-privileging of the domain which contains the OS root image becomes tricky and requires a reliable trusted boot scheme, as discussed in [50]. That, however, as we saw in the previous chapter is still something we lack on x86 platforms...

Still, assuming the keyboard is connected through a "static bus" connection, and the OS boots normally from the internal HDD (connected through SATA controller, not USB), there are lots of opportunities to keep USB subsystems outside of the TCB. This includes e.g. using an encrypted USB storage, through untrusted USB VM, with the actual decryption terminated in another, more trusted VM. Thus, a potentially malicious firmware, or partition table on the USB device, or even potentially backdoored USB controller, cannot really extract the secrets stored on the stick.

More discussion about USB attacks can be found in [42], and more about how the OS can help alleviating some of the attack vectors coming through USB devices using sandboxing in [46]. Also, as one of the reviewers pointed out, the Linux kernel provides a mechanism that could be used to blacklist certain USB

[12] More technically speaking, the Embedded Controller (discussed later) exposes the keyboard functionality to the chipset over an LPC bus.

[13] Another advantage is that there typically are no untrusted input-processing devices on the same bus.

devices (and soon interfaces) from being used in the system [34], [31]. While this could be used to block some of the simpler attacks, such as preventing the keyboard-pretending devices from being used as actual keyboards by the system, that solution does not really provide true sandboxing of the hardware and kernel-side USB-processing code. Also, being a form of black- or white-listing solution, it arguably affects the usability aspect significantly.

The graphics subsystem

The graphics card, together with the graphics subsystem, is generally considered a trusted part of the system. The reason for this is that the graphics subsystem always "sees" all the screen, including all the user's confidential documents, decrypted.

To a large degree of confidence, it is possible to keep the graphics card and the whole graphics subsystem isolated from attacks. This is possible by exposing only a very thin, strictly audited protocol to the rest of the system (see e.g. [50]). [14]

However, this does not solve the potential problem of a graphics card, or its firmware, being backdoored. The latter might become compromised not only as a result of the vendor intentionally inserting backdoors, but also as a result of an attacker reflashing the GPU firmware. This might happen as a result of an Evil Maid-like attack, in which the attacker might boot the target computer from a malicious USB stick, which would then perform a reflashing attack on the GPU. Such an attack would work, of course, irrespective of what OS is installed on the host, as the attacker would reboot the system from its own OS from the USB device. Only a solid, whitelisting, boot security mechanism could prevent this from happening, but that, as we have discussed in the previous chapter, is still not quite available on x86 platforms today...

Protecting against a potentially malicious graphics subsystem is tricky (see [39]), but protecting against only a malicious GPU might be more feasible, as this should be achievable by maintaining strict IOMMU protections fine-grained only to the frame buffer and other pages designated for communication with the graphics card, and no other pages. At least in theory. In practice this is still questionable, because of the heavy interaction of the (trusted) OS graphics subsystem with

[14] Specifically, it should be noted that naive approaches, such as running the X server as non-root, really provide little security gain on the desktop. For the attacker it is really the X server that represents a meaty target, no matter if it runs as root or not, as it is the X server that sees all the user data...

the graphics hardware, which likely opens up lots of possibilities for the latter to exploit the former.

No OS today implements the protection against a malicious graphics subsystem, unfortunately. It's worth mentioning though that the Linux kernel does seem to implement a partial protection mentioned above, because of the fine-grained IOMMU permissions it applies for (all?) PCIe devices, including GPU devices.[15]

The disk controller and storage subsystem

The disk controller (often just called: SATA controller), similarly to the GPU is often considered trusted. The reason for this is that a compromised disk controller can present modified code for the OS kernel or hypervisor during the boot sequence, and thus compromise it.[16] Assuming we had a reliable secure boot technology, this problem could be resolved and so the disk controller could be considered untrusted. Of course we assume the OS would additionally use disk encryption implemented by the OS, rather than disk-provided encryption, which is always a good idea anyway.[17]

Keeping the disk controller untrusted is only part of the story though. Just like with Networking, USB, and GPU, we talked about sandboxing of the hardware as well as of the associated drivers and other subsystems. The same applies to the disk controller – it's important to also keep the disk subsystem untrusted. This includes typically all the code that exposes the storage to the applications, e.g. the disk backends in case of a virtualized system. Fortunately this is doable, albeit a bit more complex, and does not require any additional hardware mechanisms beyond a secure booting scheme, and IOMMU. More details could be found in [50].

But more importantly, similarly to the GUI subsystem discussed above, the disk controller and hardware could be kept reasonably secure, provided the host OS uses proper care to isolate it from the untrusted parts of the system, such as the apps, and less trusted hardware (e.g. USB, networking)[50].

[15] The author hasn't researched this issue in more detail.

[16] One of the reviewers has pointed out that the SATA protocol actually allows the disk itself to pass requests for setting up DMA transactions to the SATA controller. While the controller is expected to validate these requests, this still opens up possibilities for even more attacks, given disks are usually much less trusted than the SATA controller (which is almost always part of the processor package on modern systems).

