IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 8 November 2023, accepted 17 December 2023, date of publication 25 December 2023,
date of current version 3 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3346501

RESEARCH ARTICLE

Formal Specification and Verification of Architecturally-Defined Attestation Mechanisms in Arm CCA and Intel TDX

MUHAMMAD USAMA SARDAR[1], (Member, IEEE), THOMAS FOSSATI[2], SIMON FROST[3], AND SHALE XIONG[3]
[1]Faculty of Computer Science, TU Dresden, 01062 Dresden, Germany
[2]Linaro, 1009 Lausanne, Switzerland
[3]Arm, CB1 9NJ Cambridge, U.K.

Corresponding author: Muhammad Usama Sardar (muhammad_usama.sardar@tu-dresden.de)

This work was supported by the German Research Foundation (DFG) as part of TRR 248—CPEC under Grant 389792660.

ABSTRACT Attestation is one of the most critical mechanisms in confidential computing (CC). We present a holistic verification approach enabling comprehensive and rigorous security analysis of architecturally-defined attestation mechanisms in CC. Specifically, we analyze two prominent next-generation hardware-based Trusted Execution Environments (TEEs), namely Arm Confidential Compute Architecture (CCA) and Intel Trust Domain Extensions (TDX). For both of these solutions, we provide a comprehensive specification of all phases of the attestation mechanism, namely provisioning, initialization, and attestation protocol. We demonstrate that including the initialization phase in the formal model leads to a violation of integrity, freshness, and secrecy properties for Intel's claimed trusted computing base (TCB), which could not be captured by considering the attestation protocol alone in the related work. We open-source our artifacts. Other researchers, including a team from Intel, are adopting our artifacts for further analysis.

INDEX TERMS Arm confidential compute architecture (CCA), confidential computing, formal specification, Intel trust domain extensions (TDX), remote attestation, trusted execution environment.

I. INTRODUCTION

Several privacy and data protection laws, such as the General Data Protection Regulation (GDPR) [1], mandate protecting personal data during processing (cf. Article 32 in GDPR). While data protection at rest and in transit has been well-understood, protecting data in use is relatively new. There are four primary technologies, collectively known as Privacy-Enhancing Technologies (PETs), available as potential solutions for protecting data in active use: 1) Secure Multi-party computation (MPC) [2], 2) Homomorphic Encryption (HE) [3], 3) Zero-Knowledge Proofs (ZKPs) [4], and 4) Confidential Computing (CC) using hardware-based Trusted Execution Environments (TEEs) [5]. Each of these technologies has its pros and cons. Generally speaking, while MPC, HE and ZKPs offer a small Trusted Computing Base (TCB) consisting of software only, they suffer from high computation as well as communication overhead. On the other hand, CC enables high performance at the cost of relatively larger TCB consisting of hardware, firmware as well as software components [5]. Because of this dependence on hardware, firmware, and software components in CC, it is necessary to attest [6] that: 1) the platform consisting of hardware, firmware as well as trusted software components – on which user-provided workload executes – is real (i.e., neither simulated nor emulated), authentic, and

The associate editor coordinating the review of this manuscript and approving it for publication was Zijian Zhang.

up to date 2) workload executing on the data is the expected one (i.e., unmodified). We call this architecturally-defined attestation. If such attestation of platform and workload is not executed correctly, user's sensitive data may be sent to a fake CC platform or malicious CC workload and thus overcome the protections of CC.

