Attacking Intel TXT® via SINIT code execution hijacking

Rafal Wojtczuk
rafal@invisiblethingslab.com

Joanna Rutkowska
joanna@invisiblethingslab.com

November 2011

Abstract

We present a software attack against Intel® TXT that exploits an implementation problem within a so called SINIT module. The attack allows to fully bypass Intel TXT, Intel Launch Control Policy (LCP), and additionally also provides yet-another-way to compromise SMM code on the platform.

1 What is Intel TXT?

For a basic introduction to Intel® Trusted Execution Technology (TXT), the reader is referenced to our previous paper on this topic [1], or alternatively, for a much more complete and in-depth introduction, to the updated book by David Grawrock [3], and also to the MLE Developer's Guide [4].

2 Prior work on attacking Intel TXT

In early 2009 our team presented an attack against Intel TXT that exploited a design problem with System Management Mode (SMM) being over privileged on PC platforms and able to interfere with TXT launch [1].

A year later we demonstrated a different attack against Intel TXT, this time exploiting an implementation bug in a so called SINIT module, an internal part of the Intel TXT infrastructure. The attack worked by tricking SENTER into mis-configuring VT-d setup, so that the attacker could compromise the newly loaded hypervisor using a DMA attack[2].

3 On Attacking Intel TXT

The attack presented in this paper assumes, as usual, that the attacker can execute code before the TXT launch, i.e. before the SENTER instruction. The attacker's goal is to either 1) be able to compromise the newly loaded hypervisor, even though is has just been "securely" loaded by TXT (and this is exactly how our previous two attacks worked), or 2) be able to load *arbitrary* hypervisor, yet make it seem as if it was a trusted one by making all the PCR hashes to be *correct*. This is how the attack presented today works.

Our new attack exploits a bug in an SINIT module. Before describing the bug, let's make a quick recap on what is the role of SINIT in Intel TXT.

4 About Authenticated Code (AC) modules and SINIT

SINIT is an important binary module that is used by Intel TXT. SINIT binaries are distributed by Intel for specific chipsets/processors, and the task of an SINIT module is to prepare the platform for entering the TXT secure mode.[1] SINIT module is loaded and

[1] One can download SINIT modules from http://software.intel.com/en-us/articles/

executed by the SENTER instruction. SINIT must be digitally signed by Intel for the SENTER instruction to load and execute it. SINIT is thus also called an Authenticated Code Module (AC Module). There is at least one other example of an AC Module distributed by Intel, the SCLEAN AC Module, that can be loaded by ENTERACCS instruction and is supposed to be used by a TXT-aware BIOS to wipe the system memory in the event of an unexpected system shutdown.[2]

An AC Module, such as an SINIT, when loaded using the SENTER or ENTERACCS instructions[3], executes in a specially protected and privileged environment. There seem to be some differences between the environment provided by the ENTERACCS vs that provided by the SENTER instruction, and so we focus further on the latter case, and we will call this special privileged environment, or mode of execution, an "SINIT mode".

One task of the SINIT module is to read and parse platform configuration as exposed by the BIOS ACPI tables, and specifically by the ACPI DMAR table that describes the VT-d configuration of the platform.[4]

5 The SINIT bug

SINIT code is written in a regular x86 assembly, so it is possible to disassemble it using standard x86 tools for binary analysis. Below is a fragment of the SINIT code for Sandy Bridge processors (disassembly created using the `objdump` tool, comments added manually):

```
6675:       mov    (%edi),%esi
6677:       cmpl   $0x52414d44,(%esi)
; (DWORD*)esi == 'DMAR'?

667d:       je     0x6697
...
6697:       mov    (%edi),%edi
6699:       mov    %edi,%es:0xa57
; var_a57 = &dmar

66a0:       mov    0x4(%edi),%ecx
; ecx = dmar.len

66a3:       push   %ecx
66a4:       add    %edi,%ecx
66a6:       mov    %ecx,%es:0xa5b
; var_a5b = &dmar + dmar.len

...
6701:       mov    %es:0xa47,%edi
; edi = var_a47 (memory on the TXT heap)

6708:       mov    (%edi),%eax
670a:       mov    %es:0xa5b,%ebx
; ebx = &dmar + dmar.len

6711:       sub    %es:0xa57,%ebx
; ebx = dmar.len

...
6738:       mov    %es:0xa57,%esi
; var_a57 = &dmar

673f:       mov    %ebx,%ecx
6741:       rep movsb %ds:(%esi),%es:(%edi)
; memcpy (var_a47, dmar, dmar.len)
```