[17] It's a good idea, because the disk-controller-provided, or disk-provided encryption could not be audited and so we could never know if it has not been backdoored in some way, e.g. by leaking key bits in response to magic signals presented to the hardware interface.

Assuming the interface exposed by the OS to the (trusted) storage subsystem is indeed secure, then the only way for the disk controller hardware to get malicious and present a serious security concern to the system, is via vendor-introduced backdoors in the hardware or firmware, or by means of a physical attacker able to reprogram the flash chip on the device[18] by connecting a programming device.

The audio card

The audio card presents three potential security problems:

1. It controls the microphone, so if the attacker compromised the host OS (or just the audio card firmware) they could potentially listen to the user's conversations,
2. It controls speakers, which in case of a compromised host OS (or some firmware, such as the BIOS/SMM or ME) allows to leak stolen sensitive information even from an "air-gapped" (network-disconnected) machine by establishing some communication channel, perhaps using some inaudible frequencies [13].
3. Finally, as always with a bus-master device, the potentially backdoored firmware of the audio card (which typically is a PCIe device) can compromise the host OS if it doesn't use proper IOMMU protection.

The author believes the best protection against the first two attacks is via physical kill switches, discussed below. As for protection against the last vector, again, just like it was with the GPU and SATA controller, the best protection should be via application of strict IOMMU permissions only to audio data pages, combined with proper OS design that exposes only a very strictly controlled interface to untrusted software for interaction with the Audio subsystem.

Microphones, speakers, and cameras

The actual "spying" devices such as microphones and cameras are notoriously bundled with modern laptops, and users often have no possibility to opt out.

[18] Most likely only on the disk, as it seems rather unlikely for this attack to work against the firmware for the Intel integrated devices, such as the SATA controller, due to the ME most likely checking the signature on it. That note, of course, does not apply to the attacks performed by those who are in the possession of the Intel signing key, such as, presumably, some law enforcement agencies.

While the audio card (to which the microphones are typically connected), as well as the USB controllers (to which the camera(s) are typically connected), might be sandboxed as discussed above, this still doesn't solve the problem that microphones and cameras might present to the user. Whether trusted or not by the OS, leaking recorded audio of a user's private conversations, or images taken with the built-in camera, can present a serious problem.

Many laptops offer software buttons, or BIOS settings (in the so called "Setup") to disable microphones and/or cameras, but such approaches are far from ideal, of course. A compromised BIOS or SMM firmware will likely be able to get around such software-enforced kill switches.

Thus, it only seems reasonable to desire physical kill switches, which would be operating on the actual signal and power lines for the discussed devices. One laptop vendor has already implemented these [35], in case of other devices, the user can often remove these themselves (see e.g. [45]). This problem is, of course, not specific to the x86 platform, it applies to all endpoint devices. Sadly most are being equipped with microphones and cameras these days.

The Embedded Controller

The Embedded Controller (see e.g. [21]) is a little auxiliary microcontroller connected through an LPC or SPI bus to the chipset, and responsible for 1) keyboard operation, 2) thermal management, 3) battery charging control, 4) various other OEM-specific things, such as LEDs, custom switches, etc.

The good news is that the EC is not a bus-master device, thus it has no access to the host memory. The bad news, though, is that it (typically) is involved with keyboard handling, and so is able to sniff and/or inject (simulate) keystroke events. This means a malicious EC firmware can e.g. sniff the disk decryption/login passwords. Theoretically it can also sniff the content of some confidential documents or emails, or chats, again, by sniffing the scan codes flying through it. This is somehow questionable in practice though, due to highly multi-tasked nature of desktop OSes, and lack of any mechanism the EC could use to detect the context. In other words: knowing whether the given keystroke was part of writing a secret document, or a keystroke for some other application the user just switched to.[19]

[19] Arguably the EC, being able to see all keystrokes and touchpad events might be able to accurately model the window placement on the screen and figure out which one is currently active (focused), as one of the reviewers pointed out. In practice, the author thinks, this seems really hard, and very system and applications software version-specific.

However, a reasonably practical attack could be for the EC to inject a pre-defined series of keystrokes that e.g. open a terminal window on the host and then paste in some shell script and execute it. This would be a very similar attack to one of the popular malicious-USB-device-pretends-to-be-keyboard attacks (e.g. [55]).

While the protection against password sniffing attacks is rather simple – it's just enough to use some form of OTP or challenge-response, preventing the EC from injecting lethal keystrokes on behalf of the user is more challenging. One solution might be de-privileged GUI domain, or kiosk-like lock-down of the trusted desktop environment.

All these concerns apply, of course, only if we cannot afford the comfort of treating the EC as a trusted element. Sadly, this seems like a reasonable assumption, given ECs in most laptops today are black-box controllers running OEM's proprietary code.[20]

The Intel Management Engine (ME)

There is another embedded microcontroller on modern Intel platforms, which should by no means be confused with the previously discussed EC, for it is significantly more powerful in terms of potential damage it could do the user. The security problems arising from it are also much more difficult to address. A full chapter is devoted to this Intel Management Engine embedded controller below...

