OSLO: Improving the security of Trusted Computing

Bernhard Kauer
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
kauer@os.inf.tu-dresden.de

Abstract

In this paper we describe bugs and ways to attack trusted computing systems based on a static root of trust such as Microsoft's Bitlocker. We propose to use the dynamic root of trust feature of newer x86 processors as this shortens the trust chain, can minimize the Trusted Computing Base of applications and is less vulnerable to TPM and BIOS attacks. To support our claim we implemented the Open Secure LOader (OSLO), the first publicly available bootloader based on AMDs `skinit` *instruction.*

1 Introduction

An increasing number of Computing Platforms with a Trusted Platform Module (TPM) [33] are deployed. Applications using these chips are not widely used yet [5, 37]. This will change rapidly with the distribution of Microsoft's Bitlocker [2], a disk encryption utility which is part of Windows Vista Ultimate. As the trusted computing technology behind these applications is quite new, there is not much experience concerning the security of trusted computing systems. In this context we analyzed the security of TPMs, BIOSes and bootloaders that consitute the basic building blocks of trusted computing implementations. Furthermore, we propose a design that can improve the security of such implementations.

1.1 Trusted Computing

Trusted Computing [9, 23, 25, 33] is a technology that tries two answer two questions:

- Which software is running on a remote computer? (*Remote Attestation*)
- How to ensure that only a particular software stack can access a stored secret? (*Sealed Memory*)

Different scenarios can be built on top of trusted computing, for example, multi-factor authentication [37], hard disk encryption [2,5] or the widely disputed Digital Rights Management. All of these applications are based on a small chip: the Trusted Platform Module (TPM).

1.2 Technical Background

As defined by the Trusted Computing Group (TCG), a TPM is a smartcard-like low performance cryptographic coprocessor. It is soldered[1] on various motherboards. In addition to cryptographic operations such as signing and hashing, a TPM can store hashes of the boot sequence in a set of Platform Configuration Registers (PCRs).

A PCR is a 160 bit wide register that can hold an SHA-1 hash. It cannot be directly written. Instead, it can only be modified using the `extend(x)` operation. This operation calculates the new value of a PCR as an SHA-1 hash of the concatenation of the old value and x. The `extend` operation is used to store a hash of a chain of loaded software in PCRs. The chain starts with the BIOS and includes Option ROMs[2], Bootloader, OS and applications.

Using a challenge-response protocol, this *trust chain* can attest to a remote entity which software is running on the platform (*remote attestation*). Similarly it can be used to *seal* some data to a particular, not necessarily the currently running, software configuration. Unsealing the data is then only possible when this configuration was started. Figure 1 shows such a trust chain based on a Static Root of Trust for Measurement (SRTM), namely the BIOS.

$$TPM \Longrightarrow BIOS \Rightarrow OptionROMs \Rightarrow$$
$$BootLoader \Longrightarrow OS \Longrightarrow Application$$

Figure 1. Typical trust chain in a TC system

1.3 Chain of Hashes

Three conditions must be met, to make a chain of hashes trustworthy:

1. The first code running and extending PCRs after a platform reset (called SRTM) is trustworthy and cannot be replaced.

2. The PCRs are not resetable, without passing control to trusted code.

3. The chain is contiguous. There is no code in-between that is executed but not hashed.

The reasons behind these conditions are the following: If the initial code is not trustworthy or can be replaced by untrustworthy code, it cannot be guaranteed that any hash value is correct. This code can in fact modify any later running software to prevent the undesirable hashing. The second condition is quite similar and can be seen as a generalization of the first one. If PCRs are reset and untrustworthy code is running then any chain of hashes can be fabricated. The first two points describe the beginning of the trust chain. The third point is needed to form a contiguous chain by recursion. It forces the condition that every program occupying the machine must be hashed, before it is executed. Otherwise, the trust chain is interrupted and unmeasured code can be running. Every program using sealed memory has to trust the code running before it to not open a hole in the chain. Similarly, a remote entity needs to find out during an attestation whether the trust chain presented by a trusted computing platform contains any hole in which untrusted code could be run. We will see later how current implementations do not meet the three conditions.

Organization

This paper is structured as follows. We describe bugs and ways to attack trusted computing systems based on SRTM in the next section. After that we present the design and describe the implementation of OSLO. A section evaluating the security achievements follows. The last section proposes future work and concludes.