We see that the above code fragment first reads the DMAR ACPI table length, as indicated by the *length* field in the *untrusted* ACPI DMAR header, and then attempts to copy this ACPI table onto a buffer located on the TXT heap. The copying loop (the `rep movsb` instruction) is happily copying as many

`intel-trusted-execution-technology/`

[2] If (an incorrectly implemented) BIOS doesn't execute SCLEAN module after an unexpected platform shutdown occurred, so no clean TXT exit was performed, the chipset will block access to DRAM until SCLEAN module is loaded an executed. This will effectively make the platform "bricked", as we have an occasion to witness ourselves a few times...

[3] Intel pointed out that SINIT will fail if loaded using ENTERACCS.

[4] In fact the primary's job of SINIT module is to *verify* the ACPI tables, not to really *use* the information they provide. SINIT module is smart enough to extract most (all?) information that the ACPI tables communicate using various chipset registers, often undocumented. However, verifying ACPI tables correctness is an important task because the MLE (e.g. a hypervisor) that loads later relies on those ACPI tables.

bytes as the untrusted ACPI DMAR table indicated. The ACPI tables are untrusted because an attacker who controls the pre-TXT boot environment, e.g. infected the system's MBR, is able to freely modify the in-memory ACPI tables that have originally been published by the BIOS. The ACPI tables are not digitally signed, so it's not possible to detect such malicious modifications of the ACPI tables. [5]

6 The SINIT code execution exploit

Normally the SINIT code is placed before the TXT heap as illustrated on figure 1. This is the default layout that is expected to be created by system software as described in [4].

In order to take advantage of the overflow in the SINIT code discussed above, we should rearrange the memory layout and place the TXT heap (or at least some parts of it) above the SINIT code, so that the overflowing data could overwrite the SINIT code.

It seems like a trivial task, because the system software (so also the attacker in this case) controls the base address of the TXT heap base address via TXT.HEAP.BASE register, see [4].

Unfortunately, it turned out that if one attempted to reverse the layout of the heap and SINIT, then SENTER would return an error. Specifically, we have seen a code in the SINIT module that specifically checked whether the heap is located *below* the SINIT code. This might suggest Intel has been anticipating potential bugs in SINIT code, and wanted to prevent SINIT code overwrites in such cases.

We have found, however, another way of how to relocate heap before the SINIT code. It turned out this could be achieved by changing the *size* field of the

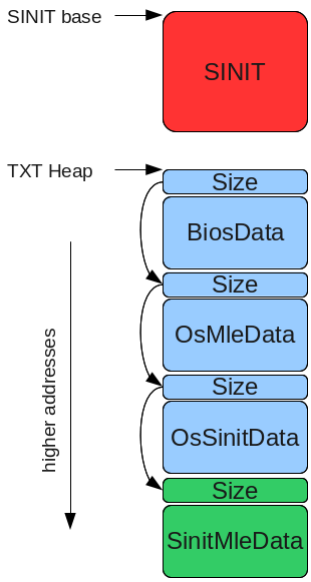

Figure 1: Normally the TXT heap is expected to be below the SINIT code (in terms of numerical addresses).

first TXT heap's chunk[6] to be effectively negative as illustrated as illustrated on figure 2.

