Security Policy Enforcement in Modern SoC Designs

Sandip Ray
Strategic CAD Labs, Intel Corporation
Hillsboro, OR 97124. USA
sandip.ray@intel.com

Yier Jin
EECS Dept., University of Central Florida
Orlando, FL 32816. USA
yier.jin@eecs.ucf.edu

Invited Paper

***Abstract*— Modern SoC designs contain a large number of sensitive assets that must be protected from unauthorized access. Authentication mechanisms which control the access to such assets are governed by complex *security policies*. The security policies affect multiple design blocks and may involve subtle interactions among hardware, firmware, OS kernel, and applications. The implementation of security policies in an SoC design, often referred to as its *security architecture*, is a subtle composition of coordinating design modules distributed across the different IPs. Toward this direction, this paper gives an overview of SoC security architectures in modern SoC designs and provides a glimpse of their implementation, as well as their design complexities and functional shortcomings. Design of security architectures involves a complex interplay of requirements from functionality, power, security, and validation. We also outline some of the research needs in the area for developing robust, trustworthy SoC designs.**

I. Introduction

Modern embedded and mobile computing devices, *e.g.*, smartphones, tablets, wearables, implants, smart sensors, etc. are increasingly getting used in a large number of personalized activities, including shopping, banking, providing driving directions, and tracking health and wellness conditions. Consequently, these devices have access to significant sensitive, personal data including our bank and credit card information, email contacts, browsing history, location, even intimate physiological information such as heart-rates and sleep patterns. In addition to personalized end-user information, these devices contain highly confidential collateral from architecture, design, and manufacturing, such as cryptographic and digital rights management (DRM) keys, programmable fuses, on-chip debug instrumentation, defeature bits, etc. Malicious or unauthorized access to secure assets in a computing device can result in identity thefts, leakage of company trade secrets, even loss of human life. Consequently, a crucial component of a modern computing system architecture includes authentication mechanisms to protect these assets.

Modern computing systems are typically developed as system-on-chip (SoC) designs, *i.e.*, a single integrated circuit encompassing the system functionality. An SoC design involves composition of a large number of design modules (often referred to as *intellectual properties* or IPs) that coordinate with through a number of on-chip communication fabrics to implement the system functionality. Secure assets in such a design are sprinkled across the different IPs, and their access control requirements are defined by a collection of highly complex *security policies*. The policies specify the conditions under which a security asset can be accessed at any point in the system execution. An SoC design consequently requires a *security architecture*, *i.e.*, a mechanism of authentication to ensure that the system enforces and manages these policies.

Unfortunately, in spite of its obvious importance, there has been little work on standardization or systematic definition of the security architecture of SoC designs. Consequently, authentication mechanisms used in current industrial practice are ad hoc, point approaches for specific policies and system implementations, and typically depend on low-level, often unspecified, architectural and design invariants. Exacerbating the issue is the fact that the policies themselves, as well as the design invariants used in their implementation, are rarely formalized or even documented, making it non-trivial and sometimes impossible to validate if the final design indeed enforces proper authentication for all design assets: such information is buried within a plethora of architectural and design documents, specified in ambiguous natural language descriptions, and often left implicit. Unsurprisingly, security vulnerabilities are abound in modern SoC designs, as evidenced by the frequency and ease in which activities like identity theft, DRM override, device jailbreaking, etc. are performed.

In this paper, we describe some of the key design considerations involved in the systematic definition of an SoC security architecture. The goal is to facilitate a global understanding of the problem, and the constraints that must be addressed in order for the architecture to be usable. We believe that such a solution would require significant collaboration among a large number of research communities, including processor architecture, SoC designs, security, and validation, and our hope is that this paper would facilitate such collaborations by providing a unified exposition of the issues involved.

The remainder of the paper is organized as follows. Sections II and III identify two key ingredients of security architecture of modern SoC designs, *viz.*, the kind of policies being implemented and the adversarial threat models involved. In Section IV we discuss the considerations involved in the design of a modern SoC security architecture from these

978-1-4673-8388-2/15/$31.00 ©2015 IEEE

ingredients, and point out the complexities involved. Section V briefly discusses some of the efforts undertaken recently, both in industrial practice and academic research, to mitigate these complexities through standardization of skeletal architectural designs. We also point out some of the limitations and weaknesses of the current state of the practice. We conclude in Section VI.

