2015 16th International Workshop on Microprocessor and SOC Test and Verification

Modeling and Analysis of Trusted Boot Processes based on Actor Network Procedures

Mark Nelson
Dept. of Information and Computer Sciences
University of Hawai'i at Manoa
Honolulu, HI
marknels@hawaii.edu

Peter-Michael Seidel
Dept. of Information and Computer Sciences
University of Hawai'i at Manoa
Honolulu, HI
pseidel@hawaii.edu

***Abstract*—We are discussing a framework for formally modeling and analyzing the security of trusted boot processes. The presented framework is based on actor networks. It considers essential cyber-physical features of the system and how to check the authenticity of the software it is running.**

***Keywords*-trusted boot process; secure boot process; trusted platform module; formal model; actor network procedures.**

I. Introduction

The existence and widespread distribution of bootkits, rootkits and hardware viruses [1], [2], [3], [4] indicate increasing attempts to penetrate computer systems at lower levels in order to escape traditional observations. Even overwriting non-volatile random-access memory (NVRAM) can trigger a Denial-of-Service attack [5]. This type of malware takes control before anti-virus software can detect it. Recent improvements to firmware standards [6], [7], [8] and the introduction of trusted boot processes [9], [10], [11], [12], [13], [14], [15] show the commercial need and interest to address and prevent such low level attacks.

Trusted boot processes incorporate many layers and phases, each requiring the ability to be updated. Early implementations of trusted boot processes contained vulnerabilities demonstrating that the development of commercial trusted boot solutions is not straightforward and may be an error-prone task. Yet, describing the protocols and features of a specific trusted boot process in a rigorous framework that would allow for formal analysis is non-trivial and requires a methodology to not only model hardware and software components, but also physical devices, identities and locations. Conventional models to describe computer hardware, software and network protocols typically fail to describe distinguishing physical features such as sensors, actuators and locations.

In [16], a multi-factor authentication process is analyzed using actor networks and Protocol Derivation Logic (PDL). We are applying this framework to model trusted boot processes and analyze the integrity of the layers of software used to administer, update and run the system.

II. Related Work

The evolution from BIOS (Basic Input/Output System) to UEFI (Unified Extensible Firmware Interface) is still in progress. Standards organizations such as the Trusted Computing Group (TCG), the UEFI Forum and the National Institute of Science and Technology (NIST) have collaborated with industry partners to define a common set of implementation guidelines how secure chains of trust should be implemented.

NIST has developed recommendations for BIOS Integrity and Measurements [8], [17] intended to help establish a secure measurement and reporting chain. However, BIOS vendors are free to implement and market 'secure boot' as they see fit.

Originally developed by Intel, the UEFI Specification [6] and the UEFI Platform Initialization Specification describe a framework that BIOS manufacturers, hardware designers and operating system vendors can use to securely boot a system.

The TCG encourages vendor-neutral collaboration and publishes an extensive set of specifications for a broad spectrum of technologies. One of TCG's working groups manages the Trusted Platform Module (TPM) specifications [7], [18], [19] that describe the API (application program interface) and use models for a critical phase in the secure boot process. TPMs were not part of the original PC-AT design. A commonly accepted TPM specification [20] was published in 2003 and by 2009, TPMs had reached critical mass in that they were broadly available on most PC platforms [21], and vendors could rely on their presence. TPMs have also been adopted in mobile and networking devices.

TPM vendors such as Atmel [22], Infineon [23] and Intel [24] publish application notes documenting recommended procedures and details to help designers secure their systems. While these references describe established and commercially used methods, protocols and responsibilities for implementing trusted boot, there is a growing number of studies that illustrate various shortcomings of implementations or weaknesses within the specifications themselves.

BIOS implementation flaws in laptops are described in [25]. In [3], the authors describe weaknesses in the BIOS update process such that if malware were to gain kernel access, undetectable code could be written to flash memory. In 2013, procedures for bypassing Windows 8 Secure Boot by writing into NVRAM and disabling trusted boot have been found [2]. In 2014, overflow vulnerabilities in UEFI firmware were discovered that allowed application software (helped by Windows 8) to introduce malware as a pre-boot driver [1].

III. Actor Network Procedures

Today's computers do not work autonomously. Being operated by a user and being connected to many other computers and devices, they are part of a heterogeneous network that extends into the physical and into cyber space and may involve many active participants. This connectivity has significant implications for the operation and security of any computer.