The SINIT code that reads and copies the DMAR table will place it within the TXT heap chunk called *SinitMleData* (the last one). This is where the overflow will start at. Now, by providing large enough DMAR table the attacker is able to overflow the SINIT code.

In order to exploit the overflow for arbitrary code execution, we have found a function that is located before the `rep movsb` instruction, but which is called afterwards – see figure 3. Thus, by overwriting this function with a custom shellcode, we can easily get arbitrary code executed in the context of the SINIT mode.

Even if there was no such function, we could still exploit it by overflowing past the `rep movsb` instruction, while preserving the `rep movsb` (overwriting it with itself), although we haven't tried that in prac-

[5] In fact, even if the ACPI tables were digitally signed by the BIOS, this would still not be a satisfactory solution here, as it would mean the TXT trusts the BIOS to be non-malicious, while one of the main selling points of Intel TXT is that it doesn't need to trust the BIOS.

[6] See [4] for description of the heap chunks

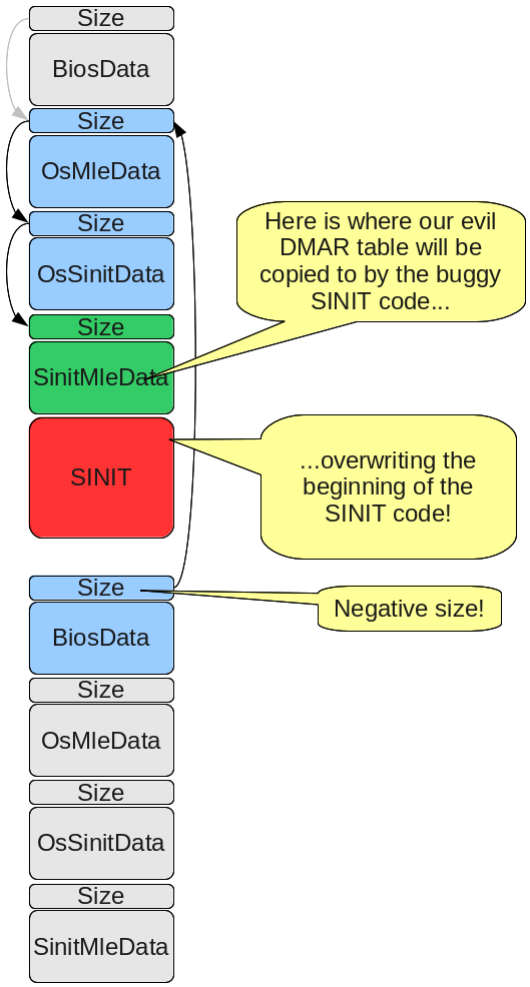

Figure 2: We have modified the first chunk's *size* field to be negative, and now most of the heap is considered to be *above* the SINIT code.

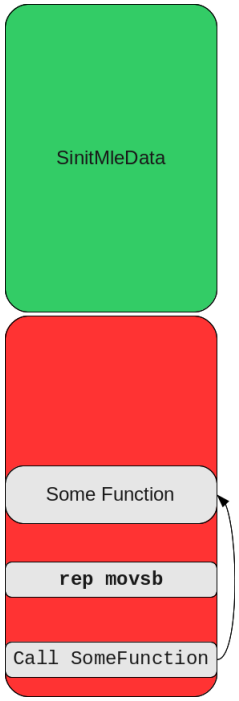

Figure 3: The function *Some Function* is a convenient target to overwrite...

