PA-Boot: A Formally Verified Authentication Protocol for Multiprocessor Secure Boot Under Hardware Supply-Chain Attacks

Zhuoruo Zhang, Rui Chang, Mingshuai Chen, Wenbo Shen, *Member, IEEE*, Chenyang Yu, He Huang, Qinming Dai, and Yongwang Zhao

***Abstract*—Hardware supply-chain attacks are raising significant security threats to the boot process of multiprocessor systems. In this paper, we investigate critical stages of the multiprocessor system boot process and identify a new, prevalent hardware supply-chain attack surface that can bypass secure boot due to the absence of processor-authentication mechanisms. To defend against such attacks, in this paper, we present PA-Boot, the first formally verified processor-authentication protocol for secure boot in multiprocessor systems. PA-Boot is proved functionally correct and is guaranteed to detect multiple adversarial behaviors, such as processor replacements and man-in-the-middle attacks. The fine-grained formalization of PA-Boot and its fully mechanized security proofs are carried out in the Isabelle/HOL theorem prover with 348 lemmas/theorems and ~7,100 LoC. We further implement in C an instance of PA-Boot. Experiments on the proof-of-concept implementation indicate that PA-Boot can effectively identify boot-process attacks with a minor overhead (4.98% on Linux boot process) and thereby improve the security of multiprocessor systems.**

***Index Terms*—Formal verification, theorem proving, secure boot, authentication protocol.**

I. Introduction

Attacks during the boot process are notoriously hard to detect because at this early stage of a device's lifecycle, traditional countermeasures like firewalls and anti-viruses are not yet in place [1]. A widely adopted defence against boot attacks is known as *secure boot* [2], [3], which enforces every boot stage to authenticate the subsequent stage such that only the firmware signed by an authorized entity (i.e., the device manufacturer) can be loaded and thereby establishes a *chain-of-trust* in the entire boot process. During this process, the processors of a device serve as the *root-of-trust* (RoT) to bootstrap the trust chain [3]. The authenticity of processors is thus of vital importance to system security.

Received 21 April 2025; revised 27 October 2025; accepted 28 November 2025. Date of publication 11 December 2025; date of current version 2 January 2026. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4501903 and in part by the NSFC under Grant 62572427. The associate editor coordinating the review of this article and approving it for publication was Dr. Yang Zhang. *(Corresponding author: Rui Chang.)*

Zhuoruo Zhang, Rui Chang, Mingshuai Chen, Wenbo Shen, Chenyang Yu, He Huang, and Qinming Dai are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: zhangzhuoruo@zju.edu.cn; crix1021@zju.edu.cn; m.chen@zju.edu.cn; shenwenbo@zju.edu.cn; 22151279@zju.edu.cn; huanghe3776707@zju.edu.cn; qinm_dai@zju.edu.cn).

Yongwang Zhao is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China (e-mail: zhaoyw@zju.edu.cn).

Digital Object Identifier 10.1109/TIFS.2025.3642638

However, the globalized and increasingly complicated hardware supply chains are threatening the trustworthiness of processors – the RoT in secure boot – therefore exposing various modern devices to inevitable *supply-chain attacks* [4], [5], [6], [7], [8], [9], [10], [11]. Specifically, many original equipment manufacturers (OEMs) nowadays outsource their hardware and/or firmware development to third-party suppliers without full inspection into their cybersecurity hygiene [12], [13], where the devices can be intercepted and implanted with compromised components during multiple hands of trade. Such supply-chain attacks raise significant security threats and thereby an urgent request in identifying device vulnerabilities [14], particularly, in the boot process [15], [16].

We focus on new hardware supply-chain attacks that can bypass secure boot of *multiprocessor systems*. A multiprocessor system includes a *bootstrap processor* (BSP) responsible for initializing and booting the operating system and multiple *application processors* (APs) activated after the operating system is up.[1] As instances of hardware supply-chain attacks, an attacker can intercept a customer's multiprocessor device and either (i) replace an AP with a compromised one, e.g., an AP with a pre-installed bootkit; or (ii) implant an extra chip sabotaging the inter-processor communications (*man-in-the-middle attacks* [17]). Such supply-chain attacks can give attackers control early in the boot process, allowing them to load a malicious bootloader or OS kernel. This enables high-privilege arbitrary code execution and evades detection by conventional security measures that rely on the OS. For instance, see the proof-of-concept in Bloomberg's "Big Hack" [18], [19].

Existing research efforts focus on firmware integrity and provide no countermeasures against hardware supply-chain attacks in multiprocessor secure boot. Specifically, both the authenticity of APs and the inter-processor communications are conventionally *trusted by default* universally across all modern multiprocessor systems. In fact, defending against this new hardware attack surface is challenging: It is difficult to examine all steps through the global supply chain from manufacturers to customers; moreover, identifying malicious components via hardware tampering detection techniques, e.g., circuit-based sensors and X-ray imaging, requires expertise and is time consuming [20], [21]. Some work uses run-time monitors to record external behaviors of CPU chips

[1]The same convention applies to *symmetric* multiprocessing (SMP): Whereas all the processors in an SMP system are considered functionally identical, they are distinguished as two types in the boot process.

(i.e., I/O, and memory read/write) and verifies chip integrity [22]. However, specialized hardware components are required to extend the system. It is thus desirable to equip the existing multiprocessor secure boot process with a mechanism for authenticating APs and securing communications without requiring custom hardware changes.

In this paper, we present a processor-authentication proto col, called `PA-Boot`, to assure both the *authenticity of APs* and the *confidentiality of inter-processor communications* in the early stage of secure boot process for multiprocessor systems. `PA-Boot` is capable of detecting multiple adversarial behaviors including AP replacements and man-in-the-middle attacks. The boot process is aborted if any of the adversarial behavior is detected to prevent the attacker from taking control of the system. The security and functional correctness of `PA-Boot` is verified based on deductive reasoning techniques: The formalization of `PA-Boot` and its fully mechanized security proofs (in terms of the AP authenticity, certificate integrity, etc.) are conducted in the (interactive) theorem prover Isabelle/HOL [23]. This fine-grained formalization of `PA-Boot` in Isabelle/HOL succinctly captures its key components, the system behaviors, and a full range of adversarial capabilities against the protocol. To the best of our knowledge, *`PA-Boot` is the first formally verified processor-authentication protocol for secure boot in multiprocessor systems*. We further implement in C an instance of `PA-Boot` called `CPA-Boot`. Experiments simulated via ARM Fixed Virtual Platforms (FVP) suggest that `CPA-Boot` can effectively identify multiple boot-process attacks – by either manipulating the APs or tricking the AP-authentication mechanism – with a minor overhead and thus essentially improve the security of multiprocessor systems. We have open-sourced our code, available at https://zenodo.org/records/14513558.

A. Contributions

The main contributions are summarized as:

- **Design of `PA-Boot`**: We inspect critical steps in multiprocessor secure boot and identify a new, prevalent attack surface – exhibiting hardware supply-chain attacks – that may bypass secure boot due to the lack of AP authentication. To defend against such attacks, we design the first processor-authentication protocol `PA-Boot` for secure boot in multiprocessor systems that is amenable to formal verification via theorem proving.
- **Verification of `PA-Boot`**: We formalize `PA-Boot` based on multi-level abstraction-refinement in Isabelle/HOL via 91 locale/definitions. Meanwhile, we formalize multiple properties on both functional correctness and security (e.g., authenticity and integrity). The proof that `PA-Boot` satisfies these properties is then fully mechanized in the form of 348 lemmas/theorems and ~7,100 LoC
- **Implementation and evaluation**: We implement `CPA-Boot` in compliance with the formalization of `PA-Boot`. To build confidence in their consistency, we introduce a validation framework that extracts executable code from our Isabelle/HOL model and validates it against `CPA-Boot`. We integrate `CPA-Boot` in a real-world bootloader and show on FVP that `CPA-Boot` can effectively identify multiple adversarial behaviors like replacing APs and man-in-the-middle attacks, with a minor overhead (4.98% on Linux boot).

II. Background

This section recapitulates secure boot in multiprocessor systems and hardware supply-chain attacks.

A. Normal Flow and Limitation of Secure Boot

Secure boot [2] establishes a chain-of-trust (CoT) to ensure firmware integrity. It serves as a default (or often mandatory) feature in modern secure devices like laptops, desktops, smartphones, and IoT devices [3], [24]. The process begins with the immutable bootloader in read-only storage, considered the hardware root-of-trust (RoT). As the system's security foundation, the RoT must be immutable and tamper-resistant. It initializes the hardware, locates the next boot image in Non-Volatile Memory (NVM), and loads it into memory. The RoT then verifies the image's integrity and authenticity using the Root of Trust Public Key (RoTPK), an OEM key stored immutably as a hash or in full. The image, signed with the Root of Trust Private Key, ensures it comes from a trusted source. Execution proceeds only if the image is verified. Each subsequent boot stage follows this process, verifying the next image before transferring control. This chain-of-trust ensures that only verified components progress through the boot sequence, ultimately loading the kernel image securely. Secure boot implementation varies across platforms [25], with room for platform-specific operations. For enhanced security and efficiency, many systems integrate cryptographic engines for operations like signature verification and hash calculations.

Nevertheless, the secure boot's chain-of-trust (CoT) can be undermined by a malicious RoT. The integrity of the system depends on the assumption that the code in the first boot stage (acting as the RoT) is trustworthy or at worst buggy but non-malicious. Early systems stored this code on a write-protected flash memory (e.g., BIOS in PCs), but BIOS ROM is poorly protected and easily writable, leading to attacks exploiting the lack of access control during BIOS reflashing [26]. In response, modern systems use hardwired bootROM, an immutable hardware component integrated into the CPU [27], [28]. Most research assumes the CPU chip, as part of the trusted computing base (TCB), is tamper-resistant and trustworthy, while off-chip components are vulnerable [27], [28], [29]. However, we show that the hardware supply chain poses a risk, as malicious actors with access to the hardware supply chain can replace genuine CPU chips, compromising the RoT and gaining control over the system.

B. Secure Boot in Multiprocessor Systems

Fig. 1 illustrates the key components of a multiprocessor secure boot system. Without loss of generality, we assume throughout the paper that the motherboard of a multiprocessor device is equipped with two processors, each in physically separate chip sockets: one as the BSP and the other as the AP. Each processor has a unique on-chip private key as its

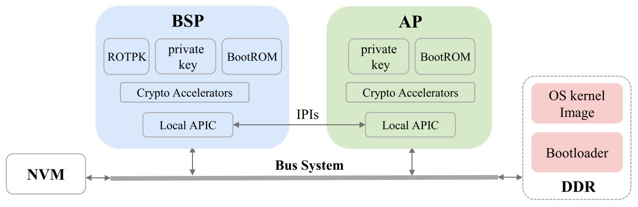


Fig. 1. Key components in multiprocessor secure boot.

hardware identifier. The processors are connected via a shared bus to a mutable NVM and shared memory. Each processor includes a cryptographic extension module to accelerate operations. Except the shared bus, the BSP and AP communicate via a dedicated channel for inter-processor interrupts (IPIs). The motherboard is soldered a baseboard management controller (BMC), which is connected to the BSP and handles initialization and monitoring tasks. Note that the roles of BSP and AP are determined by their socket positions on motherboard, allowing the BMC to identify them accordingly.