In [16], a formal framework of actor networks is developed to analyze processes in cyber-physical networks of active participants. The active participants in this framework are the actors. They could be people, computers or devices. The interactions between actors are realized through physical, visual and network channels. Each actor coordinates with other actors by sending and receiving stimuli through available channels following a protocol. Actors do not work autonomously. They interact and collaborate to perform tasks. Groups of actors can form hierarchical structures. A thorough introduction to the formal Actor-Network framework is given in [16].

A. Definitions

Actors are the active entities in this framework. They have a state that advances based on internal processes of the actor and according to the protocols that determine their interaction with other actors through available channels. Actors can be computers, programs, devices or people. Actors may also be physical sensors or actuators. This paper uses upper-case identifiers to refer to actors, e.g. A, CPU_1, TPM.

Predicates identify the state of an actor. They are denoted by the actor and the state as in: $<TPM, reset>$.

Transactions are executed by actors. A transaction is initiated by a single actor and advances the state of the actor itself or of other actors. Transactions are denoted in square brackets, as in: $[encryptdata]$ or $[executecode]$.

Channels define a connection for an interaction between actors. Channels may be visual, physical or 'cyber'.

A *Conversation* or *View* is the protocol of an interaction as seen from the viewpoint of one of the actors.

A *Protocol* is providing the rules to constrain the actors' interactions, and allow executing transactions to move and make progress through their states.

B. Analysis Methodology

For each security property we go through the following steps:

1) define the actors involved,
2) relate the actors by mapping the channels between them and define static assertions,
3) model the protocol - for each actor:
   a) determine the initial predicates and assumptions,
   b) analyze and propagate the actors' states by following transactions to reach target security assertions,
4) generate the local views of the actors for their security assertions (optional),
5) analyze the target security property, extract local requirements and identify potential vulnerabilities.

IV. Secure Boot Processes

A. Manufacturing and Branding

There are seemingly infinite ways to manufacture computer systems and every permutation has its own name (Original Equipment Manufacturers, Private Label, etc.). We use the term CM for Computer Manufacturer, however, the term could be extended to everything from automobiles to watches.

Factory issue TPMs are typically customized by the CM. Customizations[1] include removal of test keys, establishing a physical presence detection mode, setting the TPM authorization mode and endorsement key generation.

Because the TPM is at the root of a system's trust, the authors of the TPM specification include special protections around commands that can write low-level cryptographic information into the TPM. A Physical Presence (PP) sensor is intended to raise the bar for attackers by requiring physical access to a system. PP detection is required to reset the TPM's Endorsement Keys. PP can also be added to the authorization policies for commands or to access objects or keys. In the context of this presentation, PP is the main requirement that exposes user interaction and physical features of the system. By showing that such system features can be included in the model, we are opening the analysis of computer security concerns to more general cyber-physical mechanisms. Other examples of physical aspects of the boot process that we do not explicitly mention in this presentation are the physical on/off switch of the system, a physical key to enter the firmware configuration menu, a physical button or switch on the motherboard to reset the BIOS code to factory settings or to make it read-only, or the option to boot from devices that are physically attached to the computer system.

It is important to note that the term 'Physical Presence' for the TPM is not always an intuitive description. The TPM specification allows for two forms of PP[2]:

- Hardware asserted PP through a pin of the TPM module,
- PP asserted via software commands.

Both methods have benefits and weaknesses. Hardware asserted PP has the benefit of requiring some type of physical access to a system. For those who are responsible for fleets of devices, the security benefit may not be worth the operational expense[3]. The threat in this scenario would be a PP sensor that is controlled by a malicious actuator.

Physical Presence asserted by software commands is intended to be used by CMs and BIOS manufacturers to confirm PP via a console or CAPTCHA[4]. Of course, the downside is the potential thread for having malware mimic PP.

System designers must decide how to implement Physical Presence detection and then lock down the intended method(s) for the lifetime of the TPM.