2 Security Analysis

2.1 Bootloader Bugs

We look at the three publicly available TPM-enabled bootloaders and analyze whether they violate the third condition of a trust chain, executing code that is not hashed.

The very first publicly available trusted bootloader was part of the Bear project from Dartmouth College [19, 20]. They enhanced Linux with a security module called Enforcer. This module checks for modification of files and uses the TPM to seal a secret key of an encrypted filesystem. To boot the system they used a modified version of LILO [7]. They extend LILO in two ways: the Master Boot Record hashes the rest of LILO and the loaded Linux kernel image is also hashed. Only the last part of the image, containing the kernel itself, is hashed here. But the first part of the image, containing the real-mode setup code, is executed. Hence, this violates the third condition. A fix for this bug would be to hash every sector which gets loaded.

A second trusted bootloader is a patched GRUB v0.97 from IBM Japan [21, 36]. This bootloader is used in IBMs Integrity Measurement Architecture [28]. It has the same security flaw as our own experiments with a TCG enabled GRUB [16]: it loads files twice, first for extraction and later for hashing into a PCR. A cause for this bug lies certainly in the structure of GRUB. GRUB loads and extracts a kernel image at the same time instead of loading them completely into memory and extracting them afterwards. This leads to the situation that measuring the file independently from loading is the easiest way for a programmer to add TCG support to GRUB. Such an implementation is unfortunately incorrect. As program code is loaded twice from disk or from a remote host over the network, an attacker who has physical access either to the disk or to the network can send different data at the second time. This violates again the third condition, as hashed and executed code may differ.

Another GRUB based trusted bootloader called TrustedGRUB [35] solves this issue in a recent version by moving the hash code to a lower level. Hashing is simply done on each `read()` call that loads data from disk or network, before the actual data is returned to the caller. The hash is then used after loading a kernel to extend a PCR.

The current version 1.0-rc5 of TrustedGRUB (August 2006) contains at least two other bugs. The hashing of its own code when starting from hard disk is broken. The corresponding PCR is never extended and always zero. Furthermore TrustedGRUB never contained any code to

use it securely from a CD. Nevertheless, it is used on a couple of LiveCDs [6].

All publicly available TPM-enabled bootloaders violate the third assumption, which makes systems booted by them unable to prove their trustworthiness. To analyze this it was not necessary to look at more sophisticated attack points such as missing range checks or buffer overflows. Both of these will become more interesting if the aforementioned bugs are fixed.

2.2 TPM Reset

In July 2004 we discovered that setting the reset bit in a control register of a v1.1 TPM[3] resets the chip without resetting the whole platform. This violates the second condition. As it results in default PCR values, this breaks the *remote attestation* and *sealing* features of those chips: Any PCR value can be reproduced without the opportunity for a remote entity to see the difference via remote attestation. Unsealing protected secrets of a security critical program is possible after resetting as well. The reset feature was added for maintenance reasons but does not have broad security consequences, because sealing and remote attestation are not used in any product application with v1.1 chips. Instead the chips are solely used as smartcard for signing and key management.

This case demonstrates the security risk of a resettable TPM. As other chips have different interfaces and can therefore not be reset in the same way, we experimented with a simple hardware attack. The Low Pin Count (LPC) bus was the point of attack. Most TPMs are connected to the southbridge through it and the bus has a separate reset line. We used different TPMs on external daughterboards for this experiment.

By physically connecting the LRESET# pin to ground we were able to perform a reset of the chip itself. We separated the pin from the bus as otherwise the PS/2 keyboard controller received such a reset signal, too. We had to reinitialize the chip which we did by reloading the driver and then sending a `TPMStartup(TPM_CLEAR)` to the chip. This process gave us an activated and enabled TPM in a state normally only visible to the BIOS: As expected all PCRs were in their default state. We presume that this attack could be mounted against any TPM in a similar way.

The simplicity of the reset makes this hardware attack a threat to trusted computing systems. In particular in use cases where physical access, for example, through theft, can not be excluded. This attack also affects another use case of trusted computing, the widely disputed Digital Rights Management scenario where the owner of a device is untrusted and can use the system unintendedly.