Bottom line

As we have seen, perhaps to much surprise for many, peripherals such as WiFi, USB controllers and devices, the disk controller, and even the GPU can all be reasonably well de-privileged and treated as either completely untrusted or partly-trusted (GPU). This requires substantial help from the host OS, however.

Nevertheless, in the author's opinion, it's an important point to realize the peripheral hardware is not as much worrying, as is the previously discussed problem of boot security, and even less so than the threat from Intel ME which we discuss next...

[20] And even in case of laptops that boast open source EC, such as Google Chromebook, it's worth to consider whether the user has a reliable method to verify if the EC firmware on the laptop has indeed been built from the very source code that is published on the website...

Chapter 4

The Intel Management Engine

The Intel ME is a small computer (microcontroller) embedded inside *all* recent Intel processors [33], [37]. In the past ME was located inside the Northbridge (the Memory Controller Hub) [72], but since Intel's move towards integrated packaging which contains the MCH and the CPU (and more recently also the Southbridge) all inside one physical package, the ME has become an integral part of Intel CPUs without any hope of removing or disabling it.

Intel ME is very much similar to the previously discussed SMM. Like SMM it is running all the time when the platform is running (but unlike SMM can also run when the platform is shut down!). Like SMM it is more privileged than any system software running on the platform, and like SMM it can access (read or write) any of the host memory, unconstrained by anything. That's actually a bit of a difference between ME and SMM, since, as we have discussed above, the SMM can, at least in theory, be constrained by an STM hypervisor. ME's desire for accessing the host memory cannot be constrained in any way, on the other hand, not even by VT-d.

Also, while the SMM can be disabled (even if that might be strongly discouraged by the vendors), the ME cannot, according to the "ME Book" [37][1], as there are way too many inter-dependencies on it in modern Intel CPUs.

Finally, while the SMM code can still be reliably dumped from the SPI flash chip using an EEPROM programmer, and analyzed for vulnerabilities and backdoors, the same, sadly, is not true for the ME code, for reasons explained below.

[1]While admittedly not an official Intel publication, this book has been written by one of Intel's architects involved with ME design and implementation, and the book itself has also been selected for "Intel's recommended reading list".

ME vs. AMT vs. vPro

Originally Intel ME has been introduced as a mean to implement Intel Advanced Management Technology (AMT), which is an out-of-band, always-on, remote management toolkit for Intel platforms [33]. Indeed, AMT has been the first "application" running on the ME infrastructure, but over the years more and more additional applications, such as PTT (ME-implemented TPM), EPID (ME-rooted chain of trust for remote attestation), Boot Guard, Protected Audio and Video Path (PAVP), and SGX, have been introduced, and it's expected the number of such applications will grow with time [37].

Thus, one should not consider AMT to be the same as ME. Specifically, it is possible to buy Intel processors without AMT, but that does not mean they do not feature the ME system – as already stated, according to the available documentation, all modern Intel processors do have the Management Engine, no exceptions.

Another source of confusion has been the "vPro" marketing brand. It's not entirely clear to the author what "vPro" really means, but it seems it has been used extensively in various marketing materials through the years to cover all the various Intel technologies related to security, virtualization and management, such as VT-x, VT-d, TXT, and, of course, AMT. In some other contexts the use of vPro might have been narrowed to be a synonym of an AMT.

In any case, one should not consider a processor that "has no vPro", or "vPro disabled", to not have Intel ME – again, ME currently seems such a central technology to modern Intel processors that it doesn't seem possible to buy a processor without it.

Two problems with Intel ME

An alert user clearly must have noticed already that Intel ME might be a troublesome technology in the opinion of the author. We will now look closer why is that so. Actually, there are two reasons, inter-connected, but distinct.

Problem #1: zombification of general-purpose OSes?

The first reason why Intel ME seems so wrong, is what can be called "zombification" of the general purpose x86 platform. When reading through the "ME Book" [37] it is quite obvious that Intel believes[2] that 1) ME, which includes its own custom OS and some critical applications, can be made substantially more secure than any other general purpose system software written by others, and 2) ultimately all security-sensitive computing tasks should be moved away from the general purpose OSes, such as Windows, to the ME, the only-one-believed-to-be-secure-island-of-trust...

This thinking is troublesome for a few reasons. First, Intel's belief that its proprietary platform, unavailable for others to review, can be made significantly more secure than other, open platforms, sounds really unconvincing, not to say arrogant.[3]

Furthermore, if the industry buys into the idea this would quickly lead to what we can call "zombification" of these systems. In other words systems such as Windows, Linux, etc., would quickly become downgraded to mere UI managers (managing the desktop, moving windows around) and device managers (initializing and talking to devices, but never seeing any data unencrypted), while all logic, including all user data this logic operates on, would be processed inside the ME, behind the closed doors without others being able to see how the ME processes this data, what algorithms it uses, and whether these are secure and trustworthy or not.

Does a hypothetical email encryption implemented by the ME also adds an undesirable key escrow mechanism that the user cannot opt out of? Does it perform additional filtering of (decrypted) emails and documents, perhaps also samples of audio from microphone for "dangerous" words and phrases? Does it store the disk encryption key somewhere for (select) law enforcement to obtain in some cases? Is the key generation indeed secure? Or is the random number generation scheme maybe flawed somehow (intentionally or not)?

