Hardware Security in Practice: Challenges and Opportunities

Nachiketh Potlapally
Security Center of Excellence (SeCoE)
Intel Corporation, Hillsboro, OR 97124
nachiketh.potlapally@intel.com

***Abstract*—Computing platforms used in practice are complex and require interaction between multiple hardware components (such as processor, chipset, memory and peripherals) for their normal operation. Maintaining security of these computing platforms translates to verifying there are no known security exploits present in the run-time interaction between these hardware units which can be exploited by attackers. However, given the large number of state elements in the hardware units and many control signals influencing their mutual interaction, validating security of a commercial computing platform thoroughly can be complicated and intractable. We believe this real-world perspective of hardware security is crucial to building secure systems in practice, but it has not been sufficiently addressed in security research community, and our paper is a step in covering this gap. In this paper, we exemplify the challenges in correctly implementing security in commercial hardware platforms through representative examples of various classes of hardware-oriented security attacks. We present an overview of methods adopted to deal with the complexity of validating security of hardware in an industrial setting, and enumerate opportunities present for the security research community to contribute to hardware security validation.**

Keywords-Hardware security, Vulnerabilities, Formal methods, Security validation, Security Development Lifecycle.

I. INTRODUCTION

Computing platforms are being increasingly deployed in many critical infrastructures such as smart grid, financial systems, sensitive governmental organizations *etc.,* where consequences of a successful security attack could be potentially serious. Thus, these applications of computing platforms in high-risk areas motivate the need to build platforms with enhanced security. However, the complexity of building platforms with enhanced security has increased due to two emergent trends, in the nature of newer security attacks, described below,

- *Increase in skill and resources*: Previously, many of the notable security attacks were perpetrated by skilled hackers motivated by fame. Nowadays, we see attacks being launched increasingly for economic reasons by well-funded criminal organizations or as information warfare orchestrated by sophisticated organizations with access to significant resources and talent. An example of such an attack is the Stuxnet worm attack intended to target sensitive industrial machinery of a specific nation state [1].
- *Increase in hardware-oriented attacks*: Computing platforms comprise a software stack consisting of user applications, operating system (OS) and device drivers, executing on a hardware consisting of processor, chipset, memory, and peripherals. Previously, security attacks largely targeted the software stack and were based on exploiting software vulnerabilities such as buffer overflows, format string *etc*, in order to get privileges for executing the attacker's malicious code [2, 9]. In recent times, security attacks have been steadily gravitating toward hardware with the aim of maliciously manipulating hardware settings to facilitate bypassing of protection mechanisms [3, 4, 5, 6]. In a platform, hardware is the most privileged entity, and ability to manipulate it has the potential to give the attacker considerable flexibility and power to launch malicious security attacks. In addition, many of the hardware-oriented attacks have the ability to escape detection by OS-based mechanisms such as anti-virus scanners.

Thus, in order to be robust against emerging security threats, we believe it is important for manufacturers to build computing platforms which will be resistant to increasingly sophisticated hardware-oriented attacks.

The hardware in general-purpose computing platforms such as desktops and laptops consists of many components where each component contains a large number of state elements (*e.g.*, configuration registers) that control its operation. In addition, the components interact with each other using numerous control signals. For secure operation of the hardware, we must ensure that an adversary cannot control or observe any state elements or control signals such that a security objective (*e.g.*, confidentiality, integrity, availability *etc*) is compromised or a protection mechanism is bypassed. Consider the example of kernel-level cryptographic software libraries (*e.g.*, Microsoft Cryptography API: Next Generation, Linux Cryptographic API) where all the cryptographic operations are performed in the kernel space. In these implementations, the cryptographic keys are stored by the kernel (running at Ring 0 privilege) in files within the kernel space where they are protected from user-level applications (running at Ring 3 privilege) by a combination of paging mechanism and processor privilege mode separation. However, memory access requests from a direct memory access (DMA) peripheral device are not subjected to access control enforced by the paging mechanism. Thus, an attacker can maliciously

