State considered harmful

A proposal for a stateless laptop

Joanna Rutkowska

December 2015

Version: 1.0

Contents

Chapter 1

Introduction

Modern Intel x86-based endpoint systems, such as laptops, are plagued by a number of security-related problems. Additionally, with the recent introduction of Intel Management Engine (ME) microcontroller into *all* new Intel processors, the trustworthiness of the Intel platform has been seriously questioned.

In a recently published paper [10] the author has presented an in-depth survey of these topics. In this paper the author proposes what she believes might be a reasonable, practical and relatively simple solution to most of the problems.

The main principle introduced below is the requirement for the laptop hardware to be *stateless*, i.e. lacking any persistent storage. This includes it having no firmware-carrying flash memory chips. All the state is to be kept on an external, trusted device. This trusted device is envisioned to be of a small USB stick or SD card form factor.

This clean separation of state-carrying vs. stateless silicon is, however, only one of the requirements, itself not enough to address many of the problems discussed in the article referenced above. There are a number of additional requirements: for the endpoint (laptop) hardware, for the trusted "stick", and for the host OS. We discuss them in this paper.

The author thinks the solution proposed here is not limited to solving the Intel-specific challenges and might be useful for other future platforms also.

Those readers who can't help but think that the (Intel) x86 is an already a lost battle, and that we should be moving to other architectures, are advised to have a look at the end of the paper where such alternatives are quickly discussed, and then... potentially jump back here to continue reading.

Chapter 2

State(-carrying) considered harmful

There are several fundamental reasons why endpoint computing devices, such as laptops, without clearly defined separation of state-carrying elements are problematic:

1. The presence of persistent storage intermixed with the hardware makes it possible for the attacker to persist malware on the platform, without the user to have any simple way of learning about it, nor removing it (e.g. via OS re-installation).

2. This also allows dishonest vendors, such as the OEMs or shipping agents, to deliver already infected hardware without the user being able to easily find out. This also includes Evil Maid attacks, which might be executed by other actors than vendors.

3. The malware, once installed on the platform somehow, is given places where to store stolen secrets from the user. This is especially worrisome in the context of disk encryption keys, which could be exfiltrated this way even on air-gapped machines.[1]

4. Finally, these state-carrying elements make it possible to identify platforms and ultimately their users, due to various personally identifiable information, such as MAC addresses for WiFi or BT, which make the hardware unique.

[1] In such cases the leaked keys would become readily available to whoever has seized the laptop and has the keys to decrypt the backdoor-created data with the keys inside.

It's worth stressing that modern computer architectures make it very hard, sometimes impossible, for the user to inspect what firmware has really been programmed into the flash memory on the platform[2]. This is especially true for any so called tamper-proof chips. The use of such "secure" chips on endpoint computing devices should be avoided at all cost (see also discussion at the end of the paper). Any tamper-proof electronics on client systems should be considered harmful to the user as they jeopardize any form of transparency or verification.

[2]Indeed, by merely asking a flash-hosting device, such as the SPI flash chip, or some other u-controller such as one used on a NIC, to tell us what firmware it has inside, we can only get as trustworthy a response as is the device itself, or worse even: as is the current firmware that is used to serve our request... A classic chicken and egg problem.

Chapter 3

The Stateless Laptop

In this chapter a vision for the stateless laptop is described. The author believes the clean separation of state introduced by these modification would be attractive not just on current x86-based platforms, but also on any future platforms, be they based on ARM or any other processor architectures.

Also discussed are additional modifications needed to make the laptop more trustworthy, or to state it in a more direct way: much less of a threat to its user.

State-carrying elements on modern laptops

We start by identifying the state-carrying (persistence-carrying) elements on a modern x86 laptop. These are:

1. The SPI flash chip carrying the BIOS, ME, and other firmware.
2. The Embedded Controller (EC), which is an OEM-specific microcontroller (uC), and which requires its own flash memory, which might be either implemented inside the uC itself or as a discrete chip on the board (potentially shared with the SPI flash chip mentioned in the previous point).
3. Additional discrete devices, such as e.g. the WiFi or BT modules. Typically they would contain their own flash memories to hold their own firmware.
4. Finally, there is the hard disk.[1]

[1]The disk should not be confused with the SATA controller, which is part of the processor package on the latest Intel models.

Notably the above list does not contain the processor package, which includes the actual processor (CPU), and what was previously known as "the chipset", comprising the MCH (formerly the "northbridge"), and ICH/PCH (formerly the "southbridge"). Indeed, it seems that none of the modern processors are being equipped with flash-able memory. The reason for this seems to be resulting from the limitation of the manufacturing technology as used for modern processors. If it was otherwise, we would likely not see discrete SPI flash chips for holding of the BIOS, ME, and other firmware on notebook motherboards anymore...

Although it's worth mentioning that Intel processor packages still contain a residual form of persistent state storage: so called fuses. It's unclear to the author if it's possible for the processor itself to blow its own fuses.[2] Even if that was possible, however, it seems like the usefulness of this form of state storage would be very limited to the attacker: it could potentially only be used once, and only for storing very short secrets. Notably, it doesn't seem it could be used for platform re-infection.[3]

Because we don't have any control over the processor package, i.e. we must accept it the way it is, at least if we want to build an x86 laptop today, and also because of the limitations mentioned above, we will treat the processor package as a stateless element in the rest of this paper. Nevertheless, it would be desirable if the processor vendors used such a technology for fuses implementation, as it would not be possible for the processor itself to self-blow these.

Let us now discuss what we could potentially do about all the above mentioned state-carrying elements.

The SPI flash chip

The platform's firmware-carrying flash chip (the SPI flash as it's often called) presents the biggest challenge for us, the state-less laptop proponents. The SPI flash chip is tasked with several crucial goals on modern Intel x86 laptops:

1. It provides the firmware to the Intel ME processor. Failure to do so would, most likely, result in the platform shutdown.[4]

[2] More specifically: for either the instructions running on the host CPU, or those running on the ME processor to blow the fuses.

[3] Although it could be used to implement reply-protection for hypothetical CPU-based backdoors, as discussed in [9].

[4] While there is no clear officially statement in the Intel platform specs about this, it's considered a tribal knowledge among many experts.

2. It provides the BIOS firmware. Failure to provide a valid BIOS firmware would render the platform un-bootable.

3. It provides the firmware to some of the integrated devices, such as Ethernet network controller, potentially also other devices. Also, it might provide some of the crucial personally identifiable information, such as the MAC address(es) to be used by the networking device(s).

4. Additionally, the flash chip serves as a storage space for various persistent platform configuration settings, for both the BIOS as well as the Intel ME.

The general idea is to remove the SPI flash chip from the motherboard, and route the wiring to one of the external ports, such as either a standard SD or a USB port, or perhaps even to a custom connector[5]. A Trusted Stick (discussed in the next chapter) would be then plugged into this port before the platform boots, and would be delivering all the required firmware requested by the processor, as well as other firmware and, optionally, all the software for the platform.