All these questions would be very difficult to answer, significantly more difficult than they are today, even in case of proprietary OSes such as Windows or OSX. This is because with the ME it is orders of magnitude more difficult for us mortals to reverse engineer its code, analyze and understand it, than it is with systems

[2] Or at least some of its key architects do.

[3] Indeed, Intel has been proven wrong on this account several times, as pointed out throughout this report.

such as Windows or OSX.

Problem #2: an ideal rootkiting infrastructure

There is another problem associated with Intel ME: namely it is just a perfect infrastructure for implanting targeted, extremely hard (or even impossible) to detect rootkits (targeting "the usual suspects"). This can be done even today, i.e. before the industry moved all the application logic to the ME, as theorized above. It can be done even against users who decided to run open, trustworthy OS on their platforms, an OS and apps that never delegate any tasks to the ME. Even then, all the user data could be trivially stolen by the ME, given its superpowers on the x86 platform.

For example, just like the previously discussed LightEater SMM rootkit [32], a hypothetical ME rootkit might be periodically scanning the host memory pages for patterns that resemble private encryption keys, and once these are identified they can be stored somewhere on the platform: as encrypted blobs on the disk, or on the SPI flash, or perhaps only inside the ME internal RAM, as laptops are very rarely shutdown "for real" these days.

The trigger for such a ME rootkit could be anything: a magic WiFi packet, which might work great if the rootkit operator is able to pin-point a physical location of the target, or a magic stream of bytes appearing in DRAM at predefined offset[4], or perhaps a magic command triggered by booting the system from an "Evil Maid's" USB stick, something the adversary might do upon interception of the hardware during its shipment to the customer.

Disabling Intel ME?

Intel ME is so tightly integrated into Intel processors that there doesn't seem to be any way to disable it. A large part of the ME firmware is stored on an SPI flash chip, which admittedly could be reprogrammed using a standard EEPROM programming device to wipe all the ME partition from the flash. But this approach does not have a good chance of succeeding. This is because the processor itself contains an internal boot ROM [37] which is tasked with loading and verifying

[4]Which might appear there as a result of the OS allocating e.g. a network buffer and copying some payload there, in response to the user visiting a particular website perhaps.

the rest of the ME code. In case the bootloader is not able to find or verify the signature on the ME firmware, the platform will shutdown.[5]

Auditing Intel ME?

If Intel ME cannot be disabled, perhaps it could at least be throughly audited by 3rd parties, ideally via an open-sourced effort? Unfortunately there are several obstacles to this:

1. On at least some platforms the firmware stored on the SPI flash is encoded using a Huffman compression with an unknown dictionary [54], [6][6]. Even though this is not encryption, this still presents a huge obstacle for any reverse engineering efforts.

2. At least on some platforms, the internal boot ROM additionally implements a library of functions being called from the code stored on the flash, and we lack any method to read the content of this internal ROM.

3. The documentation of the controller is not available. Even if we could guess the architecture of the processor, and even if the code was not encoded, nor encrypted, it would still be very difficult to meaningfully interpret it. E.g. the addresses of internal devices it might be referring to (e.g. DMA engines) would be mostly meaningless. See also [72].

4. There does not seem to be any way for mere mortals to dump and analyze the actual internal boot ROM code in the processor.

This last point means that, even if all the firmware stored on the SPI flash chip was not encoded, didn't make use of any calls to the internal ROM, and the whole microcontroller was thoroughly documented, still the boot ROM might contain a conditional clause that could be triggered by some magic in the firmware header. This would tell the boot ROM to treat the rest of the flash-stored image as encrypted. This means Intel might be able to switch to using encrypted blobs for its ME firmware anytime it wants, without any modification to the hardware. Do the existing, shipped, platforms contain such a conditional code in their boot ROM? We don't know that, and cannot know.[7].

[5] Interestingly, the author was unable to find any official statement from Intel stating the platform will unconditionally shutdown, nor is aware of any practical experiments supporting this, but the "ME book" makes a clear suggestion this would be the case indeed.

[6] The dictionary was most likely stored in the internal ROM or implemented in the silicon.

[7] We would only know if Intel started shipping encrypted ME firmware one day...

While it is common for pretty much any complex processors to contain internal boot ROM, it's worth pointing out that not every product makes the boot ROM inaccessible to 3rd parties, see e.g. [2].

Summary of Intel ME

We have seen that Intel ME is potentially a very worrisome technology. We cannot know what's really executing inside this co-processor, which is always on, and which has full access to our host system's memory. Neither can we disable it.

If you think that this sounds like a bad joke, or a scene inspired by George Orwell's work, dear reader, you might not be alone in your thinking...

Chapter 5

Other aspects

In this last chapter we discuss a few other security-related aspects of Intel x86-based systems, which, however, in the author's opinion, represent much smaller practical problems than the topics discussed previously.

CPU backdoors

Every discussion on platform security, sooner or later, gets down to the topic of potential backdoors in the actual processor, i.e. the silicon that grinds the very machine instructions. Indeed, the possibilities that the processor vendor has here are virtually limitless, as discussed e.g. in [17], [38].