II. Security Policies

Security policies identify the authentication, access, and protection requirements for the different assets in the design. At a high level, the policies are typically instances of confidentiality, integrity, and availability requirements [1]. The role of a policy is to define an instantiation of these requirements for specific assets, and provide an "actionable" specification for the SoC system architect and designer on the protection mitigation strategies that need to be implemented. For example, the following sample policies define some of the policies for cryptographic keys, programmable fuses, and executable firmware. Note that these policies are merely illustrative and do not represent the security policy set of any specific company or design.

- **Boot confidentiality:** During boot, no IP can access any internal registers of the crypto engine.
- **Fuse integrity:** A programmable fuse can be updated for silicon validation but not after production.
- **Key authentication:** The crypto engine can respond to key access requests by other IPs with actual keys only if the IP has been authenticated, and the system is not executing in debug mode; in debug mode, the crypto engine will respond with dummy keys.
- **Firmware integrity:** Firmware executing on any IP must have been previously signed or authenticated by the firmware authentication engine.

The above examples, albeit hypothetical, illustrate some important characteristics of security policies. In particular, access to an asset $\mathcal{S}$ by an agent $\mathcal{A}$ can be restricted depending on the stage of the execution (*e.g.*, boot vs. normal) or the point of the design in the system life-cycle (*e.g.*, debug vs. deployment). Furthermore, policies may need to be updated on-field, either in response to bugs or errors detected after deployment or in response to changing environmental conditions. For example, assume the original security policy permitted the access to a cryptographic key $\mathcal{K}$ by a trusted IP $\mathcal{A}$. However, if subsequently a bug or vulnerability is discovered in $\mathcal{A}$ that could compromise the security of $\mathcal{K}$ then the policy may need to be revised.

Roughly, security policies can be categorized into the following four classes. A companion paper [2] provides some additional details of these policy classes.

Access control [3], [4]: This common class of policies defines which IP has access to an asset at any point in the system execution. In our examples, boot confidentiality, fuse integrity, and key authentication are access control policies.

Information flow [5]: Information flow policies go one step ahead of access control by constraining what can be *inferred* from accessed data. Typically, information flow policies are implemented by a collection of access control policies together with additional constraints as necessary. For example, the key authentication policy above may be one component of an information flow policy that requires that no unauthenticated IP can infer the original crypto keys in debug mode.

Liveness: Liveness refers to the requirement that the functionality of the system is not compromised through implementation of protection mechanisms. Note that a trivial system that simply denies access to all assets would typically satisfy access control requirements; such system would not satisfy liveness requirements. Typical liveness policies ensure protection against denial-of-service attacks, by requiring legitimate access requests to eventually succeed.

Time-of-check vs. time-of-use (TOCTOU) [6], [7]: TOCTOU policies ensure that the authentication mechanisms deployed to ensure access control cannot be bypassed or "spoofed", by requiring that the authenticated agent is really the agent accessing the asset it is authenticated for. The firmware integrity policy above is an example of a TOCTOU policy, *e.g.*, it requires that an authenticated firmware cannot be tampered with in the intermediate stages of system execution between its authentication and final deployment.

In addition to ensuring access restrictions on assets at individual IPs, there are policies protecting the integrity and confidentiality of assets during communication. Such policies, also referred to as *fabric policies*, form a significant portion of the security requirements of a modern SoC design. The following categories provide a flavor of the diversity of requirements that must be accounted for during on-chip communications.

Message immutability: If IP $\mathcal{A}$ sends a message m to IP $\mathcal{B}$ then the message received by $\mathcal{B}$ must be exactly message m.

Redirection and masquerade prevention: If $\mathcal{A}$ sends a message m to $\mathcal{B}$, then the message must be delivered to $\mathcal{B}$. In particular, it should be impossible for a (potentially rogue) IP $\mathcal{C}$ to masquerade as $\mathcal{B}$, or for the message to be redirected to a different IP $\mathcal{D}$ in addition to, or instead of $\mathcal{B}$.