978-1-4577-1058-2/11/$26.00 ©2011 IEEE

program configuration registers in the platform such that a DMA device can access any memory location, and use that privilege to dump the entire memory contents including kernel code and data [7]. Next, the attacker can use offline analysis (such as reverse engineering of binaries and searching for high-entropy bit strings) of the memory dump of the kernel address space to extract the cryptographic keys, and subvert confidentiality of sensitive data protected with those keys. This is an example of a hardware-oriented attack based on directly manipulating hardware to enable the attack. However, given the exponential number of combinations of state elements and control signals controlling the behavior of hardware in a computing platform, verifying that an adversary cannot compromise the security of a platform becomes very challenging. In this paper, we highlight this challenge through illustrative examples based on real attacks, and identify different classes of hardware attacks based on root cause analysis of vulnerabilities behind these attacks. We believe that by understanding different ways in which hardware security is compromised in real world computing platforms, we can incorporate changes in future hardware to prevent similar attacks, and thereby make them more robust. We consider this to be the main contribution of the paper.

The rest of the paper is organized as follows. Section II gives a description of hardware in a typical computing platform, and goes onto illustrate various classes of hardware security bugs. Section III describes a security development lifecycle (SDL) process adopted by us to validate security of complex hardware designs. In this section, we also present a formal methods based approach for security validation which has demonstrated promising results. Finally, we conclude the paper with a section describing areas of further research to help effectively deal with the complexity of hardware security validation.

II. HARDWARE SECURITY ATTACK TYPES

In this section, we define various categories of hardware attacks, and illustrate them through representative examples based on real security attacks. We believe that an important component of building platforms with enhanced security is to study the different ways in which they are attacked in practice, and incorporate lessons into future designs. Before we go onto describing the various classes of hardware attacks, we present some background on organization of hardware in a typical computing platform.

A. Hardware Organization Basics

The hardware in a computing platform consists of many components interacting with each other to implement platform functionality. Figure 1 illustrates organization of hardware in a typical computing platform. The platform hardware comprises three main components, namely integrated circuit (IC) chips, peripheral interfaces and buses. The functionality and examples of each of these hardware types are described below,

- *Integrated circuit (IC) chips*: These chips implement control logic, perform data processing and store firmware code. Besides the central processing unit (CPU), we have other chips such as *memory controller hub* (MCH) which regulates accesses from CPU and peripherals to memory, *input/output controller hub* (ICH) which regulates accesses between CPU and the different peripherals, *dynamic random access memory* (DRAM) which implements memory, *programmable interrupt controller* (PIC) which maps platform interrupts to entries in interrupt descriptor table, *trusted platform module* (TPM) which implements cryptographic functionality, *flash* which stores BIOS firmware, and *embedded controller* (EC) which regulates platform thermal and power events. Usually, the MCH and ICH are collectively referred to as the *chipset*. Thus, in addition to the CPU, there are multiple chips which control the flow of information on the platform that has implications for overall platform security.
- *Peripheral interfaces*: These electrical interfaces are responsible for implementing communication protocols underlying various standards for connecting peripherals to the platform. Examples of such interfaces include USB, SATA (*e.g.*, hard drives), PCI/e, LPC (*e.g.*, TPM device), and GbE (*e.g.*, wired and wireless network cards).
- *Buses*: These buses are responsible for transferring data between the chips and the peripherals. Notable examples include front side bus (FSB) between CPU and MCH, and direct media interface (DMI) bus connecting MCH to ICH.