One problem is that when the system wants to read the ME or the BIOS firmware, none of the devices, not even the DRAM memory is initialized at this stage. This means we cannot use e.g. a USB controller, and consequently a USB "stick" easily to provide the firmware at this stage. What we can do, however, is to reuse some of the pins in a USB port for the purpose of passing the SPI connections to our Trusted Stick. Ideally these could be multiplexed with original USB port connections, so that after the platform boot is complete, the USB port could be used as a fully featured USB port.[6]

In either case, the goal is to relocate the SPI flash element from the motherboard – where it cannot be neither properly protected (e.g. against software-based reflashing attacks, or physical Evil Maid attacks), nor reliably verified by the user. By relocating it to the Trusted Stick, we

1. provide a reliable way to enforce read-only property of the (select) firmware partitions,

2. allow the user to reliably inspect its content, perhaps using some other (more trusted) machine,

[5]While use of a custom connector might increase the cost of manufacturing of a Stateless Laptop, it might have some advantages related to usability (clear indication to the user where to plug the Trusted Stick), and messaging ("This laptop is *supposed* to be implementing the stateless laptop standard").

[6]Without dynamic multiplexing of these extra signals, we would need to downgrade a USB 3.0 port to USB 2.0, as we would likely need to use the 4 "Super Speed" signals to pipe SPI over them.

3. also allow the user to reliably write content to the stick (e.g. an image for a trustworthy BIOS the user decides to use).

The Embedded Controller's flash memory

The Embedded Controller (which should not be confused with Intel ME) is a little auxiliary, discrete, microcontroller connected through an LPC bus to the chipset. It is responsible for 1) keyboard operation, 2) thermal management, 3) battery charging control and 4) various other OEM-specific things, such as LEDs, custom switches, etc. More discussion about how this uC might be compromising platform's safety has been provided in [10].

In this paper, however, we're more concerned with the fact that the uC that is used to implement EC would typically also contain an internal flash memory (e.g. see [5]), yielding it a state-carrying element on the platform, something we would like to avoid.[7]

We thus would like to use a uC without a built-in flash-able memory, one that expects the firmware for execution to be provided by an external chip. One example of such a chip is the one used by the OLPC and Purism laptops [7].

The hard disk

The internal hard disk is an obvious device which is capable of storing the state. In fact this is the very reason disks are made.

What might be less obvious though, is that disks typically contain their own uC with their own internal flash-able memory. This naturally breaks the stateless requirement for the platform even further...

Also, due to potentially backdoored firmware, or just due to how modern solid-state disks work (wear-leveling mechanisms), some information stored directly on the disk by malware (such as the stolen user disk encryption key) might not be easy for the user to wipe using traditional disk shredding methods.

[7] Admittedly the EC, no matter how evil firmware it executes, would not be able to interfere with the platform boot sequence, and thus would not be able to compromise the system or any other software execution directly. However, as already discussed in [10], the EC might pull out a few other, more subtle attacks, such as e.g. injecting keystrokes that could trigger some actions that might also be fatal. Or, as one of the reviewers noted, might pretend the system is off when it really is not, which might be problematic e.g. when switching between supposedly separate boot environments, or trying to prevent potential Cold Boot attacks.

There are two ways how to solve the above problems:

1. Get rid of the hard disk, and rely on external storage only (perhaps also implemented on the Trusted Stick) connected e.g. through USB or SD protocols. Of course this solution is neither elegant nor convenient. Also an internal disk will always excel in terms of speed and capacity for a given cost.

2. Use an internal disk with *trusted* firmware satisfying the requirements discussed in the next section.

The trusted internal hard disk requirements

The first requirement for using an internal disk would be for it be flash-less, of course. The disk uC would need to obtain its firmware from the trusted stick, just like the EC described above is expected to do that.

Additionally, the firmware that would power the disk would need to be *trusted* (this is in contrast to e.g. the ME firmware which we do not assume to be trusted!). Trusted, to do a few things:

1. Implement reliable read-only protection for select partitions on the disk (e.g. those containing the /boot and root filesystems),

2. Implement reliable transparent encryption for anything that is ever written to the disk. In other words make it impossible (e.g. for the malware in the ME or on the host) to store anything on the disk that would not be encrypted with a *user* controllable key. This requirement has an added advantage that wiping of all the user data on the disk can be implemented by simply throwing away the encryption key, something that could be done very quickly and easily.

The above requirements demand, in practice, the disk hardware to be of open-hardware design, running open-firmware. Fortunately it appears significant work has already been made in this area [15], which should be a good starting point.

It should be reiterated here the requirement for the trusted internal disk is an optional one, and it is envisioned that meanwhile an external disk could used, ideally integrated into the Trusted Stick.

Other discrete elements

Occasionally there might be additional discrete devices on the laptop, such as a discrete GPU. Such devices will likely come with their own internal flash memory, thus breaking the stateless principle. In most cases these discrete devices would also be bus-mastering devices (capable of issuing DMA to host memory), which means they could not only be used as a secret storage, but also interfere with the platform boot process if it is not properly secured against DMA from devices.[8]

It's thus best to ensure no discrete devices are present on the laptop, especially no discrete GPUs. We talk more about the discrete wireless cards, such as WiFi and cellular modem, below.[9]

Other modifications to the laptop

In addition to removing persistent state-carrying elements from the laptop, there are also a few other minor, yet important, modifications that are needed to assure the laptop is not harmful to the user. We discuss these below.

The wireless devices

All the wireless devices (WiFi, BT, 3G/LTE modems, etc.) deserve special consideration (even if they do not have their own flash memory) because they provide a very convenient way for the malware that runs on the platform (e.g. in the ME or SMM, or even on the host OS) to leak information using a wireless channel (so, a channel very difficult to block or notice). This could happen irrespectively of whether the user decided to consciously enable and use the actual device or not (e.g. turned on WiFi in the host OS and connected to a WiFi network).

Additionally any wireless device could be used to gather information about the user surroundings, such as e.g. the list of active WiFi networks (SSIDs) or BT devices MAC addresses.

[8] The author is not aware of any BIOS implementation that would actively try to protect itself against potential DMA attacks originating from devices during boot, especially early boot.

[9] One of the reviewers also pointed to battery as a potential element containing embedded microcontroller with its own flash memory. Needless to say such "smart batteries" should be avoided and all the charging/monitoring logic implemented by the EC, or using "dumb" electronics without persistent state storage.

Admittedly though, most such exfiltration channels would require the attacker to be physically close to the user's laptop, so for some of the users this might not be a realistic threat[10]. Notably with one exception – if the malware managed to interpose on legitimate traffic generated by the user, e.g. by finding and modifying network buffers in the host memory, it might then easily leak the stolen secrets at least to the user's ISP, or with some luck, to whatever server on the Internet the user chose to establish connection with. We discuss this problem as well as potential countermeasures later in this paper.

Similarly not every user would be concerned about their physical location being leaked (through the information sniffed by the wireless devices). But for those who care, a mechanism is needed to prevent this from happening.

The easiest way to address all the above mentioned problems is to fit a physical kill switch for each (or all) of the wireless devices. Care should be taken for the switch(es) to control the actual power supply wires to the devices, rather than merely asking the devices to disable themselves, a request which a malicious device (or one with an infected firmware) might simply ignore.

