Towards Verifiably Secure Systems-on-Chip Platforms

Sujit Kumar Muduli and Pramod Subramanyan
Indian Institute of Technology, Kanpur.

***Abstract*—Verification and validation of system-level security primitives is a pressing challenge in systems-on-chip (SoC) design and verification. This is a difficult problem to tackle for three reasons. First, no general frameworks exist that can enable adversary modeling for SoC platforms. Second, succinct specification of the desired security properties is not possible with current property specification languages. Finally, verification of a security specification is more challenging than functional verification. In this paper, we introduce a formal framework that enables general adversary modeling for SoC platforms and a security property specification language for this framework. We present formal semantics for the framework and illustrate its utility through a case study of an authenticated firmware load protocol.**

I. Introduction

Today's systems-on-chip (SoC) designers do not have an end-to-end verification methodology to ensure security of their platforms. At heart of this problem is a lack of techniques for: (i) formal *modeling* of the system and adversary behavior, (ii) formal *specification* of platform security properties, and (iii) systematic (not necessarily formal) *verification* of these properties. As a result, it is difficult to reason about security vulnerabilities in SoC platforms. We argue that this difficulty has contribution to the explosion in hardware security vulnerabilities discovered over the past few years. For instance, reported hardware vulnerabilities in the National Vulnerability Database (NVD) have increased in count from 1350 in 2013 to 3961 in 2017; an increase of 290% in just four years [24]. Viewed as a fraction of total vulnerabilities, hardware vulnerabilities have increased from being an insignificant fraction of all vulnerabilities in 2003 to being 15% of reported vulnerabilities in the NVD in 2017. These statistics point towards the pressing need for systematic SoC security verification.

It is important to note that recent years have seen significant progress in the formalization of *functional* specifications of hardware platforms. This includes formalization of instruction set architecture (ISA) functionality [6, 18, 19], memory consistency models [11, 12, 14] and accelerator functionality [13, 23]. However, only a few efforts have tackled formalization of the *security guarantees* provided by hardware platforms. Formalization of security guarantees has two important aspects that go beyond formalization of functionality. The first of these is the construction of an adversary model that captures expected threats to the platform. This typically has two components: (i) what are the ways in which an adversary can affect (modify) system state, and (ii) what parts of system state are visible (observable) to an adversary. Both of these components require going beyond the features of a functional model. For example, both fault injection attacks and the RowHammer attack require modeling behavior not present in a functional model of the system: viewed formally these attacks introduce transitions in system state that are non-existent in a functional model of the system. Similarly, many side channels are adversary-observable and models of these side channels (e.g., caches/branch predictors) are not included in functional models. As a result, SoC modeling for security analysis needs to incorporate more detail than a functional correctness framework. Furthermore, the framework will need carefully delineate actions performed by trusted components from the actions of untrusted components.

Current efforts at formalization of security guarantees have been restricted to the very specific hardware platform features like secure enclaves [25] or authenticated firmware loaders [9, 16]. There are no *general* frameworks for reasoning about platform security guarantees. Construction of a such a framework for the specification and verification of hardware platform security guarantees is challenging because a useful framework will need to satisfy the following three criteria.

The specification framework must have first-class support for adversary modeling. In particular, the adversary model must be clearly separated from the system model as well as the property specification. To see why this is useful, consider a scenario where a hardware design house produces two versions of an IC: a hardened version for use in safety-critical applications and a "normal" version for commercial off-the-shelf application. Suppose both versions of the IC implement privilege separation. What differentiates the two versions is not the security requirement of privilege separation, but that adversary model for the hardened IC allows more tampering actions by an adversary. Ideally, the model used for reasoning about security of privilege separation for these two versions of the IC will only exhibit minor changes.

Second, the framework must ensure that properties specified are independent of any particular implementation. Property specification must not result in spurious violations when the implementation is changed in some way.

Finally, it must be possible to specify end-to-end or platform security guarantees. It is relatively easier to capture low-level security requirements; e.g., no dereferencing of pointers to deallocated memory (use after free), no writes to a read-only register etc. However, we need the ability to reason about platform security features that span multiple modules and abstraction layers. For example, we would like properties to capture security of complete protocols like Secure Boot [1].