[1]Infineon calls this process 'personalization'[23].
[2]Either or both may be used.
[3]Home DSL routers, cellphones or automobiles
[4]Completely Automated Public Turing Test To Tell Computers and Humans Apart: http://www.captcha.net

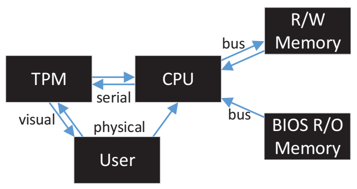

Fig. 1. Secure Boot Channels

There have been documented instances of CMs misconfiguring BIOS firmware and TPMs [2]. The TCG has detailed instructions describing the implementation requirements[19]. However, this level of customization remains a technically complex process that has resulted in products with a degraded root of trust.

B. Routine Secure Boot

The detailed boot process is as varied as the number of products in the market. The purpose of the paper is to study protocols and not a specific implementation, therefore we model a representative computer system with a TPM that boots into a trusted environment. A refinement of our analysis can be targeted to a specific system implementation. We focus in our presentation on selected boot and update protocols to illustrate our analysis framework. In the presented model we are mostly concerned *if the system be constrained to exclusively run cryptographically checked authentic software.*

Four actors participate in the initial process of secure boot: TPM, CPU, General Purpose Read-Write Memory (RWM) and Read-Only Memory (BIOS)[5]. Figure 1 shows the channels between the actors. This includes the USER with physical and visual channels which is not taking part in the first phase.

The channels establish static assertions for the model:

- {TPM} serial {CPU}, {CPU} serial {TPM}
- {CPU} bus {RWM}, {RWM} bus {CPU}
- {BIOS} bus {CPU}

Actor Network models may identify the set of transactions that are valid for the actors and channel, but that level of detail is not needed in this phase.

With the channels established, we model the first boot process *to verify that the BIOS software is authentic.* This is called the Core Root of Trust Measurement or CRTM. The actors in the boot process follow the protocol in figure 2.

The actors' initial states (predicates) are: $<TPM, reset>$, $<CPU, reset>$, $<RWM, undefined>$ and $<BIOS, versionX>$.

It is unrealistic to assume that the first step a BIOS will perform is to measure the TPM. The BIOS software needs to initialize the CPU's addressing mode, configure the memory manager and initialize the TPM.

In time, the BIOS will start the CRTM process. This test is intended to be the simplest, smallest-scope measurement

[5]Another important element is NVRAM, but it does not play a unique role in the protocol of this phase of the boot process.

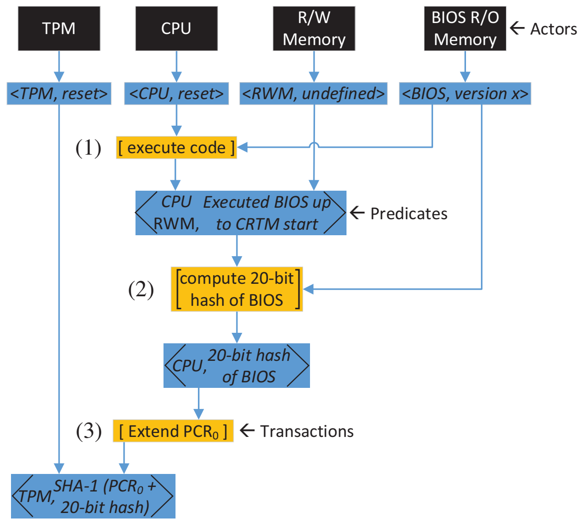

Fig. 2. Core Root of Trust Measurement

of the pre-boot environment. The CPU reads the BIOS and computes a 20-bit hash of its contents.

The length of this hash is too short to be cryptographically secure. This insecurity is ameliorated by the TPM's 'extend' operation. The PCR register does not store the 20-bit hash. Instead, it computes a 160-bit SHA-1[6] hash that is computed by appending the PCR's current 160-bit value with the 20-bit hash. This process is called 'extension' and the resulting 160-bit hash is stored in the PCR. This rolling process makes predicting or forcing the PCR value to a specific 160-bit string computationally difficult.

C. Model Verification

This section builds on the previous step to check *if a system can be constrained to exclusively run cryptographically checked authentic software* by making several assertions. The transactions are listed in figure 3 as three equations. We refer to the equation numbers from this figure in the descriptions below and by the numbers in figure 2.

1) The CPU must only execute code from BIOS ROM (see Fig. 3, Eq. 1): This model assumes that the BIOS is actually ROM and is in no way writable. One challenge that our Actor Network model identifies is the inability to verify in software that the BIOS is indeed stored in ROM while maintaining a secure thread of execution. Attempting a BIOS ROM write would trigger a segmentation fault, which flows through an interrupt vector table stored in RWM.