Of course physical kill switches are not an elegant solution, as in most cases the user would like to have some form of wireless connectivity. After all there is a reason we want to have these networking devices in the first place... As mentioned we will consider this problem in more details later in this paper. For now suffice to say that it would be beneficial to either: 1) not have any internal WiFi or BT card, or 2) a simple networking proxy implemented on an external (trusted) uC, not directly connected to the host processor.

It should be pointed out for completeness, that a GPS receiver (if fitted to the laptop), while a one-way radio device, should also be fitted with a kill switch, for the reasons discussed above.

The audio and camera devices

The audio and video (camera) devices can compromise user's privacy similarly to the above discussed wireless devices. In addition to the obvious threats posed by these devices, it's perhaps worth mentioning a possibility of using the mic and the camera not only to sniff the conversations in the room where the laptop is kept, but also to allow the attacker to sniff the user's disk and login password. Also, it seems possible, in theory at least, for the malware to use the speakers to

[10] Although the adversary might use e.g. the user's phone as a relaying device

communicate with other devices (such as the user's phone or even an internet-connected TV) in order to exfiltrate some low-bandwidth information (e.g. the disk decryption key stolen from the host DRAM page or registers).

For this reason it seems only reasonably to put all the audio and video devices behind physical kill switches, just like it was recommended for all the wireless ones. Again it should be stressed the physical switches should be cutting the actual power or signal lines to the devices, accounting for potentially misbehaving ones.

Volatile memory quick wiping

Finally, one additional aspect of building a stateless laptop is to account for all the state accumulated in the *volatile* memory, specifically DRAM and the processor internal SRAM used by the ME. Even though we're talking about volatile memory, it's a well know fact that residual information might remain there for a surprisingly long time [6]. Additionally, the ME internal memory (SRAM) is believed to remain to be sustained despite platform normal shutdown state, as the ME is still in operation, albeit it might be in sleep mode (again, the platform does not need to be in e.g. S3 for this).

Thus a mechanism is needed to ensure, upon user's request, a reliable and quick clearing of all the volatile memories fitted on the platform. This might be the default behaviour every time the platform is going to be shutdown for hibernation. One of the reviewers suggested short-circuiting Vcc with GND pins might do the trick for the processor and DRAM.

Putting it all together

The diagram below wraps up our discussion so far:

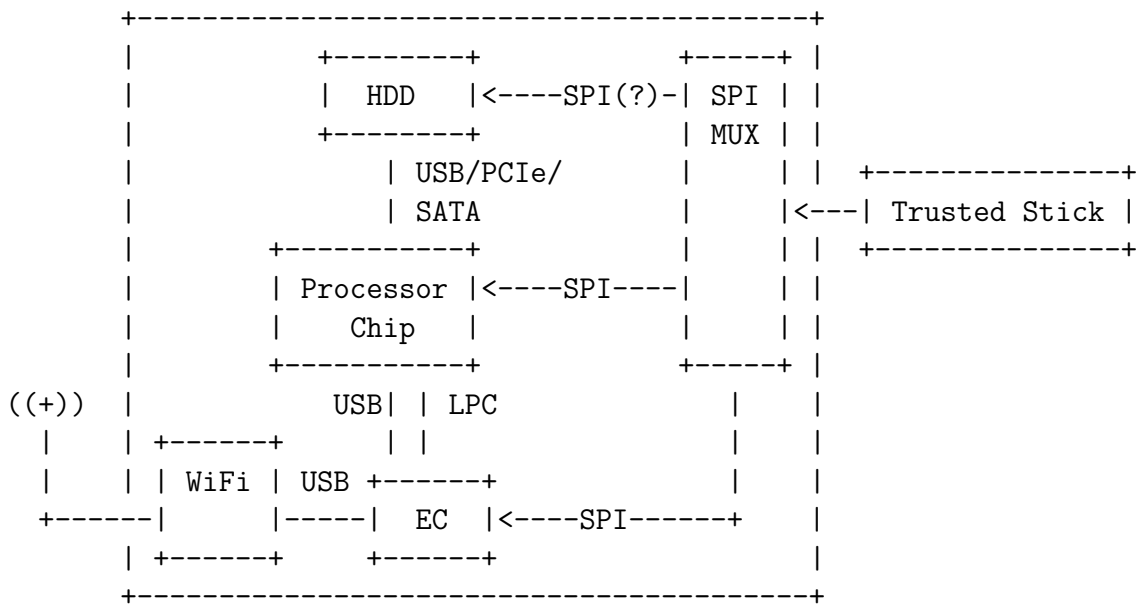

Fig 1. The stateless laptop with trusted internal disk and network proxy.

A new element introduced on the above diagram is the "SPI MUX" box. It's a multiplexer for the SPI buses coming from different devices, which are normally expecting to be the only ones talking to the SPI flash chip. It should be possible to implement this using an FPGA (for a prototype) or an inexpensive ASIC (for production models).

The next diagram shows a simplified laptop: without trusted internal disk (implemented according to the requirements laid out above), and also without networking proxy (as discussed later), but rather using an external USB-connected WiFi device.