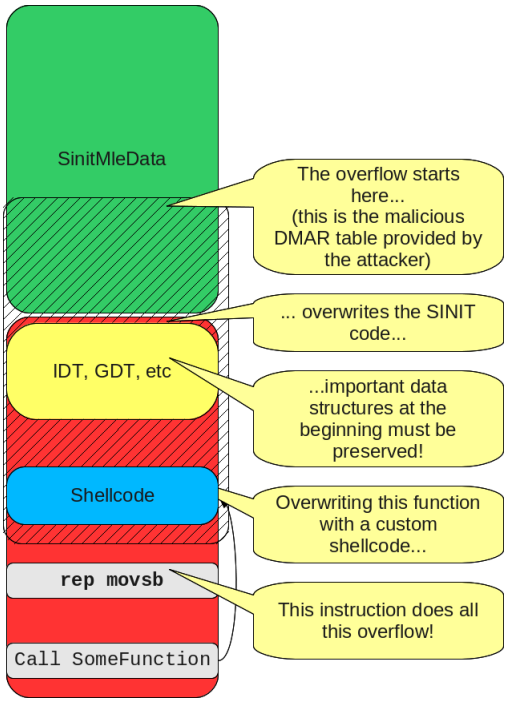

Figure 4: Exploiting the SINIT overflow in practice.

tice.

As it turned out, we also needed to preserve some important data structures at the beginning of the SINIT area, such as e.g. GDT. We do this by overwriting this part with the original bytes extracted from SINIT. This is possible, because we control the whole DMAR table we use for overwrite (except some early headers that must be in compliance with the DMAR spec). Figure 4 illustrates the whole exploitation process.

The exploitation process described above is reliable, because all the offsets are deterministic and depend only on the actual version of the SINIT module used for exploitation. The attacker can always download and analyze the specific SINIT version, before the attack.

As we can see the actual exploitation process has been rather straightforward, with only one non-standard task being the tricky heap layout relocation. Combining this with a fact that exploitation is reliable, we believe this attack is very practical and could be easily used by malware in the wild to bypass TXT and LCP as described below.

7 The SINIT exploitation consequences: TXT bypass

One obvious consequence of the SINIT code execution exploit is the immediate ability to directly bypass TXT-based trusted boot. This is because the SINIT code executes before most dynamic PCR registers[7] are extended, yet after they have been reset to 0 by the SENTER microcode.[8]

In fact only PCR17 is extended at the time the first instruction of SINIT is being executed – it has been extended by the SENTER with a hash of the SINIT module and parameters passed to the SENTER instruction. Extension of the other PCRs, specifically the PCR18 that is normally extended with a hash of the to-be-loaded MLE, is left as a task for SINIT...

Because the SINIT module that we use in this attack is a fully legitimate, Intel distributed, original SINIT, the PCR17 will be extended with a *correct* hash before handing execution to SINIT.

It turned out that SINIT will first parse the DMAR table, before measuring the MLE and extending the PCR18. This means that at the time our shellcode is executing, the PCR18, and other higher-numbered PCRs, are still set as zero. This means our shellcode can now freely extend PCR18 with an arbitrary hash, e.g. a hash of some legitimate MLE, yet can pass execution to some *other* MLE, e.g. a malicious one.[9]

This has been illustrated on figure 5.

[7] That is PCR registers resettable by SENTER

[8] All dynamic PCR registers have initial value of -1 after system boot. Only the SENTER instruction can instruct TPM to reset those dynamic registers so they have a value of 0.

[9] A decent SINIT would always pass execution to the same entity that is has just measured, but our "hacked" SINIT doesn't need to be decent anymore!

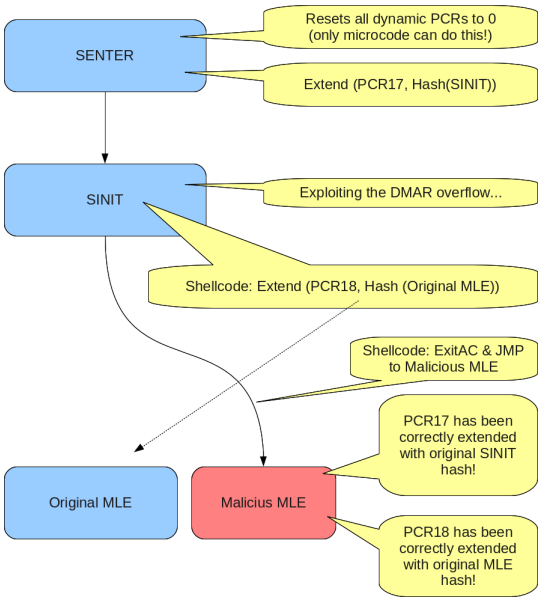

Figure 5: Illustration of TXT bypass process.