The multiprocessor secure boot begins by executing the immutable code in the processors' bootROM, considered the hardware root-of-trust (RoT), which initializes the hardware. The AP enters a suspended state while the BSP proceeds with the following steps: (i) Locating the bootloader image in the NVM, (ii) Loading it into shared memory, (iii) Verifying its authenticity and integrity using cryptographic accelerators, detecting any tampering with the certified images in NVM, and (iv) Executing the image to load the next firmware stage. This authentication process propagates through all subsequent layers until the OS kernel is loaded. The BSP then broadcasts an IPI with the address for the next execution via its built-in *local advanced programmable interrupt controllers* (LAPIC). The AP, which monitors the IPI channel, is activated and runs the OS kernel with the same privileges as the BSP. Variants of the typical multiprocessor architecture in Fig. 1 may include, e.g., different inter-processor communication methods.

However, in the context of secure boot, as identified in our threat model below, an attacker with access to the hardware supply chain may bypass secure boot by exploiting either a compromised AP or an additional chip sabotaging the IPI channel and thus taking control of the target system.

III. Overview of Our Approach

This section identifies our threat model exhibiting the new attack surface in multiprocessor secure boot as hardware supply-chain attacks, and outlines our defense approach.

A. Threat Model

In our threat model, we trust the CPU chip manufacturer (e.g., Qualcomm), PCB board manufacturer, and device OEM (e.g., Apple). In other words, the design and fabrication process of original CPU chips and boards are trusted, since they come from reliable hardware supply chains controlled by the OEM. The OEMs can also take measures for tampering detection, like checking credentials, to ensure they receive genuine components from the vendors. In addition, attacks at the manufacturing steps can't easily target a specific end-product, making attacks highly unlikely.

We assume a realistic and powerful attacker who has physical access to the target multiprocessor device in the post-manufacturing hardware supply chain, i.e., in-transit from the OEM to the end-user or in the field. Therefore, the attacker can intercept the device and then replace the pluggable CPU chips on board and modify the PCB, such as by implanting an interposer chip on the inter-processor channel wires, which are susceptible to probing attacks [30]. These capabilities enable the attacker to launch the following two *attack vectors* (see in Fig. 2) under no awareness of the BSP:

a) *AP-replacement attack*: The attacker replaces the original AP with a malicious one that has, e.g., a factory-installed bootkit in its bootROM. Such a malicious AP can obtain via the shared memory bus secret data or high-privilege resources. It can also manipulate memory content and overwrite (parts of) the bootloader to be executed to take control of the system already at the boot stage.

b) *Man-in-the-middle attack*: The attacker implants an extra interposer chip snooping on the IPI channel for inter-processor communications. The chip can sabotage traffic along the IPI channel by, e.g., substituting the memory entry pointer encoded in the activation IPI from BSP to AP, leading to control-flow hijacking and arbitrary code executions; moreover, it can sniff the IPI channel to capture secret data or interfere with runtime inter-processor communications concerning, e.g., remote TLB shootdowns.

B. Assumptions

We make the following assumptions: (i) The BSP is trusted. Its authenticity is ensured by the onboard baseboard management controller (BMC) before our protocol executes, and the trustworthiness of the BMC itself can be validated through mechanisms like measured boot [31]. (ii) The on-chip bootROM is designed to be immutable [27], [32] and cannot be modified by an attacker, as modifying it after manufacture is exceptionally difficult [33] (see Sect. VIII for potential extension of our approach against very powerful attackers able to tampering with the bootROM). While the bootROM inside the AP remains immutable after manufacturing, as discussed in (a), attackers can more feasibly create a malicious AP by pre-installing malware during chip production. Later, in the post-manufacturing supply chain, they can replace the genuine AP plugged on device with the malicious one. (iii) Even if an attacker introduces a malicious AP, they cannot extract or replicate the genuine AP's on-chip private key. While advanced physical attacks using techniques like scanning electron microscopy (SEM) could theoretically target on-chip data, they are impractical due to their high demand for knowledge, time and equipment [34]. Modern hardware protections mitigate these risks by binding the key to hardware identities [1] or intrinsic physical properties rather than storing it statically. For example, techniques like Physically Unclonable Functions (PUFs) [35] regenerate keys at runtime using unique physical

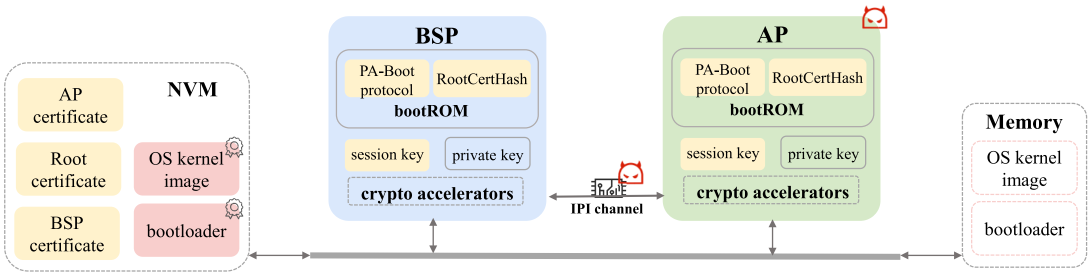


Fig. 2. Threat model of PA-Boot. marks potential vulnerabilities to adversarial behaviors (a) and (b) in multiprocessor secure boot. Components of PA-Boot are colored in yellow.

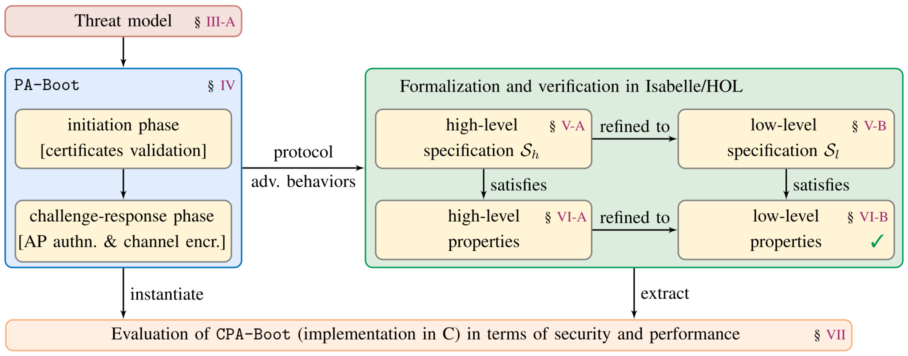

Threat model § III-A
PA-Boot § IV
initiation phase [certificates validation]
challenge-response phase [AP authn. & channel encr.]
protocol
adv. behaviors
Formalization and verification in Isabelle/HOL
high-level specification $\mathcal{S}_h$ § V-A
refined to
low-level specification $\mathcal{S}_l$ § V-B
satisfies
satisfies
high-level properties § VI-A
refined to
low-level properties § VI-B ✓
instantiate
extract
Evaluation of CPA-Boot (implementation in C) in terms of security and performance § VII

Fig. 3. The workflow of our approach. ✓ indicates verified properties encoding security and functional correctness.

characteristics. (iv) Cryptographic accelerators, like those used for elliptic-curve cryptography (ECC) and SHA-256 integrity checks, are trusted.

Note that hardware replacement/modification is a strong attack model, the adversary can physically tamper with all components on the device. Uniquely in our case, we only focus on the two new novel attack vectors. For instance, we do not consider physical attacks such as cold boot attacks or snooping attacks of the shared memory bus, given that such attacks have been studied and mitigated as in prior studies [36], [37]. Physical attacks other than internal hardware replacement, such as side-channel attacks are out of our scope.

C. *Workflow of Our Approach*

Fig. 3 sketches the overall workflow of our approach. To defend against the hardware supply-chain attacks (a) and (b) identified in our threat model, we develop the processor-authentication protocol PA-Boot, which orchestrates the secure boot process via an *initiation phase* to validate the processor certificates and a *challenge-response phase* to authenticate the AP identity and to secure the inter-processor communication channel. We then formalize it and the possible (adversarial and normal) behaviors thereof in Isabelle/HOL as a *high-level specification* $\mathcal{S}_h$ and a set of *high-level properties* (encoding security and functional correctness). These high-level ingredients – capturing the core components of PA-Boot – are further *refined* into their low-level counterparts, i.e., the *low-level specification* $\mathcal{S}_l$ and a set of *low-level properties*. We then conduct a fully mechanized proof via theorem proving that $\mathcal{S}_l$ satisfies the low-level properties. Finally, we derive an implementation of PA-Boot from the formalized model as CPA-Boot based on a code-to-spec review [38]. We show that CPA-Boot effectively identifies various boot-process attacks with a minor overhead and thereby improves the security of multiprocessor systems.

Separating a protocol's formalization into high- and low-level specifications is a common strategy for abstraction and refinement (e.g., [38], [39]). The high-level specification $\mathcal{S}_h$ provides a simple system behavior description, while the low-level specification $\mathcal{S}_l$, closer to implementation, details all possible protocol executions and attacker interactions. We verify security properties and functional correctness using $\mathcal{S}_l$, as it captures specific protocol configurations and execution traces. $\mathcal{S}_h$ helps to gain confidence in the correctness of the system's specification and its alignment with expectations. Its abstract, declarative nature reduces specification errors and facilitates their discovery. The refinement proof between $\mathcal{S}_l$ and $\mathcal{S}_h$ further extends that confidence to $\mathcal{S}_l$.

IV. Design of PA-Boot

This section details the design of our processor authentication protocol PA-Boot. It augments multiprocessor secure boot with several key components as depicted in Fig. 2. PA-Boot steers the secure boot process in a certificate-based, two-phase manner: The processor certificates are validated in the *initiation phase* and thereafter, in the *challenge-response phase*, the AP identity is authenticated and the inter-processor communication channel is encrypted. We first explain the necessary operations to set up the stage for running PA-Boot and then elucidate key message flows in the abovementioned two phases. Frequently used notations are collected in Table I.