2) If the CPU has only executed code from BIOS ROM, then when it computes the 20-bit hash of the entire BIOS, it must be accurate (see Fig. 3, Eq. 2): This model demonstrates the danger of storing the 20-bit hash in R/W Memory. The Actor Network model can not use any *guarantee* that RWM will maintain its state between storing a value in RWM and retrieving it. One could assume that as long as the thread

[6]The SHA-1 algorithm can be supplanted in TPM 2.0 modules.

$$<BIOS, version\,X> \quad \text{bus} \quad \{<CPU, reset>\ [execute\,code]\ <CPU, CRTM\ start>\}_{CPU} \tag{1}$$

$$<BIOS, version\,X> \quad \text{bus} \quad \{<CPU, CRTM\ start>\ [compute\ 20\text{-}bit\ hash]\ <CPU, 20\text{-}bit\ hash>\}_{CPU} \tag{2}$$

$$\{<BIOS, version\,X>\ \text{bus}\ \{<CPU, 20\text{-}bit\ hash>\ [extend\ PCR_0]\}\} \quad \text{serial} \quad \{<TPM, accurate\ PCR_0>\}_{TPM} \tag{3}$$

Fig. 3. Routine Secure Boot Equations

of execution stays in BIOS, the contents of RWM should be predictable; however, multiprocessong, NUMA memory models and PCI-based DMA all have the potential to change the state of memory between storage operations while a CPU is running code from the BIOS ROM. Furthermore, a careful reading of the model[7] shows that the safest way to implement code for the BIOS hash process would avoid any branching that uses the stack[8] or data reads from RWM. Data reads from BIOS ROM would be trustworthy.

Although the model does not explicitly demonstrate this, it is worthwhile to note that a simple checksum of the BIOS ROM is insufficient. If the BIOS ROM were manipulated together with additional compensations that keep the same checksum, then a modified BIOS ROM could go undetected. The model explicitly calls for a strong 20-bit hash function that has the characteristics of a one-way function.

3) The TPM's PCR_0 is extended by an accurate 20-bit hash (see Fig. 3, Eq. 3): Our Actor Network model reveals that, from the TPM's perspective, the only security statement that Step 3 can make on its own is that at one point, the CPU had a 20-bit hash value. This is well understood in the TPM community, but it is critical that CMs understand this as well. Should malware get access to the thread of execution or the 20-bit hash before it is extended into the PCR, the malware could extend a seemingly correct 20-bit hash into the PCR and all of the downstream code would be under the impression that the system is trusted.

If all of the assumptions hold, Equation 3 will have an accurate value in PCR_0. This implies that the value is a true reflection of the contents of the BIOS. *Checking the BIOS integrity* explores how to determine the trustworthiness of the BIOS.

V. Boot Process Primitives

TPM modules provide considerable flexibility with respect to how secrets are stored and later accessed. The details of these mechanisms are beyond the scope of this paper. Furthermore, the standards committees have published guidelines but they are not as specific as a protocol that can be modeled. The following sections describe a constructed Secure Boot scenario that we can model with Actor Networks.

The process for secure BIOS updates is significantly different between TPM 1.2 and 2.0. The term *PCR Fragility* refers the challenge of securely updating a BIOS whereby the original BIOS's PCR values have been sealed and a new set of PCR values representing the new BIOS must be authorized – without leaving the system vulnerable. In TPM 1.2, every policy or encryption key sealed with a PCR must be unsealed, the PCR updated and then resealed. This leaves the system vulnerable for a period of time and requires careful programming. In TPM 2.0, the PolicyAuthorize command allows policies to change by letting an authority authorize a new policy that can be substituted for an old policy. This should reduce the vulnerability time to zero.

The operational concept is to store a value in the TPM. The value does not have to be a secret, what is important is to satisfy the TPM policy that is required to read or write the value[9].

A. Manufacturing a Secure BIOS

To enable secure boot, CMs must configure information about the BIOS into the TPM during the manufacturing process. This information should be digitally signed so that the BIOS' authenticity can be verified.

A public key will be stored by the Platform Owner (requiring Physical Presence verification) that will be the basis for trusting BIOS firmware updates.

