Verification of Authenticated Firmware Loaders

Sujit Kumar Muduli Pramod Subramanyan
Indian Institute of Technology, Kanpur
{*smuduli, spramod*}*@cse.iitk.ac.in*

Sayak Ray
Intel Corporation, Hillsboro, OR
sayak.ray@intel.com

***Abstract*—An important primitive in ensuring security of modern systems-on-chip designs are protocols for authenticated firmware load. These loaders read a firmware binary image from an untrusted input device, authenticate the image using cryptography and load the image into memory for execution if authentication succeeds. While these protocols are an essential part of the hardware root of trust in almost all modern computing devices, verification techniques for reasoning about end-to-end security of these protocols do not exist.**

In this paper, we take a step toward addressing this gap by introducing a system model, adversary model and end-to-end security property that enable reasoning about the security of authenticated load protocols. We then present a decomposition of the security property into two simpler hyperproperties. This decomposition enables more scalable verification. Experiments on a protocol model demonstrate viability of the methodology.

I. Introduction

Many system security guarantees in systems-on-chip (SoC) designs rely on authenticated firmware loaders. These loaders are programs that read in a binary image (aka an executable file) from an input device and authenticate the image using public key cryptography to ensure it is from a trusted source. If authentication succeeds, the loader copies the image into memory for execution. Authenticated firmware loaders are used in many important security-critical scenarios in modern SoCs. For instance, they are the most important component of secure boot protocols [1, 14, 21, 24, 25, 38].

While secure boot protocols are almost ubiquitous in modern SoCs, we illustrate its importance using a concrete example of the Sanctum enclave processor [12]. In Sanctum, a secure bootloader loads a trusted *security monitor* into memory when the system boots [25]. The security monitor ensures isolations between different enclaves, as well as isolation between enclaves and other software on the system. A malicious security monitor renders the enclave platform's software isolation guarantees meaningless. Therefore, it is essential to ensure that an untrusted security monitor is never loaded at boot time. This is achieved by using an small immutable loader, typically stored in ROM, that loads the actual security monitor from an untrusted input device, typically flash storage. This loader authenticates the security monitor using public key cryptography and aborts if authentication fails.

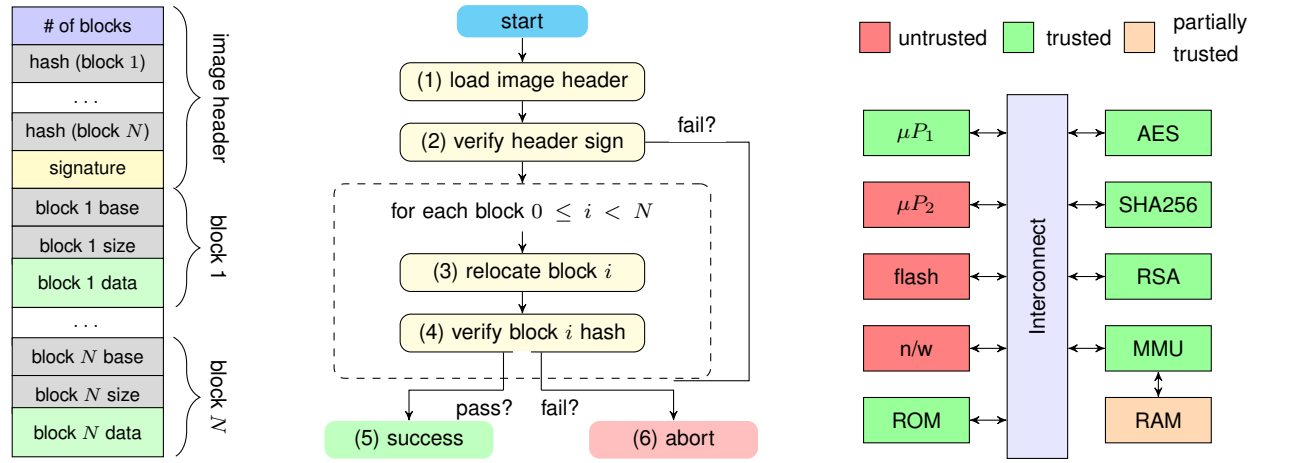
A second important usage scenario for authenticated loaders is trusted firmware updates to a system [13, 21, 41]. Consider a scenario where trusted but possibly vulnerable firmware is being executed on a system. The original equipment manufacturer (OEM) may wish to update this firmware by replacing it with a version where certain security vulnerabilities are fixed. Here too, it is essential to ensure that only authenticated updates/patches are applied on the firmware. Otherwise, attackers would just use the update feature to load malicious firmware onto victims' devices. In the worst case, this would result in malicious firmware being permanently installed and used on victim devices. But even if the worst case scenario is prevented by secure boot, a vulnerable updater allows remote attackers to cause permanent denial of service by installing an unauthenticated bootloader.

The above examples demonstrate the security-critical nature of authenticated firmware loaders. It would not be an exaggeration to say that security guarantees of almost all modern SoCs rest on the security of the authenticated firmware loaders used by them. However, ensuring security of authenticated loaders has many subtleties associated with it. As prior work by Krstic et al. [24] has noted, and as we discuss in Section II of this paper, loaders are vulnerable to a number of different classes of bugs: race conditions, time-of-check to time-of-use (TOCTOU) attacks, confused deputy attacks and control-flow hijacking attacks. The chief difficultly in reasoning about security is the presence of an active attacker executing concurrently with the loader protocol. The attacker can manipulate shared state in order to trick the protocol into validating a bad image.

As a result, formal verification of the security of authenticated loaders is extremely important. Unfortunately, despite their importance to system security, verification techniques for *end-to-end* security verification of authenticated loaders do not exist. There are two reasons for this. The first is a lack of techniques for adversary modeling. As we are dealing with an active adversary, it is important to formulate an adversary model that captures the full range of attacker behavior.