Our protocol requires each processor to request its certificate from the certificate authority (CA) at the OEMs and store the

TABLE I
FREQUENTLY USED NOTATIONS

Notation	Intuitive meaning
$Cert_{root}/Cert_{BSP}/Cert_{AP}$	certificate of root/BSP/AP, resp.
$RootCertHash$	hash value of $Cert_{root}$
$\langle PubK_{BSP}, PrivK_{BSP}\rangle$	public-private key pair of BSP
$\langle PubK_{AP}, PrivK_{AP}\rangle$	public-private key pair of AP
$\mathcal{N}_{AP}/\mathcal{N}_{BSP}$	nonce generated by AP/BSP, resp.
$\langle EPubK_{BSP}, EPrivK_{BSP}\rangle$	ephemeral public-private key pair generated by BSP for $\mathcal{K}_s$
$\langle EPubK_{AP}, EPrivK_{AP}\rangle$	ephemeral public-private key pair generated by AP for $\mathcal{K}_s$
$\mathcal{K}_s$	shared session key for the secure channel eventually established between BSP and AP

signed certificate once issued. More concretely, the OEM first generates authorized certificates $Cert_{BSP}$ and $Cert_{AP}$ respectively by signing $PubK_{BSP}$ and $PubK_{AP}$ with its own private key (these certificates thus can be validated by the OEM's public key in $Cert_{root}$), and then stores all necessary certificates $\langle Cert_{root}, Cert_{BSP}, Cert_{AP}\rangle$ in the NVM. Meanwhile, the hash value of the root certificate $Cert_{root}$ is stored in the bootROM of the processors. Recall that any attacker that can compromise the BSP's bootROM is beyond the scope of this paper.

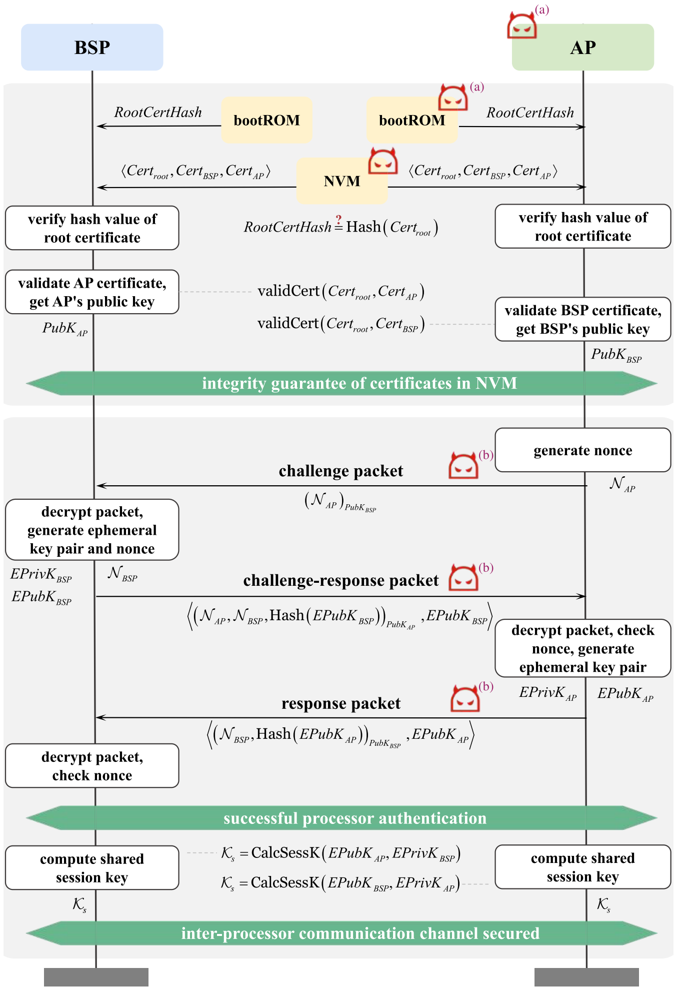
Remark 1: PA-Boot is *not* tailored to dual-processor devices, rather, it applies to the setting of *multiple* APs where the BSP can authenticate these APs in sequence.◁

A. The Initiation Phase

The two processors BSP and AP behave *symmetrically* in the initiation phase: As depicted in (the upper part of) Fig. 4, each processor first reads the precomputed hash value $RootCertHash$ of $Cert_{root}$ from its bootROM as well as the chain of certificates $\langle Cert_{root}, Cert_{BSP}, Cert_{AP}\rangle$ from the NVM. It then checks the validity and integrity of $Cert_{root}$, namely, checking whether $RootCertHash = Hash(Cert_{root})$. If this is indeed the case, then $Cert_{root}$ is used to validate the certificate of the other processor by applying the function $validCert(Cert_{root}, \cdot)$, which further reveals the public key of the other processor. The non-volatile memory (NVM) is easily modifiable, giving the attacker the ability to alter its contents. The attacker could replace legitimate certificates with those associated with a malicious AP, thereby attempting to deceive PA-Boot. However, since the forged certificate is not signed by a trusted certificate authority (CA), the system can detect its invalidity during the certificate validation process.

B. The Challenge-Response Phase

As depicted in (the lower part of) Fig. 4, once confirming the validity of $Cert_{BSP}$ in the initiation phase, the AP sends – via the inter-processor communication channel – a *challenge packet* $(\mathcal{N}_{AP})_{PubK_{BSP}}$, that is, a randomly generated nonce $\mathcal{N}_{AP}$ encrypted with $PubK_{BSP}$. Note that a typical nonce is a 32-byte random number which is practically infeasible to guess. Upon



BSP
AP
(a)
RootCertHash
bootROM
bootROM
RootCertHash
$\langle Cert_{root}, Cert_{BSP}, Cert_{AP}\rangle$
NVM
$\langle Cert_{root}, Cert_{BSP}, Cert_{AP}\rangle$
verify hash value of root certificate
$RootCertHash \overset{?}{=} Hash(Cert_{root})$
verify hash value of root certificate
validate AP certificate, get AP's public key
$validCert(Cert_{root}, Cert_{AP})$
$validCert(Cert_{root}, Cert_{BSP})$
validate BSP certificate, get BSP's public key
$PubK_{AP}$
$PubK_{BSP}$
integrity guarantee of certificates in NVM
generate nonce
(b)
challenge packet
$\mathcal{N}_{AP}$
$(\mathcal{N}_{AP})_{PubK_{BSP}}$
decrypt packet, generate ephemeral key pair and nonce
$EPrivK_{BSP}$
$EPubK_{BSP}$
$\mathcal{N}_{BSP}$
challenge-response packet
$\langle(\mathcal{N}_{AP}, \mathcal{N}_{BSP}, Hash(EPubK_{BSP}))_{PubK_{AP}}, EPubK_{BSP}\rangle$
decrypt packet, check nonce, generate ephemeral key pair
$EPrivK_{AP}$
$EPubK_{AP}$
response packet
$\langle(\mathcal{N}_{BSP}, Hash(EPubK_{AP}))_{PubK_{BSP}}, EPubK_{AP}\rangle$
decrypt packet, check nonce
successful processor authentication
compute shared session key
$\mathcal{K}_s = CalcSessK(EPubK_{AP}, EPrivK_{BSP})$
$\mathcal{K}_s = CalcSessK(EPubK_{BSP}, EPrivK_{AP})$
compute shared session key
$\mathcal{K}_s$
$\mathcal{K}_s$
inter-processor communication channel secured

Fig. 4. Key message flows in PA-Boot. marks potential vulnerabilities to adversarial behaviors (a) and (b).

receiving this packet, the BSP decrypts the packet via $PrivK_{BSP}$ to get $\mathcal{N}_{AP}$ and then sends a *challenge-response packet*

$$\langle(\mathcal{N}_{AP}, \mathcal{N}_{BSP}, Hash(EPubK_{BSP}))_{PubK_{AP}}, EPubK_{BSP}\rangle$$

where $\mathcal{N}_{AP}$ is the challenge nonce generated by AP, $\mathcal{N}_{BSP}$ is the response nonce generated by BSP, and $EPubK_{BSP}$ is the ephemeral public key generated by BSP (for computing the shared session key $\mathcal{K}_s$ later); this challenge-response packet is encrypted by $PubK_{AP}$ except for the $EPubK_{BSP}$ part. After receiving the challenge-response packet, the AP first decrypts the packet and checks its integrity, i.e., checking whether the received $\mathcal{N}_{AP}$ is identical to the previously generated one *and* whether the decoded $Hash(EPubK_{BSP})$ is identical to the hash value of the received $EPubK_{BSP}$. If this is indeed the case, the AP stores $EPubK_{BSP}$ and generates its ephemeral key pair $\langle EPubK_{AP}, EPrivK_{AP}\rangle$. Then, analogously, the AP responds to the BSP with a *response packet*

$$\langle(\mathcal{N}_{BSP}, Hash(EPubK_{AP}))_{PubK_{BSP}}, EPubK_{AP}\rangle \ .$$

The BSP then decrypts the packet and checks its integrity by similar means as mentioned above.

These message flows by now in the challenge-response phase are able to detect the aforementioned attack vector (a), i.e., AP replacements, and (b) man-in-the middle attacks attempting to sabotage the communication packets. To protect the communication channel from future interference, `PA-Boot` further produces a *shared session key* $\mathcal{K}_s$ such that subsequent inter-processor communications can be secured using $\mathcal{K}_s$. This concludes the entire challenge-response phase of `PA-Boot`. We remark that the (symmetric) shared session key $\mathcal{K}_s$ can be calculated by BSP (resp. AP) based on $\mathit{EPubK_{AP}}$ and $\mathit{EPrivK_{BSP}}$ (resp. $\mathit{EPubK_{BSP}}$ and $\mathit{EPrivK_{AP}}$) using the Diffie-Hellman (DH) key exchange algorithm [40].

V. Formalization in Isabelle/HOL

This section presents the formalization of `PA-Boot` in the theorem prover Isabelle/HOL [23]. The formal model consists of a *high-level specification* $\mathcal{S}_h$ and a refined *low-level specification* $\mathcal{S}_l$. $\mathcal{S}_h$ captures core components of `PA-Boot` and gives the simplest description of the system behavior, whereas $\mathcal{S}_l$ is a *symbolic model* closer to the implementation layer, encoding a more fine-grained characterization of all possible executions of the system. We opt for deductive verification as implemented in Isabelle/HOL due to its scalability and inherent support of abstraction refinement [41] and code generation [42].

Adversary Model: For the adversarial behavior (a) identified in Sect. III-A, we consider the possibility that the agent AP is compromised, allowing the compromised agent to manipulate its long-term keys. For the adversarial behavior (b) identified in Sect. III-A, we explicitly model a classical Dolev-Yao-style adversary [43] who has full control over the insecure BSP-AP communication channel. However, the adversary is limited by the constraints of the cryptographic methods used: he cannot forge signatures or decrypt messages without knowing the key (the black box cryptography assumption).