In this example, the CM will create an NV index with a firmware build number. This index will be protected by the following policies:

Read Policy requires all of the following

- *TPM2_PolicyPCR*: PCR_0 must be a specific value
- *TPM2_PolicyTicket*: CPU must get a ticket from TPM2_VerifySignature that verifies a hash value has been signed by the CM's private key

Write Policy requires all of the following

- *TPM2_PolicyPhysicalPresence*
- *TPM2_PolicyTicket*: The new BIOS should be hashed and verified that it's been signed by the CM's private key
- *TPM2_PolicyNV*: The new version should be $\geq$ the current contents of this NV index
- *TPM2_PolicyPCR*: PCR_0 must be a specific value
- *TPM2_PolicyAuthorize*: This policy allows a new PCR_0 value to be substituted into this Write Policy

B. Checking BIOS integrity

Figure 4 documents the protocol used to determine if an *accurate* PCR_0 should be trusted or not.

As before, it's vital that the thread of execution stays within the BIOS during this protocol.

[7]We don't explicitly include this in the presented discussion.
[8]Such as JSR/RET subroutine calls.

[9]Values stored in the TPM's NVRAM are called NV indexes which have separate policies for reading and writing.

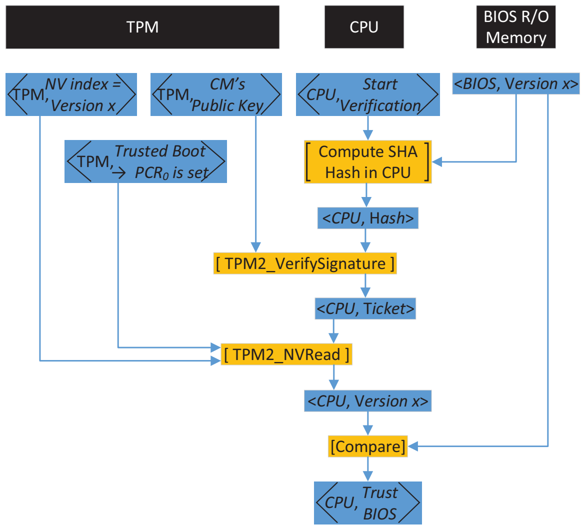

Fig. 4. BIOS Verification Protocol

C. *Updating the BIOS*

NIST has published BIOS Protection Guidelines[17], but the standards committees have not published a recommended procedure for BIOS updates. They do, however, require that vendors of consumer products publish a set of security attributes which documents the BIOS Update Mechanism for a vendor's product[8]:

- None - It is not possible to update the BIOS
- Capsule - Update the BIOS from the OS
- System Management Mode
- Other

While this information is helpful to establish a secure BIOS update from a vendor, The Practical Guide to TPM states that as of 2015, "We know of no OEM who are currently providing platform certificates"[20].

The TPM would not normally act directly as an approving authority for write operations into the BIOS. In our constructed system, the BIOS may be updated at any time; the goal is to re-establish trust when the BIOS is modified. The protocol to do this is straightforward. By satisfying all of the criteria for the NV index's Write Policy, the BIOS software can securely update both the firmware build number and the hash sealed to PCR_0.

For PCR_1, which by convention holds information about the motherboard's CMOS NV RAM (such as boot order, USB legacy mode, etc.), the same protocols could be used, but the TPM Platform's private key could be used instead of the CM's private key.

VI. Extending the Trusted Platform

After the BIOS has started and the CRTM has been verified, the system will use the UEFI Secure Boot process to load UEFI binaries[6]. The OS boot loaders can extend the trusted platform until the system is both fully operational and trusted. For example, Microsoft utilizes technologies such as Measured Boot and DeviceGuard[26] – which maintains a whitelist of allowed applications. The common theme for these post-CRTM technologies is that they inspect the software *before* loading and executing it. Trust is derived from a database of trusted public keys (certificates) maintained by the device. This database could be kept in a TPM module[10] for use during the UEFI Secure Boot phase.

UEFI's Secure Boot binaries are UEFI applications, such as diagnostics tools, boot menus or graphical configuration applications. UEFI binaries can also be device drivers and OS boot loaders. All signed UEFI binaries must be Portable Executable (PE) files that conform to Microsoft's Authenticode format.

The Authenticode standard embeds a signed hash, checksum and public key into the PE file. This file format, however, permits unsigned content to be introduced into an image file[27]. It can also allow sections to be rearranged. Weak implementations may also allow unsigned content to be introduced between sections or at the end of the image file[11].