The second is a security specification problem: it is unclear what property or class of properties can ensure loader security. On the one hand, trace properties are too weak to capture security requirements of authenticated load as they cannot reason about flow of information. On the other hand, secure information flow properties like non-interference [20] and observational determinism [27, 29, 40] are too strong as they require that the loader must succeed despite adversary interference. (We discuss the specification issue further in Sections II-C and IV-B1.)

In this paper, we address both of the above challenges and introduce a methodology for the verification of authenticated loaders. Our solution is based on the formulation of an abstract model of loader protocols and specification of correct behavior in the absence of adversary. We then extend this model to allow flexible modeling of adversary interference. We then



(a) Image data structure. (b) Authenticated load protocol. (c) Example SoC for authenticated load.

Fig. 1: Overview of authenticated firmware load protocols used in contemporary SoC designs.

introduce a security property that captures correctness of authenticated load. The property relates protocol behavior with and without adversary actions. Intuitively, it states that every execution where the protocol verifies an image in the presence of adversarial behavior must correspond to an equivalent execution in the model without the adversary. This security property is a hyperproperty [10] with quantifier alternation and is not subset-closed and so it is difficult to verify. We address this verification challenge by introducing a decomposition of the above property into two simpler 2-safety properties [36]. We show that satisfaction of the 2-safety properties implies satisfaction of the loader security property.

A. Contributions

This paper makes the following contributions.

- We introduce a methodology and adversary model reasoning about the security of authenticated load protocols.
- We introduce a formal security guarantee that captures security of authenticated load protocols.
- We introduce a decomposition of the security guarantee into two simpler-to-verify 2-safety properties. We show that satisfaction of these properties guarantees security of authenticated load.
- We demonstrate proof of concept verification of the security property on simple but illustrative models.

The rest of this paper is organized as follows. Section II describes the class of authenticated load protocols and their security vulnerabilities. Section III describes the adversary model and security property. Section IV presents the decomposition into 2-safety properties. Section V describes the experimental evaluation. Section VI discusses related work and finally section VII provides concluding remarks.

II. Overview of Protocols and Vulnerabilities

In this section, we present an overview of authenticated firmware load implementations and describe some of the associated security requirements and potential vulnerabilities.

A. Authenticated Firmware Load Protocols

Figure 1b shows a simplified flowchart for one example of an authenticated load protocol while Figure 1a shows the image data structure the protocol operates on. The protocol we describe is representative; in particular, it is based on implementations in commercial SoCs [24] as well as the secure boot implementation in the open source Sanctum processor [25].

The image data structure depicted in Figure 1a has two parts. The image header contains the number of blocks and the hash of each block. This header is signed using a private key, and this signature can be verified using the corresponding trusted public key. Typically, this public key is stored in ROM or loaded from fuses. The second part of the image is a contiguous sequence of blocks. The blocks themselves are not signed and their authenticity is verified by computing the hash of the block contents and comparing these with the hashes in the header. The benefit of this image organization is that expensive public key cryptography need only be performed over the header, which is much smaller than the rest of image.

The steps involved in the protocol are as follows.

1) The protocol loads a binary image from the untrusted input device (e.g., flash storage, hard disk or network interface) to the RAM.
2) It then checks the authenticity of the *header* of the loaded image using cryptographic signatures. Often this authenticity check is implemented using dedicated hardware accelerators.
3) Note that the image stored on the I/O device is a contiguous block of bytes. However, when placed into memory, the different segments/blocks may not be adjacent to each other. Therefore, steps 3 and 4 iterate over each block of the image. Step 3 moves block i of the image to its eventual location in the RAM.
4) Finally, step 4 computes cryptographic checksums over the relocated block and compares this checksum with the checksum stored in the header.

It is important to note that this is one (simplified) instantiation of an authenticated load protocol. Many variants are used in practice. For example, blocks may need to be decrypted and/or decompressed before relocation. Some blocks may be stored as binary diffs aka "patches" w.r.t to an existing binary, rather than a contiguous block of data. The loader will need to apply each patch rather than just copy a block of bytes. The loader may be a multi-stage algorithm where the first stage loads the second stage loader into memory and the second stage loader fetches and authenticates the actual binary. These variants all share a common set of security guarantees and are vulnerable to similar attacks. *In this paper, we study the class of protocols and not a specific instance of the protocol. Our theoretical results are not restricted to just the specific variant shown in Figure 1b.*

1) System and Threat Model: The protocol is typically executed on a SoC which consists of both trusted and untrusted components. In the simple SoC shown in Figure 1c, the protocol executes on a trusted microcontroller (μP_1) and makes use of two trusted crypto engines: SHA256 for computing cryptograhic checksums and RSA for public key cryptography. While the protocol is being executed on μP_1, untrusted code is running in parallel on μP_2. This code can attempt to configure and initiate operation of the other accelerators and modify memory arbitrarily.

2) Protocol Security Requirements: A secure implementation of the protocol must ensure that despite arbitrary adversarial actions from untrusted components, only images with valid signatures and cryptographic checksums must be loaded. We will make this informal definition precise in Section III-B2.

B. Potential Vulnerabilities in the Protocol

The protocol as shown in Figure 1b is deceptively simple. In practice, there are many subtleties to its implementation and if these are not handled correctly, invalid images may be loaded and executed with disastrous results for system security. To help understand some of these subtleties, we now describe three categories of protocol vulnerabilities.

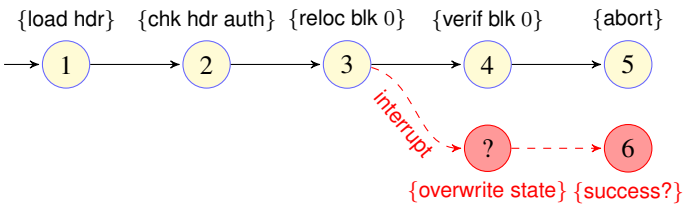


Fig. 2: Example of protocol state hijacking.

1) Protocol State Hijacking: The protocol consists of a sequence of checks, each of which must be carried out faithfully to ensure its security. If the adversary is able to modify system state in order to "trick" the loader into skipping steps, this may allow invalid/insecure images to be loaded.