Non-observability: A private message from $\mathcal{A}$ to $\mathcal{B}$ must not be accessible to another IP during transit.

The above policies may seem to be "obvious". However, it may still be highly subtle and nontrivial to enforce these policies. As an example of the subtleties that need to be accounted for, consider the SoC configuration in Fig. 1. Suppose that IP IP0 needs to send a message to the DRAM. Ordinarily, the message would be routed through Router3, Router0, Router1, and Router2. However, such a route permits message redirection via software. To understand how this can be done, note that each router includes a base address register (BAR) which is used to route messages for specific destinations. However, one of the routers in the proposed path, Router0 is connected to the CPU; the BARs in this

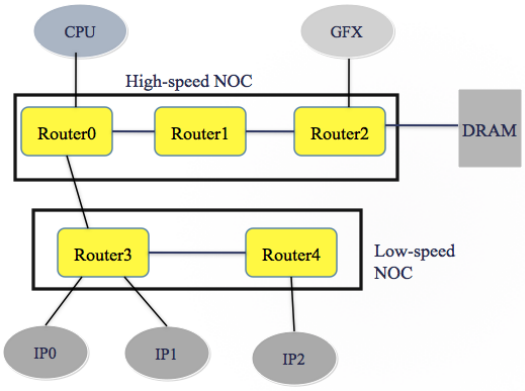

Fig. 1. An Illustrative Toy SoC Configuration. Typical SoC designs include several on-chip fabrics with differing speed and power consumption profiles. For this toy configuration, we assume a high-speed fabric with three routers connected linearly, and a low-speed fabric with two routers also connected linearly.

router are subject to potential overwrite by the host operating system, which can consequently redirect a message passing through Router0 to a different destination. Consequently, a secure message cannot be sent from IP0 through DRAM via this route unless the host operating system is trusted. Note that understanding the potential of redirection in this scenario requires knowledge of operation of the fabrics, functioning of routers within a NoC (*e.g.*, the use of BARs), as well as the capabilities of the software potentially in an adversarial role.

We conclude the discussion on policies by noting that the above fabric policies are generic requirements for *all* on-chip communication networks used for communicating sensitive information. In addition to these, most SoC designs typically include additional asset-specific communication constraints. For instance, a potential fabric policy relevant to secure boot is listed below. This policy ensures that a key generated by the fuse controller cannot be sniffed during propagation to the crypto engine for storage.

- **Boot-time key nonobservability:** During he boot process, a key from the fuse controller to the crypto engine cannot be transmitted through a router to which any IP with user-level output interface is connected.

III. Threat Models

In order to ensure that an asset is protected, the designer needs, in addition to the security policy governing the protection requirements, a comprehension of the power of the adversary against which to protect. Indeed, effectiveness of virtually all security mechanisms in SoC designs today are critically dependent on how realistic the model of the adversary is, against which the protection schemes are considered. Conversely, most security attacks rely on breaking some of the assumptions made regarding constraints on the adversary while defining protection mechanisms. When discussing adversary and threat models, it is worth noting that the notion of adversary can vary depending on the asset being considered: in the context of protecting DRM keys, the end user would be considered an adversary, while the content provider (and even the system manufacturer) may be included among adversaries in the context of protecting private information of the end user. Consequently, rather than focusing on a specific class of users as adversaries, it is more convenient to model adversaries corresponding to each policy and define protection and mitigation strategies with respect to that model.

Defining and classifying the potential adversary is a highly creative process. It needs considerations such as whether the adversary has physical access to the system, which components they can observe, control, modify, or reverse-engineer, etc. Recently, there have been some attempts at developing a disciplined, clean categorization of adversarial powers. One potential categorization, based on the interfaces through which the adversary can gain access to the system assets, can be used to classify them into the following six broad categories (in order of increasing sophistication). Note that there has been significant research into specific attacks in different categories, and a comprehensive treatment of different attacks is beyond the scope of this paper; the interested reader is encouraged to look up some of the references for a thorough description of specific details.

