Platform Trust Beyond BIOS Using the Unified Extensible Firmware Interface

Vincent J. Zimmer
System Software Division
Intel Corporation
DuPont, Washington, USA

Abstract **-** *The Unified Extensible Firmware Interface (UEFI) provides a consistent set of interfaces designed to support the booting of shrink-wrap operating systems, loading of drivers that replace legacy PC/AT option ROM's, and support operating-system absent diagnostics and applications. In addition to this, UEFI capabilities are exported by C-callable interfaces, thus allowing for UEFI platforms to span a large class of platform and CPU microarchitecture. These interfaces run in the native machine mode and go beyond today's 16-bit real-mode BIOS.*

Inherent in the business deployment and interoperability of "extensibility", there is also the peril, namely control of system policy. One such policy, such as integrity control of the platform firmware and authorization of module launch, will be discussed in light of these business challenges and the emergent wave of malware in the market.

Keywords: Firmware, BIOS, Platform Trust, TPM, Measured Boot, Secure Boot

1 Introduction

The first several million instructions executed when a system is powered on are firmware, software stored in a chip. The firmware is responsible for initialization of much of the system, including important components such as RAM, video, and keyboards and mice. The firmware is responsible for finding and loading the operating system (such as Microsoft® Windows XP ® or Linux) from a number of different types of media, ranging from hard disks to LANs. Firmware then cooperates with the operating system to load further parts of the operating system before the operating system completely takes over. Today, on PC-class machines at least, that class of software is known as "BIOS."

What do we mean when we mention "UEFI" and "Beyond BIOS?" "UEFI" [1] is an industry group that is standardizing what were the EFI1.10 specification and the Framework PEI (Pre-EFI Initialization) and DXE (Driver Execution Environment) specifications [3][4]. Within UEFI, the main UEFI specification (beginning with UEFI2.0) is handled by the UEFI Specification Working Group (USWG) and the Platform Initialization (PI) Architecture PEI/DXE content are handled in the Platform Initialization Working Group (PIWG). The differences are that USWG is focused on the OS-to-platform interfaces, whereas the PIWG is focused upon platform initialization. The USWG parties of interest are the OS (operating system) vendors, pre-OS application writers, and independent hardware vendors who write boot ROM's for block or output console devices. The PIWG parties of focus include the platform builders, such as Multi-National Corporations/Original Equipment Manufacturers (MNC's/OEMs), chipset vendors, and CPU vendors. The PIWG work can accommodate both UEFI operating systems and today's conventional/Int19h-based OS's.

PIWG-based components provide one means of producing the UEFI interfaces. What is common across both UEFI and PI Arch is the extensibility of code and interfaces. For UEFI and PI DXE, the extensibility point is a loadable driver model and for PI PEI extensibility is through firmware files with PEI Modules. Although it is common to think of firmware as being stored in a ROM (Read-Only Memory), most firmware is stored in Flash devices, which act like a ROM but may be updated to add enhancements or fix bugs. The Flash devices are divided into the equivalent of sectors on a disk.

1.1 Chernobyl

In 1999 and 2000, the Chernobyl virus (also known as CIH 2) spread around the world. Although it did attack files on disks like most viruses, Chernobyl was the first major virus to attack firmware. Chernobyl simply sent the few commands telling a common type of flash device to erase the block of data where the first instructions executed are stored. Many systems were unaffected because they did not use that type of device. Others had built in safeguards.
Chernobyl's attack on firmware was very primitive but it left hundreds of thousands of systems non-functional because of a few tens of instructions.

Viral attacks on system software are becoming more fiendish. There is every reason to believe this trend will continue and that Chernobyl's attack will become the brother of many more dangerous viruses. Many vendors have taken steps to close the easy path Chernobyl used so its attack is not as successful anymore. Both sides are busy in the now escalating virus versus anti-virus war.