A. High-Level Specification

The high-level specification $\mathcal{S}_h$ of `PA-Boot` is encoded as a finite-state *acyclic labelled transition system* representing the protocol executions under certain security contexts:

Definition 1: [High-Level Specification of PA-Boot] The *high-level specification of* `PA-Boot` is a quintuple

$$\mathcal{S}_h \triangleq \langle S, s_0, \bar{s}, \Lambda, \Delta \rangle \ , \quad \text{where}$$

- S is a finite set of *states* encoding the protocol configurations,
- $s_0 \in S$ is the *initial state*,
- $\bar{s} \in S$ is the *ideal state* signifying attack-free authentication,
- Λ is a finite set of *event labels* representing actions of the processors and the attacker, and
- $\Delta \subseteq S \times \Lambda \times S$ is a finite set of *labelled transitions*.

A labelled transition $\delta = (s, \alpha, s') \in \Delta$, denoted by $s \xrightarrow{\alpha} s'$, yields a jump from the *source state* s to the *target state* s' on the occurrence of event α. We consider deterministic transitions, i.e., if $s \xrightarrow{\alpha} s', s \xrightarrow{\alpha} s''$ are both transitions in Δ, then $s' = s''$. The set of *terminal states* is defined as $S^{\downarrow} \triangleq \{s \in S \mid \forall \alpha \in \Lambda.\ \nexists s' \in S : s \neq s' \wedge (s, \alpha, s') \in \Delta\}$, i.e, a state s is *terminal* iff s has no successors other than itself. Note that the ideal state $\bar{s}$ is necessarily a terminal state in $S^{\downarrow}$. A *run* π of $\mathcal{S}_h$, denoted by $s_0 \overset{A}{\hookrightarrow} s_n$ with $A = \alpha_1\alpha_2\cdots\alpha_n \in \Lambda^*$, is a finite sequence $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} s_n$ with $s_n \in S^{\downarrow}$, $s_{i-1} \xrightarrow{\alpha_i} s_i \in \Delta$ for any $1 \leq i \leq n$. We denote by Π the set of all possible runs of $\mathcal{S}_h$. Given a run $\pi = s_0 \overset{A}{\hookrightarrow} s_n \in \Pi$, $\mathit{Tail}(\pi)$ denotes the tail state s_n of π.

1) Security Contexts and Indicator Functions: Every state $s \in S$ of $\mathcal{S}_h$ encodes, amongst others, the current *security context* $c \in C$ consisting of security-related system configurations of the underlying *security assets*, including the two processors, the NVM, and the inter-processor communication channel. We apply the function $\Gamma\colon S \to C$ to extract the security context pertaining to a state; for simplicity, we write Γ_s as shorthand for $\Gamma(s)$. Let $\mathbb{B} \triangleq \{\mathit{true}, \mathit{false}\}$. We employ two indicator functions to witness the presence/absence of security threats: The (partial) *benignity function* $\mathcal{B}\colon C \rightharpoonup \mathbb{B}$ signifies whether a security context c is benign (in the state that c is associated with), i.e., whether the AP and the certificates in the NVM are genuine; the *free-of-attack function* $\mathcal{M}\colon \Pi \to \mathbb{B}$ determines whether a man-in-the-middle attack has *not* occurred along a run of $\mathcal{S}_h$, i.e., during one possible execution of the protocol.

$\mathcal{S}_h$ together with, e.g., its indicator functions are formalized as a *locale module* [41] in Isabelle/HOL – an emerging mechanism for abstract reasoning – consisting of abstract types and primitive operations that can be interpreted in different contexts. We omit the detailed locale formulation here.

B. Low-Level Specification

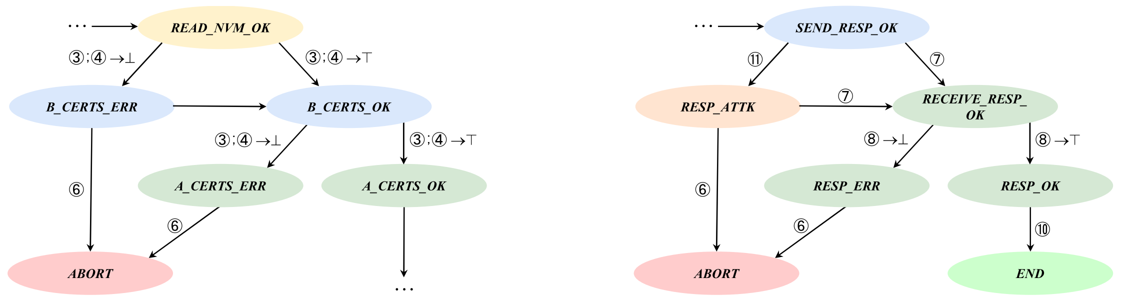
Next, we refine $\mathcal{S}_h$ of `PA-Boot` to its low-level counterpart $\mathcal{S}_l$ by instantiating the state space as concrete protocol configurations and the labelled transitions as event-triggered actions of the processors or the attacker. The fact that $\mathcal{S}_l$ *refines* $\mathcal{S}_h$, written as $\mathcal{S}_h \sqsubseteq \mathcal{S}_l$, is proved in Isabelle/HOL by interpreting locales as parametric theory modules [41].

A state in $\mathcal{S}_l$ is encoded as a *record* construct in Isabelle/HOL collecting all fields related to the protocol configuration:

$$\begin{aligned}\textbf{record}\ \mathit{State}\ =\ \{\ & \mathit{bsp}\ :: \mathit{Processor},\quad \mathit{ap}\ :: \mathit{Processor},\\ & \mathit{env}\ :: \mathit{Envir},\quad \mathit{status}\ :: \mathit{Status}\ \}\ .\end{aligned}$$

Specifically, *Processor* encapsulates all the BSP/AP-related ingredients in `PA-Boot` (cf. Fig. 4); *Envir* encodes the *environment* of the processors, i.e., the NVM storing certificates and the inter-processor channel carrying communication packets; *Status* signifies the current status of `PA-Boot`, which can be *INIT* (initialization), *OK* (normal execution), *ERR* (failure in certificate validation or packet parsing), *ATTK* (presence of man-in-the middle attacks), *END* (normal termination), and *ABORT* (abnormal termination). In particular, a man-in-the-middle attack will be recognized by `PA-Boot` when parsing the attacked communication packet and thus leads to an *ERR* state; moreover, once a run visits an *ERR* state, it raises an error-specific alarm and terminates in the (unique) *ABORT* state. In fact, *END* and *ABORT* represent terminal states $S^{\downarrow}$ as defined in $\mathcal{S}_h$, where *END* particularly marks the ideal state $\bar{s}$. Note that *OK*, *ERR*, and *ATTK* are refined to more fine-grained status types using prefixing, see examples in Fig. 5.

As listed in Table II, the set Λ of event labels in $\mathcal{S}_h$ is instantiated in $\mathcal{S}_l$ as concrete actions of the protocol participants. The



(i) validating NVM certificates (ii) attack on the resp. packet

Fig. 5. Snippets of the state machine $\mathcal{S}_l$ of PA-Boot. A state s is identified by its status and colored in light green (normal termination), red (abnormal termination), or in blue, dark green, yellow, or orange if the transition to s is triggered by BSP, AP, both BSP and AP, or the attacker, respectively; The (condensed) transition ⓟ;ⓠ … → ⊤, for $p, q \in \{3, 4, 8\}$ (cf. Table II), is triggered if all the checks in the event sequence ⓟ, ⓠ,...are successful, otherwise ⓟ;ⓠ;...→ ⊥ is triggered.

TABLE II
THE SET Λ OF EVENTS IN $\mathcal{S}_l$

ID	Name	Description of the event
Processor behaviors (executed by BSP or AP)		
①	Read_ROM	read *RootCertHash* in bootROM
②	Read_NVM	read $\langle Cert_{root}, Cert_{BSP}, Cert_{AP}\rangle$ in NVM
③	Verify_RCHash	verify if $RootCertHash = \texttt{Hash}\,(Cert_{root})$
④	Verify_Cert	verify the other processor's cert. via $Cert_{root}$
⑤	Gen_Nonce	generate a fresh nonce
⑥	Send_Packet	send a packet to the channel
⑦	Receive_Packet	receive a packet via the channel
⑧	Parse_Packet	decrypt a packet and check its integrity
⑨	Gen_EpheKey	generate ephemeral public-private key pair
⑩	Gen_SessKey	calculate $\mathcal{K}_s$ to secure the comm. channel
Adversarial behavior (executed by the attacker)		
⑪	Attack	perform a man-in-the-middle attack

set Δ of event-triggered transitions in $\mathcal{S}_l$ is specified based on *State* and Λ. Fig. 5 depicts parts of the state machine $\mathcal{S}_l$, which represent typical scenarios in PA-Boot, e.g., validating NVM certificates (Fig. 5i) and detecting AP replacements or man-in-the-middle attacks on the response packet (Fig. 5ii).

The security-related utility functions Γ, $\mathcal{B}$ and $\mathcal{M}$ as described in the high-level specification are instantiated in the lower-level counterpart as well. For example, for a state s in $\mathcal{S}_l$, $\Gamma_s = \Gamma(s)$ yields the security context associated with s, that is, the components $\langle bsp, ap, env\rangle$ in s; moreover, given a run π of $\mathcal{S}_l$, $\mathcal{M}(\pi)$ determines whether or not a man-in-the-middle attack, i.e., ⑪, is absent along π.

VI. Formal Verification

This section presents the mechanized proof in Isabelle/HOL that PA-Boot is functionally correct and suffices to detect aforementioned adversarial behaviors (a) and (b). The core ingredient to the proof is the formalization of the corresponding properties on functional correctness and security – first at the level of $\mathcal{S}_h$ and then at $\mathcal{S}_l$ in a refined form.

A. High-Level Properties

1) Functional Correctness: We require PA-Boot to be *functionally correct*, in the sense that the benignity of the security context remains *invariant* during any possible execution of PA-Boot:

Property 1: [Functional Correctness w.r.t. $\mathcal{S}_h$]

$$\forall\ \ \pi = s_0 \overset{A}{\hookrightarrow} s_n \in \Pi,\quad i \in [1, n].\ \ \mathcal{B}\left(\Gamma_{s_i}\right) \iff \mathcal{B}\left(\Gamma_{s_0}\right) \qquad (\dagger)$$

Recall that $\mathcal{B}(\Gamma_{s_0}) = true$ iff the AP and the certificates in the NVM are genuine upon the start of PA-Boot, thereby witnessing the absence of adversarial behaviors (a). Provided a reasonable assumption that these two adversarial behaviors can only be conducted during the hardware supply chain, i.e., before the launch of PA-Boot, Property 1 ensures that our protocol per se does not alter the benignity of the security context during all of its possible executions.