Unprivileged software adversary: This form of adversary models the most common type of attack on SoC designs. Here the adversary is assumed to not have access to any privileged information about the design or architecture beyond what is available for the end-user, but is assumed to be smart enough to identify or "reverse-engineer" possible hardware and software bugs from observed anomalies. The underlying hardware is also assumed to be trustworthy, and the user is assumed no physical access to the underlying IPs. The importance of this naïve adversarial model is that any attack possible by such an adversary can be potentially executed by any user, and can therefore be easily and quickly replicated on-field on a large number of system instances. For these types of attacks, the common "entry point" of the attack is assumed to be user-level application software which can be installed or run on the system without additional privileges. The attacks then rely on design errors (both in hardware and software) to bypass protection mechanisms and typically get a higher-privilege access to the system. Examples of these attacks include buffer overflow, code injection, BIOS infection, return-oriented programming attacks etc. [8], [9].

System software adversary: This provides the next level of sophistication to the adversarial model. Here we assume that in addition to the applications, potentially the operating system itself may be malicious. Note that the difference between the system software adversary and unprivileged software adversary can be blurred, in the presence of bugs in the operating system implementation leading to security vulnerabilities: such vulnerabilities can be seen as unprivileged software adversaries exploiting an operating system bug, or a malicious operating system itself. Nevertheless, the distinction facilitates defining the root of trust for protecting system assets. If the operating system is assumed untrusted, then protection and

mitigation mechanisms must rely on lower-level (typically hardware) primitives to ensure policy adherence. Note that system software adversary model can have a highly subtle and complex impact on how a policy can be implemented, *e.g.*, recall from the masquerade prevention example above that it can affect the definition of communication fabric architecture, communication protocol among IPs, etc.

Software covert channel adversary: In this model, in addition to system and application software, a side-channel or covert-channel adversary is assumed to have access to non-functional characteristics of the system, *e.g.*, power consumption, wall-clock time taken to service a specific user request, processor performance counters, etc., which can be used in subtle ways to identify how assets are stored, accessed, and communicated by IPs (and consequently subvert protection mechanisms) [10], [11].

Naïve hardware adversary: Naive hardware adversary refers to the attackers who may gain the access to the hardware devices. While the attackers may not have advanced reverse engineering tools, they may be equipped with basic testing tools. Common targets for these types of attacks include exposed debug interfaces and glitching of control or data lines [12]. Embedded systems are often equipped with multiple debugging ports for quick prototype validation and these ports often lack proper protection mechanisms, mainly because of the limited on-board resources. These ports are often left on purpose to facilitate the firmware patching or bug-fixing for errors and malfunctions detected on-field. Consequently, these ports also provide potential weakness which can be exploited for violating security policies. Indeed, some of the "celebrated" attacks in recent times make use of available hardware interfaces including the XBOX 360 Hack [13], Nest Thermostat Hack [14], and several smartphone jailbreaking techniques.

Hardware reverse-engineering adversary: In this model, the adversary is assumed to be able to reverse-engineer the silicon implementation for on-chip secrets identification. In practice, such reverse-engineering may depend on sniffing interfaces as discussed for naïve hardware adversaries. In addition, they can depend on advanced techniques such as laser-assisted device alteration [15] and advanced chip-probing techniques [16]. Hardware reverse engineering can be further divided into two categories: (1) chip level reverse engineering; and (2) IP core functionality reconstruction. Both attack vectors bring security threats into the hardware systems, and permit extraction of secret information (*e.g.*, cryptographic and DRM keys coded into hardware), which cannot be otherwise accessed through software or debugging interfaces.