Pre-UEFI firmware was predominately written by the company who sold the computer or video card or SCSI card. Each vendor's system had a different architecture so it was difficult for a virus author to attack a variety of systems in these more subtle ways. UEFI's driver architecture offers more standardization and, as such, creates the type of commonality that is the benefit and curse of systems today: enhanced interoperability and enhanced methods for viral attack.

1.2 UEFI as an Enabler

The main dilemma in any firmware security discussion is that different applications require different levels of security. Just as the security in one's home isn't usually the same as (hopefully) in one's bank, one size does not fit all.
UEFI was designed using advice from security experts as to their expectations of requirements in the future as well as from those developing systems today. The architecture is designed to allow future expansion into developing areas of security without unduly burdening the normal requirements of us normal folks.

In the simplest case, no security is required. Consider, for example, a system that forms part of an advanced copier. Such a system is protected by its physical security. Only trusted staff can add or remove programs, so the risk of viral attack is small. Additionally, there is no secret data stored by the computer in this environment.

The majority of systems will fall into categories between 'none' and 'high security', with the spectrum describing platform that are willing to trade of some security for other concerns such as economics and performance. They are willing to trust to simple complexity and a locking cover for most physical security. Their systems are, as we have seen, increasing under viral attack.

Many flash devices have mechanisms to 'write protect' blocks. The concept is quite similar to the write protect switch on 3½ inch floppy disks except that the switch is reset at each boot and is electronic. Using this form of hardware support, a good level of security against viruses can be achieved. What happens when the firmware needs to be updated? Haven't we just locked out valid updates as well as invalid ones?

UEFI defines a 'capsule' update mechanism. The update, known as a capsule, is received by the Operating System from e.g. a web site. The operating system leaves the capsule at a predictable location and shuts down (more or less as if it is going into a save-to-RAM (S3) power saving state) which un-write-protects the blocks. The firmware gains control, validates the capsule using the public key validation, and performs the actual write. It then write-protects the blocks again and continues.

A capsule is distributed as a file and consists of the update image itself, descriptive data (so users can know if they need the update or not), possible programs to perform the update, and possible security data.

The program files inside UEFI are stored as a list of 'sections'. Some optional sections are defined as security sections. These sections contain the data, such as, in the case of the public key-based validation, the encrypted hash and the public key either for the file containing the section or for a group of files. This grouping shrinks the amount of overhead consumed by security.

The DXE Core supports the dispatch of the capsule by means of a new boot mode. PEI shall discover that the restart is a non-destructive restart with respect to memory. This can be activated via an INIT or an S3-like restart. PEI shall then coalesce the memory pages of the capsule. Within DXE, the BDS shall interact with the DXE Core in order to create a firmware volume with the capsule drivers and data.

In an interesting twist on the capsule concept, UEFI has a construct whereby a new key is provided with an update to an existing driver or set of drivers. If the old key is no longer valid (for example, if the secret part was published on a web sight), an update with a new key could be provided. To a certain extent, this addresses the issue of key replacement. Even without a secure clock, UEFI can provide a reasonable mechanism for rendering keys obsolete by way of monotonically increasing version numbers.

2 Static Measured Boot

The Trusted Platform Module (TPM) is a passive hardware device on the system board. It servers as the Root of Trust for Storage (RTS) and Root of Trust for Reporting (RTR). The former is the use of the Storage Root Key (SRK) and the Platform Configuration Registers (PCR's).

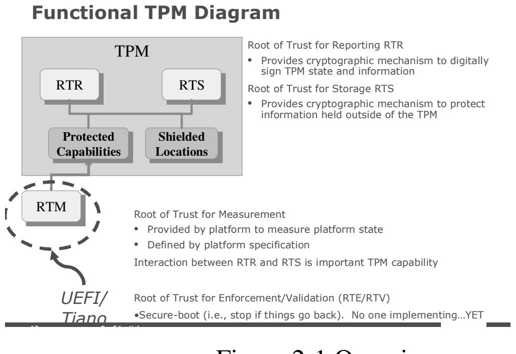
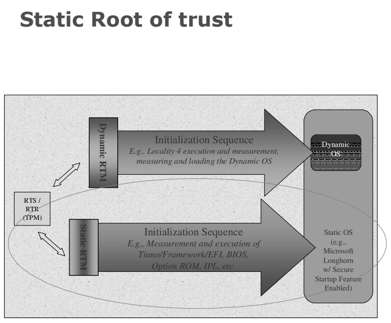