2) Security Property: PA-Boot should be able to *secure the boot process*, in the sense that the successful authentication by PA-Boot (indicated by reaching the ideal state $\bar{s}$) guarantees that every boot flow thereafter involves only trusted security assets and is free of man-in-the-middle attacks:

Property 2: [Security w.r.t. $\mathcal{S}_h$]

$$\forall\ \pi \in \Pi.\ Tail\,(\pi) = \bar{s} \iff \mathcal{B}\left(\Gamma_{s_0}\right) \wedge \mathcal{M}\,(\pi) \qquad (\ddagger)$$

Intuitively, Property 2 ensures that a run π of $\mathcal{S}_h$ terminates in the ideal state $\bar{s} \in S^{\downarrow}$ if and only if $\mathcal{M}(\pi) = true$ and $\mathcal{B}(\Gamma_{s_0}) = true$ (so is $\mathcal{B}(\Gamma_s)$ for any s along π, due to Property 1), that is, π is free of man-in-the-middle attacks and both the AP and the certificates in the NVM remain genuine along π. Moreover, the established shared session key $\mathcal{K}_s$ ensures that PA-Boot is able to secure the entire boot process.

B. Low-Level Properties

Next, Properties 1 and 2 are refined w.r.t. $\mathcal{S}_l$ in the form of Isabelle/HOL lemmas. We use . to extract specific fields of a state in $\mathcal{S}_l$, e.g., $s.\,ap.\,private\_key$ denotes $PrivK_{AP}$ in s.

1) Functional Correctness: At the level of $\mathcal{S}_l$, Property 1 boils down naturally to the requirement that the private keys of the two processors, *RootCertHash* stored in their bootROMs, and the certificates stored in the NVM remain unchanged during all possible executions of PA-Boot:

Lemma 1 (Functional Correctness w.r.t. $\mathcal{S}_l$):

$$\forall\ \pi = s_0 \overset{A}{\hookrightarrow} s_n \in \Pi,\ i \\ \in [1, n].\ s_i.\,env.\,nvm = s_0.\,env.\,nvm$$

$$\begin{aligned}&\wedge\ s_i.ap.private\_key = s_0.ap.private\_key\\ &\wedge\ s_i.bsp.private\_key = s_0.bsp.private\_key\\ &\wedge\ s_i.ap.root\_cert\_hash = s_0.ap.root\_cert\_hash\\ &\wedge\ s_i.bsp.root\_cert\_hash\\ &\quad = s_0.bsp.root\_cert\_hash\end{aligned}$$

2) Security Properties: In the presence of adversarial behaviors (a) and (b), we aim to establish – by the end of protocol execution – three types of security goals of the underlying security assets at the level of $\mathcal{S}_l$:

- *Authenticity* (against (a)): The identity of AP is validated.
- *Integrity* (against (a) and (b)): The inter-processor communication packets and the certificates in NVM remain unmodified (even before the execution of PA-Boot).
- *Confidentiality* (against (b) for future executions): The established shared session key $\mathcal{K}_s$ (for encrypting future communication messages) is known only to BSP and AP.

The above-mentioned security goals are encoded as three individual lemmas interpreting different protocol-execution scenarios. Specifically, the protocol under normal execution (cf. Lemma 2) should eventually terminate in the *END* state where the processors are mutually authenticated and $\mathcal{K}_s$ is established; otherwise, if an adversarial behavior that violates either authenticity or certificate integrity is identified – i.e., the AP has been replaced or the certificates in NVM have been tampered with (Lemma 3), or an inter-processor communication packet has been modified by a man-in-the-middle attack (cf. Lemma 4) – then the protocol should raise error-specific alarms by visiting the *ERR* states and eventually terminate in the *ABORT* state.

Lemma 2 (Security w.r.t. $\mathcal{S}_l$ under Normal Executions):

$$\forall\ \pi \in \Pi.\ \mathcal{B}\left(\Gamma_{s_0}\right) \wedge \mathcal{M}(\pi) \implies Tail(\pi).status = END$$

Lemma 2 declares that starting from the initial state s_0 where the associated security context is benign, if there is no man-in-the-middle attack during the execution, PA-Boot must terminate in the *END* state signifying attack-free authentication and the establishment of $\mathcal{K}_s$.

Lemma 3 (Security w.r.t. $\mathcal{S}_l$ against Tampered Configurations):

$$\begin{aligned}&\forall\ \pi = s_0 \overset{A}{\hookrightarrow} s_n \in \Pi.\ \exists\ i \in [1,n].\ \neg\mathcal{B}\left(\Gamma_{s_0}\right) \implies\\ &\left(Tail(\pi).status = ABORT\ \wedge\ s_i.status = \%\_ERR\ \right)\\ &\text{where } \% \text{ matches } A\_CERTS,\ B\_CERTS \text{ or } RESP,\ \text{ see Fig. 5}\end{aligned}$$

Lemma 3 states that in case Γ_{s_0} is compromised (due to adversarial behaviors (a) and attacker's possible modification to necessary certificates in the NVM to impersonate a benign AP), PA-Boot should raise an error-specific alarm by visiting the corresponding *ERR* state until it eventually terminates in the *ABORT* state.

TABLE III
PROOF EFFORTS IN ISABELLE/HOL (~7.1K LOC). (H)SP FOR (HIGH-LEVEL) SECURITY PROPERTIES, FC FOR FUNCTIONAL CORRECTNESS, SP FOR SECURITY PROPERTIES, AND RP FOR REFINEMENT PROOF

Level	Specifications		Proofs		
	#locale/definition	LoC	property	lemma/theorem	LoC
$\mathcal{S}_h$	1	~50	HSP	4	~50
$\mathcal{S}_l$	90	0.7k	FC	28	0.3k
			SP	302	5.9k
			RP	14	0.1k

Lemma 4 (Security w.r.t. $\mathcal{S}_l$ against Man-in-the-Middle Attacks):

$$\begin{aligned}&\forall\ \pi = s_0 \overset{A}{\hookrightarrow} s_n \in \Pi.\ \exists\ i,j \in [1,n]\ s.t.\ i < j.\\ &\neg\mathcal{M}(\pi) \implies \big(Tail(\pi).status = ABORT\ \wedge\\ &s_i.status = \%\_ATTK\ \wedge\ s_j.status = \%]\_ERR\ \big)\\ &\text{where } \% \text{ matches } CHAL,\ CHALRESP \text{ or } RESP\end{aligned}$$

Lemma 4 states that if a (challenge, challenge-response, or response) packet is modified by a man-in-the-middle attack (adversarial behavior (b)), PA-Boot should raise an error-specific alarm by visiting the corresponding *ERR* state (after the *ATTK* state) and finally terminate in the *ABORT* state.

3) Mechanized Proof in Isabelle/HOL: In our specification $\mathcal{S}_l$, an agent BSP/AP can extend a trace (i.e., a *run* π of $\mathcal{S}_l$) in any way permitted by the protocol. An adversarial behavior can also change the current state and extend a trace. Low-level properties on both functional correctness and security are formalized as trace properties and proved by induction on traces of $\mathcal{S}_l$. In particular, security properties Lemmas 3 and 4 both cover different possible traces of $\mathcal{S}_l$ (corresponding to different attack scenarios). Therefore, we further decompose these lemmas into a set of *auxiliary lemmas* to account for different traces and prove by induction on each trace. Then, by chaining these auxiliary lemmas together in Isabelle/HOL, we obtain a fully mechanized proof that all possible traces of $\mathcal{S}_l$ (i.e., all possible executions of PA-Boot) fulfill the requirements on functional correctness and security. Table III collects the statistics with regard to the proof efforts conducted in Isabelle/HOL, which amount roughly to 8 person-months.

VII. IMPLEMENTATION AND EVALUATION

This section presents CPA-Boot, a proof-of-concept implementation of PA-Boot embedded in multiprocessor secure boot. To build confidence in the consistency of the formal model and the implementation, we develop a test framework that extracts executable code from our Isabelle model $\mathcal{S}_l$ and validate it against CPA-Boot. We then evaluate the security and performance of CPA-Boot on ARM Fixed Virtual Platform (FVP) based on Fast Models 11.18 [44]. The tests are conducted on the Foundation Platform [45] with ARM Cortex-A72 CPUs, with CPU0 performing as BSP and CPU1 as AP.

A. Proof-of-Concept Implementation

We derive `CPA-Boot` in C (∼1,400 LoC) for the ARM64 bare-metal environment as an instance of the (low-level) formalization of `PA-Boot` in Isabelle/HOL. This instantiation is justified by a code-to-spec review in the same way as [38], establishing a near one-to-one correspondence between the formalization and the implementation. For example, each event transition in Isabelle closely matches a correspond ing C function in `CPA-Boot` for protocol operations, with the main difference being that Isabelle models cryptography symbolically, while the C code uses concrete cryptographic operations. We build `CPA-Boot` using GCC 9.4.0 cross-compiler on a server running 64-bit Ubuntu 20.04. `CPA-Boot` is further embedded in bootwrapper v0.2 [46], a bootloader for the ARMv8 architecture, to perform authentication at the initial boot stage. To use cryptographic primitives in the boot environment, `CPA-Boot` adopts wolfSSL 5.3.0 [47]. Moreover, `CPA-Boot` uses Newlib 4.3.0 [48] as the C standard library. Functional correctness of `CPA-Boot` is evaluated on the functional-accurate simulator FVP.

1) `CPA-Boot` Bootloader: We integrate `CPA-Boot` at a later phase of bootwrapper such that necessary hardware initialization is finished before the execution of `CPA-Boot`. We refer to the resulting bootloader as the *`CPA-Boot` bootloader*. Upon the launch of `CPA-Boot`, both processors *concurrently* validate the other processor's certificate until the initiation phase (cf. Sect. IV-A) is completed, i.e., when both processors have retrieved each other's public key and the validations are successful.[2] During the challenge-response phase, both processors *sequentially* exchange encrypted packets (as the inter-processor communications cannot be parallelized) using asymmetric cryptography, then establish a shared session key (in the *concurrent* mode) to secure the communication channel. Since FVP does not support inter-processor communication buses, the channel is mimicked by the main memory where both processors can read from and write to a predefined memory location. Moreover, `CPA-Boot` uses Set Event (SEV) and Wait For Event (WFE) instructions for inter-processor synchronizations: after writing its packet to the channel, a processor executes a SEV instruction to wake the other, then immediately suspends itself by executing WFE.