As a specific example, consider an implementation where the loader uses a finite state machine (FSM), with state variables stored on the firmware stack to step through the various stages of image authentication. Further suppose the adversary can cause interrupts to occur on the microcontroller executing the protocol. The interrupt handler may have a buffer overflow vulnerability which may be exploited to change the state of the FSM. An example of this vulnerability is shown in Figure 2, where the adversary prevents checking of the hash of the block 0, by causing state 4 to be skipped.

2) Time of Check to Time of Use (TOCTOU) Vulnerabilities: This classic attack refers to the scenario where the data is changed between the time of validation and the time of its use. In our example, an attacker may wait until header authenticity is checked and then replace parts of the header with a malicious payload. This is depicted in Figure 3.

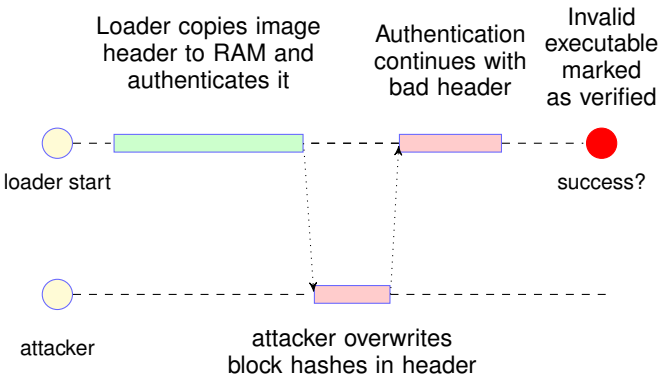


Fig. 3: TOCTOU attacks on authenticated firmware load.

3) Confused Deputy Attacks: A commonly used technique for preventing TOCTOU attacks is make the object being checked immutable before the check. One way to accomplish this is to marking regions of memory as read-only.

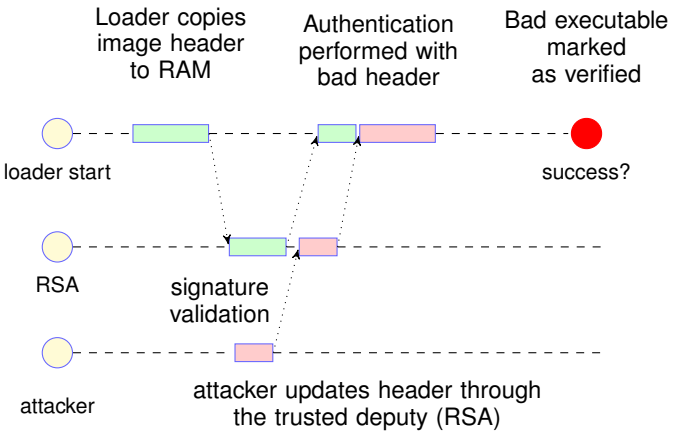


Fig. 4: TOCTOU attack mounted using a confused deputy.

However, marking regions of memory read-only has subtleties associated with it in a system containing multiple principals. Suppose the data is marked as read-only to the attacker, but read-write to cryptographic engines. This may seem reasonable because the cryptographic engines are trusted. However, although the attacker cannot directly alter the image,

she may confuse trusted components into doing the modifications for her. For instance, the attacker might invoke the engine with a command that results in its output overwriting the header. This is an example of a *confused deputy* being used to mount a TOCTOU attack and is depicted in Figure 4.

C. Challenges in Formal Specification of Protocol Security

As Section II-B demonstrates, protocols in this class have a number of subtle security vulnerabilities which can only be prevented by careful reasoning about adversarial actions. Formal verification of the protocol can help address this problem, but the property specification research challenge needs to be addressed: it is not straightforward to come up with a property specification that captures protocol security.

As a strawman, consider a property which states that authentication must succeed if and only if the initial value of the image binary has a valid header and each block in the initial image has a valid hash. While this seems like a reasonable property, an implementation which satisfies this property need not be secure. Although the property ensures the initial image is valid, intermediate steps may replace it with a malicious payload. Therefore, an implementation satisfying the property may be vulnerable to TOCTOU and confused deputy attacks.

We will also demonstrate in Sections III and IV-B1 that security of the protocol is not captured by secure information flow properties such as noninterference or observational determinism. The problem is that noninterference is too strong a requirement: it requires that regardless of what the adversary's actions are, the system must boot a valid image. This needs strict isolation between the adversary and the trusted loader, which requires specialized hardware support and so is more expensive in terms of hardware and design cost. Instead, many practical implementations only guarantee that *if* the image is marked as verified and loaded for execution, *then* it must have been valid. This subtle but important difference means techniques for verification of secure information flow cannot be directly applied to the authenticated loader verification.

III. Security Specification

In this section, we first present a model for authenticated firmware load protocols. The first model includes the protocol but not the adversary. We then extend the model to include adversary tampering, and then introduce an end-to-end security property over the extended model that captures security of an authenticated loader.

A. An Abstract Model of Authenticated Firmware Load

We will model the authenticated load protocol as a transition system $M = \langle \Sigma, init, tx \rangle$. The set of states of the transition system is given by Σ. We use $\sigma, \sigma_1, \sigma_2$ etc. to refer to individual states of the transition system, where $\sigma_i \in \Sigma$. $init$ is the set of initial states, while tx is the transition relation. A trace of the system π is a sequence of states $\pi = \langle \sigma_0, \sigma_1, \ldots, \sigma_i, \ldots \rangle$ such that: $\sigma_0 \in init$ and for all $i \geq 0$, $(\sigma_i, \sigma_{i+1}) \in tx$. We use the notation π^i to denote the ith element of the trace π. In the above example, $\pi^3 = \sigma_3$. The set of all traces of a system M is denoted by $\mathsf{TR}(M)$.

When the protocol begins execution, it reads the firmware image from an input device (e.g., flash storage or network device). Given a state of the transition system σ, we denote image data stored on the input device in state σ by the term $\mathsf{img}(\sigma)$. Note here that $\mathsf{img}(\sigma)$ refers to the *entire* block of image data, including both headers and data blocks.