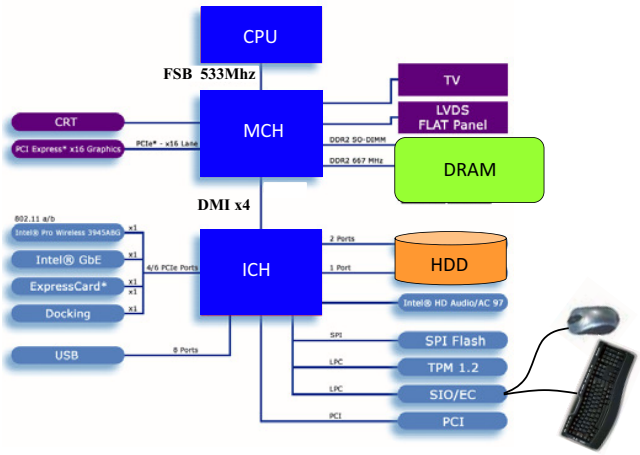

Figure 1. High-level view of hardware in a computing platform

The software running on the CPU interacts with other chips and peripheral interfaces using memory-mapped I/O (MMIO) or I/O instructions which are mapped to transactions on the corresponding buses. The primary reason for explaining the platform hardware organization is to enumerate the multiple components which affect overall platform security. Interestingly, the overall platform security is not a sum of the security of individual components, i.e., secure components do not guarantee a secure platform, and security vulnerabilities can result from interaction between the features (we will illustrate this, using an example of a real attack, in the next section). This latter aspect greatly increases the complexity of

realizing effective hardware security. In the next two sections, we define classes of hardware security bugs based on their root causes, and illustrate them using examples of real attacks. In this paper, we do not consider multi-processor systems (such as servers) which have additional security issues arising from race conditions, de-centralized access control *etc*.

B. Types of Hardware Attacks

At the hardware level, each of the IC chips has a large number of state elements, and the chips interact with each other and the peripheral interfaces through numerous control signals sent via the interconnection buses. In this context, we define *hardware security vulnerability* as a particular combination of state values and control signals which results in a violation of a security objective (*e.g.*, data confidentiality, data integrity, resource availability *etc*) or bypass of a platform access control mechanism [1]. A *hardware-oriented attack* results from an attacker being able to exploit hardware vulnerabilities to compromise the security of a computing platform. Based on the ways in which hardware attacks have been observed to occur in practice, we classify them into following types,

1. *Active adversarial manipulation of hardware control signals*: In this class of attacks, an adversary actively manipulates the control signals on the platform in order to subvert platform access control mechanisms. These attacks may require physical access to the platform for attaching a mod chip or a probe to influence the hardware control signals.
2. *Security gap in interaction of multiple platform features*: The overall platform functionality is realized through a collection of features implemented in different components of the platform. However, a particular interaction of two or more features can create security vulnerabilities which can be exploited by an attacker to subvert protections built into hardware. Given the exponential amount of state and control signals on the platform, it is quite difficult to identify all such interactions which could result in a potentially exploitable vulnerability.
3. *Insecure platform initialization by boot-up firmware*: Hardware needs to be configured correctly for normal platform operation and this initialization is done by platform boot-up firmware such as basic input/output system (BIOS). For example, the BIOS is responsible for configuring the memory controller which converts a physical address, output by a page table, into a combination of parameters (*i.e.*, channel, DIMM, rank, bank, row and column) required to index a location in the DRAM chip. If the memory controller is not configured correctly, then paging-based access control could be bypassed. Thus, we see that BIOS plays an integral role in hardware security of a platform (*i.e.*, an improper platform initialization by the BIOS can open multiple ways to launch hardware attacks), and this vital contribution is not normally highlighted.
4. *Ability of untrusted or lesser privileged entities to maliciously influence hardware operation*: Along with hardware, which is the most privileged platform entity, other entities (such as software, firmware, peripherals *etc.*) with different privileges or trust levels are present on the platform. Overall platform security is compromised when an adversary is able to exploit a lesser privileged or an untrusted entity on the platform in order to maliciously influence operation of hardware implementing protection mechanisms.

However, as a caveat, we would like to state that the above categories do not encapsulate all the possible ways of enabling hardware attacks, since there are many degrees of freedom in formulating these attacks. As newer attacks are uncovered, additional categories can be added to the above list. In the next section, we give examples of real world hardware attacks.

C. Examples of Hardware Attacks

In this section we present four representative examples of real hardware attacks to illustrate the attack classes defined in the previous section. All these attacks were against commercial products deployed in the field, and have been described in publicly available product security advisories or in presentations given at hacking-related security conferences (*e.g.*, BlackHat, CanSecWest, PacSec, Chaos Communications Congress *etc*). Also, mitigations have been released for all these attacks.