In this paper, we take a step towards tackling these chal-

978-1-7281-2695-1/19/$31.00

lenges. We describe a framework for modeling and verifying hardware platform security. We introduce a property specification language that captures security requirements over this modeling framework. The modeling framework is extends the transition system view of the system with adversarial behavior while the property specification builds on top of HyperLTL [5]. The combination of the two is novel and enables system-level reasoning about security. We provide a formal semantics for this modeling and specification framework. We demonstrate the utility of our framework by a case study of security specification of an authenticated firmware load protocol and review verification results [16].

This paper makes the following contributions.

- We provide an overview of the challenges in SoC security validation using a concrete example of an authenticated firmware load protocol.
- We discuss the design of a framework for SoC security validation. The framework brings together adversary modeling, security property specification and property verification. The combination of these features for general SoC security analysis in a unified framework is novel.
- We introduce a formal modeling framework and property specification language for capturing SoC security properties. While other works have studied modeling of specific protocols (e.g., authenticated load [16]) and security features (e.g., enclave execution [25]), ours is the first general framework for SoC security reasoning. We also provide formal semantics for our framework.

The rest of this paper is organized as follows. Section II presents the motivating example of the authenticated boot protocol used in the rest of this paper. Section III provides an overview of the validation framework. Section IV describes the specification language. Section V presents the case study and verification results. Section VI describes related work and finally section VII provides some concluding remarks.

II. Motivating Example

Figure 1 shows a small SoC design. We use this design to motivate the need for, and describe the features of a high-level security property specification language. We will use an authenticated firmware load protocol as a running example.

A. Features of the Example SoC

The SoC consists of a trusted microprocessor core (labelled μP_1). This trusted core includes a trusted ROM which functions as a hardware root of trust. It also contains an untrusted microprocessor core (μP_2). The two devices used for input and output (I/O) are an untrusted network interface (n/w), and an untrusted flash storage interface (flash). Three trusted accelerators are included for symmetric key cryptography (AES), cryptographic checksum computation (SHA256) and public key cryptography (RSA). A trusted memory management unit (MMU) is used to control access to various memory regions. Permissions for each page can be set to read-only, execute-only or read-write. The MMU controls access to the RAM

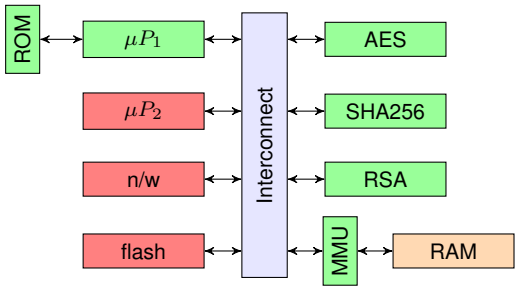

Fig. 1: Example SoC to illustrate the authenticated firmware load protocol. Trusted components are in green, while untrusted components are in red; components of mixed provenance are shown in orange.

which is partially trusted because depending on the MMU configuration, some parts may be adversary accessible.

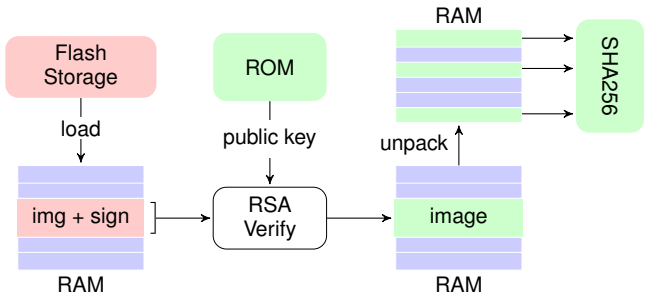

Fig. 2: Authenticated Firmware Load Protocol.

B. The Authenticated Firmware Load Protocol

In this section, we describe a security-critical firmware load protocol. Figure 2 is a depiction of a protocol, and the steps involved in it are as follows.

1) The protocol loads a binary image from the untrusted flash I/O device to the RAM.
2) It then authenticates the loaded image using the RSA accelerator. The public key used for signature verification is stored in the trusted ROM attached to μP_1.
3) If the authentication check passes, the image needs to unpacked. Unpacking involves copying various parts of the contiguous binary file into a set of possibly non-contiguous RAM locations.
4) The unpacked components of the image have to re-verified by computing cryptographic checksums to ensure they have not been modified after unpacking but before execution.
5) If the final checksum verification succeeds, the image is ready for execution. The protocol now jumps to the image's entry-point.