There is one problematic thing with CPU-based backdoors, though. This is lack of plausible deniability for the vendor if some malware gets caught using the backdoor. This itself is a result of lack of anti-replay protection mechanisms, which are a result of modern CPUs not having flash, or other persistent form of memory, which in turn is dictated by the modern silicon production processes.[1]

Of course the existence of Intel ME might change that, as ME itself is capable of storing data persistently as it has its own interface to the SPI flash chip. True. But in case we consider a platform with ME[2], then... it doesn't seem to make much sense to use CPU-based backdoors because it's much easier, more convenient, and more stealthy, to implement much more powerful backdoors inside the ME microcontroller.

[1] Although this might not be true for some processors which use so called eFuse technology [64].

[2] As previously stated, virtually any modern Intel processor seems to have ME anyway, but older generations might not.

Backdooring can apparently be done at the manufacturing level also [4], although it seems this kind of trojaning is more suitable for targeting crypto operations (weakening of RNG, introducing side-channels), rather than targeting code execution. It seems this kind of backdoor is less of a problem, because, theoretically at least, it might be possible to protect against them by using carefully written crypto code (e.g. avoiding use of the Intel RDRAND instruction).

It's generally unclear though, how one should address the problem of potential code-execution backdoors in CPUs. About the only approach that comes to mind is to use emulation[3], ie. not virtualization(!), in the hope that untrusted code will not be able to trigger the real backdoor in the actual silicon through the emulated layer.

Isolation technologies on Intel x86

There has been lots of buzz in the recent years about how the new "hardware-aided" virtualization technology can improve the isolation capabilities for general purpose OSes on x86. In the author's opinion these are mostly unfounded PR-slogans. It's important to understand that none of the attacks we saw over the last two decades against popular OSes had anything to do with x86 isolation technologies weaknesses. All these attacks had always exploited flaws exposed by system software through overly complex and bloated interfaces (e.g. all the various syscalls or GUI subsystem services, or drivers IOCTLs).

On the other hand, some of the complexity added by system software has sometimes been the result of the underlying complex x86 architecture and a misunderstanding of some of its subtleties by OS developers, such as demonstrated by the spectacular SYSRET attack [73].

Admittedly also some things, such as the complexity needed to implement memory virtualization, can be reduced with the help of silicon (and this is where VT-x and EPT indeed come handy).

It's also worth mentioning that Intel has a history of releasing half-baked products without key security technologies in place and without explicitly warning about the possible consequences. This includes e.g. releasing of systems implementing VT-d without Interrupt Remapping hardware, which has been shown to allow for practical VT-d bypass attacks [69]. Another example has been the previously discussed problem of introducing Intel TXT without proper protection against

[3] Ideally of some other architecture than that of the host CPU

SMM attacks, or at least explicitly educating developers about the need for an STM.

Covert and side channel digression

Strictly speaking the existence of potential covert and side channels on x86 is outside the scope of this paper. In practice it might be relevant for OSes that implement Security by Isolation architecture, in part to resolve some of the problems mentioned in this paper. Thus a few words about these seem justified here.

First, it's important to understand the distinction between covert channels and side channels: the former require two *cooperating* processes, while the latter do not. In the context of a desktop OS the existence of covert channels would be a problem if the user was concerned about malware running in one security domain (VM or other container), being able to establish covert communication with cooperating malware running in another domain, and leak some information to it.

This could be the case perhaps if the first domain was intended to be kept off-line, and used for some dedicated, sensitive work only, while the other domain was a more general-purpose one. Now imagine the user opens a maliciously modified PDF document in the off-line domain which exploits a hypothetical flaw in the PDF viewer running there. The user would like to think that even then the data in this protected domain would remain safe, as there should be no way for the malware to send any data to the outside world (due to the VM being kept off-line by the VMM). But if the malware running in the other domain can now establish a covert channel to the malware running in the protected domain, perhaps exploiting some sophisticated CPU cache or other semantics [36], then this would present a problem to the user.

It's generally believed that it is not possible to eliminate covert channels on x86 architecture, at least not if the system software was to make use of the multi-core and/or multi-thread features of the processors. Some work has been done in order to reduce such covert channels though, e.g. through special scheduling policies, but they often impact the performance significantly.

Now, the side channel attacks are much different in that they do not require the source domain to be compromised. This means the user, in the example above, does not need to infect his or her protected domain – all they need to do is to have some malware running in another domain, and this malware might now be able to e.g. sniff the content of some keys or other data as used by the software

running in the protected domain. The malware would, again, exploit subtle timing or other differences in caching or other elements of the processor, in order to deduce about the execution flow in other processes. (See e.g.: [5], [24]).

In the author's opinion[4], the side channel attacks are in general difficult to exploit on a typical general-purpose desktop system, where not only many programs run at the same time, but where the attacker typically also has very little control over triggering of various operations (such as crypto operations operating on the specific key or data), multiple times. This should be especially hard in a VMM-based desktop OS with ASLR (memory layout randomization) used inside each of the VM.