`CPA-Boot` uses ECC for its asymmetric cryptographic operations due to its efficiency and security over RSA. It employs elliptic-curve digital signature algorithm (ECDSA) for certificate generation and validation, elliptic-curve Diffie-Hellman (ECDH) for secure key exchange, and ephemeral ECDH (ECDHE) for session key generation to ensure forward secrecy. These cryptographic primitives are implemented using the wolfSSL library (see Appx. for operational details). Private keys, certificates, and hash values are linked to the `CPA-Boot` bootloader during compilation and loaded into memory alongside it.

2) Library Compatibility: We tune compile-time configurations of the invoked libraries for compatibility with the bare-metal environment: bootwrapper, Newlib, and wolfSSL are compiled with the flag -mstrict-align to disable unaligned accesses; For Newlib, we adapt bootwrapper's linker script to make memory-management functions such as `malloc` and `free` work correctly, e.g., to avoid heap-memory conflicts; For wolfSSL, we define compile-time macros to fit for the bare-metal environment, which mainly include

- NO_FILESYSTEM to disable loading of keys and certificates in the system's file system,
- WOLFSSL_USER_IO to remove automatic setting of default I/O functions, and
- NO_WRITEV to disable `writev` semantics simulation.

We further add macros to enable all ECC-related operations. The libraries are statically linked to `CPA-Boot` bootloader as the bare-metal environment does not support dynamic load.

[2]The processor that finishes the validation earlier than the other halts and waits for the other processor.

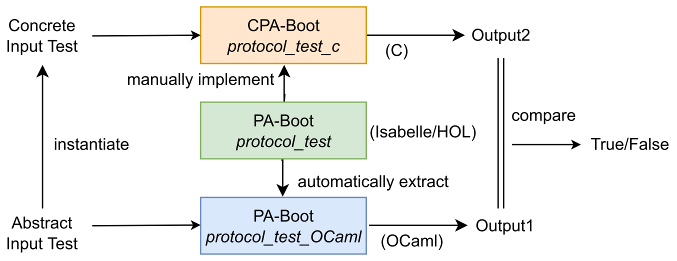


Fig. 6. Validation framework.

B. Validation Framework

Here, we present the process to validate the consistency between the C implementation and Isabelle specification. A test framework is developed to extract executable OCaml code from our Isabelle model $\mathcal{S}_l$ and validate it against `CPA-Boot`, covering both normal protocol execution and attack scenarios.

1) Consistency Validation: The validation is performed using an executable version generated in OCaml via Isabelle/HOL's built-in extraction mechanism. Based on the extracted OCaml code, as illustrated in Fig. 6, our validation framework operates as follows. Our focus is on test cases that simulate normal protocol execution as well as the actions of a skilled attacker carrying out real attack effects, as described in Sect. VII-C. Given an abstract input test case, the extracted OCaml function *protocol_test_OCaml* executes the test case and produces an output. Simultaneously, we instantiate the input test case as corresponding concrete ones, and `CPA-Boot` produces an output. If the two results match, the test is valid, demonstrating consistency between the two implementations. If the results differ, the framework identifies an inconsistency between the low-level C code and OCaml code (and, by extension, the high-level Isabelle/HOL model).

The OCaml implementation extracted from the formal model acts as a higher-level abstraction of the C implementation. For instance, private keys are abstractly represented as an enumerated type with cases of *ROOT_PRIVK*, *M_PRIVK*, *S_PRIVK* and *BAD_PRIVK* in Isabelle/HOL and thereby in OCaml. The OCaml side uses *BAD_PRIVK* to represent a scenario involving a tampered AP. In contrast, the C side simulates an AP-replacement attack by statically linking a modified private key file of CPU1.

TABLE IV
INSTANCES OF ADVERSARIAL BEHAVIORS SIMULATED VIA FVP. ACRONYMS: APR FOR AP REPLACEMENT, RCT FOR ROOT-CERTIFICATE TAMPERING, APCT FOR AP-CERTIFICATE TAMPERING, CPM FOR CHALLENGE-PACKET MANIPULATION, CRPM FOR CHAL.-RESP.-PACKET MANIPULATION, AND RPM FOR RESPONSE-PACKET MANIPULATION

Category	(a)			(b)		
Instance	APR	RCT	APCT	CPM	CPRM	RPM
Detected	✓	✓	✓	✓	✓	✓

In the OCaml implementation, the output result indicates whether the protocol reaches the terminal state with an *END* status or *ABORT* status. Similarly, in CPA-Boot, the result reflects whether the protocol executes successfully and progresses to the next boot stage, or aborts with a detected attack. The results demonstrate the consistency of the outputs for matching inputs across various scenarios particularly those covered in Table IV, thereby increasing confidence in the accuracy of our C implementation.

C. Security Evaluation

We perform empirical security evaluations of CPA-Boot to detect aforementioned adversarial behaviors simulated in FVP covering (a) and (b). In particular, we introduce an extra processor CPU2 mimicking the interposer chip to launch man-in-the-middle attacks. As summarized in Table IV, CPA-Boot succeeds in detecting all different instances of these adversarial behaviors where the BSP returns error-specific alarms to abort the boot process. We show below how the adversarial behaviors are implemented and how CPA-Boot detects them.

1) Ap Replacement: We modify the private key $PrivK_{AP}$ of CPU1 to simulate the AP-replacement attack. Such attack does not trigger alarms in certificate validation and CPA-Boot enters the challenge-response phase. In this phase, CPU1 attempts to forge packets to pass the authentication. However, since the modified private key of CPU1 does not match the stored certificate $Cert_{AP}$ that has been validated in initiation phase, CPU0 cannot decrypt the challenge packet and hence raises an AP-replacement error and aborts the boot process.

2) Man-in-the-Middle Attacks: CPU2 attempts to eavesdrop on or tamper with the (challenge, challenge-response or response) packets transmitted over the inter-processor communication channel (mimicked by memory) during protocol execution. Particularly, for the latter two types of packets, CPU2 acts as a skilled attacker who attempts to replace $EPubK_{BSP}$ in the challenge-response packet and $EPubK_{AP}$ in the response packet aiming to establish a shared session key for future communications. However, as the attacker does not know the private keys of CPU0 and CPU1, he/she cannot manipulate the encrypted hash values of ephemeral public keys. As a consequence, CPA-Boot observes unmatched hash values and thus raises an alarm and aborts the boot process.

3) Tampering With Certificates: We modify certificates linked to CPA-Boot bootloader to simulate certificate manipulations. Specifically, the attacker tries to modify the root certificate or the AP certificate to bypass the processor authentication in the challenge-response phase. However, since the attacker cannot modify the hash value of the root certificate stored in the bootROM, CPU0 detects the root-certificate manipulation using the hash and terminates the boot process. For the AP-certificate manipulation, CPU0 uses the validated root certificate to verify the manipulated AP certificate (signed by $Cert_{root}$) which leads to abortion as well.

TABLE V
#INSTRUCTIONS CONSUMED BY THE MAIN BOOT STAGES, I.E., BOOTWRAPPER (B.W.P.), CPA-BOOT, AND LINUX KERNEL BOOT

Processor(s)	**b.w.p.** (α)	CPA-Boot (β)	**Kernel boot** (γ)	**Overhead** ($\beta/(\alpha+\gamma)$)
CPU0	4,939	52,091,762	1,683,639,749	3.09%
CPU1	650	52,181,244	1,759,591,720	2.97%
CPU0 ‖ CPU1	4,939*	87,570,653◇	1,759,591,720*	4.98%

α, β, or γ denotes the number of instructions executed at each stage, respectively, e.g., $\alpha_{\text{CPU0} \parallel \text{CPU1}}$ for bootwrapper executed instructions of the dual-processor system.
* $\alpha_{\text{CPU0} \parallel \text{CPU1}} = \max\{\alpha_{\text{CPU0}}, \alpha_{\text{CPU1}}\}$ because bootwrapper is assumed to be fully parallelized. The same applies to the boot stage of the Linux kernel.
◇ $\max\{\beta_{\text{CPU0}}, \beta_{\text{CPU1}}\} < \beta_{\text{CPU0} \parallel \text{CPU1}} < \beta_{\text{CPU0}} + \beta_{\text{CPU1}}$ since CPA-Boot is partially parallelized, i.e., communications in the chal.-resp. phase must be in sequence.

D. Performance Evaluation

we report the performance of CPA-Boot within a complete boot process from bootloader to shell login on FVP. We measure the number of executed instructions rather than CPU cycles of the boot process, as FVP is not cycle-accurate and each instruction takes equally one cycle to execute [49]. To capture this, we enable performance monitor unit (PMU [50]) monitoring by setting a PMU control register at key points in the boot stages, recording the total number of instructions executed on FVP. To assess the impact of CPA-Boot on boot performance, we additionally measure the boot process of a Linux system, namely, the Gentoo Linux distribution (stage archive 3, with systemd as init system, kernel version 5.16.0) [51]. Moreover, we disable Linux kernel PMU support to avoid kernel's modification on the PMU counters.

Table V quantifies the experimental performance of the entire boot process averaged over three boot trials, consisting of three main boot stages, i.e., bootwrapper, CPA-Boot, and Linux kernel boot. The integration of CPA-Boot adds a 4.98% overhead to the multiprocessor secure boot, roughly 3% per CPU, which is relatively small. Moreover, we report the performance of different operations conducted in CPA-Boot. As depicted in Fig. 7, the main overhead of CPA-Boot stems from asymmetric cryptography used for certificate validation, packet encryption and decryption, and key exchange.

VIII. DISCUSSION AND FUTURE WORK

In this section, we briefly discuss some issues pertinent to our approach and several interesting future directions.