Malicious hardware intrusion adversary: A hardware intrusion adversary (or hardware Trojan adversary) is a malicious piece of hardware inside the SoC design. It is different from a hardware reverse-engineering adversary in that instead of "passively" observing and reverse-engineering functionality of the rest of the design components, it has the ability to communicate with them (and "fool" them into violating requisite policies). Note that as with the difference between system software and unprivileged software adversaries above, many attacks possible by an intrusion adversary can, in principle, be implemented by a reverse-engineering adversary in the presence of hardware bugs. Nevertheless, the root of trust and protection mechanisms required are different. Furthermore, in practice, hardware Trojan attacks have become a matter of concern specifically in the context of SoC designs that include untrusted third-party IPs as well as those integrated in an untrusted design house. Protection policies against such adversaries is complex, since it is unclear a priori which IPs to trust under this model. The typical approach taken for security in the presence of intrusion adversaries (and in some cases, reverse-engineering adversaries) is to ensure that a rogue IP $\mathcal{A}$ cannot subvert a non-rogue IP $\mathcal{B}$ into deviating from a policy.

IV. Designing A Security Architecture

Given a plethora of complex policies and protection requirements under different classes of potential adversaries, how would we go about designing authentication mechanisms to ensure policy enforcement? Unfortunately, the state of the practice in this area by far is extremely manual and depends heavily on human creativity and observation. In particular, a security architect roughly iterates through the following five steps until convergence.

1) **Asset Definition.** Identify all the system assets governing protection. This requires identification of IPs and the point of system execution where the assets originate. Some assets (*e.g.*, fuse configurations, e-wallet keys, etc.) are "hard-coded" in specific IPs as provisioned by manufacturers or original equipment manufacturers (OEMs). Others are generated under specific scenarios at different points of system execution.
2) **Policy Specification.** For each asset, identify the policies that involve it. Note that a policy may "involve" an asset without specifying direct access control for it. For example, a policy may specify how a secure key $\mathcal{K}$ can be accessed by a specific IP. This in turn may imply how the controller of the fuse where $\mathcal{K}$ is programmed can communicate with other IPs during boot process for key distribution.
3) **Attack Surface Identification.** For each asset, identify potential adversarial actions that can subvert policies governing the asset. This requires identification, analysis, and documentation of each potential "entry point", *i.e.*, any interface that transfers data relevant to the asset to an untrusted region. The entry point depends on the category of the potential adversary considered in the attack, *e.g.*, a covert-channel adversary can make use of non-functional design characteristics such as power consumption or temperature to infer the ongoing computation.
4) **Risk Assessment.** The potential for an adversary to subvert a security objective does not, in and of itself, warrant mitigation strategies. The risk assessment and analysis are defined in terms of the so-called

DREAD paradigm, composed of the following five components: (a) Damage potential;(b) Reproducibility; (c) Exploitability, *i.e.*, the skill and resource required by the adversary to perform the attack; (d) Affected systems, *e.g.*, whether the attack can affect a single system or tens or millions; and (e) Discoverability. In addition to the attack itself one needs to analyze the likelihood that the attack can occur on-field, motives of the adversary, etc.

5) **Threat Mitigation:** Once the risk is considered substantial given the likelihood of the attack, protection mechanisms are defined and the analysis must be performed again on the modified system.

Implementation example: Consider protecting a system against code injection attacks by malicious or rogue IPs by overwriting code segments through direct memory access (DMA) access. The assets being considered here are appropriate regions of memory hierarchy (including cache, SRAM, secondary storage), and the governing policy may be to define DMA-protected regions where DMA access is disallowed. The security architect needs to go through all memory access points in the system execution, identify memory access requests to DMA-protected regions, and set up mechanisms so that DMA requests to all protected accesses will fail. Once this is done, the enhanced system must be evaluated for additional potential attacks, including attacks that can potentially exploit the newly set-up protection mechanisms themselves. Such checks are performed typically via *negative testing*, *i.e.*, looking beyond what is specified to identify if the underlying security requirements can be subverted. For our DMA protection example, such testing may involve looking for ways to access the DMA-protected memory regions, other than directly performing a DMA access. The process is iterative and highly creative, resulting in a collection of increasingly complex line-up of protection mechanisms, until the mitigation is considered sufficient with respect to the risk assessment.

It should be clear that performing the above activities manually over the range of system assets and policies, in the presence of subtleties related to implicit expectations, potential adversaries that break the risk/mitigation analysis, and complex interplay between functional behavior and security constraints, is a daunting task. Admittedly, there is a host of available tools to assist in the different steps, *e.g.*, tools for documenting steps in threat and severity identification identifying security scenarios, etc. [17], [18] Nevertheless, the key architectural decisions and analysis still depend highly on human insight.