Another interesting attack, that we could also classify as a side-channel type of attack, is the rowhammer attack exploiting physical faults in DRAM modules [52], [53].

All these covert- and side-channel attacks do not seem to be inherent to the x86 platform, especially the rowhammer attack. Nevertheless it's worth to keep in mind that we do have these problems on Intel x86 platforms, and that there are few technologies available to effectively protect against these attacks. With an exception of the rowhammer attack perhaps, for which, it seems, the proper solution is to use DRAM modules with error correcting codes (ECC).

[4]It should be stressed the author does not consider herself an expert in side channel attacks.

Summary

We have seen that the Intel x86 platform offers pretty advanced isolation and sandboxing technologies (MMU, VT-x, VT-d) that can be used to effectively de-privilege most of the peripheral devices, including their firmware, as well as their corresponding drivers and subsystems executing as part of the host OS. This allows to build heavily decomposed systems, which do not need to trust networking or USB devices, drivers and stacks. This does require, of course, specially designed operating systems to take advantage of these technologies. Sadly most systems are not designed that way and instead follow the monolithic paradigm in which almost everything is considered *trusted*.

But one aspect still presents a serious security challenge on x86 platform: the boot security. Intel has introduced many competing and/or complementary technologies which are supposed to solve the problem of boot security: support for TPM and TXT, support for SMM sandboxing, finally Boot Guard and UEFI Secure Boot. Unfortunately, as we have seen in the first chapter, none of these technologies seem satisfactory, each introducing more potential problems than it might be solving.

Finally, the Intel Management Engine (ME) technology, which is now part of all Intel processors, stands out as very troublesome, as explained in one of the chapters above. Sadly, and most depressing, there is no option for us users to opt-out from having this on our computing devices, whether we want it or not. The author considers this as probably the biggest mistake the PC industry has got itself into she has every witnessed.

And what about AMD?

In this paper we have focused heavily on Intel-based x86 platforms. The primary reason for this is very pragmatic: the majority of the laptops on the market today use... Intel processors.

But is the situation much different on AMD-based x86 platforms? It doesn't seem so! The problems related to boot security seem to be similar to those we discussed in this paper. And it seems AMD has an equivalent of Intel ME also, just disguised as Platform Security Processor (PSP) [1].

The author, however, does not have enough 1st-hand experience with modern AMD desktop platforms to facilitate a more detailed analysis, and hopes others will analyze this platform in more detail, perhaps writing a similar paper dedicated entirely to AMD x86...

Credits

The author would like to thank the following people for reviewing the paper and providing insightful feedback: Peter Stuge, Rafał Wojtczuk, and Rop Gonggrijp.

About the Author

Joanna Rutkowska has authored or co-authored significant portion of the research discussed in the article over the timespan of the last 10 years. In 2010 she has started the Qubes OS project, which she has been leading as chief architect since then.

She can be contacted by email at: `joanna@invisiblethings.org`

Her personal master key fingerprint[5] is also provided here for additional verification:

```
ED72 7C30 6E76 6BC8 5E62  1AA6 5FA6 C3E4 D9AF BB99
```

[5] See http://blog.invisiblethings.org/keys/

References

[1] AMD TATS BIOS Development Group. AMD security and server innovation. http://www.uefi.org/sites/default/files/resources/UEFI_PlugFest_AMD_Security_and_Server_innovation_AMD_March_2013.pdf, 2013.

[2] Andrea Barisani. Internal boot ROM. USB Armory Wiki, https://github.com/inversepath/usbarmory/wiki/Internal-Boot-ROM.

[3] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, and Mickey Shkatov. Attacking and defending BIOS in 2015. Presented at ReCon Conference, http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf, 2015.

[4] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'13, pages 197–214, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Daniel J. Bernstein. Cache-timing attacks on aes. http://cr.yp.to/papers.html#cachetiming, 2005.

[6] bla. Intel ME (manageability engine) Huffman algorithm. http://io.smashthestack.org/me/, 2015.

[7] Caspar Bowden. Reflections on mistrusting trust. Presented at QCon London, http://qconlondon.com/london-2014/dl/qcon-london-2014/slides/CasparBowden_ReflectionsOnMistrustingTrustHowPolicyTechnicalPeopleUseTheTWordInOppositeSe
pdf, 2014.

[8] BSDaemon, coideloko, and D0nAnd0n. System Management Mode hack: Using SMM for "other purposes". *Phrack Magazine*, 2008.

[9] Yuriy Bulygin. Remote and local exploitation of network drivers. Presented at BlackHat USA, https://www.blackhat.com/presentations/bh-usa-07/Bulygin/Presentation/bh-usa-07-bulygin.pdf, 2007.

[10] Yuriy Bulygin, Andrew Furtak, and Oleksandr Bazhaniuk. A tale of one software bypass of Windows 8 Secure Boot. Presented at Black Hat USA, http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf, 2013.

[11] J. Butterworth, C. Kallenberg, and X. Kovah. BIOS chronomancy: Fixing the Core Root of Trust for Measurement. In *BlackHat*, 2013.

[12] Johnny Cache, H D More, and skape. Exploiting 802.11 wireless driver vulnerabilities on Windows. *Uninformed*, 6, 2007.