As an example of active adversarial manipulation (Type 1 in previous section), we present an attack on the hypervisor in PS3 [11]. Hypervisor is the most privileged software entity in PS3, and is protected from kernel-level and user-level processes. In this attack, an attacker gains write access to some entries in the page table maintained by the hypervisor, and circumvents hypervisor access control protection. In order to achieve this, the attacker uses normal software functions to allocate a buffer in memory, and requests multiple pointers to the buffer, *i.e.*, the hypervisor initializes multiple pointers to the same buffer in memory. Next, the attacker issues a software command to de-allocate the buffer which causes the hypervisor to free the buffer memory, and issue commands to remove page table entries for the multiple pointers previously allocated to this buffer. At this point, the attacker glitches the memory bus (using some hardware techniques) such that some of the pointer de-allocation commands fail to reach DRAM. This results in the attacker controlled software process still retaining valid pointers to a memory location which the hypervisor marks as free memory. The attacker waits until the hypervisor uses that memory to store its page table entries (since the memory has been marked free, the hypervisor can use it), and then uses the pointers under his control to alter the hypervisor page table entries to gain control over hypervisor execution.

To illustrate security gaps resulting from interaction of multiple platform features (Type 2 in previous section), we consider a security vulnerability arising from caching behavior interacting with system management mode (SMM) hardware

[1] The severity of a vulnerability is based on an evaluation of the likelihood and impact or consequences of the potential exploit. By itself, however, the term "vulnerability" does not signify the likelihood of the potential exploit occurring or the impact or consequences if the exploit were to occur.

protections in an unintended manner [6]. SMM is the most privileged mode in x86 processors, and is used for system-level functions such as power management, debugging *etc* [15]. When a system management interrupt (SMI) is issued, the processor goes into SMM, executes a SMI software handler, and exits SMM using RSM instruction to resume execution from the last instruction before SMI was issued. The SMI software handler is stored in a memory region called TSEG which is protected from user-level processes (Ring 3) and the kernel (Ring 0) by chipset hardware protection mechanisms. However, attackers found a way to get around this hardware protection by exploiting processor caching behavior. In an x86 processor, the L1 caching behavior of all physical addresses can be configured to either *write through* (WT) or *write back* (WB) using the memory type range registers (MTRRs) [15]. The sequence of steps in the attack is as follows,

- The attacker maps the SMI handler memory address range*, i.e.*, TSEG region, to WB
- Next, the attacker triggers a SMI interrupt to enter SMM. The SMI handler gets copied into cache as part of normal processor operation, and is executed out of the cache. Then, the processor exits SMM. This is shown in left side of Figure 2.
- The attacker corrupts the cached copy of the SMI handler with his malicious code. Since, the SMI handler memory range is mapped to WB, the write accesses occur only to the cache and do not go to TSEG. If the accesses went to TSEG, since the processor was not in SMM, the chipset hardware would have dropped all the write accesses. At this point, the attacker managed to successfully corrupt the SMI software handler copy in L1 cache with malicious code. This is shown in right side of Figure 2.
- Now, the attacker triggers a SMI, and the corrupted SMI software handler in the L1 cache gets executed. The attacker is able to get his malicious code executed with SMM privileges.

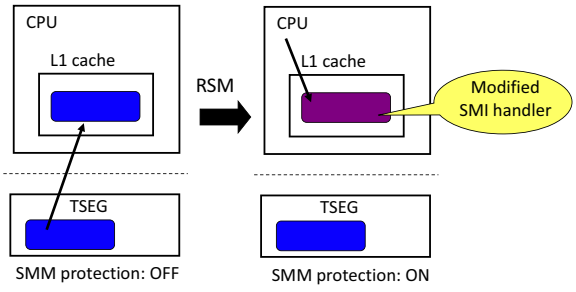

Figure 2. WB caching and SMI handler execution

In this attack, the SMM hardware protections worked in the way they were architected. However, this particular sequence of events in interaction of SMM execution with caching behavior was not anticipated, and it created an exploitable security vulnerability. This allowed an attacker to circumvent SMM-related hardware protections.