Protocol Security Requirements: A secure implementation of the protocol must ensure integrity of the load process. Despite arbitrary adversarial actions from untrusted components, only an image which satisfies the RSA signature verification and cryptographic checksum verification must be executed.

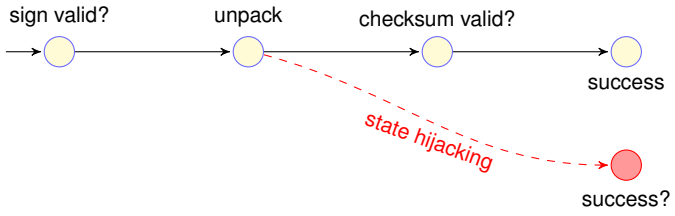

Fig. 3: Traces showing valid boot and compromised boot.

C. A Partial List of Potential Vulnerabilities

To illustrate the difficulty in verification of the protocol's security requirements, we now review two potential classes of vulnerabilities. A more detailed explanation of these vulnerabilities may be found in [16]

1) Protocol State Hijacking Attacks: Protocols like the one shown in Figure 2 are typically implemented as state machines. In this specific example, the first state loads the image from the I/O device to memory while the second state verfies the digital signature of the image header using public key cryptography. The penultimate state computes cryptographic checksums of blocks in the unpacked image and compares these computed values to corresponding values stored in the header. If an adversary could somehow modify the state of the state machine to skip certain validity checks, then invalid images may be loaded by the protocol. These attacks are referred to as protocol state hijacking attacks and an example is shown in Figure 3. Here the adversary causes the state where checksums are verified to be skipped. There are many ways in which such an attack may be mounted. For instance, the untrusted code running on μP_2 may exploit a buffer overflow vulnerability in an interrupt handler executing on μP_1 to overwrite the return address and set the PC to an invalid value.

2) Time of Check to Time of Use (TOCTOU) Vulnerabilities: A TOCTOU attack occurs when the data is changed between the time of validation and the time of its use. In our scenario, an attacker may wait until HMAC validity is checked and then replace parts of the image with a malicious payload.

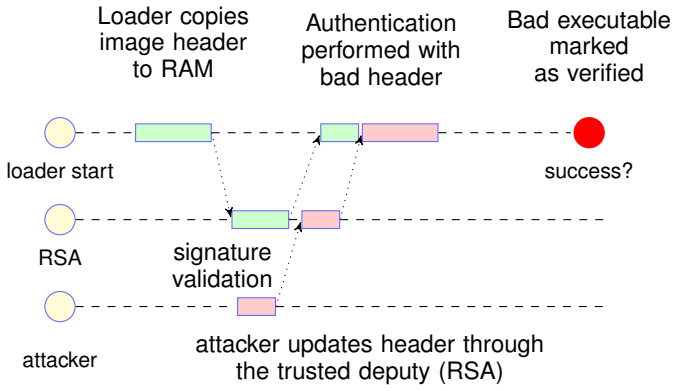

Fig. 4: TOCTOU attack mounted using a confused deputy. Figure taken from [16].

An example is shown in Figure 4. Notice that the attacker does not directly modify the image, but instead "tricks" the trusted RSA accelerator into overwriting the image. Naïve verification techniques which only check access control properties (i.e. check that untrusted devices cannot overwrite to the image) will not be able to detect this bug.

III. SoC Security Validation

As Section II-C demonstrates, verification of the example protocol's security requirements are non-trivial. We discuss some challenges unique to security validation first and then provide an informal overview of our ongoing research that is building a framework to tackle some of these challenges.

A. Security Validation vs. Functional Validation

It is important to emphasize that security validation is in fundamental ways different from functional validation. For the case of functional validation, well-developed techniques for formal specification and formal/semi-formal verification exist. For instance, SystemVerilog Assertions (SVA) [28] is a specification language for hardware functionality that is based on the formalism of linear temporal logic (LTL) [17]. SVA has seen wide adoption in industry and both formal and semi-formal tools for the validation of SVA properties exist.