[13] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, August 2014. USENIX Association.

[14] Loic Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU System Management Mode to circumvent operating system security functions, 2006.

[15] Loic Duflot, Olivier Levillain, and Benjamin Morin. ACPI: Design principles and concerns. http://www.ssi.gouv.fr/uploads/IMG/pdf/article_acpi.pdf, 2009.

[16] Loic Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card? http://www.ssi.gouv.fr/uploads/IMG/pdf/csw-trustnetworkcard.pdf, 2010.

[17] Loïc Duflot. CPU bugs, CPU backdoors and consequences on security. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 580–599. Springer Berlin Heidelberg, 2008.

[18] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. Getting into the SMRAM: SMM reloaded. https://cansecwest.com/csw09/csw09-duflot.pdf, 2009.

[19] Shawn Embleton, Sherri Sparks, and Cliff Zou. Smm rootkits: A new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 11:1–11:12, New York, NY, USA, 2008. ACM.

[20] Matthew Garrett. Anti Evil Maid 2 Turbo Edition. https://mjg59.dreamwidth.org/35742.html, 2015.

[21] Google Chrome Project. Chrome EC. Chrome OS Firmware Summit, https://docs.google.com/presentation/d/1Xa_Z5SjW-soPvkugAR8__TEJFrJpzoZUa9HNR14Sjs8/pub?start=false&loop=false&delayms=3000&slide=id.p, 2014.

[22] David Grawrock. *The Intel Safer Computing Initiative*. Intel Press, 2006.

[23] David Grawrock. *Dynamics of a Trusted Platform*. Intel Press, 2009.

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[25] Intel. Intel TXT software developer's guide. http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[26] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[27] Intel. SMI Transfer Monitor (STM) user guide, revision 1.00. https://firmware.intel.com/sites/default/files/STM_User_Guide-001.pdf, 2015.

[28] Karl Janmar. FreeBSD 802.11 remote integer overflow. Presented at BlackHat Europe, https://www.blackhat.com/presentations/bh-europe-07/Eriksson-Janmar/Whitepaper/bh-eu-07-eriksson-WP.pdf, 2007.

[29] Mateusz Jurczyk. One font vulnerability to rule them all. Presented at ReCon, http://j00ru.vexillium.org/dump/recon2015.pdf, 2015.

[30] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell. Extreme privilege escalation on UEFI Windows 8 systems. Presented at Black Hat USA, https://www.blackhat.com/docs/us-14/materials/us-14-Kallenberg-Extreme-Privilege-Escalation-On-Windows8-UEFI-Systems-WP.pdf, 2014.

[31] Stefan Koch. USB interface authorization patch. Proposed patch for the Linux kernel, https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6ef2bf71764708f7c58ee9300acd8df05dbaa06f, 2015.

[32] X. Kovah and C. Kallenberg. How many million BIOSes would you like to infect? http://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf, 2015.

[33] Arvind Kumar, Purushottam Goel, and Ylian Saint-Hilaire. *Active Platform Management Demystified*. Intel Press, 2009.

[34] Inaky Perez-Gonzalez. Authorizing (or not) your USB devices to connect to the system. Part of the Linux kernel documentation, https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/usb/authorization.txt?id=refs/tags/v4.2.5, 2007.

[35] Purism. Hard, NOT soft, kill switches. Purism Blog, https://puri.sm/posts/hard-not-soft-kill-switches/, 2015. Accessed: 26/09/2015.

[36] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[37] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.

[38] Joanna Rutkowska. More thoughts on CPU backdoors. The Invisibe Things Blog, http://blog.invisiblethings.org/2009/06/01/more-thoughts-on-cpu-backdoors.html, 2009.

[39] Joanna Rutkowska. (Un)Trusting your GUI subsystem. The Invisibe Things Blog, http://blog.invisiblethings.org/2010/09/09/untrusting-your-gui-subsystem.html, 2010.

[40] Joanna Rutkowska. Anti Evil Maid. The Invisibe Things Blog, http://blog.invisiblethings.org/2011/09/07/anti-evil-maid.html, 2011.

[41] Joanna Rutkowska. Playing with Qubes networking for fun and profit. The Invisibe Things Blog, http://blog.invisiblethings.org/2011/09/28/playing-with-qubes-networking-for-fun.html, 2011.

[42] Joanna Rutkowska. USB security challenges. The Invisibe Things Blog, http://blog.invisiblethings.org/2011/05/31/usb-security-challenges.html, 2011.

[43] Joanna Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (part 1). The Invisibe Things Blog, http://blog.invisiblethings.org/2013/08/30/thoughts-on-intels-upcoming-software.html, 2013.

[44] Joanna Rutkowska. Thoughts on Intel's upcoming Software Guard Extensions (part 2). The Invisibe Things Blog, http://blog.invisiblethings.org/2013/09/23/thoughts-on-intels-upcoming-software.html, 2013.

[45] Joanna Rutkowska. A practical example of an iPhone6 deprived of most of its spying devices. https://twitter.com/rootkovska/status/547496843291410432, 2014.