In order to illustrate attacks based on insecure hardware initialization by the BIOS (Type 3 in previous section), we use the example of the remapping attack [4]. Before going into the details of the attack, we present some introduction to remapping. In practice, many memory ranges within the 4GB physical address space, addressed by a processor, are pre-allocated for other purposes such as BIOS range (640KB-1MB), TSEG range (for storing SMI handlers and determined by SMBASE value), graphics stolen memory and MMIO range (which contains PCI/e configuration space). This is illustrated in Figure 3. The base address of the MMIO region is referred to as TOLUD (refer to Figure 3), and it is around 3GB, i.e., MMIO region consumes about 1GB of address space. In other words, when the processor generates accesses in (TOLUD-4GB) address range, the accesses are directed by MCH (refer to Figure 1) to configuration spaces of PCI/e devices attached to the platform, and not to DRAM. As a result, even if 4GB of DRAM is plugged into the platform, almost 1GB of it cannot be accessed by the processor, and goes unused. In order to get around this limitation, the concept of remapping was introduced in hardware. Using remapping, the 1GB of DRAM space corresponding to MMIO range (TOLUD - 4GB), which was previously inaccessible, can be reclaimed by redirecting access in the (4GB – 5GB) physical address space to this DRAM region (refer to Figure 3). The BIOS is responsible for programming the remapping registers in the chipset hardware (MCH, to be specific) such that accesses in 4GB-5GB address range are directed to (TOLUD - 4GB) region of DRAM.

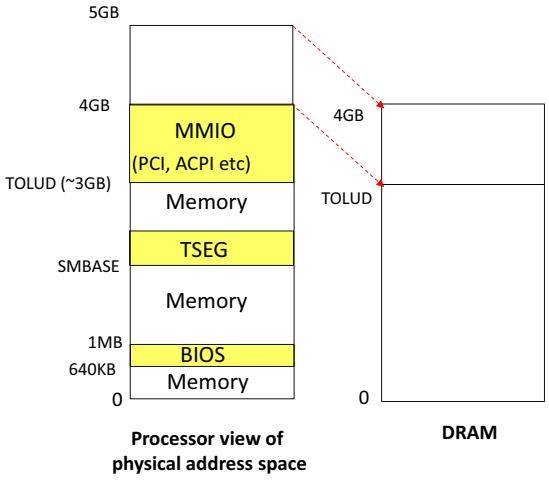

Figure 3. Illustration of remapping mechanism

In this attack, the BIOS did not lock the remapping registers after configuring them. This allowed the attackers to reprogram these registers such that the (4GB - 5GB) region of the physical address space mapped to DRAM space holding the TSEG region (which was completely different from the recommended usage of remapping mechanism). This enabled the attackers to bypass SMM-related hardware protections and corrupt SMI handlers with their malicious code. This attack was facilitated by incorrect platform configuration by the BIOS, and emphasizes the role of BIOS in enhancing platform security. This example also highlights the extent to which attackers use the platform features in a novel, creative and non-recommended manner in order to compromise the platform.

Finally, we illustrate attacks based on maliciously influencing secure hardware operation from untrusted sources (Type 4 in previous section). The Cell processor used in PS3 has dedicated security co-processors with private memories for performing security-sensitive computations in a protected and isolated manner from the host processor [11]. However, a malicious USB peripheral was used by hackers to trick the hardware to enter a debug mode, and then by sending a sequence of specially crafted and ingenuous USB messages, they were able to load their own kernel onto the platform. Next, using the compromised kernel, they could issue commands to the security co-processor to decrypt binaries encrypted by the manufacturer (and break confidentiality). In addition, by instructing the security co-processor to copy a specially constructed payload into its private memory, the hackers were able get malicious code to execute on the co-processor and subsequently, instruct it to dump cryptographic keys embedded in hardware. Thus, operation of the security co-processor was completely subverted by using malicious inputs from an untrusted peripheral and data supplied by a compromised kernel.

III. Security Validation in Industry

The exponential state space of computing platforms, along with combinations of feature interactions, makes it very difficult to detect all possible platform attacks during security validation. However, at the same time, there are attackers who are focused in using novel and creative means to find exploitable hardware vulnerabilities in these platforms. The combination of these two factors makes security validation very challenging in an industrial setting. In this section, we give an overview of the methodology called security development lifecycle (SDL) adopted by us to perform a structured security validation of complex designs like Intel products. In addition, we present details of formal methods-based approach which has shown promising results, and we are actively developing further. Normally, it is very difficult to prevent physical attacks on hardware, and during security validation, we focus primarily on vulnerabilities resulting in attacks of types 2, 3 and 4 (defined in Section II.B).