When Intel releases a fixed SINIT module, all users will need to re-seal their secrets to the new SINIT's hash in order to prevent the above attack.

8 The SINIT exploitation consequences: LCP bypass

Closely related to TXT bypass (understood as bypassing of a seal-based or remote attestation-based trusted boot) is also an ability to use the exploit above to also bypass the Intel Launch Control Policy (LCP). Intel LCP is described in detail in [4].

The LCP is enforced by the SINIT module and the enforcement is done after DMAR table parsing, so after we exploit SINIT to get our shellcode executed. This means we can simply skip the LCP enforcing code, fully ignoring any LCP policy the user might have set up.

One practical scenario is to ignore a potential LCP policy that might be blacklisting the very SINIT module we're exploiting[10]. If such an LCP policy could be enforced, it would be a convenient solution against our attack. However, because we can trivially bypass any LCP policy, this would not work to stop our attack.

9 The SINIT exploitation consequences: SMM hijacking

Another interesting consequence of our SINIT attack is the ability to compromise the platform's SMM handler. This is because the SINIT module is normally granted unlocked access to the SMRAM, as it is indicated in the Intel SDM [5]:

```
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP -> ACEntryPoint;
```

The above is a fragment of a pseudocode for the SENTER instruction. We can see that just before the execution is passed to the AC module, in this case our SINIT module, SMRAM, TXT private space, and locality 3 are all unlocked. Because our shellcode executes in the context of the SINIT module, it means it will also have access granted to the SMRAM.

Theoretically the SINIT module could be written in such a way that it voluntarily locks down SMRAM access right at the beginning, before attempting to parse DMAR table (so, before our shellcode gets a chance to execute), but we have verified this was not the case.

Apart from other SMM-related attacks (e.g. SMM rootkits), this allows for yet another interesting attack...

Let's assume that, after we had informed Intel about the SINIT attack, Intel released an advisory urging customers to re-seal their secrets to a new PCR17 hash for an updated SINIT module, as explained

[10] This, of course, assumes that Intel was informed about this very problem, and that Intel released an advisory for the attack.

above. Now, our direct TXT bypassing attack would no longer work, because PCR17 would contain a wrong hash at the time when our shellcode executes.

However, we can now use SINIT exploit to gain access to SMRAM (using the buggy SINIT), install a backdoor in the SMI handler, exit AC mode, and then re-launch original MLE using the new, patched SINIT (so performing a legitimate TXT launch).

We have verified that we can indeed modify the SMI handler from within the SINIT exploit's shellcode, exit the SINIT mode (using EXITAC instruction) and that our SMM modifications survive this operation. Because the compromised SMI handler survives also the TXT launch, our backdoor in SMM will now be able to compromise the newly and legitimately loaded MLE on the first occurrence of an SMI interrupt (which usually happens immediately after the MLE enables SMIs), as we have demonstrated in our first paper on attacking TXT [1].

Intel should be able to fix this problem in two ways:

1. A proper way of addressing this would be to (finally) release an STM, i.e. a special component, distributed as part of the BIOS, that is supposed to sandbox a potentially malicious SMM code. Interestingly, Intel seems to have been ignoring this problem for almost 3 years now, that is since we have demonstrated the first practical TXT-bypassing attack using a software-compromised SMI handler back at the beginning of 2009.