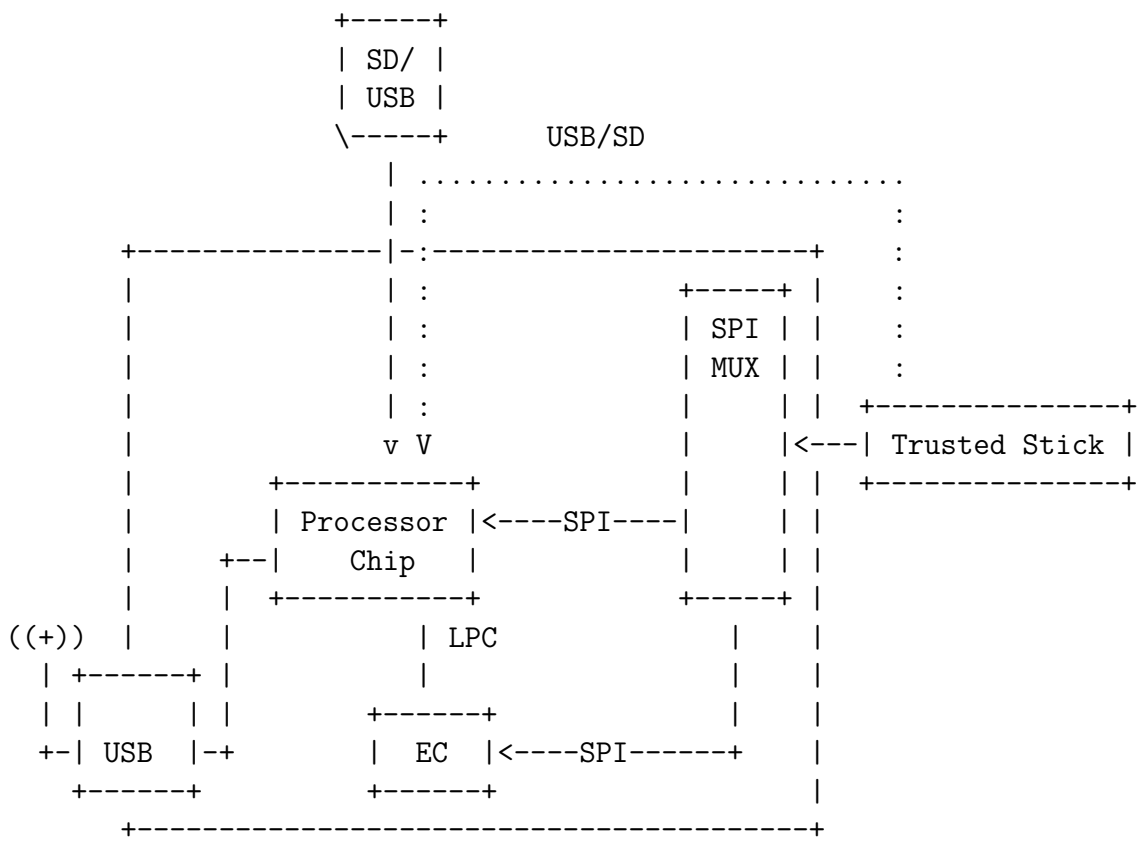

Fig 2. The simplified stateless laptop (no internal disk, USB WiFI)

It might also be possible to use a microcontroller with one-time-programmable (OTP) memory in order to avoid the need to do the SPI multiplexing, and so to further simplify the construction of the laptop and the Trusted Stick. While an OTP would not provide state persistence, it would still be a sub-optimal solution because the user would (likely) not be able to inspect the firmware, or load one the they want.

Chapter 4

The Trusted Stick

The Trusted Stick, a small device of a "USB stick" or an SD card form factor, is an element that the user always carries with themselves and which contains all the "state" for the platform. This includes the (encrypted) user files and platform configuration. It also is expected to carry all the software and – what is unique as of today – firmware for the platform, and also enforce read-only'iness of these.[1]

As the name suggests, it is assumed the device is to be *trusted*. In other words, should this device malfunction (due to a bug in its own firmware), or get compromised by the attacker somehow, the security of the user data is in jeopardy.

It is thus expected this device should be as simple as possible to assure it's reasonably secure, and also to make it possible for various vendors, ideally by users themselves, to be able to build it. It goes without saying the device should be an open-source, open-hardware device. The author believes there is no excuse for entrusting proprietary products with such important things as ones digital life.

We are now considering what functionality should the Trusted Stick implement.

Firmware Storage

First of all it should provide read-only (from the host perspective at least) storage for all the platform firmware. This includes the Intel ME, the BIOS (including any blobs it might depend on, such as the FSP, ACMs, etc.), any of the standard

[1] A mechanism for updating the software and firmware on the stick should be explicitly under the control of the user. One can easily imagine this to be implemented using a physical switch on the stick, i.e. something that software could not be able to interfere with.

integrated devices firmware (e.g. the GbE firmware), as well as firmware for the OEM-specific Embedded Controller, and potentially other devices, such as the already discussed (optional) internal disk, and perhaps any discrete networking devices.

The above (read-only) firmware storage should cover also any platform configuration. Typically the BIOS, ME, and potentially other devices would want to use some parts of the flash partitions to store their own configuration (e.g. which devices to boot from, the MAC address, etc).

It should be stressed that all this firmware should be exposed to the platform (e.g. to the host processor or the EC u-controller) using the standard protocols that would normally be used to fetch the firmware. In most cases this is the SPI protocol.

Disk Storage

In addition to playing the role of a firmware storage (in practice: an SPI flash device), the Trusted Stick might also act as a normal mass storage device, seen by the host as e.g. a USB mass storage device, or an SD card.

Here we should further distinguish between two types of storage that is going to be exposed to the platform (the same applies also in the scenario with an internal trusted disk):

1. A read-only non-encrypted storage containing the system code (i.e. the bootloader, the boot partition, and the root filesystem),
2. A writeable (but encrypted) partition for the user files (i.e. the home directory and perhaps some additional system configuration). The key for the encryption could be derived from: 1) the user provided passphrase (provided via keyboard), optionally combined with: 2) a TPM-released secret which can be used, to some extent, to prevent laptop-replacing Evil Maid attacks (which we discuss at the end of this paper in more detail), 3) and also a secret generated by the Trusted Stick and subject to wiping in case the user requested secure deletion of all user-specific data.

It should be noted that it might not be possible to obtain the user passphrase using the standard keyboard during early phase of the platform boot. It is not expected this to be necessary because all the early boot firmware should not be encrypted, but only read-only protected by the Trusted Stick. However, in case

it turned out that e.g. Intel ME refused to run having only read-only access to its flash partition, then we might need to encrypt the flash partitions on the Trusted Stick holding this early boot firmware. More on this at the end of the paper.

An alert user might be wondering what a TPM device is doing on a stateless laptop? After all the TPM has its own non-volatile memory, doesn't it? Interestingly on the recent Intel platforms the TPM has been integrated into the processor package (it's in fact an application running on the ME processor), and so it uses the system's SPI flash memory as its own non-volatile storage. Of course everything that is written there is encrypted with a key that is tamper-proof protected inside the processor, so the mere fact the attacker is able to read the SPI flash content with an external programmer does not compromise safety of this TPM's storage. While it hasn't been confirmed experimentally if such a processor-internal TPM would work with a read-only storage exposed by the Trusted Stick, it seems plausible to expect it should[2]. Of course the user would be expected to let the TPM write its generated keys during the platform initialization, by operating the read-protect switch on the Trusted Stick.

The variant with internal trusted disk

As already discussed earlier, assuming a trusted, open implementation of an internal hard disk was available, then the stick would not need to act as (fast) storage. It would only have to provide the decryption key to the (trusted) internal disk device.[3]

The primary benefit in this case would be the simplification of the stick: no need to fit high-capacity, high-performance flash memory. Depending on the application this could be an important benefit.

Self-destruct

Optionally, at least for some groups of users, it might be desirable for the Trusted Stick to implement quick and reliable wiping of its content, especially of the user partition.[4] This should be easily implemented by securely erasing just the

[2]And in case it didn't work with a read-only flash, we might still be able to use it with an encrypted writeable flash, as discussed later in the paper

[3]Potentially it might also be providing the /boot partition, although the benefit of this is unclear.

[4]Although, there might be scenarios extending this requirement also for other partitions, i.e. these holding the firmware and system image.

encryption key, for which even a small battery or perhaps even a capacitor should be enough.

Chapter 5

Addressing leaks through networking

Assuming the platform might be compromised with sophisticated rootkits, e.g. running in SMM or ME, that are actively trying to steal e.g. GPG private keys from the host memory, it is important to ensure the malware cannot leak the data using networking. It should be realized that for malware running in ME or SMM it might be possible to leak data using networking irrespectively of what specific networking hardware is in use by the host OS. It should be just enough for the malware to (asynchronously) find pages containing what looks like specific data structures (e.g. Linux sk_buff structures) and modify just a few fields there in order to implement some form of covert channel for exfiltration (see e.g.[8]).