For example, for the protocol in Figure 1b, $\mathsf{img}(\sigma) \doteq \sigma.inputDev[\mathsf{baseAddr} : \mathsf{baseAddr} + \mathsf{len}]$. Here $inputDev$ is an array that models the contents of the input device. We use notation $\mathsf{arr}[start : end]$ to denote the slice of array between the indices $start$ and end. We are using the notation $\sigma.var$ to refer to the valuation of the state variable var in the state σ.

Viewed abstractly, the protocol has to perform a number of checks to determine validity and authenticity of the image. The precise number of checks to be performed may be a function of the image data and system state. We denote this by the term $\#\mathsf{chks}(\sigma)$. For the protocol in Figure 1b, the number of checks to be performed is $1 + \sigma.header.numBlocks$: one check for the header and one for each block contained in the image.

Each of the checks is denoted by the predicate $\mathsf{valid}_i(\sigma)$ where $1 \leq i \leq \#\mathsf{chks}(\sigma)$. Returning to the example protocol in Figure 1b, $\mathsf{valid}_1(\sigma)$ is true if the header signature is valid in state σ, $\mathsf{valid}_2(\sigma), \ldots, \mathsf{valid}_i(\sigma), \ldots$ etc. are true if the appropriate block's hash is equal to the corresponding value stored in the header.

$$\mathsf{valid}(\sigma) \doteq \bigwedge_{i=1}^{\#\mathsf{chks}(\sigma)} \mathsf{valid}_i(\sigma) \tag{1}$$

We will use the predicate $\mathsf{valid}(\sigma)$ to indicate that all the checks are valid. Like $\#\mathsf{chks}$, valid is a predicate over system state σ rather than just the image data $\mathsf{img}(\sigma)$. This is because the type and number of checks to be performed may depend on system state. For example, some patches may load only on a system with a specific version of firmware.

Finally, when the protocol completes execution it marks an image as verified and therefore eligible for execution, or as an invalid image, in which case the protocol aborts. These are denoted by the state predicates $\mathsf{verif}(\sigma)$ and $\mathsf{aborted}(\sigma)$ respectively. If verification succeeds, the executable data in memory ready for execution is denoted by the term $\mathsf{exec}(\sigma)$.

Security of Authenticated Load without Adversary: An implementation of authenticated load without an adversary is secure if whenever an image is verified, all of the required validity checks on it pass. Given a trace of the transition system M, $\pi = \langle \pi^0, \pi^1, \ldots \rangle$, the above informal definition can be precisely stated as follows:

$$\forall \pi \in \mathsf{TR}(M).\ \left(\exists i.\ \mathsf{verif}(\pi^i)\right) \implies \mathsf{valid}(\pi^0) \tag{2}$$

Property 2 is a trace property and can be expressed in linear temporal logic as $\Diamond \mathsf{verif} \implies \mathsf{valid}$. This property can be verified using standard model checking techniques. In practice,

it turns out be to be somewhat challenging to verify because of the need for modeling cryptography [3, 7, 17, 26].[1] As we will see in the next subsection, it is the introduction of adversarial behavior that makes the above property incomplete.

Implication vs. Bi-Implication: Property 2 uses an implication rather than a bi-implication. We only require that *if* the protocol declares an image as verified, *then* the image data be valid. In other words, this property only requires the *detection* of invalid images; valid images may sometimes not be authenticated. The loader cannot always guarantee that a good image will be loaded even in the absence of adversarial interference. For example, the loader may run out of memory or be unable to access shared resources (e.g., cryptographic accelerators may be unavailable).

B. Abstract Model Including Adversarial Behavior

To extend the model presented in Section III-A to include adversarial behavior, we augment the transition system definition with a *tamper* relation over states. Specifically, our transition systems is the now defined as the tuple $M_{adv} = \langle X, init, tx \circ tmpr \rangle$. This system's transition relation $tx \circ tmpr$ is the composition of the relations tx and $tmpr$. Every step of the system consists of a state update due to the tamper relation and a state update due to the trusted transition relation tx. The former corresponds to transitions initiated by the trusted components in the system while the latter captures the adversary's ability to make untrusted updates to system state. Note that an adversary's state updates are visible to the trusted component, so adversary actions may cause a chain reaction in the trusted code causing so-called confused deputy attacks.

A trace of the augmented transition system M_{adv} is defined as a sequence of states $\pi = \langle \sigma_0, \sigma_1, \dots \rangle$ such that:

- $\sigma_0 \in init$ is true,
- for all $i \geq 0$, there exists σ_i' such that $(\sigma_i, \sigma_i') \in tmpr$ and $(\sigma_i', \sigma_{i+1}) \in tx$.

The above definition says that system state starts in some initial state and then evolves by the composition of the tampering relation $tmpr$ and the trusted transition relation tx.

The predicates $\mathsf{valid}_i(\sigma)$, $\mathsf{valid}(\sigma)$, and $\mathsf{verif}(\sigma)$ as well as the terms $\mathsf{img}(\sigma), \#\mathsf{chks}(\sigma)$ and $\mathsf{exec}(\sigma)$ all have the same definitions for M_{adv} as they do for the transition system model without the adversary M.

1) Defining the Tamper Relation: The tamper relation is the most crucial component of the adversary model and captures how an adversary can affect system state. For the example protocol shown in Figure 1b, our definition of the tamper relation states that untrusted modules (μP_2, flash and network devices) can make arbitrary reads and writes on the shared interconnect. This definition simulates all *functional* attacks carried out by an adversary involving these modules under the assumption that trusted modules do not interact with the untrusted modules except via the shared interconnect.

[1]Our model uses the Dolev-Yao technique [18] and cryptography is modeled using uninterpreted functions along with axioms that state properties like collision resistance and pre-image resistance.