A. Hardware Adaptation

We implemented our protocol on FVP to demonstrate its security and usability. Here, we discuss how to deploy it on real hardware. Based on the analysis of required components (highlighted in yellow in Fig. 2), we identify necessary

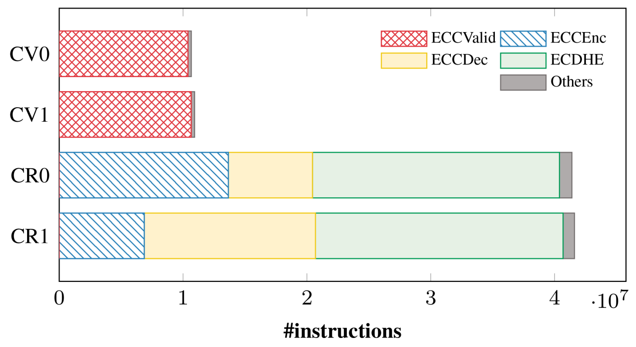


Fig. 7. #Instructions by different operations in `CPA-Boot`. CV0 denotes certificate validation in the initiation phase performed by CPU0; CR0 denotes the challenge-response phase on CPU0. The notations for CPU1 are analogous. ECCValid: ECC-based certificate verification; ECCEnc/ECCDec: ECC-based public-key encryption/decryption via the ECDH algorithm; ECDHE: ECC-based key exchange, including the generation of ephemeral key pairs and $\mathcal{K}_s$; Others: memory read/write, SHA-256 processing, etc.

hardware modifications during manufacturing. The primary requirement is customizing the CPU during manufacturing, embedding the `PA-Boot` protocol code and the hash value of the root certificate signed by the CA into the immutable bootROM. This process is achievable through collaboration between OEMs and silicon vendors. The root certificate hash serves as a trust anchor for processor certificate validation. If the CA is compromised, devices must be returned to the factory to update this anchor. Standard Public Key Infrastructure (PKI) certificate revocation mechanisms, such as Certificate Revocation Lists (CRLs) and Online Certificate Status Protocol (OCSP), rely on online validation and are not applicable in offline secure boot environments. In addition, the OEM must modify the NVM to store certificates, which is easily reprogrammable. The generated session key does not require hardware modifications and can be stored in a CPU register to secure communication during the subsequent boot process.

As noted in Sect. III-A, our approach is particularly useful when the OEM is located far from the end-user, such as overseas, and lacks full control over the supply chain. This is essential for devices performing critical tasks, such as data center servers that support enterprise workloads like cloud services and big data processing.

B. Protocol Extensions

We envision three key extensions of `PA-Boot`: (i) Supporting multiple APs with sequential or parallel authentication. Our protocol supports sequential authentication of multiple APs by the BSP, which aligns with industrial practices where each AP is activated by BSP individually. While parallel authentication, where the BSP broadcasts signals to initialize all APs simultaneously, is possible, addressing this scenario is left for future work. (ii) Enabling a permissive mode that disables APs that fail verification, and allowing the system to continue booting with only verified APs in scenarios with multiple APs; (iii) Incorporating bootROM integrity into AP's key derivation process to enhance security against very powerful attackers who can tamper with the bootROM. For instance, the protocol can use Physically Unclonable Functions (PUFs) to generate keys at runtime, combining the PUF response with the bootROM hash. Any change to the bootROM alters the derived key, enabling the protocol to detect bootROM compromise through a mismatch with the stored certificate $Cert_{AP}$.

1) C Code Generation: Our proof-of-concept prototype is implemented in C to leverage its efficiency. While Isabelle/HOL's can generate executable code (e.g., in OCaml), this code is not optimal in terms of performance. A promising direction for future work is to derive verified C code. One approach is to extend Isabelle's toolchain to directly synthesize C code, as has been observed in [52]. To achieve this, we plan to refine our low-level specification in Isabelle/HOL using the Simpl language [53] (with C-like syntax) and develop a verified compiler from Simpl to C within Isabelle/HOL. Another option is to use the Igloo framework [54] to model our protocol and extend it to support our security properties. This would allow us to translate Isabelle's formal specifications into program assertions, which can then be verified with existing tools to ensure the C implementation meets the specified properties. Alternatively, we could rewrite our protocol in Low* and use the KaRaMeL compiler [55] to generate certified C code.

IX. Related Work

This section reviews research closely related to our approach, focusing on formal methods in bootstrap authentication and protocol verification.

A. Secure Boot/Authenticated Load Verification

Formal methods have witnessed a spectrum of applications in enhancing confidence in boot code security [56], [57], [58]. More pertinently to our work, Muduli et al. [59] verified the end-to-end security of authenticated firmware loaders using model-checking techniques [60]. The targeted scenarios and security properties are different from ours. Muduli et al. focuses on vulnerabilities such as protocol-state hijacking, time-of-check to time-of-use (TOCTOU), and confused deputy attacks *during* firmware loading, whereas our protocol ensures processor authenticity and inter-processor communication confidentiality *before* firmware loading. Similar differences apply to [61], which models firmware loading flows in Promela and verifies the absence of TOCTOU attacks using the Spin model checker [62]. Cremers et.al. [63] formalized the SPDM 1.2 protocol standard in the Tamarin prover [64]. SPDM aims for device attestation, authentication and secure communication and can be used during the boot process. However, due to its size and complexity, the formalization is not fully verified. Although they split the protocol into four separate models and verified them separately, they do not analyze cross-protocol attacks or verify security properties on the complete model. In summary, no existing work addresses defending against the hardware supply-chain attack surface, as identified in Sect. III-A.

B. Protocol Verification

Formal methods have been widely applied to ensure security of real-world protocols, like TLS 1.3 [65], messaging protocols [66], and entity authentication protocols [67], [68].

There are dedicated tools for protocol verification, such as Tamarin [64] and ProVerif [69]. However, protocols verified in these frameworks are abstract models that may not guarantee the same properties in actual implementations [70]. Works using F* [71], Coq, and Isabelle/HOL seek to address this limitation. Protocols in F* can be compiled into OCaml or F# code, while protocols in Low* [72], a low-level subset of F*, can be translated into C via the KaRaMeL compiler. Low* supports the development of high-assurance cryptographic libraries, such as HACL* [73], and implementations of protocols like TLS 1.3 [74] and protocols for measured boot [75] or bootloaders [76]. The DY* symbolic protocol framework [77], written in F*, combines the precision of dependent types from F* with the automation of symbolic execution in dedicated provers like Tamarin, offering efficient and expressive verification.

The following discusses the most closely related works that formalize protocols using Isabelle/HOL and probably derive verified implementation.

Paulson's seminal work [78], [79] uses an inductive approach to model network protocols and prove their security properties. However, it does not verify our security properties Lemma 3 and Lemma 4, which ensures that once an adversarial behavior is performed during protocol execution, the protocol should detect it at some later state along the trace, and ultimately terminate in an error state and trigger an alarm. These properties are crucial for identifying attack vectors and preventing attackers from gaining control of the system.

Igloo [54] also uses Isabelle/HOL with multiple specification layers and guides the correctness proof of implementation code but follows a different approach. It establishes a compositional refinement framework using forward simulation to prove refinement between layers. The soundness theorem establishes that refinement implies trace inclusion. Igloo shows that the abstract protocol model satisfies a trace property, while the lower layer, though introducing more events, refines the higher layer and preserves the trace property. In contrast, our work leverages Isabelle's built-in locale module for abstraction-refinement reasoning. Unlike Igloo, which is based on Paulson's approach, our work verifies security properties Lemma 3 and Lemma 4. While both link high-level models to low-level implementations, Igloo manually converts formal requirements defined in Isabelle into program specifications annotated in programs and uses tools like VeriFast for verification. Our approach prioritizes automation, ensuring consistency between formal specification and implementation through extracted executable code and testing.

X. Conclusion

We have identified a new, prevalent hardware supply-chain attack surface that can bypass multiprocessor secure boot due to the absence of processor-authentication mechanisms. To defend against these attacks targeting “”assumed-safe components (e.g., processors and inter-processor communication channels), we presented `PA-Boot`, the first formally verified processor-authentication protocol for multiprocessor secure bootstrap. We showed – using a machine-checked mathematical proof in Isabelle/HOL – that `PA-Boot` is functionally correct and is guaranteed to detect multiple adversarial behaviors, e.g., man-in-the-middle attacks and processor replacements. Experiments on ARM FVP suggested that our proof-of-concept implementation `CPA-Boot` can effectively identify boot-process attacks with a minor overhead and thereby improve the bootstrap security of multiprocessor systems.

Future work includes expanding the functionality of `PA-Boot` and generating verified C code (see Sect. VIII).

Appendix

A. Cryptographic Primitives and Operations Implemented in `CPA-Boot`

appendix]appx:config-proto In `CPA-Boot`, an ephemeral ECC key pair is generated using the P-256 curve, producing a 256-bit private key and its corresponding public key. The shared session key, derived via ECDHE between BSP and AP, is also 256 bits. In the challenge-response phase, each party computes a shared secret using ECDH with its private key and the other party's public key. This shared secret is processed with HKDF-SHA256 to derive cryptographic keys. AES-128-CBC is used for data encryption, and HMAC-SHA256 ensures message authentication. These operations, including HKDF-SHA256 and AES-128-CBC, are internally invoked by the wolfSSL library during ECDH execution.

References

[1] Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. Thakur, “DICE*: A formally verified implementation of DICE measured boot,” in *Proc. USENIX Secur.*, 2021, pp. 1091–1107.

[2] J. D. Tygar and B. Yee, *Dyad: A System for Using Physically Secure Coprocessors*. Pittsburgh, PA, USA: Carnegie Mellon Univ., 1991.

[3] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay, “Lightweight secure-boot architecture for RISC-V system-on-chip,” in *Proc. 20th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2019, pp. 216–223.

[4] O. Shwartz, A. Cohen, A. Shabtai, and Y. Oren, “Shattered trust: When replacement smartphone components attack,” in *Proc. WOOT*, 2018, pp. 1–13.

[5] F. Bohling, T. Mueller, M. Eckel, and J. Lindemann, “Subverting Linux' integrity measurement architecture,” in *Proc. 15th Int. Conf. Availability, Rel. Secur.*, Aug. 2020, pp. 1–10.

[6] J. P. Skudlarek, T. Katsioulas, and M. Chen, “A platform solution for secure supply-chain and chip life-cycle management,” *Computer*, vol. 49, no. 8, pp. 28–34, Aug. 2016.

[7] J. Christensen, I. M. Anghel, R. Taglang, M. Chiroiu, and R. Sion, “DECAF: Automatic, adaptive de-bloating and hardening of COTS firmware,” in *Proc. USENIX Secur.*, 2020, pp. 1713–1730.

[8] S. N. Dhanuskodi, X. Li, and D. Holcomb, “COUNTERFOIL: Verifying provenance of integrated circuits using intrinsic package fingerprints and inexpensive cameras,” in *Proc. USENIX Secur.*, 2020, pp. 1255–1272.

[9] Z. Han, M. Yasin, and J. Rajendran, “Does logic locking work with EDA tools?,” in *Proc. USENIX Secur.*, 2021, pp. 1055–1072.

[10] J. F. Miller, “Supply chain attack framework and attack patterns,” MITRE Corp., McLean, VA, USA, Tech. Rep. MTR140021, 2013.

[11] M. Gaithersburg, “Guide for conducting risk assessments,” Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. NIST Special Publication 800-30 Revision 1, 2012.

[12] K. Shiralkar, A. Bongale, S. Kumar, K. Kotecha, and C. Prakash, “Assessment of the benefits of information and communication technologies (ICT) adoption on downstream supply chain performance of the retail industry,” *Logistics*, vol. 5, no. 4, p. 80, Nov. 2021.