Functional TPM Diagram
TPM
RTR
RTS
Protected Capabilities
Shielded Locations
RTM
UEFI/ Tiano
Root of Trust for Reporting RTR
• Provides cryptographic mechanism to digitally sign TPM state and information
Root of Trust for Storage RTS
• Provides cryptographic mechanism to protect information held outside of the TPM
Root of Trust for Measurement
• Provided by platform to measure platform state
• Defined by platform specification
Interaction between RTR and RTS is important TPM capability
Root of Trust for Enforcement/Validation (RTE/RTV)
•Secure-boot (i.e., stop if things go back). No one implementing…YET

Figure 2-1 Overview

The active agent on the platform is the Root of Trust for Measurement (RTM). The RTM can be either Static or Dynamic (SRTM versus DRTM, resp). The DRTM is something in hardware with a CPU instruction to initialization, such as the SENTER instruction in Intel ® Trusted Execution Technology ® (TXT) [12]. The SRTM, on the other hand, entails trust chain from the platform reset vector going forward.

The definition of the RTM for UEFI is defined in the TCG Platform Specification [5].



Static Root of trust
Dynamic RTM
Initialization Sequence
E.g., Locality 4 execution and measurement, measuring and loading the Dynamic OS
Dynamic OS
RTS / RTR (TPM)
Static RTM
Initialization Sequence
E.g., Measurement and execution of Tiano/Framework/EFI, BIOS, Option ROM, IPL, etc.
Static OS (e.g., Microsoft Longhorn w/ Secure Startup Feature Enabled)

Figure 2-2 SRTM

There need to be UEFI API's available so that the UEFI OS loader can continue to measure the operating system kernel, pass commands to the TPM to possibly unseal a secret (e.g., symmetric key for Microsoft ® Vista BitLocker ® with the volume disk encryption), and perform other TPM actions prior to the availability of the OS TPM driver. In addition, this API can be installed at the beginning of DXE to enable measurement of the DXE and UEFI images.

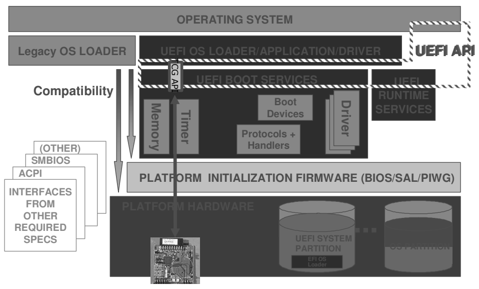
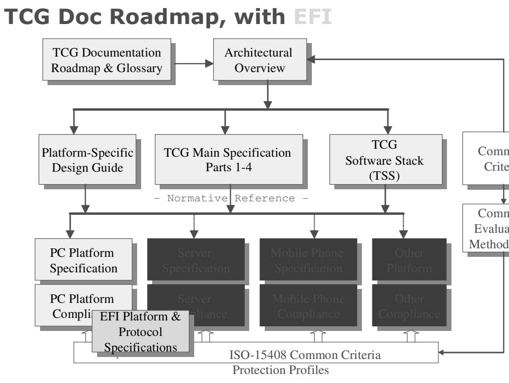


Figure 2-3 Overall UEFI Picture



TCG Doc Roadmap, with EFI
TCG Documentation Roadmap & Glossary
Architectural Overview
Platform-Specific Design Guide
TCG Main Specification Parts 1-4
TCG Software Stack (TSS)
Common Criteria
- Normative Reference -
Common Evaluation Methodology
PC Platform Specification
Server Specification
Mobile Phone Specification
Other Platform
PC Platform Compliance
Server Compliance
Mobile Phone Compliance
Other Compliance
EFI Platform & Protocol Specifications
ISO-15408 Common Criteria Protection Profiles

Figure 2-4 Document layout

The UEFI specifications are cross-listed in the TCG PC and Server Working Groups. Since UEFI is supported on both the Itanium ® Processor Family and x64, UEFI offers an opportunity to deploy a platform security solution across all of Intel's offerings.