While MPC, HE, and ZKPs have existed for decades and are based on well-defined mathematical concepts, CC is a relatively emerging technology. In addition to the existing TEEs – such as Intel Software Guard Extensions (SGX) [7] and AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) [8] – new TEEs, such as Arm Confidential Compute Architecture (CCA) [9], [10], Intel Trust Domain Extensions (TDX) [11], RISC-V Confidential Virtual Machine Extensions (CoVE) [12], [13], and IBM Protected Execution Facility (PEF) [14], are being introduced. Each of these TEEs has a distinctive attestation architecture and design (Table 1). For example, Intel SGX and TDX, and AMD SEV-SNP are deployed products where all details of the platform attestation have been established (called vendor solutions), while Arm CCA and RISC-V AP-TEE have common architectural specifications that multiple products may share (called architecture led solutions). Architecture led solutions only provide a common design for attestation, allowing detailed differentiation between the actual products from their partners. Moreover, some technologies, such as Intel SGX and TDX, have a layered structure for attestation (called Layered Attester), while attestation in Arm CCA comprises multiple Attesters (called Composite Attesters) [15]. Similarly, Intel SGX performs isolation at the process level (called process-based), while others like Intel TDX and Arm CCA operate at Virtual Machine (VM) level (called VM-based).

Because of the distinctive attestation architecture and security-critical nature of the architecturally-defined attestation mechanisms in TEEs, it is crucial to understand and perform a systematic security analysis of such mechanisms for emerging TEEs. Formal methods provide a systematic and mathematically rigorous approach to understanding, specifying, and verifying system behavior. Specifically, symbolic security analysis tools [16] are widely used to precisely define and rigorously analyze the behavior and properties of a protocol such as TLS 1.3 [17]. Compared to such protocols, a major challenge in the formalization of architecturally-defined attestation is that all vendors present specifications of architecturally-defined attestation in natural language. Such vague specifications can cause legal uncertainty [18] and thus hamper compliance with the regulations; for instance, GDPR (cf. Articles 5(1)(a) and 12) [1] requires that personal data is processed transparently. Apart from a legal perspective, a better understanding helps develop better systems. With this motivation, in our previous works [19], [20], [21], we took the challenge of high-level formal specification and verification of attestation protocol phase in Intel SGX - namely EPID [19] and DCAP [20] - and TDX [21]. However, these works cover only the attestation protocol phase of attestation and do not cover the provisioning and initialization phases.

In this work, we demonstrate that including the initialization phase of attestation in Intel TDX leads to a violation of integrity, freshness, and secrecy properties on Intel's claimed TCB that could not be captured by performing the formal verification of attestation protocol phase alone in our previous work [21]. Moreover, we aim to provide wide coverage of the different flavors of attestation in TEEs to generate verification primitives that may be partly reused to analyze other TEEs. As shown in Table 1, Arm CCA is the only TEE with a composite attester. Moreover, it is the only TEE in architecture led solutions that provides a good level of available specification documents. In contrast, attestation specifications of RISC-V CoVE are still to be ratified at the time of writing. Hence, we formally analyze Arm CCA in addition to Intel TDX. In a nutshell, this work provides a comprehensive formal analysis of Arm CCA and Intel TDX attestation covering all the phases. Specifically, the novel contributions of this work include:

- first formal specification and analysis of attestation in architectural specification group and composite attester (Arm CCA)
- most detailed formal model (including certificate chain and verifier steps) of Intel TDX attestation with initialization phase and variable measurements. We formally prove the insecurity of Intel's claimed TCB. Moreover, the formal specification led to several improvements in Intel's specifications.

We release our artifacts [22] under generous Apache License v2.0 for further research and development. Other researchers are adopting our artifacts for further analysis. This includes a team from Intel performing formal verification of the virtual Trusted Platform Module (vTPM) solution for the Trust Domain (TD). Moreover, a Confidential Computing Consortium project[1] on attested Transport Layer Security (TLS) is planning to utilize our artifacts, in addition to the formalization of TLS [17], for the formal specification and verification of protocol allowing a generic way of passing Evidence and Attestation Results in the TLS handshake [23].

The rest of the paper is organized as follows: Section II presents our approach for the specification of the attestation mechanisms. Section III and IV present the specification of the attestation mechanism in Arm CCA and Intel TDX, respectively. Section V highlights our contributions compared to the related work. Finally, Section VI concludes the work and highlights some interesting directions for future work.

II. APPROACH AND TERMINOLOGY

This section first presents abstract architecture and terminology for architecturally-defined attestation in Section II-A. We then describe the phases of attestation mechanisms –

[1] https://github.com/ccc-attestation/attested-tls-poc

TABLE 1. Heterogeneity of attestation in confidential computing.