We have to admit that the TCG does not claim to protect against hardware attacks. But scenarios using trusted computing technology have to be aware of these restrictions.

2.3 BIOS Attack

We have shown that bootloader and TPM implementations have some weaknesses. Now we look at the entity in-between them: the BIOS.

The BIOS contains the Core Root of Trust for Measurement (CRTM), a piece of code that extends PCR 0 initially. A CRTM has only to be exchanged with vendor signed code. Currently, the CRTM of many machines is freely patchable. It is stored in flash and no signature checking is performed on updates. This violates the first condition needed by a trust chain.

We used a HP nx6325, a recent business notebook with a TPM v1.2, for this experiment. The fact that the BIOS is flashed from a raw image eased an attack. Other vendors are checking a hash before flashing the image to avoid transmission errors, a feature that is missing here. Checking a hash is irrelevant from a security point of view but it would make the following steps slightly more complicated, as we would have to recalculate the correct hash value.

The part of the BIOS we choose to patch is the TPM driver. This has the advantage that all commands to the TPM, whether they come from the CRTM or from a bootloader through the `INT 1Ah` interface, can be intercepted. Our BIOS has only a memory-present TPM driver. These drivers need access to main memory for execution and can therefore only run after the BIOS has initialized the RAM. The interface of the TPM drivers are defined in the TCG PC client specification for conventional BIOS [34]. The function that we want to disable is `MPTPMTransmit()` which transmits commands to the TPM. We found the TPM driver in the BIOS binary quite easily. Strings like `'TPM'` and the magic number of the code block as well as characteristic mnemonics (e.g., `in` and `out`) in the disassembly point to it.

Figure 2 shows the start of the BIOS TPM driver. It starts with a magic number and entry point, both as defined in the specification. The code itself starts at address $0x28$. We now search for an instruction that allows us to disable `MPTPMTransmit()`. The first instructions of the driver are quite uninteresting. They just save some registers to the stack and calculate the drivers starting address in register `edi` in order to make the code position independent. The first interesting instruction is the comparison at address $0x3a$. By look-

```
 0:  aa 55        /* magic number */
 4:  28 00        /* entry point */
     ...
28:  57          push   %edi
29:  56          push   %esi
2a:  53          push   %ebx
2b:  33 ff       xor    %edi,%edi
2d:  e8 00 00    call   0x32
     00 00
32:  5f          pop    %edi
33:  81 ef 33    sub    $0x33,%edi
     00 00
39:  47          inc    %edi
3a:  3c 04       cmp    $0x4,%al
3c:  74 23       je     0x61
3e:  3c 01       cmp    $0x1,%al
40:  74 0a       je     0x4c
     ...
```

Figure 2. Start of BIOS TPM driver

ing further into the disassembly we found out that this instruction is part of the branch where the code distinguishes between `MPTPMTransmit()` (where `al=4`) and other functions. By changing this comparision to `cmp $0x14,%al`, which just requires to flip a single bit, we can avoid that the branch at $0x3c$ is taken and any command is transmited to the TPM. An error code is returned to the caller instead.

We now have to flash the BIOS with this modified image. As there is no hash of the BIOS image checked during flashing we use the normal BIOS update procedure. After a reboot we have a TPM in its default power-on state, without any PCR extensions.

The ability to easily exchange the CRTM violates the TCG specifications. A result of this bug is that the trust into these machines can not be brought back anymore without an expensive certification process.

2.4 Summary

We found weaknesses in bootloaders and the possibility of a simple hardware attack against TPMs. Furthermore by just flipping a single bit we disabled the CRTM and any PCR extension from the BIOS. These cases show that current implementations do not meet all three conditions of a trust chain.

In summary, we conclude that current BIOSes and bootloaders are not able to start systems in a trusthworthy manner. Moreover, TPMs are not protected against resets.

$$TPM \Rightarrow OSLO \Rightarrow OS \Rightarrow Application$$

Figure 3. Trust chain with a Dynamic Root of Trust for Measurement (DRTM)

3 Design and Implementation of OSLO

3.1 Using a DRTM

The main idea behind a secure system with a resettable TPM, an untrusted BIOS and a buggy bootloader, is to use a Dynamic Root of Trust for Masurement (DRTM). A DRTM effectively removes the BIOS, OptionROMs and Bootloaders from the trust chain (cf. Figure 3).