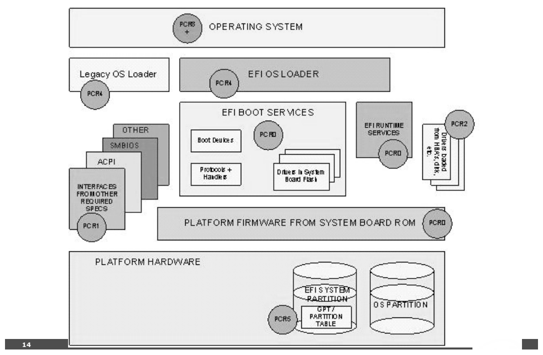


Figure 2-5 Measured objects in UEFI

The UEFI TCG Platform specification [5] describes which objects to measure in an UEFI System, such as the imagines, on-disk data structures, UEFI variables, etc.

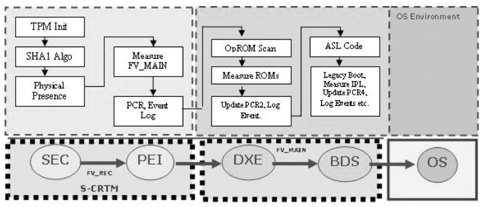


Figure 2-6 Measurement temporal flow

Prior to the "UEFI Phase" of platform execution, the Framework & PIWG describe the PEI and DXE phases. Therein, the CRTM is mapped to the PEI phase and what is thought of as BIOS POST is mapped to DXE. There are interfaces in PEI (namely, the PEIM-to-PEIM interface, or "PPI") to allow for fine grain measurement in that phase of execution, too.

What is the point of measurements?

The process of measurements records the state of the platform, both executable code and data hashes, into the TPM's platform configuration registers (PCR's). These PCR's are write-only and cleared upon a platform reset (at least the static PCR's for SRTM). The PCR's reflect the platform state. They are used such that software, when installed upon the platform, can "Seal" some information to the platform. A seal is like an encryption that also includes PCR's. There is a corresponding "Unseal" operation which is a decryption that also uses the PCR's.

What this practically means is that if the state of the platform changes between the installation of some software (and the Seal operation) and successive invocations of software on later restarts (and the use of Unseal operation), unauthorized changes to the platform in the interim will be detected (i.e., PCR's changed).

This is sort of the Resurrecting Duckling [6] security model wherein the initial state of the platform (i.e., PCR values upon installing application) are considered good. A stronger usage would be to have a platform validation credential/platform certificate that listed the expected values of the PCR's for the as-shipped-from-manufacturer state of the platform. The use of platform cert's, etc, has not been prevalent.

UEFI offers an opportunity here, though, since the addition of UEFI entails a large disruption in the platform model. Ubiquity of x64 boot, more stringent ACPI [7] implementation checking by the OS, etc, are other disruptions in the platform that will occur in the UEFI wave. As such, a more complete TCG implementation can also be driven with these other changes.

In addition, the CRTM update needs to occur in an appropriate manner. One recommendation is to use RSA-2048/SHA-256 [13] capsule update.

This is important to enable the CRTM maintenance described in the Trusted Building Block (TBB) protection profile. Either the CRTM is immutable, or never changed in the field, or appropriate cryptographic techniques need to be employed in order to update the CRTM. The signed capsule update is one way to enable the latter capability and ensure the platform firmware integrity.