V. Towards Architecture Standardization

The issue of non-standard, ad hoc security mechanisms is acknowledged in the industry today as a key road-block in ensuring trustworthiness of critical computing systems. The situation is exacerbated with increasingly stringent time-to-market requirements which provide little time for comprehensive manual review or analysis of system assets, threats, and mitigation strategies.

To circumvent the problem, there have been recent attempts at standardizing a set of mechanisms for implementing access control for different assets. Below, we discuss three such frameworks: Samsung KNOX, Intel® Software Guard Extension (SGX), and ARM Trustzone®. Note that while there are differences in design, the overall goal for all the above mechanisms are similar, *viz.*, provide a mechanism for ensuring a Trusted Execution Environment (TEE) with guaranteed isolation of sensitive data. The underlying architectural plans are also similar, *viz.*, a combination of hardware support (*e.g.*, secure operating modes, virtualization), and software mechanisms (*e.g.*, context switch agents, integrity check).

Samsung KNOX [19]: This architecture is specifically targeted towards smartphones and provides secure separation features to enable information partition between business and personal content coexisting on the same system. In particular, it permits hot swap between these two content worlds (*e.g.*, without requiring system restart). The key ingredient of this technology is a *separation kernel* that implements information isolation. This architecture permits several system-level services, including the following:

- Trusted boot, *i.e.*, preventing unauthorized OS and software from being loaded onto the device at startup;
- Trust-zone based integrity measurement architecture (TIMA), which continually monitors kernel integrity;
- Security enhancement (SE) for Android, an enforcement mechanism providing protection of system/user data based on confidentiality and integrity requirements through separation; and
- KNOX container, which offers a secure environment in which protected business applications can run with guaranteed information separation from the rest of the device.

ARM Trustzone [20]: TrustZone technology is a system-wide approach to security on high performance computing platforms. The TrustZone implementation relies on partitioning the SoC's hardware and software resources so that they exist in two worlds: secure and non-secure. Hardware supports access control and permissions for the handling of secure/non-secure applications and the interaction and communication among them. The software supports secure system calls and interrupts for secure run-time execution in a multitasking environment. These two aspects ensure that no secure world resources can be accessed by the normal world components, except through secure channels, enabling an effective wall-of-security to be built between the two domains. This protection extends to I/O connected to the system bus via the TrustZone enabled AMBA3 AXI bus fabric, which also manages memory compartmentalization.

Intel SGX: SGX [21] is an architecture for providing a trusted execution environment provided by the underlying hardware to protect sensitive application and user programs against

potentially malicious operating systems. Rather than providing two isolated worlds for trusted and untrusted executions as in Trustzone, SGX permits applications to initiate secure *enclaves* or *containers* which serve as so-called "islands of trust". It is implemented as a set of new CPU instructions that can be used by applications to set aside such secure enclaves of code and data. This enables (1) applications to preserve the confidentiality and integrity of sensitive data without disrupting the ability of legitimate system software to manage the platform resources; and (2) end users to retain control of their platforms, applications, and services even in the presence of malicious system software.

The above infrastructures provide a foundation (*i.e.*, a mechanism of isolation) for implementing security policies. However, they are a far cry from a standardized approach for implementing policies themselves. To provide such approaches, it is necessary to (1) develop a language for succinctly expressing security policies; (2) creating a parameterized "skeleton" design that can be easily instantiated to diverse policy implementations; and (3) developing techniques for synthesizing policy implementation from high-level descriptions. Recent academic and industrial research has attempted to mitigate some of these issues. Li *et al.* [22] provide a language and synthesis framework for certain security policies. Basak *et al.* [2] provide a microcontrol-based framework for implementing certain class of security policies. There have been optimized architectural support for specific classes of policies, *e.g.*, control-flow integrity [23], Trojan resistance [24]. Finally, there has been work on a generic, distributed access control protection mechanism by permitting the asset owner to assign *security attributes* [25]; this provides a flexible mechanism to permit or restrict further redistribution (or even declassification/reclassification of the security attributes) of the asset by any agent during any subsequent access. However, in spite of such work on pieces of the problem, we are still far away from a robust, configurable security architecture as necessary for robust system design. Some key deficiencies include interplay of secure access control with on-chip instrumentation, definition of security architectures that are configurable for different phases of system life-cycle, and lack of a centralized IP for policy implementation in the SoC design, which makes it difficult to evaluate policy compliance.