On the other hand, such advanced malware (e.g. especially when running in the ME) might be reluctant to (somehow blindly) modify outgoing networking packets without fully understanding the bigger picture of the specifics of the environment and the user setup. This is because such modifications might easily be detected by more inquisitive users or admins, using more or less standard network analysis tools, risking detection of the malware. Again, for malware located that deep in the hardware, in the processor itself, this might not be acceptable. Nevertheless, let's discuss what we could do to prevent such leaks anyway. We will do that starting from the simplified scenario of an air-gapped system, then move on to increasingly more connected scenarios.

Scenario 0: An air-gapped system (no network)

Contrary to what it might seem at first sight, the mere fact that we are keeping the laptop not connected to any network does not automatically make it a truly air-gapped system! If there is malware on the laptop it can still establish communication with the outside world through a number of channels: it might use the existing WiFi or BT, or LTE/3G devices to send packets to other attacker-controlled devices[1], ostensibly without connecting to any network. It might even use more exotic means of establishing covert channels, such as the audio spectrum using the built-in speakers, as mentioned previously in this document.

Also, even if the system is not yet compromised (i.e. no malware or backdoors running on it yet), it might get compromised when devices such as WiFi or BT are exposed to the environment and are processing the (untrusted) input "from the air" around the laptop.[2]

Thus to keep the laptop truly air-gapped one must ensure access to all these devices is forbidden, and not just to the host OS, but also to any of the hardware on the platform, including the processor. The physical kill switches seem to be a reliable way for guaranteeing this, as discussed previously. Obviously, assuming such kill switches have been fitted (and set to the "off" positions), and assuming that the stateless laptop is indeed lacking any persistent memory, and that even if the ME (or any other rootkit) managed to steal any of the user data, it would not be able to leak them anyway.[3]

A truly world-disconnected computer is of very limited use, however. In practice we would like to transfer some files from/to such an air-gapped system. One popular approach is to use a USB storage device (stick) for that purpose. Such an approach, however, exposes the air-gapped computer to potential infections when its host OS is processing the device, volume, and filesystem metadata brought by this device. Additionally, and more importantly, a potential backdoor, e.g. in the ME, might now dump all the previously stolen data onto the stick (and these blobs might now not be easy discoverable by the user, thanks to e.g. the wear levelling mechanisms used on the stick, or potentially backdoored firmware on the said USB device).

A better approach is to use physically read-only media, such as DVD-R. While such a medium can still bring infection to the air-gapped system, it wouldn't be

[1] Which might be the user phone or Smart TV, for example.

[2] This is especially true if the host OS does not explicitly try to sandbox the devices, drivers, and corresponding stacks, which is often the case.

[3] One reviewer pointed the malware might try to e.g. modulate CPU usage, thus indirectly trying to leaking the data via electro-magnetic field...

possible to use it to exfiltrate the stolen data[4]. Of course, this would result in a "black-hole" use model – the air-gapped system can only accept files from the outside world, but never give anything back to the universe – again, possibly a sub-optimal use of computer technology...

Scenario 1: A closed network of trusted peers

Now, let's consider a closed network of trusted peers who would like to communicate securely with each other, also exchanging files.[5] Of course the humankind has researched this problem extensively over the last couple of decades, which resulted in an abundance of cryptographic protocols allowing to build secure tunnels over insecure networks.

However, assuming a rootkit running in the ME or SMM, we're suddenly facing a significantly more difficult challenge. This is because the ME might be now piggybacking stolen information (such as the session keys for the crypto tunnels we're trying to build) on the existing network packets, allowing an adversary – who e.g. controls the user's ISP – to receive them on a plate.

In order to prevent this from happening we need to move the actual networking device away from the jurisdiction of the ME and the host processor. It seems convenient, at first thought, to place the networking device on the Trusted Stick. Indeed, if the trusted module was implemented as a USB-pluggable device then it would be able to provide emulated Ethernet device to the host. The Trusted Stick would then perform simple tunneling to establish the virtual trusted network with other peers (hopefully using also similarly designed laptops). This way, even if a hypothetical ME rootkit was trying to leak some information over networking, this would get encapsulated into the encrypted tunnel, which only the trusted peers were able to see.[6]

Implementing Ethernet-emulation and networking proxy on the Trusted Stick has several disadvantages though:

1. It complicates the Trusted Stick design, increasing its cost, as well as its

[4] Although one should remember the DVD-R driver will likely be fitted with its own uC featuring its own flash memory, which might be a good candidate for malware to store stolen secrets to.

[5] Again, this means, by definition, that any of these "trusted peers" is able to compromise the whole network.

[6] Admittedly, as several reviewers noted, the rootkit might try to leak the stolen keys by interfering with the timings of packet transmissions, or using some other sophisticated side-channel attack...

size (which is an important factor given the stick is assumed to be carried by the user with themselves all the time, perhaps in a form of a necklace, or maybe even a ring in the future).

2. Even more importantly: it significantly enlarges the attack surface on this trusted device. Admittedly the uC used for networking proxy implementation might be a physically different one than the chip used for SPI firmware exposure, although this would now complicate the host-stick interface, in addition to further increasing the cost and size.

However, just as we discussed the use of a stateless internal disk (which runs a trusted firmware from the stick), we could similarly envision a simple networking proxy implemented using a stateless (i.e. flash-less) uC, which would then connect to a traditional WiFi card. The WiFi, however, would *not* be directly connected to the host CPU.

Incidentally we have already outlined the need for a stateless uC on the laptop – this is to implement the Embedded Controller. It seems thus logical to use this same uC for both realization of the EC as well as for the (trusted) networking proxy.

Obviously it would take time to write firmware implementing the envisioned proxy, and before this one is ready, a temporary solution could be to use an external, USB-connected or Ethernet-connected network proxy (similar in nature to e.g. [13]).

Scenario 2: Tor-ified or VPN-ed open Internet

Let's now consider the traditional scenario in which the user wants to interact with any computer on the Internet, whether trusted or not.

In this scenario we would also like to use the previously discussed networking tunneling proxy. Of course at some point the tunnel would need to be terminated and the user connection will now be visible to some 3rd party Internet infrastructure, including the final 3rd party server (e.g. a cat-photo-serving website the user might be addicted to). The termination of the tunnel would take place at a VPN service provider (which we assume to be a trusted service provider for the user), or at a Tor exit node (which itself is not assumed to be trusted, but the Tor network, as a whole, should be in that case).

Now, assuming the malware has modified the content of the user-generated packets high enough (OSI-layer wise), such as e.g. modified some of the HTTP(s)

headers or data payloads, the 3rd party infrastructure or the final server would be able to read any potentially covertly transmitted data from the compromised machine.

But the attacker (who controls the cat-pictures service server), even though receiving some of the user sensitive data, e.g. disk encryption key, might not be able to figure out which user do they belong to. Of course the user might have plenty of identifiable information on their laptop, and the malware might be smart enough to search around for them and include them with the blobs sent over the covert channel. Theoretically, if the user was careful enough this might not be the case, but in reality expecting the user to be so careful with regards to *all* of the activity performed on their laptop, might be unacceptable for most users.