Technology	Intel SGX	Intel TDX	AMD SEV-SNP	Arm CCA	RISC-V CoVE	IBM PEF
Attestation granularity	Process	VM[1]	VM	VM	VM	VM
Attestation solution	Product	Product	Product	Arch. specs[2]	Arch. specs	Product
Attester	Layered	Layered	Layered	Composite	Layered	Layered

[1] Virtual Machine [2] Architectural specifications

provisioning, initialization, and attestation protocol – in Section II-B. Finally, we give an overview of formal specification consisting of the formal model, threat model, and properties to be verified for architecturally-defined attestation in Section II-C.

Throughout the paper, we use capitalization of key terms. To describe the public/private nature of cryptographic keys precisely, we use Smart's colour coding [24] as a visual supplement throughout the text and figures, i.e., blue colour to show public information, e.g., public keys, and red colour to show secrets like private or symmetric keys. In addition, we use green colour to represent the key-pair.

A. ABSTRACT ARCHITECTURE AND TERMINOLOGY

At a very high level, architecturally-defined attestation consists of generation of Evidence at Attester and Appraisal of Evidence at Verifier, as shown in Fig. 1. Our architecture – inspired by the RATS standard [15] – predefines a number of roles and conceptual messages.

1) MAIN ROLES

In our abstract architecture, there are two main roles, namely Attester and Verifier. An *Attester* – often referring to the combination of hardware, firmware and trusted software deployed in the cloud – creates attestation Evidence about itself. It consists of at least one Attesting Environment and at least one Target Environment. An *Attesting Environment* is the measuring portion of an Attester. It collects the relevant information about the Target Environment by reading system registers and variables, calling into subsystems, and taking measurements on code, memory, or other security-related assets. It then formats the Claims appropriately and typically uses private key material to digitally sign the Claims and generate attestation Evidence about itself. A *Target Environment* represents the measured portion of an Attester. Anything that may have an impact on the correctness of the TCB is a candidate target. Note that Attesting and Target Environments may be combined. A single device can host more than one Attester, called Composite Attester. Attesters can be chained, for instance DICE [25] Attester.

A *Verifier* conducts Appraisal of Evidence in order to evaluate the trustworthiness of the Attester. The Verifier has trusted relationships with the supply chain – Endorsers and Reference Values Providers – through which it acquires accurate and timely information about Attesters from supply chain roles.

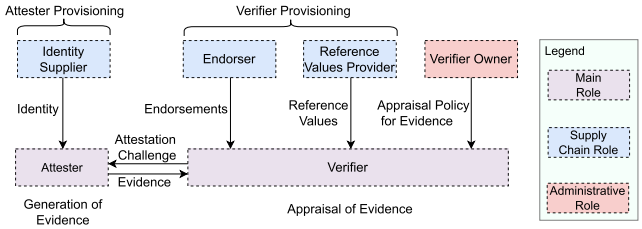

FIGURE 1. Architecture for architecturally-defined attestation.

2) SUPPORTING ROLES

To support the attestation, we define three supply chain roles. *Identity Supplier* is associated with the action of giving the Attester its cryptographic Identity. Depending on the attestation technology, this role may be instantiated in different ways. For example, it could be the manufacturer injecting secrets into the device in the plant, or a local Certificate Authority (CA) when the device is deployed into a network domain. As another example, in privacy-preserving schemes like Direct Anonymous Attestation (DAA) [26], the anonymized Identity is given by the DAA Issuer, provided the join sub-protocol completes successfully. *Endorser* and *Reference Value Provider* supply Endorsements and Reference Values, respectively, to the Verifier. Endorser helps Verifier appraise the authenticity of Evidence, while Reference Values Provider helps Verifier appraise Evidence to determine if the Claims are acceptable.

Finally, the only administrative role, *Verifier Owner*, is authorized to configure *Appraisal Policy* for Evidence in a Verifier. The Appraisal Policy for Evidence is the set of rules that define the Evidence Appraisal procedure.