With a DRTM, the CPU can reset the PCR 17 at any time. This is provided through a new instruction that atomically initializes the CPU, loads a piece of code called Secure Loader (SL) into its cache, sends the code to the TPM to extend the reseted PCR 17, and transfers control to the SL.

A design based on a DRTM is not vulnerable to the TPM reset attack because of a TPM property that can be easily missed. A TPM can distinguish between a reset and a DRTM due to CPU and chipset support. A reset of the TPM sets all PCRs to default values, which is “0” for the PCRs 0 - 16 and “-1” for PCR 17. Only a DRTM, with its special bus cycles, will reset the PCR 17 to “0” and immediately extend it with the hash of the SL. Therefore, an attacker is unable to reset PCR 17 to “0” and fake other platform configurations. Only by executing the `skinit` instruction it is possible to put the hash of an SL into PCR 17. An attacker can not hash an SL and directly afterwards executing code outside of it, since `skinit` jumps directly to the SL.

An SL is also not affected by the BIOS attack. With the presence of a DRTM, the BIOS need not be trusted anymore to protect its CRTM and hash itself into the TPM. Nevertheless, a statement that claims the BIOS can be fully untrusted is oversimplified: We still have to trust the BIOS for providing the System Management Mode (SMM) code as well as correct ACPI tables. As both can be security critical, a hash of them should be incorporated at boot time into a PCR by the operating system.

3.2 Implementation

AMD provides a DRTM with its `skinit` instruction which was introduced with the AMD-V extension [1]. On Intel CPUs, the Trusted Execution Technol-

ogy (TET) includes a DRTM with the `senter` instruction [9, 14]. AMD was generous to provide us with an AMD-V platform nearly one year earlier than we were able to buy an Intel TET platform.

Our implementation, called OSLO (Open Secure LOader), is written in C with some small parts in assembler. As OSLO is part of the Trusted Computing Base (TCB) of all applications, we wanted to minimize the binary and source code size. Furthermore, we had to avoid any BIOS call, as otherwise the BIOS would be part of the TCB again.

OSLO is started as kernel from a multi-boot compliant [22] loader. It initializes the TPM to be able to extend a PCR with the hashes of further modules. After that other processors are stopped. This is required before executing `skinit` and inhibits potential interferences during the secure startup procedure. For example, malicious code running on a second CPU could modify the instructions of the Secure Loader. The cache consistency protocol would then propagate the changes to the other processor.

Since the needed platform initialization is done, OSLO can now switch to the "secure mode" by executing `skinit`. Before starting the first module as a new kernel, OSLO hashes every module that is preloaded from the parent boot-loader.

We used chainloading via the multiboot specification to be flexible with respect to the operating system OSLO loads and who can load OSLO. Normally, this will be a multiboot-compliant loader started by the BIOS such as GRUB or SysLinux [31] but loading OSLO from the Linux `kexec` environment [17] should also be possible.

As we could not rely on the BIOS for talking to the TPM, we also implemented our own TPM driver for v1.2 TPMs. As all of these TPMs should follow the TPM interface specification (TIS) only a single driver was needed. Using this memory mapped interface is, compared to the different interfaces needed to talk to the v1.1 TPMs, rather simple. Therefore our TPM driver consists of only 70 lines of code.

Currently two features of OSLO are still unimplemented:

- protection against direct memory access (DMA) from malicious devices, and
- extension of the TPM event log for remote attestation.

The TPM event log is used to ease remote attestation. It can store hashes used as input for `extend` and optionally a string describing them. The log provides a breakdown of the PCR value into smaller known pieces. It is itself not security critical and therefore not protected by the bootloader or the operating system. An attacker can only perform Denail of Service attacks by for example overwriting the log. It is not possible to compromise the security of a remote attestation by modifying the log. The TPM event log makes it much easier for a remote entity to check a reported hash values against a list of good known values, for example if the order of the `extends` is not fixed. OSLO should extend the event log to support applications relying on it for remote attestation.

The source code of OSLO is available under the terms of the GPL [24]. The source includes three additional tools that can be multi-boot loaded after OSLO: **Beirut** to hash command lines, **Pamplona** to revert the steps done by `skinit` for booting OSLO unaware OSes, and **Munich** to start Linux from a multiboot environment.