VI. Summary

We have provided an overview of the state of practice in SoC security architecture. We have summarized the state of the practice and research directions, and discussed how it falls short of our needs for trustworthy and secure system design.

This paper does not propose any solution to the problem: our goal has been to bring a holistic picture of the problem to the attention of the research community. We hope that our exposition will provide a starting point for researchers interested in developing a unified, standardized security architecture, and help comprehend the main design choices involved in a usable solution. We believe that a solution to this problem is crucial to the future of computing, and that collaborative research between architects, designers, and security researchers is critical to developing such a solution.

References

[1] S. J. Greenwald, "Discussion Topic: What is the Old Security Paradigm," in *Workshop on New Security Paradigms*, 1998, pp. 107–118.

[2] A. Basak, S. Bhunia, and S. Ray, "A Flexible Architecture for Systematic Implementation of SoC Security Policies," in *Proceedings of the 34th International Conference on Computer-Aided Design*, 2015.

[3] M. Miettinen, S. Heuser, W. Kronz, A. Sadeghi, and N. Ashokan, "ConXsense: automated context classification for context-aware access control," in *ASIACCS*, 2014, pp. 293–304.

[4] R. Hull, B. Kumar, P. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas, "Enabling Context-aware and Privacy-conscious User Data Sharing," in *2004 IEEE International Conference on Mobile Data Management*, 2004, pp. 187–198.

[5] J. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proc. 1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[6] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, "Fixing Races for Fun and Profit: How to Abuse Atime," in *Proceedings of the 14th USENIX Security Symposium*, 2005, pp. 303–314.

[7] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of SoC Firmware Load Protocol," in *IEEE HOST*, 2014.

[8] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, ser. STC'09, 2009.

[9] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[10] P. C. Kocher and B. J. J. Jaffe, "Differential Power Analysis," in *19th Annual International Cryptology Conference*, 1999, pp. 398–412.

[11] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *16th Annual International Cryptology Conference*, 1996, pp. 104–113.

[12] S. Ray, J. Yang, A. Basak, and S. Bhunia, "Correctness and Security at Odds: Post-silicon Validation of Modern SoC Designs," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015.

[13] Homebrew Development Wiki, "JTAG-Hack," http://dev360.wikia.com/wiki/JTAG-Hack.

[14] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart Nest Thermostat: A smart spy in your home," in *Black Hat USA*, 2014.

[15] R. Rowlette and T. Eiles, "Critical Timing Analysis in Microprocessors Using Near-IR Laser Assisted Device Alteration (LADA)," in *IEEE International Test Conference*, 2003, pp. 264–273.

[16] Http://www.chipworks.com/.

[17] "Microsoft Threat Modeling & Analysis Tool version 3.0," 2009.

[18] J. Srivatanakul, J. A. Clark, and F. Polac, "Effective Security Requirements Analysis: HAZOPs and Use Cases," in *7th International Conference on Information Security*, 2004, pp. 416–427.

[19] Samsung, "Samsung KNOX," www.samsungknox.com.

[20] ARM, "Building a secure system using trustzone technology," *ARM Limited*, 2009.

[21] Intel, "Intel® Software Guard Extensions Programming Reference," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[22] X. Li, V. K. anf J. Oberg, M. Tiwari, V. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A Language for Hardware-Level Security Policy Enforcement," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[23] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015.

[24] L. Changlong, Z. Yiqiang, S. Yafeng, and G. Xingbo, "A System-On-Chip bus architecture for hardware Trojan protection in security chips," in *EDSSC*, 2011.

[25] M. R. Sastry, I. T. Schoinas, , and D. M. Cermak, "Method for enforcing resource access control in computer system," in *US Patent 20120079590 A1*.