Note that in general, roles can be coalesced into the same entity or split among different entities. For example, the same supply chain entity could be at the same time an Endorser and a Reference Values Provider. To clearly state which entity takes on a role in the rest of the paper, we use rectangles with a dotted boundary to represent the roles and rectangles with a solid boundary to represent the entity that takes on this role.

3) CONCEPTUAL MESSAGES

In our abstract attestation, there are several key messages between roles. Each of these is explained below:

Attestation Challenge is used to establish the freshness of the Evidence. It can be implicit (nonce, epoch identifier) or explicit (timestamp). In this work, we consider nonce. The Verifier sends a randomly generated nonce, and the nonce

is then signed and included along with the Claims in the Evidence. The Verifier checks that the sent and received nonces are the same, ensuring that the Claims were signed after the nonce was generated.

Evidence is a statement made by the Attester containing a set of trust metrics associated with the Attester's current state. Typically, such trust metrics are encoded into key/value pairs also known as attestation Claims. Claims may include measurements related to the boot sequence, measurements of the run-time state, and also telemetry (e.g., values sampled from a *secure* sensor). We use the prefixes Local and Remote with Evidence to distinguish whether the entities taking on the roles of Attester and Verifier are on the same or different *platforms*. The entities may be on different sockets in the case of multi-socket processor but as long as they are on the same platform, it is classified with the prefix Local.

Reference Values are *known good* values for the trust metrics reported by an Attester. They are *known good* in the sense that they describe the desired state of a trustworthy device. A number of related Claims are expected to be found in attestation Evidence which will be directly compared with the corresponding Reference Values during the Evidence Appraisal procedure.

Endorsements consist of Identity Endorsements as well as Endorsed Values. Identity Endorsements establish the cryptographic identity of the Attester, for example, a public key corresponding to the Attester's signing key. Endorsed Values are Attester's features that are not measured, for example, a security certification associated with the attesting device, etc. Endorsed Values are accepted after Reference Values have been checked.

Identity includes both long-term and ephemeral identities of Attester. It can also be anonymized, as in the case of DAA [26].

B. PHASES OF ATTESTATION MECHANISM

With all the roles in abstract attestation architecture, we further define three phases of attestation mechanisms, namely *Provisioning*, *Initialization* and *Attestation Protocol*. Attestation researchers often focus only on the attestation protocol. However, we present a holistic view of all the phases in this work since provisioning and initialization are also crucial from a security perspective. The level of detail in the specification of all phases should be such that one can make a security argument from a cryptographic – in contrast to systems – perspective.

Provisioning encompasses Attester Provisioning as well as Verifier Provisioning. *Attester Provisioning* covers the tasks typically performed by a manufacturer to provision the platform with the secrets (shown as Identity in Fig. 1) or certificate required to create the Attestation Root of Trust (RoT). This phase may also involve registration of the device with an authority to obtain the certificate. *Verifier Provisioning* covers the supply of Endorsements and Reference Values from the Endorser and Reference Values Provider, respectively, to the Verifier, as shown in Fig. 1.

Once the chips are shipped and deployed in the production environment, often the cloud, the device may go through a set of *initialization* steps that create the basis for supporting the Attestation Protocol. These steps are often conducted during platform boot. A typical scenario includes loading the firmware and then deriving dynamic secrets from the manufacturer-provisioned root secret via Key Derivation Functions (KDFs). The newly derived secrets are used in the actual attestation protocol.

Attestation Protocol – the main phase of operation – is the execution of a series of message exchanges on every attestation request. As shown in Fig. 1, it consists of two abstract steps, namely *generation* and then *appraisal* of *Remote Evidence*. The generation of Remote Evidence should specify at least: 1) completion of KDFs for keys used in the process and 2) the security-relevant contents of all messages conveyed between different entities within the process. The appraisal of Remote Evidence should precisely specify the algorithm for the Remote Verifier.

The Verifier can make a trust decision at the end of the attestation protocol. If the Verifier decides to trust the Attester, it can perform any trustworthy operations relevant to the CC use case. For example, it may release secrets into the CC execution environment.