Typically users would be willing to be careful only with regards to some of the *domains*, while would like to "live a normal life" in others. Operating systems such as Qubes OS [16] try to resolve this problem by using Virtual Machine-based compartmentalization. Sadly in case of malware operating in the ME or SMM[7] the Virtual Machine technology (even augmented by technology such as Intel VT-x and VT-d) is of little help.[8]

On the other hand, forcing the attacker's malware to modify only high-level protocol payloads to leak data might already be considered a significant win. The higher protocol the attacker needs to intercept, the higher the complexity of the malware, which increases the probability of getting caught by curious users or administrators.

In addition the attacker has little control over which servers or infrastructure she should control in order to be able to receive stolen data from a given user.

Scenario 3: Unconstrained Internet access?

Not every user would like to forward all their networking through Tor or even a fast VPN gateway. The primary reason not to do that might be the limitation on the bandwidth and latency imposed by such proxies.

[7] Although systems that properly use compartmentalization might make it very hard for the SMM to ever get infected. On the other hand, they can do nothing against the backdoors built in by vendors right from start.

[8] Admittedly Intel VT-x allows for SMM sandboxing using Dual Monitor Mode, although in practice there seem to be lots of problems with this approach, as the author has discussed in [10].

A user might typically want to use such proxies for only some of their activities (say to follow the news surrounding anti-government protests), while still enjoying "un-handicapped" Internet for other activities (such as watching full HD cat movies).

The problem with such an approach, again, is that the potential malware might choose to piggyback the stolen information onto the innocent traffic.

About the only one left solution here would be to keep an eye on the traffic generated by the user. The adversary knowing that the user might be closely monitoring their traffic should be reluctant to (somehow blindly) piggyback a covert channel on top of it, afraid of getting caught. Thus, it would seem more reasonable for the adversary to target higher-level protocols also in this scenario, facing also the same problems as discussed in the previous section.

Chapter 6

(Un)trusting firmware and the host OS

Firmware considerations

We would like to treat most of the platform firmware as untrusted. This applies to the Intel ME, other devices, and the BIOS. While it should be obvious why Intel ME should be considered untrusted, it's also prudent to treat the BIOS as untrusted even if we decided to use an open-source implementation, such as coreboot [14]. This is because the task of creating a truly secure, i.e. attack-resistant, BIOS implementation for Intel x86 platform seems like a very challenging task. Not to mention that it is currently very difficult (impossible?) to have a truly open source BIOS which would not need to execute Intel-provided blobs such as the Intel FSP.

The trick of keeping the platform's firmware on the trusted stick is a game-changer here, because we can be reasonably confident the stick will: 1) implement proper read-only protection, this way stopping any potential flash-persisting attacks originating from the platform, and 2) even if the firmware was to be somehow malicious, the construction of our stateless laptop leaves no places for the malware to store any data stolen from the user. (It could still try to leak it through networking, a problem we discussed in more detail in the previous chapter.)

There are two important exceptions with regard to trusting the firmware though:

1. If we decided to use an internal disk, as discussed earlier, then we would need to trust the disk's firmware to properly implement encryption, and read-only protection for select sectors/partitions,

2. If we decided to use the Embedded Controller (again, let's not confuse this with the Intel ME) to implement internal networking proxy (as discussed below), then we would need to trust its firmware also.

Of course, as already discussed, both of these devices would be fetching the firmware from the Trusted Stick.

Host OS considerations

It's tempting to also assume the host OS could be treated as untrusted, using the similar argumentation we just used to convince ourselves we didn't need to trust Intel ME or the BIOS...

Indeed, at least for the networking scenarios #0 (air-gap) and #1 (closed network of trusted peers), as discussed in the previous chapter, that might indeed be a justified assumption.

However, for the more open networking scenarios #2 and #3, this might no longer be the case. Indeed, an insecure OS might allow malware infections that could now use all the convenience of a locally-executing program to steal user data, collect additional personal identifying information, and exfiltrate all this to some remote server using one of the million of ways how modern malware typically would do that. This would naturally lower the bar for the adversary significantly, almost negating the benefits of a stateless laptop...

This means it is still prudent to run a secure OS on the stateless laptop.

Reconsidering BIOS and ME (un)trusting?

An alert user might, however, now point out that we cannot assume the host OS to provide any security if we don't trust the BIOS, or ME. In theory this is true, of course. In practice, however, we should consider how a malicious ME or BIOS could potentially inject malware into our (otherwise secure) host OS.

The only way for such an infection to occur would be either for the Intel ME, or the BIOS, to inject malware into the host memory. In practice this means that Intel would release a processor which, under certain circumstances (yet not depending on any persistent state) writes malware to the host memory pages.

Alternatively this might have been done by the Intel FSP blob.[1].

The author believe such a move would be extremely risky for a vendor like Intel. Again, we should remember that such malware insertion (by either the processor or FSP blob) could not be conditioned on any persistent state, and so would be subject to reply "attack". In other words, once the processor or the FSP got caught while pulling this off, it should be possible for the user to reproduce and demonstrate this malicious behaviour arbitrary number of times subsequently.

Of course, Intel ME, or a malicious SMM, instead of injecting malware into the host memory, might chose a more subtle approach and instead only expose a privilege escalation backdoor which could then be used by some malware to undermine security isolation offered by the host OS.[2]

Again, by using a largely open source BIOS implementation we can practically rule out such a backdoor in an SMM[3]. This leaves us with the possibility of the Intel ME providing this hidden escalation trap. That, however, is something that a processor vendor might always do *trivially*, without introducing technology such as Intel ME, as discussed e.g. in [9]. In that case, again, our only hope is that Intel would not risk being caught red-handed, given the hypothetical backdoor would need to be *stateless*.

We thus see that, while we cannot fully eliminate the problem of subversion of the host OS security by potentially malicious processor, the construction of the stateless laptop allows us to force the adversary into a very dangerous territory, requiring them to take high risk and also making the attack very complex.

It's worth nothing, however, how we have silently started assuming that we need to have a largely open source BIOS (so largely trustworthy), even on our stateless laptop. Needless to say, the coreboot project [14] is a natural candidate for such a BIOS, and we are very lucky there is such a project in the wild already.

[1] Here we assume a mostly open source BIOS has been used. Such a BIOS will still likely need to execute the Intel FSP blob, and this blob would be the only place which might inject the malware

[2] E.g. the backdoor might allow to escape a virtual machine, allowing some more-or-less standard malware which came through some standard channels, such as an email attachment, and which would otherwise be contained to some untrusted VM, to now spread over the whole system.

[3] Indeed, it's hardly imaginable for the FSP blob to bring such a backdoor into the SMM.

Chapter 7

Addressing Evil Maid Attacks

Originally the term Evil Maid Attack [11] was used to describe attacks on the full disk encryption schemes. In such scenarios the attacker (the Evil Maid) was replacing or infecting part of the code which was asking the user for the disk decryption passphrase. Once the passphrase was obtained from the unsuspecting user (who thought they provided it to the legitimate system software), the malicious code could have store it somewhere (e.g. save on unused disk sectors), or leak through networking, allowing the attacker to decrypt the laptop once the attacker somehow got access to it subsequently (e.g. after physically stealing it from the user, or perhaps covertly making a copy of the hard disk).