More interesting definitions of the tamper relation can capture sophisticated attacks. To illustrate this, consider fault injection attacks which refer to scenarios where the attacker induces bit-flips in the SoC. These are typically carried out by a physical adversary who launches an electromagnetic pulse at the SoC [2, 28]. If carefully targeted, a crucial bit may be flipped and security guarantees violated. These attacks could be modelled by defining the tamper relation to non-deterministically flip a bounded number of bits in each trace. RowHammer, a software-based fault injection attack on DRAM can also modelled in a similar way [23]. Of course, the specific choice of operations to be included in the tamper relation depends on the SoC's threat model.

2) Security of Authenticated Load with Adversary: The protocol is secure in the presence of adversarial interference if two conditions are satisfied. The first condition states that when an image is verified, then it must be valid. This is the same as Property 2 except it is defined over traces of M_{adv}.

$$\forall \pi \in \mathsf{TR}(M_{adv}).\ \big(\exists i.\ \mathsf{verif}(\pi^i)\big) \implies \mathsf{valid}(\pi^0) \tag{3}$$

The second condition requires that every execution that results in the image being verified in M_{adv} also have a corresponding execution in M starting from the same input image data. This corresponding execution in M should also result in the image being verified, and the executable data loaded into memory should be identical in M_{adv} and M.

$$\begin{aligned}\big(\forall \pi_1 \in \mathsf{TR}(M_{adv}).\ \exists i.\ \mathsf{verif}(\pi_1^i)\big) \implies \\ \big(\exists \pi_2 \in \mathsf{TR}(M).\ \mathsf{img}(\pi_1^0) = \mathsf{img}(\pi_2^0) \implies \\ \exists j.\ \mathsf{verif}(\pi_2^j) \wedge \mathsf{exec}(\pi_1^i) = \mathsf{exec}(\pi_2^j)\big)\end{aligned} \tag{4}$$

Property 4 makes the above definition precise. What is the intuition behind the property? First, there must be no way for an adversary to trick the system into loading a bad image. This is captured by $\mathsf{verif}(\pi_1^i) \implies \mathsf{verif}(\pi_2^j)$. Second, the executable loaded into memory upon validation should not be influenced by the adversary, i.e. $\mathsf{exec}(\pi_1^i) = \mathsf{exec}(\pi_2^j)$.

Unfortunately, Property 4 is challenging to verify due to two reasons. The first problem is quantifier alternation, specifically the existential quantification over traces of M. If implemented naïvely this could devolve into explicit exhaustive search over traces of M. This is why tools for symbolic model checking of temporal hyperproperties, e.g. MCHyper [19], do not allow existential quantification over traces. A reader may wonder why have the existential quantifier at all. Recall the discussion in Section III-A regarding the use of implication rather than bi-implication in Property 2. The same reasoning applies for the use of existential quantification: a loader may not be able to guarantee that a good image is loaded in all executions in the absence of adversarial interference.

The second problem is hidden in the term $\mathsf{exec}(\pi_1^i) = \mathsf{exec}(\pi_2^j)$. Note that exec refers to a region of memory, so this comparison of two memory ranges typically involves universal

quantification over memory addresses. This is also challenging for symbolic model checking algorithms.

IV. Decomposing the Security Property

In this section, we present a technique for more scalable verification of the authenticated load security property applicable to certain common scenarios. We first present an overapproximation of the transition system modeling the protocol and adversary. This overapproximation ensures the tamper relation is reflexive, which in turn allows overapproximation of the existential quantifier in Property 4 by a universal quantifier. We then decompose Property 4 into two 2-safety properties. Finally, we show that if the 2-safety properties are satisfied, then so is Property 4.

A. Overapproximating the Adversary Model

Let us define the transition system $M_{adv+} = \langle \Sigma, init, tx_+ \circ tmpr_+ \rangle$. Here, Σ and $init$. tx_+ is the reflexive closure of the relation tx: $(\sigma_i, \sigma_j) \in tx_+$ if either $\sigma_i = \sigma_j$ or $(\sigma_i, \sigma_j) \in tx$. $tmpr_+$ is defined similarly.

Proposition 1. *M and M_{adv} both refine M_{adv+}.*

Since M_{adv+} simulates both M_{adv} and M, any k-safety property proven over M_{adv+} holds on both M and M_{adv}. Further, note that M does not refine M_{adv} due to the inclusion of the tamper relation in the transitions of M_{adv}.

B. Decomposition into 2-Safety

In the rest of this section, we present two 2-safety properties over M_{adv+} that imply Property 4.

Given a trace $\pi = \langle \sigma_0, \ldots, \sigma_i, \sigma_{i+1}, \ldots \rangle$ of M_{adv+}, we define the predicate $\mathsf{tmprNOP}(\sigma_i)$ to be true either when $(\sigma_i, \sigma_{i+1}) \in tx$ or $\sigma_i = \sigma_{i+1}$. In other words, if a trace satisfies $\forall i.\ \mathsf{tmprNOP}(\pi^i)$ (or equivalently $\Box\,\mathsf{tmprNOP}$), that means all adversary operations in the trace are "no-ops."

1) The No Hijacking Property: This property states that for every image and every execution which results in the image being verified with adversary interference, an execution without adversary interference must also result in the image being verified. This is specified as follows.

$$
\begin{aligned}
&\forall \pi_1 \in \mathsf{TR}(M_{adv+}).\\
&\forall \pi_2 \in \mathsf{TR}(M_{adv+}).\\
&\quad \mathsf{img}(\pi_1^0) = \mathsf{img}(\pi_2^0) && \implies\\
&\quad \mathit{resourceAvail}(\pi_2) && \implies\\
&\quad \big(\forall i.\ \mathsf{tmprNOP}(\pi_2^i)\big) && \implies\\
&\quad \big(\forall i.\ \mathsf{verif}(\pi_1^i) \implies \mathsf{verif}(\pi_2^i)\big) && (5)
\end{aligned}
$$

In the above, $\mathit{resourceAvail}$ is a trace property that guarantees resources are available for validation to succeed; thus ensuring there are no failures unrelated to adversarial actions in π_2. The property ensures that adversary operations can never turn a "bad" initial image into one that is eventually executed. This property is violated when protocol state hijacking occurs. Note that unlike Property 4, this is a 2-safety property [10, 36] and as a result, it is relatively easier to verify.