C. FORMAL SPECIFICATION AND VERIFICATION

All the phases of attestation of a specific technology can be formally specified and verified in the state-of-the-art symbolic security analysis tool, ProVerif [27]. ProVerif is based on Horn clauses and supports the verification of trace and equivalence properties for an unbounded number of sessions. For background as well as syntax and semantics of ProVerif, we refer the readers to [27]. The formal specification includes the formal model, threat model and properties, as explained below.

1) FORMAL MODEL

Since the specification documents are presented in natural language by both Arm and Intel, one of the key contributions of this work is the formal specification of all the phases of attestation in Arm CCA and Intel TDX. Our formal models are based on: (i) in-depth reading of Intel and Arm specification documents (ii) our experience with Intel SGX (on which the attestation architecture Intel TDX is largely based) and Arm CCA (iii) extensive discussions with Intel and Arm. Therefore, we believe we have a faithful interpretation and thus formalization of these two specifications. For formal modeling in ProVerif, each trusted entity in the Attester is represented with a subprocess in ProVerif, and the sequence of message exchanges between the entities is represented in ProVerif's specification language.

2) THREAT MODEL

The approach of symbolic security analysis is based on the classical Dolev-Yao [28] adversary with perfect

cryptographic assumption, i.e., all cryptographic primitives (e.g., hash, encryption, MAC, digital signatures) are assumed to be perfect. Additionally, implementation flaws, such as side channels, are out of scope. Moreover, we assume that the attestation-related keys are not leaked to the adversary. We also assume that the Verifier is stateful, i.e., it can store the generated nonce to check freshness. The adversary capabilities are then defined by: (i) which entities are considered trusted: in ProVerif, untrusted entities are part of the network adversary; thus, only trusted entities are modeled. (ii) which channels are considered secure: in ProVerif, all channels are public except those that are identified by the keyword **private**. (iii) which functions are available to adversary: in ProVerif, all functions are available to the adversary except those that are identified by the keyword **private**. (iv) technology-specific capabilities, and (v) modeling assumptions.

3) PROPERTIES

In the context of attestation, properties of interest include:

Integrity: Claims inside Evidence represent the current state of the Attester and contain critical elements such as identity fields. Hence, ensuring that the adversary does not modify Claims during transport from the Attester to the Verifier is critical. Integrity of Claims is typically protected via digital signatures using an Attestation Key. In ProVerif, integrity is formalized as correspondence assertions with Claims as variables of agreement. Events with these variables are placed just before sending Evidence and after successful verification of Evidence. Now if the correspondence assertion holds, it implies that the adversary cannot modify the variables of agreement without being detected.

Freshness: An adversary can replay valid Evidence from the older session and, in the meantime, change the state of the Attester. Hence, freshness of Evidence is important. In ProVerif, this is formalized as injective correspondence assertions (i.e., using **inj** − **event**) with Claims as variables of agreement. Intuitively, this checks that only one Evidence corresponds to each nonce.

Secrecy: It must be ensured that the adversary does not have access to attestation-related keys or secrets shared between Attester and Verifier. In ProVerif, this is formalized as reachability property, i.e., to analyze whether a state where the adversary has access to the secrets in plaintext is reachable. If such a state is not reachable, it implies that secrecy is maintained.

Authentication: Verifier must ensure it communicates with the intended Attester. Informally, if the Verifier receives the public key of Target Environment, this *uniquely* matches the public key generated within the Target Environment. To formalize this in ProVerif, an event is defined after generating the identity key in the subprocess representing TEE. Another event is defined after verifying Remote Evidence in the Verifier. The property then checks whether the variable of agreement – public part of the identity key – is the same in both events. Authentication holds if the public keys are the same in both events.

III. ATTESTATION IN ARM CCA

The Arm AArch64 architecture creates multiple privilege levels in which software can execute. These levels are referred to as *Exception Levels* (EL) and have numeric names, i.e., EL0 is the least privileged, and EL3 is the most privileged level [29]. From version Armv6, Arm A-Profile architecture has supported hardware-enforced isolation between two execution environments, namely *Normal* world and *Secure* world (also known as TrustZone).