With all of the protections offered by Secure Boot, there exists a route to introduce undetected malware into the memory of a target system; however, there is no inherent path to execute this code. Furthermore, although it may be referenced in other sources, there are no requirements in the UEFI specifications on the use of the No Execute (NX) capability to protect PE data regions (such as certificates) from being executed.

A. *Updating Secure Boot images*

Figure 5 models the process for updating the trusted public keys used by the UEFI Secure Boot Process. Other processes for producing UEFI Secure Boot PE images are straightforward and well documented[28][29]. Furthermore, because UEFI Secure Boot PE files are stored on untrusted media, it is unnecessary to model how they are updated.

UEFI maintains a database of trusted public keys that can be used to verify Secure Boot PE images. This database is maintained as a TPM NV index protected by the following policies:

READ POLICY requires all of the following

- *TPM2_PolicyPCR*: PCR_0 through PCR_n must be a specific value

WRITE POLICY requires all of the following

- *TPM2_PolicyPhysicalPresence*
- *TPM2_PolicyTicket*: The caller must have access to the Platform's Private Key or a UEFI Key Exchange Key
- *TPM2_PolicyNV*: The new version should be $\geq$ the current contents of this NV index
- *TPM2_PolicyPCR*: PCR_0 through PCR_n must be a specific value
- *TPM2_PolicyAuthorize*: This policy allows a new PCR_x value to be substituted into this Write Policy

[10]The OS could also use a small TPM-hosted database.

[11]Microsoft has had years of experience in writing Authenticode loaders, UEFI implementors may lack the benefit of prolonged exposure to attacks.

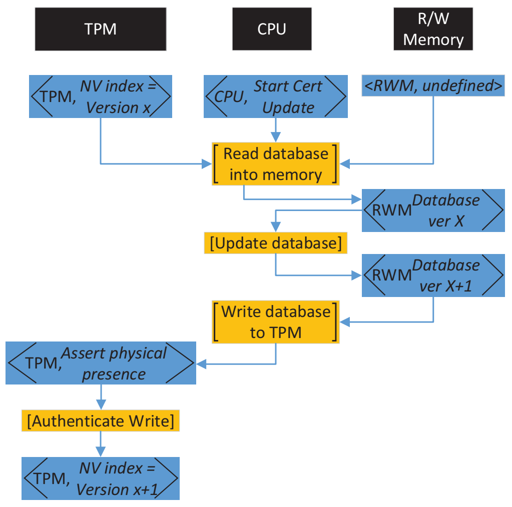

Fig. 5. Protocol to Update the UEFI Secure Boot Trust Database

VII. Conclusions

We have presented and applied a framework for the modeling and analysis of trusted boot processes. The framework is generally capable of modeling the essential details of an entire boot process from a systems perspective. The framework is also expressive enough to capture individual phases and components in greater detail and to refine implementation features to identify and analyze local requirements for components at various granularities. In an application of the presented framework we have focused on a few phases of a boot process.

Regarding the analysis we can express general security requirements for the trusted boot process and its phases. A closer analysis of individual components can translate requirements and adjust assumptions to the details of an implementation.

In our specific application we analyze the security of the BIOS verification protocol and the UEFI update protocol. Our analysis allows to find a few assumptions of system components that could be hard to guarantee in common systems and that could lead to weaknesses in the trusted boot implementation. These assumptions should be thoroughly checked in commercial systems.

Some aspects of our assumptions go beyond the UEFI recommendations. As one such example our presented model assumes there is no write channel to the BIOS. Current standards do not require UEFI implementors to test this.

The naming of the TPM Physical Presence (PP) indicator may be misleading. This indicator can be asserted entirely through software [19], [22]. Overlapping TPM policies such as secret keys and signed, time sensitive attestations should be used to strengthen access to TPM secrets.

UEFI Secure Boot relies on signed Authenticode files, which may contain unsigned binary data. While there is no direct path to execute the data, other vulnerabilities could still let it be a dangerous content in memory.

While the presented framework is expressive enough to model complex boot processes, the usability of the formalism needs some improvements for better scalability. The current formalism needs an extension to simplify switching between different actor network configurations. This will make it easier to analyze multiple phases of the process in a single diagram. It will also assist translating the framework to a form that supports machine-assisted analysis.