Noninterference vs. No Hijacking: To understand why the property uses $\mathsf{verif}(\pi_1^i) \implies \mathsf{verif}(\pi_2^i)$ rather than $\mathsf{verif}(\pi_1^i) \iff \mathsf{verif}(\pi_2^i)$ which would be similar to noninterference [20], consider the protocol shown in Figure 5a. This protocol deals with one block of data and authenticates the block by validating the block signature using a trusted public key (step 3). In order to prevent TOCTOU attacks, it marks the region of memory containing the block as read only before signature validation (step 2). This ensures that the signature is computed over the block of data that will be executed, so if signature validation succeeds, then there must not have been any adversary interference. The memory region containing the image is world-writable when it is being loaded from memory.

Figure 5b shows two executions of the protocol. In (i), the adversary is overwrites the image as it is being loaded into memory before the region is set to be read-only. However, this causes authentication to fail and the loader aborts and no damage is done. In (ii), there is no adversary interference and authentication succeeds. This pair of traces satisfies Property 5. However, if the property was $\mathsf{verif}(\pi_1^i) \iff \mathsf{verif}(\pi_2^i)$, then the depicted pair would violate this strawman property. This violation occurs even though the protocol is secure.

This example demonstrates why noninterference is too strong for security verification of authenticated loaders. From an implementation perspective, satisfying noninterference requires designing an MMU that allows fine-grained page permissions such that a page can be written to by the loader and it deputies but not by the attacker nor by attacker-invoked deputies. (Note some deputies may be invoked by both attacker and the loader.) In contrast, Property 5 only requires that page permissions not be modifiable by the attacker. The latter is much easier to implement and requires less hardware support.

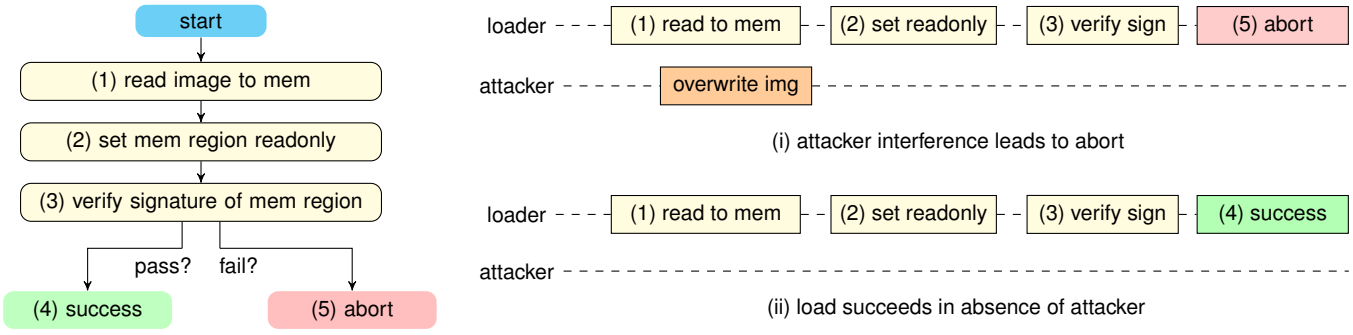
2) The No TOCTOU Attack Property: This property states that for every pair of traces which start with identical images stored in flash such that both eventually validate the image, the executables loaded into memory for these must be identical.

$$
\begin{aligned}
&\forall \pi_1 \in \mathsf{TR}(M_{adv+}).\\
&\forall \pi_2 \in \mathsf{TR}(M_{adv+}).\\
&\quad \mathsf{img}(\pi_1^0) = \mathsf{img}(\pi_2^0) \implies\\
&\quad \big(\forall i.\ \mathsf{verif}(\pi_1^i) \wedge \mathsf{verif}(\pi_2^i) \implies \mathsf{exec}(\pi_1^i) = \mathsf{exec}(\pi_2^i)\big) && (6)
\end{aligned}
$$

Property 6 is violated by TOCTOU bugs and confused deputy attacks which exploit TOCTOU bugs. Satisfaction of this property ensures that the loaded images cannot be tampered with by an adversary. This is also a 2-safety property.

C. Verification of Authenticated Firmware Load

The 2-safety properties of no hijacking and no TOCTOU attacks are important because if they are satisfied for the extended transition system M_{adv+}, then we know that M and M_{adv} satisfy Property 4.



(a) Single-block authenticated load protocol.

(b) Execution traces for the protocol in (a).

Fig. 5: Illustrating Property 5.

Lemma 2. *If the transition system M_{adv+} satisfies the no hijacking (Property 5) and the no TOCTOU attack (Property 6) properties then M_{adv} and M satisfy Property 4.*

Proof Sketch: The proof is by contraction. Suppose it is possible to satisfy Properties 5 and 6 while violating Property 4. Then there must be a counterexample trace π_1 of M_{adv} in Property 4. This trace could be one of two types.

In the first case, there exists no trace $\pi_2 \in \mathsf{TR}(M)$ which starts with the same image data as π_1 and results in successful validation. This means there exists a trace $\pi_2' \in \mathsf{TR}(M)$ which starts with the same image data as π_1 but fails validation. Then π_1 and π_2' are a counterexample to Property 5. Contradiction!

In the second case, there does exist a trace $\pi_2 \in \mathsf{TR}(M)$ which successfully validates the image. However, this trace does not have the same executable data in memory. In this case too, the trace π_2 can be padded with an appropriate number of "no-ops" to construct the trace π_2' such that π_1 and π_2' are counterexamples to Property 6. This is also a contradiction.