3.3 Lessons Learned

We have learned two lessons while implementing OSLO:

- It is hard to write secure initialization code, and
- a secure loader needs to have platform specific knowledge.

An example of the first lesson is our experience with the initialization of the Device Exclusion Vector (DEV) on AMD CPUs. A DEV is a bitvector in physical memory that consists of one bit per physical 4k-page. A bit in this vector decides whether device based DMA transfers to or from the corresponding page is allowed. DEVs could be cached in the chipset for performance reasons. We found out that the DEV initialization, if it is done in the naive way, contains a race condition.

DEV initialization is normally done in two steps: Enable the appropriate bits in the vector to protect itself and then flushing the chipset internal DEV cache. As these two operations are not atomic, a malicious device could change the DEV using DMA just before the vector is loaded into the DEV cache. An implementation has to find a workaround for this race. A secure way to initialize DEV protection is, for example, to use an intermediate DEV in the 64k of the secure loader thereby protecting the initialization of a final DEV.

The second point is a little bit more complicated. DEVs can only protect against DMA from a device. If someone puts an operating system he wants to start with OSLO into device memory it cannot be protected from a malicious device. The OS is loaded and hashed by OSLO as if it would reside in RAM, but if it is read the

Name	size	OSLO sha1	sha1sum
kernel	1.2 MB	0.070 sec	0.020 sec
initrd	4.2 MB	0.245 sec	0.064 sec
sum	5.4 MB	0.315 sec	0.084 sec

Figure 4. Performance of hashing a Linux kernel and Initrd

Name	LOC	binary in kb	gzip in kb
BIOS HP	-	1024	491
GRUB v0.97	19600	98	55
OSLO v0.4.2	1534	4.1	2.9

Figure 5. Size of BIOS, GRUB and OSLO

second time, e.g., on ELF decoding or execution, it is requested from the device memory again. Because we do not trust a device to leave its memory unmodified, we cannot be sure that the code that is executed is identical to the hashed one. As a consequence we can only protect, hash and start modules that are located in RAM. A secure loader therefore needs a reliable method to detect the distinction between RAM and device memory.

4 Evaluation

One of our design goals for OSLO was a minimal TCB size. Reducing the TCB is suitable for security sensitive applications as it increases the understandability and minimizes the number of possible bugs [30]. Furthermore, the process of formal verification will benefit from it. We achieved a minimal TCB by using two techniques: reducing functionality and trading size with performance penalties.

An example for the first is that we do not rely on external libc code but use functions with limited functionality like `out_string()` instead of a full featured `printf()` implementation.

We also implemented our own SHA-1 code trading size for performance. This resulted in an SHA-1 implementation that compiles with gcc-3.4 to less than 512 bytes. This is only a quarter of the size compared with a performance optimized version such as the one from the Linux kernel. This, on the other hand, makes the hash much slower. The Linux version has a throughput which is three to four times higher, due to, e.g., loop unrolling.

Figure 4 shows that booting linux with our SHA-1 implementation takes 0.315 seconds compared to 0.084 seconds for a heavily optimized sha1sum version. As booting a system usually takes minutes a performance penalty of 0.231 seconds is acceptable here.

Figure 5 shows the source and binary sizes for BIOS, GRUB and OSLO. We also give the size of gzip compressed binaries in this table as this reduces the effect of empty sections in the images. Unfortunately, the source code of the HP BIOS is not available. A similar but older Award BIOS consists of around 150 thousand lines of assembler code. The numbers given for GRUB do not include the drivers used to boot from a network. Adding them would nearly double the given numbers.

OSLO is an order of magnitude smaller than GRUB and two orders of magnitude smaller than the BIOS we examined. If we presume the principle *more code equals more bugs* and neglect the effect of a code size optimizing compiler, we can deduce that OSLO has a significantly smaller number of bugs due to its size compared to GRUB or the BIOS.

One could argue that in an ordinary system like Windows or Linux, where the TCB of an application consists of million lines of code with programs consuming tens or hundreds of megabytes, the size of GRUB and the BIOS does not matter. That is perhaps true, but as the trend in secure systems goes to small kernels and hypervisors [10, 13, 29, 32], architectures like L4/NIZZA or Xen can very well benefit from the TCB reduction through OSLO.