[46] Joanna Rutkowska. Software compartmentalization vs. physical separation (or why Qubes OS is more than just a random collection of VMs). http://invisiblethingslab.com/resources/2014/Software_compartmentalization_vs_physical_separation.pdf, 2014.

[47] Joanna Rutkowska and Alexander Tereshkin. IsGameOver(), Anyone? Presented at Black Hat USA, http://invisiblethingslab.com/resources/bh07/IsGameOver.pdf, 2007.

[48] Joanna Rutkowska and Alexander Tereshkin. Evil Maid goes after TrueCrypt! The Invisibe Things Blog, http://blog.invisiblethings.org/2009/10/15/evil-maid-goes-after-truecrypt.html, 2009.

[49] Joanna Rutkowska and Rafal Wojtczuk. Xen 0wning Trilogy (part 2): Detecting & preventing the xen hypervisor subversions. Presented at Black Hat USA, http://invisiblethingslab.com/resources/bh08/part2-full.pdf, 2008.

[50] Joanna Rutkowska and Rafał Wojtczuk. Qubes OS architecture. http://files.qubes-os.org/files/doc/arch-spec-0.3.pdf, 2010.

[51] Anibal L. Sacco and Alfredo A. Ortega. Persistent BIOS infection. Presented at CanSecWest conference https://cansecwest.com/csw09/csw09-sacco-ortega.pdf, 2009.

[52] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. Google Project Zero Blog, http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[53] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. Presented at Black Hat conference, https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf, 2015.

[54] Igor Skochinsky. Intel ME secrets hidden code in your chipset and how to discover what exactly it does. Presented at ReCon Conference, https://recon.cx/2014/slides/Recon%202014%20Skochinsky.pdf, 2014.

[55] Angelos Stavrou and Zhaohui Wang. Exploiting smart-phone USB connectivity for fun and profit. Presented at Black Hat DC conference, https://media.blackhat.com/bh-dc-11/Stavrou-Wang/BlackHat_DC_2011_Stavrou_Zhaohui_USB_exploits-Slides.pdf, 2011.

[56] Peter Stuge. Hardening hardware and choosing a #goodbios. Presented at 30th CCC, 2013.

[57] The coreboot project. coreboot: fast and flexible open source firmware. http://coreboot.org/.

[58] The TAILS Project. Tails FAQ. Internet Archive (July 9, 2015),https://web.archive.org/web/20150709014638/https://tails.boum.org/support/faq/index.en.html#index31h2.

[59] The TAILS Project. Tails: The amnesic incognito live system. https://tails.boum.org/.

[60] The Ubuntu Project. SecureBoot wiki page. Ubuntu Wiki, https://wiki.ubuntu.com/SecurityTeam/SecureBoot.

[61] Trusted Computing Group. TPM main specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[62] S. Türpe, A. Poller, J. Steffan, J.-P. Stotz, and J. Trukenmüller. Attacking the BitLocker boot process. http://testlab.sit.fraunhofer.de/content/output/project_results/bitlocker_skimming/, 2009.

[63] Wikipedia. BitLocker Wikipedia page. https://en.wikipedia.org/wiki/BitLocker.

[64] Wikipedia. efuse. https://en.wikipedia.org/wiki/EFUSE.

[65] Rafal Wojtczuk. Xen 0wning Trilogy (part 1): Subverting the xen hypervisor. Presented at Black Hat USA, http://invisiblethingslab.com/resources/bh08/part1.pdf, 2008.

[66] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel® Trusted Execution Technology. Presented at Black Hat DC, http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf, 2009.

[67] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM memory via Intel® CPU cache poisoning. http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf, 2009.

[68] Rafal Wojtczuk and Joanna Rutkowska. Attacking intel TXT via SINIT code execution hijacking. http://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, 2011.

[69] Rafal Wojtczuk and Joanna Rutkowska. Following the white rabbit: Software attacks against Intel® VT-d. http://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf, 2011.

[70] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another way to circumvent Intel® Trusted Execution Technology: Tricking SENTER into misconfiguring VT-d via SINIT bug exploitation. http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf, 2009.

[71] Rafal Wojtczuk and Alexander Tereshkin. Attacking Intel® BIOS. Presented at Black Hat USA, http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf, 2009.

[72] Rafal Wojtczuk and Alexander Tereshkin. Introducing ring -3 rootkits. Presented at Black Hat USA, http://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf, 2009.

[73] Rafał Wojtczuk. A stitch in time saves nine: A case of multiple OS vulnerability. Presented at BlackHat USA, https://media.blackhat.com/bh-us-12/Briefings/Wojtczuk/BH_US_12_Wojtczuk_A_Stitch_In_Time_WP.pdf, 2012.

[74] Rafał Wojtczuk and Corey Kallenberg. Attacks on UEFI security, inspired by Darth Venamis's misery and Speed Racer. Presented at 31st Chaos Communication Congress, https://media.ccc.de/browse/congress/2014/31c3_-_6129_-_en_-_saal_2_-_201412282030_-_attacks_on_uefi_security_inspired_by_darth_venamis_s_misery_and_speed_racer_-_rafal_wojtczuk_-_corey_kallenberg.html#download, 2014.