OS usages of SRTM
One of the uses of the static measurement includes the Microsoft Vista BitLocker Feature. This uses the loader, in tandem with the TPM, to encrypt the full disk volume.

Another use of the integrity measurements includes the Linux Integrity Measurement Architecture (IMA) [11].
This is an enterprise capability where it is key to ensure that the servers are patched and in the correct software state. There is no presumption of consumer usability or privacy.

3 Secure Boot

The SRTM and measured boot will just record the platform state. This is "trusted boot." To get to a "Secure Boot", or stop the boot process if unauthorized code appears, there needs to be some policy in the system.

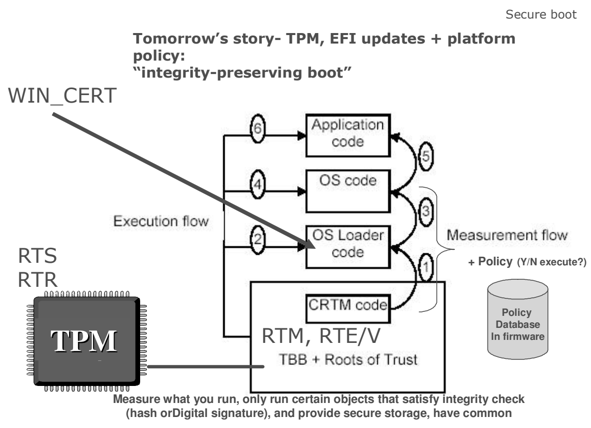
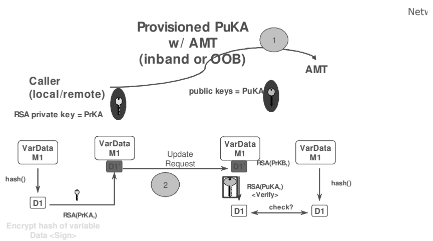

Secure boot
Tomorrow's story- TPM, EFI updates + platform policy:
"integrity-preserving boot"
WIN_CERT
Application code
OS code
OS Loader code
CRTM code
Execution flow
Measurement flow
+ Policy (Y/N execute?)
Policy Database In firmware
RTS
RTR
TPM
RTM, RTE/V
TBB + Roots of Trust
1
2
3
4
5
6
Measure what you run, only run certain objects that satisfy integrity check (hash orDigital signature), and provide secure storage, have common

Figure 3-9 Secure Boot

The "Secure Boot" is really a term from the TCG, which differs from the "Measured" or "Trusted" boot earlier in that if the system is not in a given state (i.e, unauthorized code is discovered), the boot process will omit the unauthorized module or go into some recovery mode. The original TPM 1.1b imagined this scenario with the DIR hardware, and the TPM1.2 allows for this with its more general NVData. Arbaugh initiated the concept of Secure Boot with his Aegis architecture [2].

The "Secure Boot" differs from the Trusted Boot in that the RTM is the active agent for Trusted Boot, whereas the "Secure Boot" requires an RTE/RTV (Root of Trust for Enforcement/Validation) in order to effect the policy decision. The RTE/RTV can also leverage the RTM actions and the RTS/RTR of the TPM, too (i.e., use the TPM to store white list of image hashes or public keys used to corroborate the image integrity).

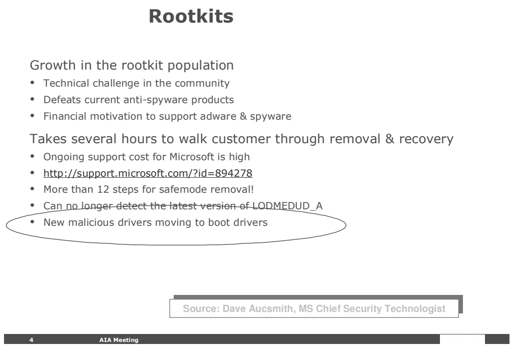


1 – Provision the public key w/ the AMT (in-band or OOB)
2 – Perform the authentication of any in-band or OOB BIOS update in the AMT. Use TPM for crypto and secure Storage.