Unfortunately, it is not possible to express information flow security properties using SVA (or LTL) [15]. As an example, consider the authenticated loader protocol described in the previous section. We can attempt to express security of this protocol using the SVA property: `eventually` $success \rightarrow (SignValid(\mathsf{img}) \wedge ChecksumValid(\mathsf{img}))$. This property states that if the authentication succeeds, then the initial value of the image must have a valid signature and checksum. Unfortunately, an implementation which satisfies this property need not be secure. In particular, it could be vulnerable to TOCTOU attacks despite satisfying the property: the property only requires the initial image be valid, but intermediate steps could replace it with a malicious payload.[1] For a formal proof see McLean [15] but the intuition is that that LTL/SVA cannot reason precisely about how adversarial interference affects execution of the protocol. In the next section, we will introduce a specification language that is capable of this reasoning.

B. Proposed Verification Framework

There three main challenges in SoC security verification are adversary modeling, security specification and verification. Our framework addresses these challenges using the combination of techniques shown in Figure 5. Besides the above, other important components include an instrumentation layer that combines the system and adversary to produce a unified model. This model is analyzed in the simulation/verification

[1] A reader might wonder why we are considering valid signatures and checksums on the initial image rather than the image at the time when loading has succeeded. The problem is that the image is loaded into memory, its signature is checked and then unpacked. In order to reduce memory pressure the initial image is deallocated after unpacking and before checksum validation. Therefore, at the time when the protocol completes, the original image does not exist in the system as its memory has been deallocated!

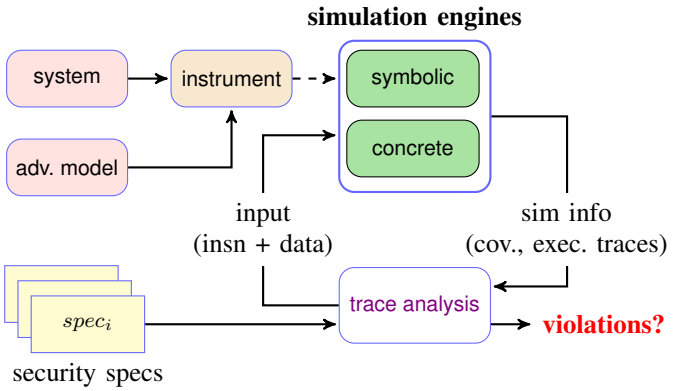

Fig. 5: Overview of the semi-formal solution

tool using the security specification provided by the user. A brief overview of the framework follows.

1) Adversary Modeling: Security verification of any system requires modeling adversarial behavior. This involves precisely specifying what actions an adversary can take in order to tamper with the security objectives of the system and what parts of system state are adversary-observable.

Our adversary model consists of two components: a model of adversarial tampering that captures how untrusted modules can change shared state and an observation function that determines what states are adversary-visible. Tampering captures how the adversary may violate system integrity while the observation function enables reasoning about whether secret information is being leaked to the adversary. Specifically, modules in the SoC can be marked as adversarial: this means that the modules perform unconstrained state updates to all state that is accessible to them. This models adversarial tampering.

On the confidentiality side, the framework allows marking adversary observable state and the framework produces proof obligations that correspond to observational determinism – adversary observations must be a deterministic function of adversarial actions [29]. In the formal model, these actions correspond to extending a standard transition system model by the inclusion of a *tamper* relation and an *observation* function. The tamper relation is composed with each step of the trusted components' transition relations. The observation function determines which parts of state are adversary-observable.

2) Security Property Specification: Consider the example protocol discussed in the previous section. What tests or formal properties can we use to determine if the protocol implementation is secure? It should be apparent that this is a challenging and application-specific problem. While it is not possible to anticipate all attacks, it is in fact possible to rule out certain classes of attacks that involve all permutations and combinations of specific actions taken by an attacker. In past work we have shown that the security properties of enclave platforms like Intel SGX can be captured by only three security guarantees [25]. Similarly, security guarantees for the authenticated load protocol can also be formally specified [16].

Despite these advances there does not yet exist a general framework for formal security property specification and our solution to this problem is described in this paper. Our solution buildss on the recent work in hyperproperties, specifically HyperLTL [5] which can be used for specifying security properties like non-interference and observational determinism [4]. An important innovation in our framework is first-class support for the *tamper* and *observe* functions in the specification language. This allows the adversary model to be parameterized in the property. Therefore, our specification language orthogonalizes these separate concerns of security specification and adversary modeling. We describe the specification language and its semantics in detail in the next section.