2. Intel could also release a microcode update that would either disable the SMRAM unlocking done as part of the SENTER instruction, and/or black-list the buggy SINIT that we use for exploitation. Additionally, for this protection to make sense, the microcode update would have to be applied by the BIOS, so that it was not possible for the attacker to opt-out the new microcode (microcode update must be applied on each platform boot, because it is non-persistent), and also the new microcode update would need to modify the existing microcode update procedure to not allow for microcode downgrade, as otherwise the attacker could simply revert to an older microcode, which would not be blocking the buggy SINIT (or disabling SMRAM access) and conduct all the attack as described above.

If Intel decided to go with the second solution and to not bring a working STM to its platforms, it would also have an interesting consequence of including the BIOS into TXT's trusted base. This is because if an attacker was able to subvert the BIOS, e.g. by compromising the BIOS anti-reflashing mechanisms[11], then the attacker would be able to opt-out the microcode update that black-lists the buggy SINIT (or disables granting access to SMRAM as part of SENTER), and conduct the TXT-bypassing attack described above.

10 Other consequences of SINIT execution hijacking?

Going further, it is an interesting question to ask what other special powers, beyond free access to SMRAM, an AC module, and specifically an SINIT module, might potentially posses?

It seems like Intel has put lots of efforts into keeping everybody away from executing arbitrary code in "SINIT mode". The whole infrastructure used for verification of an AC module's digital signatures strikes as an unnecessary complication (let's keep in mind that this verification process is part of the SENTER instruction's microcode!). It seems unnecessary because for the purpose of doing secure TXT launches, it is just enough to have SENTER measure SINIT hash and extend one of the dynamic PCRs with it, which is exactly what SENTER does (PCR17 is extended with a hash of SINIT). So, why the additional, non-trivial mechanism for signature verification?

One might argue this additional verification mechanism was needed to implement Intel LCP policy, as the policy enforcement code is itself implemented

[11] We have demonstrated how to bypass Intel vPro BIOS reflashing protection back in 2009, see [6].

within an SINIT module. However, one should remember that the Intel LCP is itself an opt-in mechanism. There is really nothing that could force an attacker into executing SENTER (and so going through the LCP policy), except only a possibility to lock down VMX mode outside SMX mode[12]. However, it seems unclear why an attacker, specifically malware, should be so keen on enabling VT-x? After all, even though the first hardware-virtualization malware has been introduced nearly 5 years ago [7], [8], it still doesn't seem to be used in the wild, because currently used operating systems offer many opportunities for much simpler, traditional compromises.

Perhaps then, it was all about the SCLEAN module, i.e. the code that is supposed to be loaded early at boot stage in case TXT was not cleanly shut down, and which is tasked with scrubbing DRAM so that no secretes could be stolen from there [3].

On the other hand, the primary role of the SCLEAN module is to prevent Cold Boot-like physical attacks [9], and it seems like an easy attack to bypass the SCLEAN-imposed protection, would be to just... remove the DRAM dies and put them into another computer, and continue with regular Cold Boot attack as usual... [13]

This brings the question again: why Intel implemented a complex digital signature verification scheme for AC modules, even though it doesn't seem to improve security properties of TXT? We shall leave the reader without an answer here...

UPDATE: While reviewing this paper Intel noted that, unlike we thought, MSEG is measured by the SINIT module[14], rather than by SENTER microcode, as we though. This explains why SINIT module needs SMRAM access. Intel also pointed out that many OEMs expect both integrity and *confidentially* for their SMM code[15], which is a reason for allowing only signed SINIT modules to be executed.

11 Intel's reaction

We have informed Intel about the vulnerability in SINIT module, together with an extensive discussion of possible consequences, and suggested patching approaches, as described above, on July 28th, 2011. Intel released the official coordinated advisory on December 5th, 2011 [10].