1) Simplification of Property 3: Further simplification is possible by noticing that if Properties 5 and 6 are satisfied, then we do not need to consider adversarial interference in Property 3.

$$\begin{aligned}&\forall \pi_1 \in \mathsf{TR}(M_{adv+}).\\ &\quad \big(\forall i.\ \mathsf{tmprNOP}(\pi^i) \wedge \mathit{resourceAvail}(\pi)\big) \implies\\ &\quad \big(\exists i.\ \mathsf{verif}(\pi^i)\big) \implies \mathsf{valid}(\pi^0)\end{aligned} \tag{7}$$

Proposition 3. *If Properties 5, 6 and 7 are satisfied then 3 is also satisfied.*

To see why this is true, suppose Property 7 is satisfied but property 3 has a counterexample. This counterexample along with any satisfying trace of Property 7 violates Property 5.

2) Verification Methodology: Lemma 2 points to a methodology for the verification of authenticated firmware load.

1) Construct the extended transition system M_{adv+}.
2) Verify Properties 5, 6 and 7 on M_{adv+}.
3) If they are satisfied, then the protocol is also secure.

Our methodology has reduced the verification problem to that of verifying two 2-safety properties and a safety property. Unlike Property 4, these properties are all subset-closed and hence preserved by refinement [10]. This means well-studied notions of abstraction and refinement [6] can be applied for scalable verification of the security property.

V. Evaluation

In this section, we describe our evaluation of the methodology presented in this paper.

A. Methodology

We implemented a model of the protocol shown in Figure 5 with only one data block in the UCLID5 modeling and verification framework [32, 37]. UCLID5 uses the Z3 SMT solver to discharge the verification conditions [16]. Our model contains the protocol state machine, an input device, shared memory and a model of the byte-wise cryptographic hash calculation. Cryptography was modelled using uninterpreted functions along with Dolev-Yao axioms [18]. The adversary was abstracted to a single writer that makes an unbounded number of unconstrained writes to the shared memory. As the memory is protected by a memory management unit (MMU), some of these writes may be blocked if target address is marked as not writeable.

Since Properties 5 and 6 are 2-safety properties, we used the standard technique of self-composition [4]. Proofs were done using induction, and several strengthening invariants had to stated in order to prove the properties of interest. Since many invariants are quantified, we specified several quantifier patterns to assist the SMT solver with these invariants.

Experiments were run on an Intel Core i7 5500U CPU operating at 2.4 GHz with 16 GB of RAM.

B. Results

Verification results are shown in Table I. The address and data widths for the memory and I/O devices used in our model of the protocol are parameterizable. We show results for these widths ranging from 8 to 32 bits. The size and location of the image are unconstrained symbolic constants.

Property 7, which is a standard safety property is proved compositionally reasoning by splitting it up into three sub-properties: properties 7a, 7b and 7c. Property 7a states that $\mathsf{digest\_value} = \mathsf{mem\_region\_hash}(\mathsf{mem}, \mathsf{base\_addr}, \mathsf{index})$.

TABLE I: Verification Results.

Bit-width	Property 5		Property 6		Property 7 Prop. 3a		Prop. 3b		Prop. 3c	
	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time
8	✓	4.3s	✓	5.2s	✓	8.2s	✓	3.0s	✓	2.8s
12	✓	4.9s	✓	12.9s	✓	12.1s	✓	3.0s	✓	2.7s
16	✓	4.6s	✓	15.0s	✓	9.0s	✓	3.4s	✓	2.9s
24	✓	6.4s	✓	13.8s	✓	26.2s	✓	3.4s	✓	3.2s
32	✓	7.8s	✓	16.5s	✓	18.5s	✓	4.3s	✓	3.2s

Property 7b shows that in the absence of adversary tampering the contents of memory region where the image is stored are equal to the contents of the input device. This proof is done assuming 7a. Finally, Property 7c states that $\Diamond\,\mathsf{verif} \implies \mathsf{valid}$ and its proof assumes both Properties 7a and 7b. The proofs required a total of 17 unique strengthening invariants across the three different proofs.

We had to add 23 strengthening invariants for properties 5 and 6. The strengthening invariants for these properties were quite similar. Of the 23 strengthening invariants, 19 invariants were relational and among the 19, four involved universal quantification over addresses.

C. Discussion of Results

The verification results shown in this paper are by no means a complete verification of an authenticated loader. However, they are an necessary and important first step towards producing a secure authenticated loader. Given the importance of authenticated loaders to SoC security, we believe it is important to produce a fully-verified implementation of such a loader. However, prior to this work it was unclear what proof obligations would need to be discharged for a fully-verified implementation. Our paper solves this problem by the introduction of Properties 5 and 6. Each of these can be verified on an abstract model of the loader protocol and a refinement proof can then show that the protocol's security guarantees also hold for the implementation.

Our work also points towards a new and important class of 2-safety properties that are not secure information flow. It opens up new avenues of research into the verification of this new class of properties, and provides challenging new benchmarks for verification tools.

VI. Related Work

1) Secure Boot/Authenticated Load Verification: The most closely related effort to ours is Krstic et al. [24] who verify models of authenticated loaders.. They develop a system model and security property that captures TOCTOU attacks. Their work also provides a wonderful exposition of the subtleties involved in the design of authenticated load protocols.

Huang et al. [22] also perform verification of the secure boot implementation in a commercial SoC design. Their innovations include use of the instruction-level abstraction (ILA) for co-verification of hardware and firmware [34] and techniques for analyzing parallel firmware. However, their verification is limited to certain access control properties of the secure boot implementation. These properties are necessary for loader security but are not sufficient to ensure loader security.

Cook et al. [11] verify memory safety of the boot code running in Amazon data centers. While this boot loader code is in fact the code that implements secure boot, they do not verify the integrity of secure boot but instead prove the absence of memory safety errors in the boot code. As with other related efforts, memory safety is a necessary condition for security of boot but by no means is it sufficient.

The main difference between our work and each of the above efforts is that our paper develops an end-to-end security property that implies security of the loader, rather than necessary (but insufficient) properties for loader security.

2) Noninterference and Hyperproperties: Seminal work in verification of secure information flow was done by Goguen, Meseguer who introduced noninteference [20] and Rushby who introduced separability [30]. Both of these, as well as observational determinism [27, 29, 40] are instances of hyperproperties [10].