3) Verification Techniques: Current work for verification of security properties has mostly relied on self-composition [2, 26]. Unfortunately, self-composition does not scale to complex systems. While a number of recent efforts have studied techniques for scaling self-composition in the context of model checking, it remains unlikely that any model checking approach will scale up to the verification of large SoC designs. Therefore, our framework is developing semi-formal techniques that build on whitebox fuzzing [7] and concolic execution [3], for the verification of SoC security properties. While fuzzing and concolic execution to find violations of safety properties is well-studied, we are interested in violations of security guarantees and this will require the development of novel extensions to the fuzzing and concolic execution algorithms. These techniques have successfully scaled to (safety) verification of very large programs, so we are optimistic about their applicability to SoC security verification.

IV. Specification Language and Formal Semantics

This section presents a formal overview of the security property specification language via its grammar and semantics.

A. System and Adversary Model

We use the standard transition system model of systems with two extensions for capturing security properties. Our first extension is the introduction of a *tamper* relation that captures an adversary's ability to make untrusted updates to system state – this is required to model integrity. The second extension captures what part of system state is adversary observable through the definition of an *observation* function.

We define transition systems as the tuple $M = \langle X, init, tx, tmpr, obs\rangle$. In the above definition X is a vector of state variables and a state of the system σ is an assignment to the variables in X. $init(\sigma)$ is a predicate that represents valid initial states. $tx(\sigma, \sigma')$ is the transition relation that models trusted state updates. Adversary tampering — which captures adversarial state updates — is modelled via the relation $tmpr(\sigma, \sigma')$. The *observation function* $obs(\sigma)$ models adversary observable parts of system state.

A trace of the system M is a sequence of states $\pi = \langle \sigma_0, \sigma_1, \dots \rangle$ such that:

- $init(\sigma_0)$ is true,
- $\forall i \geq 0.\ tx(\sigma_i, \sigma_{i+1}) \vee tmpr(\sigma_i, \sigma_{i+1})$ is true.

The above definition says that system state starts in some initial state and then evolves either through the transition relation tx which models the trusted state updates to system or through the tampering relation $tmpr$. The latter models untrusted or adversarial updates to system state. The set of all traces of a system M is denoted by TR_M.

We will refer to the individual elements of the trace as π^1, π^2, etc. In the above example where $\pi = \langle \sigma_0, \sigma_1, \dots \rangle$, $\pi^0 = \sigma_0$ and $\pi^1 = \sigma_1$. Given the trace π, we write $\pi[i, \infty]$ to denote its suffix starting from index i; i.e., $\pi[i, \infty]$ is the trace $\langle \pi^i, \pi^{i+1}, \dots \rangle$. Note that traces are of infinite length.

B. Property Specification Language

The grammar for the property specification language is shown in Figure 6. It closely follows HyperLTL [5], and thus allows quantification over traces.

$$
\begin{aligned}
\psi \quad &::= \forall \pi.\ \psi \mid \exists \pi . \psi \mid \varphi \\
\varphi \quad &::= \mathcal{P}_{\pi_1,\dots,\pi_k} \mid tmpr_{\pi} \mid \neg tmpr_{\pi} \mid \\
&\quad \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
&\quad \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi\, \mathcal{U}\, \varphi
\end{aligned}
$$

Fig. 6: Property Specification Language Grammar

In the above, $\mathcal{P}_{\pi_1,\dots,\pi_k}$ refers to a predicate $\mathcal{P}$ over the states $\pi_1^0, \dots, \pi_k^0$. $tmpr_{\pi}$ holds when the first transition in the trace π corresponds to a step of the $tmpr$ relation: $\langle \pi^0, \pi^1 \rangle \in tmpr$. $\neg tmpr_{\pi}$ holds when the first transition in the trace π corresponds to either a stuttering step: $\pi^0 = \pi^1$ or a step of the trusted transition relation tx. Note the asymmetry between $tmpr_{\pi}$ and $\neg tmpr_{\pi}$; we use $\neg tmpr_{\pi}$ refer to the adversary performing a "NOP" in the first step of the trace π.

1) Semantics: As in HyperLTL [5], the validity judgement of a property ψ by system $M = \langle X, init, tx, tmpr, obs \rangle$ is defined with respect to a trace assignment $\Pi : \mathcal{V} \to \mathsf{TR}_M$. Here, $\mathcal{V}$ is a trace variable; recall that TR_M is the set of traces of the system M. The partial function Π is a mapping from trace variables to traces. We use the notation $\Pi[\pi \mapsto \rho]$ to refer to a trace assignment that is identical to Π except that π maps to ρ.[2] We write $\Pi \models_M \psi$ if the system M satisfies the property ψ under the trace assignment Π.