Intel has informed us about the following measures that have been implemented in order to patch the vulnerability:

1. Releasing updated SINIT modules that fixes the buffer overflow for all the affected processors.
2. Releasing updated processor microcode that prevents loading and execution of the buggy SINIT modules. Additionally this microcode update provides an anti-rollback mechanism, so that it is not possible to downgrade to a previous microcode version, once this new update has been applied.
3. Coordinating with OEM vendors to release BIOS updates that load the above mentioned microcode update on system boot. Additionally, OEM vendors are expected to implement BIOS anti-rollback mechanism for their BIOSes too.
4. Additionally, as a preventive measure against possible future problems, Intel has moved the LCP-enforcing code to the beginning of SINIT code, so it was possible in the future to blacklist buggy SINIT modules without the need for microcode and BIOS updates.

[12] In other words it is possible to configure a platform, typically by the BIOS, to disallow the use of VT-x without entering secure TXT mode

[13] One of the Intel architects, while reviewing this paper, pointed out that SCLEAN module provides also a protection against a hypothetical *remote* Cold Boot attack, where an attacker is able to remotely force (unclean) system reboot, and is also able to execute code before OS loads, e.g. via infected MBR, which allows for mounting the Cold Boot attack locally and leaking the stolen secrets e.g. through a NIC back home.

[14] MSEG is a part of memory where BIOS is suppose to place an image of an STM for use with TXT, see [4]

[15] The official argument for confidentiality of the SMM code being that Intel has architecturally committed to both write- and read-protecting SMRAM when introduced the SMRAM lock feature in the chipset, years ago.

5. Releasing an advisory for customers, urging them to 1) update their BIOSes, and 2) reseal secrets to new PCR17 hashes.

All the newly released SINIT modules for the existing platforms will now have the previously reserved field *Reserved1* at offset 0x1c in the SINIT header (see [4]) set to 1. All previous (so buggy) SINIT modules have had this field set to 0. The updated microcode will refuse to load an SINIT module with *Reserved1* field being 0. Of course an attacker can not just take the buggy SINIT and change the *Reserved1*, because the SINIT module is digitally signed.

12 Summary

1. SINITs are buggy just like any other software. Intel should consider open sourcing those critical pieces of code.
2. SINIT compromise allows more than "just" TXT bypass, e.g. it also allows SMM compromise, and perhaps something else. . . ?
3. It's a shame we still don't see STMs in the wild, even on Intel platforms!
4. Preventing our attack requires: 1) SINIT patching, 2) secrets resealing by customers, 3) BIOS upgrade, and finally 4) adding the BIOS to the chain of trust. The last two might have been avoided if we had STMs. . .

13 Acknowledgments

We would like to thank Intel TXT architects, Monty Wiseman and Joe Cihula, for reviewing the paper, and providing constructive feedback.

References

[1] Rafal Wojtczuk, Joanna Rutkowska, *Attacking Intel Trusted Execution Technology*, Jan 2009, `http://www.invisiblethingslab.com/itl/Resources.html`

[2] Rafal Wojtczuk, Joanna Rutkowska, Alexander Tereshkin, *Another Way to Circumvent Intel Trusted Execution Technology*, Dec 2009, `http://www.invisiblethingslab.com/itl/Resources.html`

[3] David Grawrock, *Dynamics of a Trusted Platform: A Building Approach*, Intel Press, 2009.

[4] Intel Corporation, *Intel® Trusted Execution Technology, Measured Launched Environment Developer's Guide*, 2008.

[5] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 2b.

[6] Alexander Tereshkin, Rafal Wojtczuk, *Attacking Intel® BIOS*, Jul 2009, `http://www.invisiblethingslab.com/itl/Resources.html`

[7] Joanna Rutkowska, *Subverting Vista Kernel for Fun and Profit*, Black Hat USA, Aug 2006, `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`

[8] Dino Dai Zovi, *Hardware Virtualization Rootkits*, Black Hat USA, Aug 2006, `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf`

[9] J. Alex Halderman et al., *Lest We Remember: Cold Boot Attacks on Encryption Keys*, `http://citp.princeton.edu/research/memory/`

[10] Intel Security Advisory INTEL-SA-00030, Dec 5th, 2011, `http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00030&languageid=en-fr`