But the old Evil Maid Attack concept can be easily generalized and applied to the stateless laptop scenario. Now the Evil Maid would be replacing the whole laptop, rather than just the software on it (because there is no software to be replaced on the laptop in this case, of course). The new, fake, laptop would look identical to the user from the outside, but might be a completely different machine on the inside. E.g. it might be full of persistent memory, and also feature an army of wireless devices to leak all the user secrets to everybody in a radius of miles.

A special case of such an Evil Maid attack would be when the laptop was replaced during shipment, or simply if the vendor of the laptop turned out to be (or was forced to be) malicious.

What could we do about such attacks?

First, we should stress the primary reason behind introducing the stateless laptop idea is *not* to prevent sophisticated physical attacks, such as "full" Evil Maid attacks which replace the whole laptop with an identically-looking one.

Having said that, the author is of the opinion that the stateless laptop design

makes lots of physical attacks difficult, or simply not feasible. This applies to the "classic" Evil Maid attacks, as well as various attacks targeting the firmware.

Still, in order to somehow address (or increase the cost significantly) of the full laptop-replacing Evil Maid attacks, one can think of several solutions which include traditional physical-based protection applied to the laptop, when it is being left unattended by the user. These are things such as custom, personalized stickers, which make it more difficult to bring an identically looking laptop, as well as more classic means in a form of a vault or strong box, or a monitoring camera.

An inquisitive reader might wonder why would we need all this hassle with stateless laptops, if the user was expected to implement the physical protection, anyway? As already mentioned several times in this paper, there are many more problems with x86 platform, and which we try to resolve with the stateless laptop, than just the physical attacks. Such other problems include: software attacks on firmware, malicious firmware (backdoored by the vendor, or somewhere during the shipment), software attacks against secure boot mechanisms. A reader is, again, directed to the [10] for a more complete discussion.

The physical protections mentioned above do not, however, resolve the problem of the attackers subverting the laptop hardware at manufacturing or shipment stages. This includes, naturally, a potentially conspiring laptop vendor.

In order to address this latter problem we – the industry – need to come up with reliable and simple methods for comparing PCBs with each other. A tool analogical to 'diff', only working for PCBs rather than on files. Such a tool, implemented as a software, could e.g. take two (sets of) photos taken by the user of the two boards to compare. The photos might be taken with an ordinary camera, or, in a more sophisticated setup, using X-ray imaging to reveal also the internal layer wiring. This inititive has already been proposed by other researchers recently (e.g. [3]), so it is not unreasonable to expect some progress in this area in the near future.

Admittedly such an approach would not be able to detect sophisticated attacks which replace the original laptop board with identically looking one (connection- and chip-geometry-wise), yet with different chips. The author thinks that such attacks might be very difficult to pull off in practice, probably extremely pricey due to the need of manufacturing small series of custom integrated circuits.

Chapter 8

Select implementation considerations

Here we briefly list some of the potential challenges and some other aspects that are still left open for further discussion and research.

SPI Flash emulation challenges

One anticipated complication for emulation of the SPI flash by the trusted stick is that the processor (chipset) expects the specific timings to be met by the SPI chip when reading firmware, so it's unlikely one could use a general-purpose uC on the stick to emulate the flash chip. Also the timing requirements make it unlikely that a regular SD storage card will work for us here.[1] Rather, we need a real SPI flash chip located on the trusted stick, or better: an FPGA-based implementation.[2]

Also it does not seem trivial to use the same one SPI chip to both serve the firmware (i.e. ME, BIOS, other) to the host processor, and at the same time to also act as a flash provider to the EC, and optionally also to the internal disk. The primary reason for this might be lack of a good multiplexing mechanism built into the SPI protocol. This seems, however, merely a technical complication that, in the worst case, could be resolved by having the Trusted Stick exposing two

[1] Which otherwise sounds like a great solution, at least for prototyping, as most of these cards should be implementing the simple SPI protocol.

[2] The reason to use an FPGA-based implementation of an SPI flash is transparency, required to assure that our Trusted Stick indeed implements read-only protection for certain parts of the flash, as well as reliable encryption for other partitions, as discussed earlier in the paper.

separate SPI interfaces: one for the host processor, another to the EC uC. Of course, such an approach is far from ideal, as it increases the amount of signals required for the port to which the Trusted Stick is inserted.[3] As mentioned earlier, a temporary solution might be to use a uC with OTP memory for firmware storage.

It's also not yet clear if the Intel ME (which is part of the processor) would be happy when being put into an environment where the SPI flash it gets access to is externally forced to be read-only. Should this be the case, it might be necessary for the Trusted Stick to allow selective write-access for the ME partition accesses. In that case this region should be encrypted by the Trusted Stick, as already discussed earlier. This is to assure that in case the processor wanted to store some user-compromising secrets there, these secrets would not fall into the hands of an adversary. While this solution might seem simple enough, a slight complication might arise from the inability to ask the user for a passphrase (at least using the standard keyboard) upon early platform boot. In that case we would likely need to use a key kept on the Trusted Stick which is not conditioned on user passphrase to protect these partitions. It might be even possible to use auto-generated, discard-able keys for this purpose. Further research is needed.

Host OS implementation considerations

As previously noted the host OS should be engineered so that it was able to boot and operate efficiently from read-only storage. This is generally not a problem today: many Linux distributions support such a mode of operation (LiveUSB). It does however present some challenges for systems which aggressively try to decompose their TCB, such as Qubes OS [16]. Such systems would like to keep all the USB subsystem, drivers, and devices into separate de-privileged domains (VMs). In order to keep such USB-hosting domain(s) truly untrusted, while at the same time use it as a provider (backend) for the system root storage, special additional mechanisms would have to be used [12]. This complication could be avoided, however, when an internal trusted disk was used on the stateless laptop.

[3] And we would like to keep these down to a minimum in order to be able to re-use existing USB or SD ports.

User partition encryption considerations

It seems tempting to delegate the user partition encryption to the host – after all it runs the user approved trusted code from the stick's read-only partition, while at the same time this simplifies the construction of the stick significantly.

Unfortunately, running the encryption on the host processor we're exposing it to potential malicious interference from the ME processor. The ME can e.g. steal the encryption key from the host registers or memory pages and then try to leak it through some of the user networking activity, although this might be very difficult in practice as discussed earlier in the paper. What the ME can do, however, and very simply, is to store some of the leaked user sensitive information (such as the email private keys) on the user private partition *without* encrypting them with the user key, but rather with some other key. This would then look like random garbage for the user, if they ever decided to examine the sectors on the partition. But for the attacker who (physically) obtains access to the user stick this might be immediately readable.

On the other hand, if it was the Trusted Stick that performed the encryption, then there should be no way for the hypothetical ME rootkit to write anything onto the user partition bypassing the forced encryption with the user key.

Temper-resistance considerations

The use of tamper-resistance technology is often thought as a beneficial means to improve physical security of an endpoint device. Care must be applied however as to whether this does not compromise the ultimate trustworthiness of the product.