In summary, OSLO promises a smaller attack surface due to its minimal size and since it uses a DRTM mitigates the **TPM reset** and the **BIOS attacks** as outlined in Section 3.1.

5 Related Work

Previous research showed the vulnerability of trusted computing platforms against hardware attacks. Kursawe et al. [18] eavesdrop on the LPC bus to capture and analyse the communication between the CPU and the TPM. They only perform a passive attack, but describe that an active hardware attack on the LPC bus could be used to fool the TPM about the platform state. Untrusted code can then pretend to the TPM to be a DRTM.

Limitations of the trusted computing specification and its implementations are described in the literature multiple times. Bruschi et al. [4] showed that an authorization protocol of TPMs is vulnerable to replay attacks. Sadeghi et al. [27] reported that many TPM implementations do not meet the TCG specification. Garriss et al. [8] found out that a public computing kiosk that uses remote attestation to prove which software is running is vulnerable to boot-between attestation attacks. They suggest a reboot counter in the TPM to make reboots visible to remote parties. Such a counter will not help against our TPM reset attack as it needs to detect whether a TPM was switched on later than the whole platform[4], a property a reboot counter cannot achieve.

There are more sophisticated BIOS attacks mentioned in the literature. Heasman [12], for example, showed at the Blackhat Federal 2006 that a rootkit can be hidden in ACPI code which is usually stored in the BIOS. In a subsequent paper [11], he describes how a rootkit can persist in a system with a secured BIOS by using other flash chips. In both cases only TPM-less systems were considered. By combining our attack to disable the CRTM with Heasman's work it seems possible to hide a rootkit in the BIOS but report correct hash values to the TPM.

To generally prevent BIOS attacks, Phoenix Technologies offers a firmware called TrustedCore [26] that allows only signed updates. Intel Active Management Technology [15] has also this feature.

Sailer et al. [28] describe an architecture for an integrity measurement system for Linux using a static root of trust. As they focus on the enhancements of the operating system, the architecture is not limited to an SRTM. There implementation could easily benefit from the smaller attack surface of a secure loader like OSLO.

6 Future Work and Conclusion

OSLO is not feature complete yet. We plan to finish the implementation of the DMA protection. Moreover, we want to add ACPI event-log support. This should allow the integration of OSLO into larger projects that use the event-log for remote attestation.

A port of OSLO to use the `senter` instruction on an Intel TET platform could demonstrate that the mulitboot chainloader design is portable or show that `senter` implies an integrated design as it is proposed for Xen [38].

The search for new attack points of other trusted computing implementations is also part of our future work.

It was not necessary to look at more sophisticated attack points such as buffer overflows or the strength of cryptographic algorithms to find the bugs and attacks we presented in this paper. If we compare this to a similar analysis of another secure system, such as the one of an RFID chip [3], we have to conclude that current trusted computing implementations are not resilient to even simple attacks. Moreover, the current implementations do not meet the assumptions of a secure design. Even a small bug in them can compromise the additional security obtained by a TPM.

We suspect that most of the platforms are vulnerable to the TPM reset and many of them to the BIOS attack. As a consequence the software still based on an SRTM, such as Microsoft's Bitlocker, cannot provide secure TPM-driven encryption and attestation on these systems.

A switch to a DRTM based OSLO-like approach can shorten the trust chain, minimize the TCB, and is less vulnerable to TPM and BIOS attacks.

Acknowledgements

We would like to thank Hermann Härtig, Michael Peter, Udo Steinberg, Neal Walfield, Carsten Weinhold, and Björn Döbel for their comments. Additionally we would like to thank Adrian Perrig, Jonathan McCune and the reviewers for their suggestions to improve the paper. Special thanks go to Adam Lackorzynski for providing the hardware in time.

Notes

[1] There exist also TPMs on daughterboards. Their security value is limited as exchanging them is quite easy.

[2] Firmware on adapter cards

[3] It would be quite unfair to disclose the vendor name here.

[4] e.g., by holding the reset line of a TPM while powering the machine up

References

[1] AMD. Secure Virtual Machine Architecture Reference Manual, May 2005.

[2] BitLocker Drive Encryption: Technical Overview. URL: `http://technet.microsoft.com/en-us/windowsvista/aa906017.aspx`.