Figure 3-10 Key exchange

The key exchange can be used to pass policy into the firmware. An alternate means of configuring this capability would be a pre-OS setup screen with appropriate PoP (Proof-of-Presence).

The use of "Secure Boot" with a white list of public keys for signed images helps solve the "scalability issue" of measured boot, namely the plurality of different hashes. By signing the components, you have a level of indirection, namely a (hopefully) smaller set of public keys and root stores, versus a much larger number of possible hashes.
Is the vulnerability of a boot time OS driver real? Below is data from Microsoft on this industry concern:



Rootkits

Growth in the rootkit population
- Technical challenge in the community
- Defeats current anti-spyware products
- Financial motivation to support adware & spyware

Takes several hours to walk customer through removal & recovery
- Ongoing support cost for Microsoft is high
- http://support.microsoft.com/?id=894278
- More than 12 steps for safemode removal!
- Can no longer detect the latest version of LODMEDUD_A
- New malicious drivers moving to boot drivers

Source: Dave Aucsmith, MS Chief Security Technologist

4 AIA Meeting

Figure 3-1 Real-world threat

Specifically, malware is has moved from user into kernel space. And now the kernel-space malware is afflicting the operating system's earliest, boot-drivers.

4 Formal Security Model

In addition to the measured and secure boot description above, a system designed needs to have some guarantee or assurance that the firmware implementation can implement those capabilities. As such, one option is to apply a formal security model. Since the platform firmware mostly relates to "integrity" as a security goal, the Clark Wilson model is one candidate. Therein, we have the following terms-

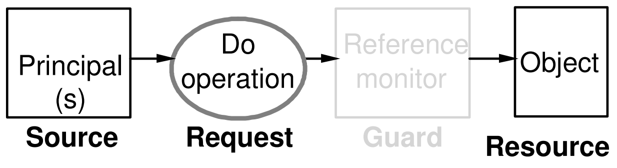
Data Items: In order to differentiate between data items that are integrity constrained and those that are not, CW uses the term "Constrained Data Items (CDI) to represent data items that are of high integrity and "Unconstrained Data Items" (UDI) to represent the others.

Subjects and data transformation procedures: Subjects transform data in the system and act on data by calling Transformation Procedures (TP), which are sequences of atomic actions in the system.

Integrity Validation Procedure (IVP): An IVP checks to see if a CDI has the specified degree of integrity, an example of this is a HMAC.

A subject can be an administrator (human), but more likely, a program (UEFI loader/application/driver/OS kernel for the UEFI runtime).

A CDI would include the UEFI boot option/global variables that describe the drivers and OS loaders to invoke.



Source = Caller (pre-OS or OS-present)
Request = Variable Write
Reference Monitor = Platform Firmware
Object = EFI Variable

Figure 4-1 Formal model picture

The IVP in UEFI secure boot would be the RsaVerify() routine in our LoadImage() service that checks the digital signature of the WIN_CERT [1] associated with the .efi image. A transformation procedure can include an invocation of the UEFI SetVariable() services (thus the Authenticated Variable work) or LoadImage()/StartImage().

RSA asymmetric signatures are used because of the expiration of the RSA patent and wide availability; 2048 key lengths are recommended. SHA-256 is used, versus SHA-1, because of recent vulnerabilities discovered in the latter [10].

Longer term, Elliptic Curve Cryptography (ECC) may be a better choice, since a 300 bit key is equivalent to a 2000 bit RSA key, thus saving on code space in flash-usage-conscious firmware. NSA Suite B [9] argues for the use of ECC, but implementation and other issues have yet to be explored with respect to productization or use in the firmware standards.