In the author's opinion it is unacceptable for any *code*, that the user is forced to entrust their digital life to, to be tamper-proof-protected if that results in an inability for the user to dump and analyze the code that runs on the device at any time the user feels a need to do that.[4]

Thus a temper-proof mechanism might only be acceptable for the actual (small) persistent memory which holds the bits of the user keys, and for nothing more, particularly not for the memory which holds the firmware for the device. Also, any tamper-proof protection on volatile memory (RAM) is not necessary, as such protection only makes sense if the threat model assumes the legitimate user to

[4] And it is completely irrelevant whether the user would, in practice, be willing or capable to do that or not – it's a matter of having an *opportunity* to do that. This is very similar to guarantees of civil liberties, such as free speech.

be a potential attacker. This admittedly is the case for various Digital Rights Management (DRM) or payment processing systems. For these systems the end user is considered a potential enemy, who might want to illegally make a copy of a movie, or clone credit card information. Indeed, only then the device would like to protect its *runtime* processing. Otherwise an attacker who managed to steal the device would not be able to get it to start doing the processing of sensitive data in its RAM, without providing a proper unlock password or key in the first place. It's worrying that the industry has been aggressively advertising various DRM-friendly technologies as protecting the user, while in fact they have an opposite effect, degrading trustworthiness of the user devices (from the user point of view, that is).

An exception would be a tamper-proof design which allowed for reliable read-only access for all the firmware (and preventing access only to key-holding storage), but it seems like existing devices (specifically microcontrollers) do not support such a mode today, at least the author is not aware of any.

Chapter 9

Alternative solutions?

Many people voice concerns that perhaps a much better strategy is to ditch the (Intel) x86 platform, and look for an alternative architecture as a foundation for secure and trustworthy personal computers... In this chapter we quickly review what options we might have, in practice, here.

ARM-based platforms?

The ARM architecture [17] seems like a natural candidate to replace x86 for desktop computers, including laptops. Indeed it has already dominated the smartphone and tablet markets, and it doesn't seem like the gap in performance is that great between these devices. This indeed might seem like a plausible direction at first sight, but there are at least two problems here:

First, there is no such thing as an "ARM processor" – rather ARM releases only a set of specifications and other IP, which are then licensed by various vendors, such as NVIDIA, Samsung, Texas Instruments, and so forth. These vendors then combine the licensed ARM IP with their own, creating unique final products: the actual processors, customary called System-on-Chips (SoCs).

This large diversity of "ARM processors", while undoubtedly beneficial in some aspects, is also problematic – e.g. it presents multiple research targets for security researchers, as well as for system architects and developers. E.g. some of the SoCs would implement IOMMU functionality adhering to the ARM-published specification, while others would use a completely different technology, invented by the OEM that makes the SoC [4].

Also, most of the ARM-based SoC's implement a so called TrustZone (TZ) extension. Of course, as with most technologies on ARM, TZ is just a specification and not malicious in itself. However, it opens a possibility for the vendors who produce TZ-compatible SoCs (which most do) to lock down their processor so that their TZ implementation will not differ significantly from Intel ME.

Also, there is nothing special in ARM-based architecture that could prevent a vendor from introducing backdoors into the SoCs they produce.

FPGA-based, true open source processors and platforms?

There are also efforts to create a fully open processor design ([1], [2]). This surely is the proper way to go for our civilization, long term. The important question is how much time it would take for such processors to become performant enough for typical desktop workflows (e.g. watching HD movies, running modern Web browsers or an office suite)?

But performance is only part of the story – another question relates to security technologies these processors should be offering? Technologies such as e.g. IOMMU and potentially also CPU and memory virtualization?[1]

Sadly, it seems like we're at least years away from having consumer-grade laptops based on such processors, and perhaps more than a decade from having these systems offering isolation technologies on par with what the current Intel processors offer.

[1] Arguably virtualization technologies might not be needed for such new processors. On the other hand, it might turn out more practical to port e.g. the existing Linux kernel and recompile many of the currently used POSIX applications for these new processors, than to write everything from scratch. In that case we would need virtualization in order to implement reasonably strong compartmentalization.

Chapter 10

Summary

Personal computers have become extensions of our brains. This symbiosis is only going to strengthen in the years to come, and not just metaphorically! The author believes it should be paramount for humankind to ensure we can trust our personal computers. Unfortunately the industry does not seem to share this opinion. Not only do we not see much effort to create secure and trustworthy hardware and Operating Systems, but we also witness the introduction of technologies, such as Intel ME, that could undermine our trust in computers, (especially personal computers) more than anytime before.

The strict separation of state-carrying (trusted) element from the rest of the hardware, proposed in this paper, is an attempt to change this game in favour of the user. While this solution might appeal to many as simple and elegant, care should be exercised in understanding various implementation-specific subtleties, many of which, hopefully, have been discussed in this paper.

The author thinks this clean separation of state might be beneficial not just for Intel x86 systems, but also for other architectures of our future personal computers.

Credits

I would like to thank the following people for many insightful discussions as well as for reviewing of this paper: Rop Gonggrijp (especially for turning my attention to the problem of "state"), Peter Stuge (for sharing his rich hardware expertise), and Rafał Wojtczuk (for being a great sparring partner in many discussions).

Contacting the author

Joanna Rutkowska can be contacted by email at: joanna@invisiblethings.org

Her personal master key fingerprint[1] is also provided here for additional verification:

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

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

References

[1] The lowRISC project. http://www.lowrisc.org/.

[2] Open processor foundation. http://0pf.org/.

[3] Jacob Appelbaum. A technical action plan. Video archives for Security in Times of Surveillance conference, https://projectbullrun.org/surveillance/2015/video-2015.html#appelbaum, 2015.

[4] Genode developers. An in-depth look into the ARM virtualization extensions. http://genode.org/documentation/articles/arm_virtualization, 2015.

[5] Google Chromium Project. Chromium embedded controller (EC) development. https://www.chromium.org/chromium-os/ec-development.

[6] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Proc. 2008 USENIX Security Symposium, https://citp.princeton.edu/research/memory/, 2008.

[7] ENE Technology Inc. KB3930 for OLPC keyboard controller data sheet. http://dev.laptop.org/~rsmith/KB3930_OLPC_v02_20100503.pdf.

[8] Joanna Rutkowska. Nushu: Passive covert channels implementation in Linux kernel. Presented at the Chaos Communication Congress, https://events.ccc.de/congress/2004/fahrplan/files/319-passive-covert-channels-slides.pdf, 2004.

[9] 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.

[10] Joanna Rutkowska. Intel x86 considered harmful. http://blog.invisiblethings.org/papers/2015/x86_harmful.pdf, 2015.

[11] 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.

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

[13] thaddeus t. grugq. P.O.R.T.A.L.: Personal onion router to assure liberty. https://github.com/grugq/portal, 2012.

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

[15] The OpenSSD Project. OpenSSD wiki. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[16] The Qubes OS Project. Qubes OS: A reasonably secure desktop os. https://qubes-os.org.

[17] Wikipedia. ARM architecture. https://en.wikipedia.org/wiki/ARM_architecture.