[3] Steve Bono, Matthew Green, Adam Stubblefield, Ari Juels, Avi Rubin, and Michael Szydlo. Security analysis of a cryptographically-enabled RFID device. In *USENIX Security Symposium*, Baltimore, Maryland, USA, July 2005. USENIX.

[4] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Attacking a Trusted Computing Platform - Improving the Security of the TCG Specification. Technical Report RT 05-05, Universit'a degli Studi di Milano, Milano MI, Italy, May 2005.

[5] eCryptfs: An Enterprise-class Cryptographic Filesystem for Linux. URL: `http://ecryptfs.sourceforge.net`.

[6] EMSCB downloads. URL: `http://www.emscb.com/content/pages/turaya.downloads`.

[7] Enforcer Project. URL: `http://enforcer.sourceforge.net`.

[8] Scott Garriss, Ramón Cáceres, Stefan Berger, Reiner Sailer, Leendert van Doorn, and Xiaolan Zhang. Towards Trustworthy Kiosk Computing. In *"Proceedings of the 8th IEEE Workshop on Mobile Computing Systems & Applications (HotMobile 2007)"*. IEEE Computer Society Press, February 2007.

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

[10] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2005)*, December 2005.

[11] John Heasman. Implementing and Detecting a PCI Rootkit. November 2006.

[12] John Heasman. Implementing and Detecting an ACPI Rootkit. In *BlackHat Federal*, January 2006.

[13] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005.

[14] Intel Corporation. LaGrande technology preliminary architecture specification. Intel Publication no. D52212, May 2006.

[15] Intel Advanced Management Technology. URL: `http://www.intel.com/technology/manage/iamt`.

[16] Bernhard Kauer. Authenticated Booting for L4. Study thesis, TU Dresden, November 2004.

[17] Kexec Article. URL: `http://lwn.net/Articles/15468`.

[18] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware*, September 2005.

[19] Rich MacDonald, Sean W. Smith, John Marchesini, and Omen Wild. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. Technical Report TR2003-471, Dartmouth College, Hanover, NH, August 2003.

[20] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with tcpa/tcg hardware, or: How i learned to stop worrying and love the bear. Technical Report TR2003-476, Dartmouth College, Hanover, NH, December 2003.

[21] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura. Trusted Platform on Demand. Technical Report RT0564, IBM Corporation, February 2004.

[22] Multiboot Specification. URL: `http://www.gnu.org/software/grub/manual/multiboot/multiboot.txt`.

[23] Chris J. Mitchell, editor. *Trusted Computing*. IEE, London, Nov 2005.

[24] OSLO - Open Secure LOader. URL: `http://os.inf.tu-dresden.de/~kauer/oslo`.

[25] Siani Pearson, editor. *Trusted Computing Platforms*. Prentice Hall International, Aug 2002.

[26] Phoenix Technologies, TrustedCore. URL: `http://www.phoenix.com/en/Products/Core+System+Software/TrustedCore`.

[27] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stüble, Christian Wachsmann, and Marcel Winandy. TCG Inside? - A Note on TPM Specification Compliance. In *The First ACM Workshop on Scalable Trusted Computing (STC'06)*, November 2006.

[28] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the USENIX Security Symposium*, August 2004.

[29] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *ACSAC*, pages 276–285, 2005.

[30] Lenin Singaravelu, Calton Pu, Hermann Hartig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *EuroSys 2006*, April 2006.

[31] SYSLINUX Project. URL: `http://syslinux.zytor.com`.

[32] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, November 2006.

[33] TCG: Trusted Computing Group. URL: `https://www.trustedcomputinggroup.org`.

[34] TCG PC Client Implementation Specification for Conventional BIOS. URL: `https://www.trustedcomputinggroup.org/specs/PCClient`.

[35] TrustedGRUB. URL: `http://www.prosec.rub.de/trusted_grub.html`.

[36] GRUB TCG Patch to support Trusted Boot. URL: `http://trousers.sourceforge.net/grub.html`.

[37] Wave's Embassy Security Center. URL: `http://www.wave.com/products/esc.html`.

[38] [Xen-devel] Intel(R) LaGrande Technology support. URL: `http://lists.xensource.com/archives/html/xense-devel/2006-09/msg00047.html`.