Noninterference on a multi-user system is defined as the commands of one group of users having no effect on what other groups of users can see. Separability is also a similar notion of isolation between mutually distrustful programs on a system. Observational determinism, when used to prove integrity, means that a trusted component's outputs must be equal if trusted inputs are equal. While these are all important classes of information flow security properties, they are too strict to capture the requirements of the authenticated boot protocol. In particular, we want adversary interference to be detected, but not necessarily prevented. These properties cannot express this requirement.

The class of k-safety properties was introduced by Terauchi and Aiken [36] while the technique of self-composition for the verification of k-safety was introduced by Barthe et al. [4]. More sophisticated approaches to self-composition have been proposed [33, 39].

3) SoC and Firmware Verification: A number of efforts have have studied techniques for firmware verification and many of these have focused on security properties of firmware. For example, S^2E [9] allows symbolic execution of system software. It was used by Bazhaniuk et al. [5] use the S^2E infrastructure to verify security properties of system manage-

ment mode software in x86 systems. FIE [15] introduces novel optimizations for scalable symbolic execution of firmware in TI MSP430 microcontrollers. Schmidt el al. [31] introduced the notion of a program netlist and used this for co-verification of firmware and hardware. Chen et al. [8] introduce CRETE, which also enables concolic testing of firmware. These efforts focus on checking safety properties of firmware and are not applicable to our scenario where we are verifying k-safety properties. Subramanyan et al. [35] use symbolic simulation of product programs to check information flow assertions. These assertions encode observational determinism which is not expressive enough to capture our properties of interest.

VII. Conclusion

This paper introduced a methodology for verification of end-to-end security of authenticated firmware loaders. Authenticated loaders are an important class of programs which are part of the hardware root of trust in almost all modern systems-on-chip (SoC) devices. Our methodology introduced a system and adversary model along with an end-to-end security property that captures security of this class of programs. We presented a decomposition of the security property into two simpler 2-safety properties that in combination with a novel abstraction enabled more scalable verification of the end-to-end security property. Experiments demonstrated the initial feasibility of our approach. Our work paves the way for the construction of a fully-verified authenticated bootloader.

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] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[3] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO '05*, LNCS 4111, pages 364–387, 2005.

[4] 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.

[5] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer. Symbolic Execution for BIOS Security. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, 2015.

[6] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131, 1988. doi: 10.1016/0304-3975(88)90098-9.

[7] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The NUXMV Symbolic Model Checker. In *Proceedings of the International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[8] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. CRETE: A Versatile Binary-Level Concolic Testing Framework. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 281–298, Cham, 2018. Springer International Publishing.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2011.

[10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, September 2010. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1891823.1891830.

[11] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R Tuttle. Model checking boot code from aws data centers. In *International Conference on Computer Aided Verification*, pages 467–486. Springer, 2018.

[12] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium*, pages 857–874. USENIX Association, August 2016.

[13] Andrew Cottrell, Jithendra Bethur, Timothy Markey, M Srikant, and Lakshmanan Srinivasan. Secure firmware update, June 29 2006. US Patent App. 11/026,813.

[14] Ang Cui, Michael Costello, and Salvatore J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[15] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, 2013.

[16] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[17] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[18] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22Nd Annual Symposium on Foundations of Computer Science*, SFCS '81, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society. doi: 10.1109/SFCS.1981.32.

[19] 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. doi: 10.1007/978-3-319-21690-4_3. URL https://doi.org/10.1007/978-3-319-21690-4_3.

[20] 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. doi: 10.1109/SP.1982.10014. URL http://dx.doi.org/10.1109/SP.1982.10014.

[21] James Hendricks and Leendert van Doorn. Secure Bootstrap is Not Enough: Shoring Up the Trusted Computing Base. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM. doi: 10.1145/1133572.1133600. URL http://doi.acm.org/10.1145/1133572.1133600.

[22] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M. Fung, and Sharad Malik. Formal security verification of concurrent firmware in socs using instruction-level abstraction for hardware. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 91:1–91:6, 2018. doi: 10.1145/3195970.3196055.

[23] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur

Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4.

[24] 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.

[25] Ilia A. Lebedev, Kyle Hogan, and Srinivas Devadas. Invited Paper: Secure Boot and Remote Attestation in the Sanctum Processor. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 46–60, 2018. doi: 10.1109/CSF.2018.00011.

[26] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[27] John Mclean. Proving Noninterference and Functional Correctness Using Traces. *Journal of Computer Security*, 1:37–58, 1992.

[28] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-based attack of rsa authentication. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 855–860, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2.

[29] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*, pages 114–127, 1995. doi: 10.1109/SECPRI.1995.398927. URL http://dx.doi.org/10.1109/SECPRI.1995.398927.

[30] 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. doi: 10.1007/3-540-11494-7_23. URL http://dx.doi.org/10.1007/3-540-11494-7_23.

[31] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. Nguyen, D. Stoffel, and W. Kunz. A New Formal Verification Approach for Hardware-dependent Embedded System Software. *IPSJ Transactions on System LSI Design Methodology*, 6:135–145, 2013.

[32] 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.

[33] Marcelo Sousa and Isil Dillig. Cartesian Hoare Logic for verifying k-safety properties. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, 2016. ISBN 978-1-4503-4261-2.

[34] 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.

[35] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung. Verifying Information Flow Properties of Firmware using Symbolic Execution. In *Proceedings of Conference on Design Automation and Test in Europe*, 2016.

[36] 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.

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

[38] Richard Wilkins and Brian Richardson. UEFI Secure Boot in Modern Computer Security Solutions. In *UEFI Forum*, 2013.

[39] Weikun Yang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Lazy Self-Composition for Security Verification. In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, July 14-17, 2018, Proceedings*, 2018.

[40] 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.

[41] Vincent Zimmer and Michael Rothman. Method for performing a trusted firmware/BIOS update, January 27 2005. US Patent App. 10/607,367.