References

[1] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell, "Extreme privilege escalation on UEFI Windows 8 systems," in *Hack.LU*, 2014.
[2] Y. Bulygin, A. Furtak, and O. Bazhaniuk, "A Tale of One Software Bypass of Windows 8 Secure Boot," *Black Hat*, 2013.
[3] R. Wojtczuk and C. Kallenberg, "Attacking UEFI Boot Script," in *31st Chaos Communication Congress*, 2014.
[4] S. Embleton, S. Sparks, and C. Zou, "Smm rootkits: A new breed of os independent malware," *ACM*, 2008. [Online]. Available: http://www.eecs.ucf.edu/ czou/research/SMM-Rootkits-Securecom08.pdf
[5] M. Yamamura, *W95.CIH*. http://www.symantec.com/security_response/writeup.jsp?docid=2000-122010-2655-99, 2002.
[6] UEFI, "UEFI Specification Version 2.5," 2015.
[7] TCG, *TPM Library Specification*, 2nd ed., October 2014.
[8] A. Regenscheid and K. Scarfone, *BIOS Integrity Measurement Guidelines*. NIST, December 2011, no. 800-155 (Draft).
[9] Intel, *Intel Trusted Execution Technology (Intel TXT) - Software Development Guide*, July 2015.
[10] Microsoft, "Secured boot and measured boot: Hardening early boot components against malware," Microsoft, Tech. Rep., 2012.
[11] V. J. Zimmer and M. A. Rothman, "System and method to secure boot UEFI firmware and UEFI-aware operating systems on a mobile internet device (mid)," 2008, US Patent App. 12/165,593.
[12] S. Sinofsky, "Protecting the pre-OS environment with UEFI," 2011, https://blogs.msdn.microsoft.com/b8/2011/09/22/protecting-the-pre-os-environment-with-uefi/.
[13] J. Bottomley and J. Corbet, "Making UEFI secure boot work with open platforms," *The Linux Foundation*, vol. 49, 2011.
[14] R. Wilkins and B. Richardson, "UEFI Secure Boot in Modern Computer Security Solutions," *UEFI Forum*, 2013.
[15] Z.-L. Zhou, N. Zhang, X. R. Zhang, and Z. Yang, "Research and Implementation of Trusted BIOS Based on UEFI," *Computer Engineering*, vol. 34, no. 8, p. 174, 2008.
[16] D. Pavlovic and C. Meadows, "Actor-network procedures: Modeling multi-factor authentication, device pairing, social interactions," *CoRR*, vol. abs/1106.0706, 2011.
[17] D. Cooper, W. Polk, A. Regenscheid, and M. Souppaya, "BIOS Protection Guidelines," NIST, Tech. Rep. 800-147, 2011.
[18] TCG, *TCG PC Client Specific Implementation Specification for Conventional BIOS*, 1st ed., February 2012.
[19] ——, *TCG PC Client Platform Physical Presence Interface Specification*, 1st ed., July 2015.
[20] W. Arthur, D. Challener, and K. Goldman, *A Practical Guide to TPM 2.0*. Apress, 2015.
[21] J. Wiens, "A tipping point for the trusted platform module?" 2008, http://www.darkreading.com/risk-management/a-tipping-point-for-the-trusted-platform-module/d/d-id/1069284?
[22] Atmel, "System Design Manufacturing Recommendations for Atmel TPM Devices," 2013.
[23] Infineon, "Infineon SLB 9645 TPM with Embedded Platform," October 2014.
[24] Intel, "Intel trusted platform module (tpm module-axxtpme3) hardware user's guide," Intel Corporation, Tech. Rep. G21682-003, 2011.
[25] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog, "BIOS Chronomancy: Fixing the Core Root of Trust for Measurement," in *ACM SIGSAC Conf. on Comp. & Comm. Security*, 2013, pp. 25–36.
[26] Microsoft, "Device guard overview," 2016. [Online]. Available: https://technet.microsoft.com/en-us/library/dn986865(v=vs.85).aspx
[27] I. Glücksmann, "Injecting custom payload into signed Windows executables," in *REcon 2012*, June 2012.
[28] Microsoft, *Windows Authenticode Portable Executable Signature Format*, 1st ed., 2008.
[29] ——, *Microsoft Portable Executable and Common Object File Format Specification*, 8th ed., 2013.