We use the notation $\Pi[i, \infty]$ as an abbreviation for the new trace assignment obtained by taking the suffix starting from index i of every trace in Π: $\Pi'(\pi) = \Pi(\pi)[i, \infty]$ for every trace $\pi \in \Pi$. We write $\Pi \not\models_M \psi$ when $\Pi \models_M \psi$ is not satisfied. We say that system M satisfies the property ψ, denoted by $M \models \psi$ if $\Pi \models_M \psi$ for the empty trace assignment $\Pi = \emptyset$.

$\Pi \models_M \forall \pi.\ \psi$	iff for all $\rho \in \mathsf{TR}_M : \Pi[\pi \mapsto \rho] \models_T \psi$
$\Pi \models_M \exists \pi.\ \psi$	iff there exists $\rho \in \mathsf{TR}_M : \Pi[\pi \mapsto \rho] \models_T \psi$
$\Pi \models_M \bigcirc\psi$	iff $\Pi[1, \infty] \models \psi$
$\Pi \models_M \Box\psi$	iff for all $0 \le i : \Pi[i, \infty] \models \psi$
$\Pi \models_M \Diamond\psi$	iff there exists $i \ge 0 : \Pi[i, \infty] \models \psi$
$\Pi \models_M \psi\, \mathcal{U}\, \varphi$	iff there exists $i \ge 0 : \Pi[i, \infty] \models_T \varphi$ and for all $0 \le j < i : \Pi[j, \infty] \models_T \psi$
$\Pi \models_M \mathcal{P}_{\pi_1,\dots,\pi_k}$	iff $\mathcal{P}(\Pi(\pi_1)^0, \dots, \Pi(\pi_k)^0)$ is true
$\Pi \models_M \neg tmpr_{\pi}$	iff $\Pi(\pi)^0 = \Pi(\pi)^1$ or $\langle \Pi(\pi)^0, \Pi(\pi)^1 \rangle \in tx$
$\Pi \models_M tmpr_{\pi}$	iff $\langle \Pi(\pi)^0, \Pi(\pi)^1 \rangle \in tmpr$
$\Pi \models_M \neg\psi$	iff $\Pi \not\models_M \psi$
$\Pi \models_M \psi \wedge \varphi$	iff $\Pi \models_M \psi$ and $\Pi \models_M \varphi$
$\Pi \models_M \psi \vee \varphi$	iff $\Pi \models_M \psi$ or $\Pi \models_M \varphi$

Fig. 7: Semantics of the property specification language.

V. Case Study: Authenticated Firmware Load

We now return to the authenticated firmware load protocol for the SoC described in Section II. We first describes properties that capture security of the protocol and then present verification results.

A. Security Properties

Protocol security is captured by the following properties.[3]

1) Tamper Evidence: This property captures the requirement that that every image that is successfully executed after verification in the presence of adversary operations is also successfully executed in the absence of adversary operations. This means the adversary cannot turn a "bad" image into one that is eventually executed. It is formally stated as follows.

$$
\begin{aligned}
&\forall \pi_1 \pi_2.\ \mathsf{flash}_{\pi_1} = \mathsf{flash}_{\pi_2} \implies \\
&\quad \big(\Box\neg tmpr_{\pi_2} \wedge \Box(success_{\pi_1} \implies success_{\pi_2})\big) \qquad (1)
\end{aligned}
$$

2) Image Invariance: This property states that for every pair of traces which start with identical images stored in flash and eventually verify and execute this image, the images loaded into memory must be identical for both traces.

$$
\begin{aligned}
&\forall \pi_1 \pi_2.\ \mathsf{flash}_{\pi_1} = \mathsf{flash}_{\pi_2} \implies \\
&\quad \Box\big((success_{\pi_1} \wedge success_{\pi_2}) \implies (\mathsf{data}_{\pi_1} = \mathsf{data}_{\pi_2})\big) \qquad (2)
\end{aligned}
$$

Equation 2 is violated by TOCTOU bugs. It ensures that loaded images cannot be tampered with by an adversary.