The Clark-Wilson (CW) model sets up the following security principles:
C1: IVP's must be available on the system for validating any CDI.
C2: Application of a TP to a CDI must preserve integrity of the CDI – the set of CDI are closed under TP actions.
C3: A CDI can be changed only by a TP: This may not always be possible, for example when a CDI is stored on a disk that is accessible to non-TPs. In this case, delete must be considered to be a TP.
C4: Subjects can only initiate certain TPs on certain CDI: For example, if we have policy data for a layer, then the policy data must be changed only through the actions of a well-known and analyzed TP.
C5: Certain TP on UDI can produce CDI as output: This is a major difference from Biba [14] policies. In this case, a guard can accept data of unknown integrity and verify or transform it into a CDI.
C6: The system must authenticate subjects performing a TP: For example, in our case, the system must enforce some authentication policy on someone seeking to edit the k-mode root store.
C7: Only special subjects (security administrators) must be able to change any authorization related lists. In our case, only security admins should be able to change layer wide policy such as the type of authentication permitted at each layer.

A Clark-Wilson evaluation helps at the system design level; namely, are we protecting the correct assets, etc. In addition to the design phase, this methodology can be used such that an evaluator or security designer can decompose a given UEFI implementation into the subjects, CDI's, UDI's, and note the IVP's. One question to ask, for example, is if these items, when refined to a given system, support the C-numbered principles listed above? By doing so, the evaluator assess whether the security goals of the measured and secure boot have been compromised by an errant implementation.

5 Conclusions

This paper has shown that the future of extensible platform firmware beyond BIOS holds many perils and opportunities. The perils include the new ability to have extensible code loading in the pre-operating system regime, but the opportunities include the use of measured and secure boot to harden the platform and authorize code loading. And in a world of ever-more-secure operating systems, the pre-OS may become a more interesting target for the Blackhat's of the world. As such, these UEFI protections are even more important to implement.

4 References

[1] Unified Extensible Firmware Interface Specification Version 2.1, January 23, 2007. http://www.uefi.org.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith, ``A Secure and Reliable Bootstrap Architecture," in *Proceedings 1997 IEEE Symposium on Security and Privacy*, pp. 65-71, May 1997.

[3] Vincent Zimmer, Michael Rothman, Robert Hale, Beyond BIOS: Implementing the Unified Extensible Firmware Interface Specification with Intel's Framework. Intel Press, September 2006. ISBN 0-9743649-0-8.

[4] Vincent Zimmer, "Advances in Platform Firmware Beyond BIOS and Across all Intel® Firmware", Technology @ Intel Magazine, January 2004. http://www.intel.com/technology/magazine/systems/it01043.pdf.

[5] Trusted Computing Group EFI Protocol Specification, Version 1.2. https://www.trustedcomputinggroup.org/specs/PCClient

[6] Frank Stajano, Ross Anderson, "The Resurrecting Duckling: Security Issues for Ad-Hoc Wireless Networks," Lecture Notes in Computer Science, Issue 1796. Springer-Verlag, 1999.

[7] Advanced Configuration and Power Interface (ACPI) Specification, Version 3.0b, http://www.acpi.info.

[8] D. Clark and D. Wilson, "A Comparison of Commercial and Military Security Policies," IEEE Symposium on Security and Privacy, 1987.

[9] NSA Suite B Cryptography. www.**nsa**.gov/ia/industry/crypto_**suite_b**.cfm

[10] X. Wang, Y.L. Yin, and H. Yu. Finding Collisions in the Full SHA-1, Advances in Cryptology -- Crypto'05.

[11] Integrity Measurement Architecture. http://domino.research.ibm.com/comm/research_projects.nsf/pages/ssd_ima.index.html

[12] Intel ® Trusted Execution Technology. http://download.**intel**.com/**technology/security**/downloads/31516803.pdf

[13] Secure Hash Standard. csrc.**nist**.gov/publications/fips/fips180-2/fips180-2.pdf

[14] K. J. Biba, "Integrity Considerations For Secure Computer Systems," ESD-TR-76-372, NTIS#AD-A039324, Electronic Systems Division, Air Force Systems Command, April 1977