A. Security Development Lifecycle (SDL)

Security vulnerabilities could arise in a product due to a variety of reasons such as gaps in architecture definition, inconsistencies in translating architectural specifications to a logic design (comprising firmware, microcode and RTL), implementation decisions, implementing the design in silicon *etc*. Thus, in order to design a product with increased security, it is important to do the following,

- Take a structured approach where security validation is performed at each stage of the product lifecycle such that security vulnerabilities related to that stage are identified and fixed in the design before it is mapped to the successive stage.
- Formulate a collection of methodologies fine-tuned to detect security vulnerabilities characteristic for each design stage. For example, techniques used to identify security vulnerabilities in the architecture stage (where the design is described only at a specification level) are different from those used to identify vulnerabilities in early implementation stage where RTL is available.

The SDL process is motivated by the above observations. This process is one possible way to deal with complexity of security validation in a structured manner such that we can verify to the best of our ability that the product satisfies stated security objectives. The SDL process consists of four phases which can be described as follows:

- *Architecture review (S1 stage)*: In this stage, the architecture specification is reviewed for gaps in architecture definition which could result in potentially exploitable security vulnerabilities in the product. At this stage, the design description is available in the form of architectural specification documents.
- *Design review (S2 stage)*: In this stage, the security review of the architecture is continued, along with evaluation of security implications of decisions related to low-level implementation choices (such as micro-architecture) for realizing the architecture.
- *Early implementation review (S3 stage)*: In this stage, the firmware, microcode and RTL of the product are available, and they are evaluated for security vulnerabilities. The activities in this stage correspond to pre-Si security validation.
- *Ship product review (S4 stage)*: In this stage, the silicon implementation of the product is available, and post-Si techniques are employed to detect security vulnerabilities in the design.

As the definitions of SDL stages imply, S1 and S2 are targeted at finding gaps in architectural specification which could result in product security vulnerabilities, while S3 and S4 are aimed at identifying security vulnerabilities resulting from errors in mapping the architecture to an implementation. As much as possible, the vulnerabilities detected in an SDL stage are fixed before the design is mapped to the next stage. However, given the product complexity and state-of-the-art in security validation techniques, it would be very difficult to detect all the platform security vulnerabilities using SDL. The SDL process is an attempt to manage the complexity of security validation of complex Intel products, and the expectation is that we can detect as many security vulnerabilities as possible in Intel products before they are shipped. We should emphasize that SDL is an evolving process where tools and methodologies are being regularly evaluated to keep the process up to date with the changing hardware security threat landscape, and as further improvements in security validation techniques are made.

B. Formal Methods for Security Validation

In order to deal with the exponential state, we have been looking into formal methods for security validation [8, 13, 14]. An attack usually involves an adversary who actively manipulates the design state in order to expose a security vulnerability which can be exploited to launch an attack. So, in our security validation approach, we give the adversary model,

description of the design-under-test and the security property to be verified to the formal tool. . In short, we are trying to verify whether the following logical formula holds true: ((*DESIGN* Λ *ADVERSARY*) => *SECURITY PROPERTY*) where *DESIGN* and *ADVERSARY* refer to design description and adversary formal model. *SECURITY PROPERTY* refers to a temporal formula describing a security property. We use a model checker to perform the formal analysis and it either gives an output showing that the given security property is a theorem for the design even under adversarial conditions (in other words, adversary has no way to compromise the design with respect to that security property), or outputs a counterexample which is a trace of design inputs along with actions of the adversary, which results in the violation of the security property. We are investigating this approach at both S1 and S3 stages of SDL. At S1 stage, we follow the high-level formal modeling approach, whereas at S3 stage, the model checker operates directly on the RTL of the design under test.

The formal approach showed promise by detecting one of the complex hardware security vulnerabilities described in Section II. We built a high-level formal model describing processor, cache, MTRRs, memory and TSEG hardware protection mechanism. We also formulated an adversary model. Both the design and adversary formal model were specified in around 225 lines of TLA+ code [10]. The TLC model checker output a counterexample detailing corruption of SMI handlers in TSEG region by exploiting WT and WB attributes of caches [8]. Based on results of more planned pilot projects and further feasibility studies, this methodology could be deployed in future Intel products.