B. Verification Results

We implemented a simplified model of the protocol described in Figure 2 which processes only one block of data using the open source UCLID5 modeling and verification language [22, 27]. We verified that the protocol satisfies the two security properties using model checking. In future work, we plan to extend this verification to use symbolic simulation and/or whitebox fuzzing. Verification results were obtained by running UCLID5 on an Intel Core i7 5500U CPU operating at 2.4 GHz with 16 GB of RAM. More details about the experiments can be found in [16].

Verification results are shown in Table I. We see that the properties are verified relatively quickly for various different

[2]Note we abuse notation and use π to refer to both trace variables and traces. Which of these is being referred to should be clear from the context.

[3]More detailed explanations and proofs of security may be found in [16].

TABLE I: Verification Times.

Bit-width	Prop. 1	Prop. 2	Result
8	4.3s	5.2s	✓
12	4.9s	12.9s	✓
16	4.6s	15.0s	✓
24	6.4s	13.8s	✓
32	7.8s	16.5s	✓

bit-widths. That said, this proof required manual guidance via the provision of inductive invariants to the verification engine. While the verifier can check validity of the provided invariants, coming up with the correct invariants is manual and time-consuming process. Future work will attempt to derive these invariants automatically.

VI. Related Work

Seminal work in security property verification was done by Goguen, Meseguer and Rushby [8, 20]. Goguen and Meseguer introduced non-interference [8] while Rushby introduced the notion of separability [20]. Each of these, as well as related ideas like observational determinism [29] are instances of hyperproperties [4]. A large body of work has studied type systems that enforce information-flow security [21]. Recent examples for hardware design are [10, 30]. The main disadvantage with these approaches is that the language comes with a "baked-in" adversary model and security property class. For instance, it is unclear how one would extend these works to verify privilege separation against a Rowhammer attack – such an attack requires a way of modeling adversary actions that are distinct from the functional transition system. The main contribution of this paper is a formal modeling and specification framework addressing these gaps.

VII. Conclusion

This paper introduced a formal framework for adversary modeling, security property specification and security verification in modern SoC designs. We provided formal semantics for this modeling and specification framework and demonstrated using a case study of an authenticated firmware load protocol that it is possible to systematically verify SoC platform security requirements using our framework.

Acknowledgements

This work was supported in part by the Semiconductor Research Corporation under task 2854.0001.

References

[1] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 65–, Washington, DC, USA, 1997.

[2] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 100–114. IEEE, 2004.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of Operating Systems Design and Implementation*, 2008.

[4] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, September 2010.

[5] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl ^*. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 30–48, 2015.

[6] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 608–621. ACM, 2016.

[7] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[8] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982.

[9] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor. Security of SoC firmware load protocols. In *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 70–75, 2014.

[10] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A Language for Hardware-Level Security Policy Enforcement. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 97–112, 2014.

[11] D. Lustig, M. Pellauer, and M. Martonosi. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646, 2014.

[12] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016.

[13] S. Malik and P. Subramanyan. Invited: Specification and Modeling for Systems-on-chip Security Verification. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 66:1–66:6. ACM, 2016.

[14] Y. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[15] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 1994.

[16] Sujit Kumar Muduli, Pramod Subramanyan, and Sayak Ray. Verification of Authenticated Firmware Loaders. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE, 2019.

[17] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[18] A. Reid. Trustworthy specifications of ARMv8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE, 2016.

[19] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi. End-to-end verification of processors with ISA-Formal. In *Proceedings of the International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[20] John M. Rushby. Proof of separability: A verification technique for a class of a security kernels. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, pages 352–367, 1982.

[21] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[22] Sanjit A. Seshia and Pramod Subramanyan. UCLID5: Integrating modeling, verification, synthesis and learning. In *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, October 2018.

[23] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik. Template-based Synthesis of Instruction-Level Abstractions for SoC Verification. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE, 2015.

[24] Pramod Subramanyan. *Deriving Abstractions to Address Hardware Platform Security Challenges*. PhD thesis, Princeton University, 2017.

[25] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2435–2450, 2017.

[26] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the International Static Analysis Symposium*, pages 352–367. Springer, 2005.

[27] UCLID5 Verification and Synthesis System. Available at http://github.com/uclid-org/uclid/, 2019.

[28] Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2005.

[29] Steve Zdancewic and Andrew C Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 29–43. IEEE, 2003.

[30] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 503–516, 2015.