[13] R. Naraine. (2022). *U.S. Gov Issues Stark Warning, Calling Firmware Security a ‘Single Point of Failure’*. Accessed: Feb. 1, 2025. [Online]. Available: https://www.securityweek.com/us-gov-issues-stark-warning-calling-firmware-security-single-point-failure

[14] J. Jaeger. (2018). *A Wake-up Call in Cyber Supply-Chain Risk*. COMPLIANCE WEEK. Accessed: Feb. 1, 2025. [Online]. Available: https://www.complianceweek.com/a-wake-up-call-in-cyber-supply-chain-risk/2106.article

[15] J. Frazelle, "Securing the boot process: The hardware root of trust," *Commun. ACM*, vol. 63, no. 3, pp. 38–42, 2020.

[16] B. Meadows, N. Edwards, and S.-Y. Chang, "On-chip randomization for memory protection against hardware supply chain attacks to DRAM," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2020, pp. 171–180.

[17] A. C. Sant'Ana, H. M. Medina, K. B. Fiorentin, and F. G. Moraes, "Lightweight security mechanisms for MPSoCs," in *Proc. 32nd Symp. Integr. Circuits Syst. Design (SBCCI)*, Aug. 2019, pp. 1–6.

[18] D. Mehta et al., "The big hack explained: Detection and prevention of PCB supply chain implants," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 4, p. 42, 2020.

[19] T. Hudson, "Lecture: Modchips of the state," in *Proc. CCC*, 2018.

[20] T. Mosavirik, P. Schaumont, and S. Tajik, "Impedanceverif: On-chip impedance sensing for system-level tampering detection," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2023, no. 1, pp. 301–325, Mar. 2023.

[21] U. D. O. COMMERCE and U. D. O. H. SECURITY. *Assessment of the Critical Supply Chains Supporting the U.S.Information and Communications Technology Industry*. Accessed: Feb. 1, 2025. [Online]. Available: https://www.dhs.gov/sites/default/files/2022-02/ICT%20Supply%20Chain%20Report_0.pdf

[22] J. Zhu et al., "Jintide: Utilizing low-cost reconfigurable external monitors to substantially enhance hardware security of large-scale CPU clusters," *IEEE J. Solid-State Circuits*, vol. 56, no. 8, pp. 2585–2601, Aug. 2021.

[23] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL—A Proof Assistant for Higher-Order Logic* (Lecture Notes in Computer Science), vol. 2283. Cham, Switzerland: Springer, 2002.

[24] O. Khalid, C. Rolfes, and A. Ibing, "On implementing trusted boot for embedded systems," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2013, pp. 75–80.

[25] A. Marchand, Y. Imine, H. Ouarnoughi, T. Tarridec, and A. Gallais, "Firmware integrity protection: A survey," *IEEE Access*, vol. 11, pp. 77952–77979, 2023.

[26] A. L. Sacco and A. A. Ortega. (2009). *Persistent Bios Infection*. Accessed: Feb. 1, 2025. [Online]. Available: https://www.coresecurity.com/sites/default/files/private-files/publications/2016/05/Persistent-BIOS-Infection.pdf

[27] (2020). *Platform Security Boot Guide*. Accessed: Feb. 1, 2025. [Online]. Available: https://developer.arm.com/documentation/den0072/0101/

[28] E. Paul, M. Andrey, M. Dennis, S. Rob, T. Stefan, and W. David. (2020). *RIoT—A Foundation for Trust in the Internet of Things*. Accessed: Feb. 1, 2025. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/06/RIoT20Paper-1.1-1.pdf

[29] (2014). *TCG Specification TPM 2.0 Mobile Reference Architecture*. Accessed: Feb. 1, 2025. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2-0-Mobile-Reference-Architecture-v2-r142-Specification_FINAL2.pdf

[30] J. Szefer, "Principles of secure processor architecture design," in *Principles of Secure Processor Architecture Design*. Cham, Switzerland: Springer, 2018, pp. 113–124.

[31] (2020). *TCG Server Management Domain Firmware Profile Specification*. Accessed: Oct. 20, 2025. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCG_ServerManagementDomainFirmwareProfile_v1p00_11aug2020.pdf

[32] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng, "Providing root of trust for ARM TrustZone using on-chip SRAM," in *Proc. 4th Int. Workshop Trustworthy Embedded Devices*, Nov. 2014, pp. 25–36.

[33] A. D. Zonenberg, A. Moor, D. Slone, L. Agan, and M. Cop, "Extraction of secrets from 40nm CMOS gate dielectric breakdown antifuses by FIB passive voltage contrast," 2025, *arXiv:2501.13276*.

[34] F. Courbon, "Practical partial hardware reverse engineering analysis: For local fault injection and authenticity verification," *J. Hardw. Syst. Secur.*, vol. 4, no. 1, pp. 1–10, Mar. 2020.

[35] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proc. 9th ACM Conf. Comput. Commun. Secur.*, Nov. 2002, pp. 148–160.

[36] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-efficient encryption for non-volatile memories," in *Proc. ASPLOS*, 2015, pp. 33–44.

[37] A. Awad, P. K. Manadhata, S. Haber, Y. Solihin, and W. G. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proc. ASPLOS*, 2016, pp. 263–276.

[38] Y. Zhao, D. Sanán, F. Zhang, and Y. Liu, "Refinement-based specification and security analysis of separation kernels," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 1, pp. 127–141, Jan. 2019.

[39] C. Hawblitzel et al., "IronFleet: Proving practical distributed systems correct," in *Proc. 25th Symp. Operating Syst. Princ.*, Oct. 2015, pp. 1–17.

[40] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.

[41] C. Ballarin, "Interpretation of locales in Isabelle: Theories and proof contexts," in *Proc. MKM* (Lecture Notes in Computer Science), vol. 4108. Cham, Switzerland: Springer, 2006, pp. 31–43.

[42] F. Haftmann and T. Nipkow, "A code generator framework for Isabelle/HOL," in *Theorem Proving Higher Order Logics: Emerging Trends Proceedings*, vol. 8. Cham, Switzerland: Springer, 2007, p. 121.

[43] D. Dolev and A. C. Yao, "On the security of public key protocols," in *Proc. 22nd Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1981, pp. 350–357.

[44] *Fast Models Fixed Virtual Platforms Reference Guide*. Accessed: Feb. 1, 2025. [Online]. Available: https://developer.arm.com/documentation/100966/latest

[45] *Foundation Platform User Guide*. Accessed: Feb. 1, 2025. [Online]. Available: https://developer.arm.com/documentation/100961/1118/

[46] *Linux Kernel Bootwrapper*. Accessed: Feb. 1, 2025. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/mark/boot-wrapper-aarch64.git/

[47] *WolfSSL Embedded SSL/TLS Library*. Accessed: Feb. 1, 2025. [Online]. Available: https://github.com/wolfSSL/wolfssl

[48] *Newlib*. Accessed: Feb. 1, 2025. [Online]. Available: https://www.sourceware.org/newlib/

[49] *FVP Reference Guide*. Accessed: Feb. 1, 2025. [Online]. Available: https://developer.arm.com/documentation/100966/1100-00/Programming-Reference-for-VE-FVPs/Differences-between-the-VE-hardware-and-the-system-model/Restrictions-on-the-processor-models

[50] *Performance Monitoring Unit*. Accessed: Feb. 1, 2025. [Online]. Available: https://developer.arm.com/documentation/100511/0401/performance-monitoring-unit

[51] *Gentoo Linux*. Accessed: Feb. 1, 2025. [Online]. Available: http://www.gentoo.org/

[52] Y. Zhang, Y. Zhao, and D. Sanan, "A verified timsort c implementation in Isabelle/HOL," 2018, *arXiv:1812.03318*.

[53] N. Schirmer, "Verification of sequential imperative programs in Isabelle/HOL," Ph.D. dissertation, Dept. Inform., Technical Univ. Munich, Munich, Germany, 2006.

[54] C. Sprenger et al., "Igloo: Soundly linking compositional refinement and separation logic for distributed system verification," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–31, Nov. 2020.

[55] *KaRaMeL*. Accessed: Feb. 1, 2025. [Online]. Available: https://github.com/FStarLang/karamel

[56] Z. Straznickas, "Towards a verified first-stage bootloader in Coq," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2020.

[57] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Model checking boot code from AWS data centers," *Formal Methods Syst. Des.*, vol. 57, no. 1, pp. 34–52, Jul. 2021.

[58] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, "Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[59] S. K. Muduli, P. Subramanyan, and S. Ray, "Verification of authenticated firmware loaders," in *Proc. Formal Methods Comput. Aided Design (FMCAD)*, Oct. 2019, pp. 110–119.

[60] C. Baier and J. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[61] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of SoC firmware load protocols," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, May 2014, pp. 70–75.

[62] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[63] C. Cremers, A. Dax, and A. Naska, "Formal analysis of SPDM: Security protocol and data model version 1.2," in *32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 6611–6628.

[64] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie–Hellman protocols and advanced security properties," in *Proc. IEEE 25th Comput. Secur. Found. Symp.*, Jun. 2012, pp. 78–94.

[65] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 470–485.

[66] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Apr. 2017, pp. 435–450.

[67] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5G authentication," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2018, pp. 1383–1396.

[68] C. Sprenger and D. Basin, "Developing security protocols by refinement," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, Oct. 2010, pp. 361–374.

[69] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. 14th IEEE Comput. Secur. Found. Workshop*, Jun. 2001, pp. 82–96.

[70] L. Arquint et al., "Sound verification of security protocols: From design to interoperable implementations," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2023, pp. 1077–1093.

[71] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," *J. Funct. Program.*, vol. 23, no. 4, pp. 402–451, Jul. 2013.

[72] J. Protzenko et al., "Verified low-level programming embedded in F*," *Proc. ACM Program. Lang.*, vol. 1, pp. 1–29, 2017.

[73] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1789–1806.

[74] A. Delignat-Lavaud et al., "Implementing and proving the TLS 1.3 record layer," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 463–482.

[75] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in webassembly," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1256–1274.

[76] S. Yuan and J.-P. Talpin, "Verified functional programming of an IoT operating system's bootloader," in *Proc. 19th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design (MEMOCODE)*, Nov. 2021, pp. 89–97.

[77] K. Bhargavan et al., "DY*: A modular symbolic verification framework for executable cryptographic protocol code," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Sep. 2021, pp. 523–542.

[78] L. C. Paulson, "The inductive approach to verifying cryptographic protocols," *J. Comput. Secur.*, vol. 6, nos. 1–2, pp. 85–128, Jan. 1998.

[79] L. C. Paulson, "Inductive analysis of the internet protocol TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 3, pp. 332–351, Aug. 1999.