IV. Opportunities For Future Work

In the previous sections, we have identified challenges of implementing hardware security in commercial platforms, and the complexity of validating security of these platforms. The two evolving trends of ever-increasing feature complexity in commercial platforms and more powerful adversarial capabilities will add newer challenges to the existing ones in security validation. Below, we enumerate some opportunities for future work by the security research community to help deal with hardware security validation challenges,

- *Static security analysis tools for hardware*: Static security analysis has been used very successfully to identify vulnerabilities in software [12]. The main idea behind this approach is to build a sufficiently descriptive representation of the source code which can be analyzed to identify anomalous conditions indicative of security vulnerabilities. The analysis can be done using techniques such as dataflow analysis, Boolean satisfiability, symbolic analysis *etc*. It would be helpful to develop similar static tools for analyzing hardware described in RTL where users provide templates describing insecure behavior and the static tools identify portions of hardware matching it. For example, by providing the ability to tag hardware with trust levels, unauthorized flow of sensitive data to untrusted sources can be identified in complex hardware modules.
- *Formal techniques for verifying hardware security*: Unlike static techniques which are limited in their ability to find deeply embedded security vulnerabilities, formal techniques have the ability to comprehensively explore the design space and identify intricate security vulnerabilities. However, use of formal techniques is not straightforward and they require lot of effort. It would be useful to have formal tools and techniques which facilitate easier verification of security properties in industrial designs. This could include tools to help translate higher-level security objectives to a collection of atomic properties to be verified in the design-under-test, improved formal semantics to capture common security notions such as privilege levels, and libraries which help unambiguous specification of security properties in temporal logic.

In this paper, based on our experience in an industrial setting, we have provided a perspective of hardware security and challenges involved, and areas of future research. We hope this would foster greater synergy between industry and security researchers to tackle hardware security challenges.

Acknowledgment

I would like to thank the members of SeCoE for their valuable feedback and suggestions.

References

[1] N.Falliere, L. Murchu, and E. Chien, "W32.Stuxnet Dossier," Symantec Security Response, Nov 2010.

[2] M. Dowd, J. McDonald and J. Schuh, The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities, Addison-Welsey, 2006.

[3] L. Duflot, "Security issues related to Pentium System Management Mode," CanSecWest Conf., March 2006.

[4] R. Wojtczuk, J. Rutkowska and A. Tereshkin, "Xen 0wning trilogy," Black Hat Conf., Oct 2008.

[5] A. Tereshkin and R. Wojtczuk, "Introducing ring -3 rootkits," Black Hat Conf., July 2009.

[6] L. Duflot, O. Levillain, B. Morin and O. Grumelard, "Getting into SMRAM: SMM Reloaded," CanSecWest Conf., March 2009.

[7] M. Dornseif, "0wned by an iPod," PacsSec Conf., Nov 2004.

[8] N. Potlapally, "High-level formal modeling-based approach for architectural security validation," Proc. Intel Design & Test Tech. Conf. (DTTC), Aug 2010.

[9] R. Kannavara, S. Mandujano and N. Potlapally, "On the security of vPro: ME stack analysis and best practices," Proc. Intel Design & Test Tech. Conf. (DTTC), Aug 2009.

[10] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Pearson Education Inc., 2002.

[11] Fail0verflow, "PS3 epic fail," 27th Chaos Comm. Congress (CCC), Dec 2010.

[12] B. Chess and G. McGraw, "Static analysis for security," IEEE Security & Privacy, vol. 2, no. 6, pp 76-79, Nov 2004.

[13] David Lie, John Mitchell, Chandramohan Thekkath, and Mark Horowitz, "Specifying and Verifying Hardware for Tamper-resistant Software", Proc. IEEE Symp. Security & Privacy, May 2003.

[14] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta, "Attacking, Repairing and Verifying Secvisor: A Retrospective on the Security of a Hypervisor", CMU Tech. Report, 2009.

[15] Intel Corp, Intel 64 and IA-32 Architectures Software Developers Manual: System Programming Guide, 2010