Arm CCA is a system consisting of both hardware and software architectures. The hardware architecture is called the Realm Management Extension (RME) [9]. RME allows the creation of a separate execution world – known as *Realm* world [30] – with a physical address space that is isolated from the existing Normal or Secure execution worlds and allows memory to be reallocated between worlds dynamically. Arm CCA firmware (EL3) uses RME to allocate memory to Realm world. Within the Realm world, individually protected execution environments at EL0-EL1, called Realms, can be created. Realms are isolated from each other using page table mechanisms. The owner of a Realm is not required to trust the software components that manage the resources used by the Realm (such as the Normal world hypervisor). The software component responsible for managing Realms is called the *Realm Management Monitor* (RMM) [10]. It executes at EL2 within the Realm physical address space. The RMM is programmed via a set of Application Binary Interfaces (ABIs) exposed to the Normal world.

In addition to the CCA-specific architecture components, additional architectural components are significant to the system: 1) *Monitor*: Executing at Root EL3, this is the most privileged software component. It is responsible for switching between the security states used at EL2, EL1 and EL0. 2) *Hardware Enforced Security* (HES): HES covers hardware trusted controllers that can store and process secrets outside of the main application processor (AP), such as secure elements. This separation aids the threat model in protecting secrets.

At the time of writing, we are not aware of any available hardware implementing CCA. Also, not all products implementing CCA will be identical because Arm defines architectural specifications while leaving certain details (such as different cryptographic algorithms) as an implementation choice. This paper discusses the reference implementation and the open-source release of the RMM specification and software implementations (TF-RMM) [31]. However, the HES is out of scope in the TF-RMM code [32]. Similarly, TF-RMM does not provide the Verifier and Realm instances. TF-RMM implements only the RMM actor interfaces [33] and business logic [34]. In contrast, we provide a holistic view covering HES, Verifier, and Realm, in addition to RMM.

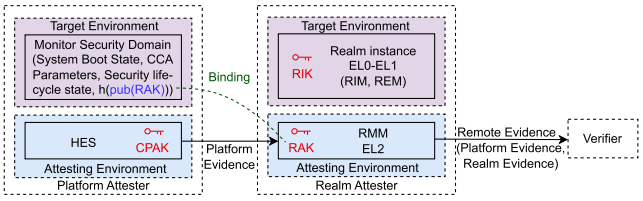

FIGURE 2. Overview of attester in Arm CCA (Composite attester with realm attester as the lead attester) showing important claims, where h represents hash.

A. ATTESTATION ARCHITECTURE OVERVIEW

Fig. 2 outlines the structure of the CCA Attester according to the conceptual model described in Section II-A. It comprises two separate but bound Attesters, namely Platform attester and Realm attester. Each Attester has its own cryptographic identity – the CCA Platform Attestation Key (CPAK) and the Realm Attestation Key (RAK), respectively – which are used to sign the corresponding Evidence. The RAK is generated and attested by the Platform Attester, specifically within the HES. The key attestation of the RAK, which is carried inside the Platform Evidence, is used as the explicit delegation mechanism between the two Attesters, ensuring that a Verifier can always verify their correct coupling in the produced Evidence. In a nutshell, CCA Attester is a Composite Attester with the Realm Attester as the lead Attester. Full details of the Evidence produced by each Attester can be found in section VI7.2 of the official specification [10].

1) PLATFORM ATTESTER

As shown in Fig. 2, the Platform Attester consists of HES as the Attesting Environment and *Monitor Security Domain* as the Target Environment. The Claims in Platform Evidence include the boot state of the system (including all firmware components and configuration within the TCB), the relevant CCA parameters (e.g., the CCA platform implementation identifier), and the security lifecycle state of the platform. Additionally, the hash of the RAK public key is included in the attested Claims, thus making the Platform Evidence output from HES a combination of platform and key attestation.

The HES generally works with the host bootloader to obtain measurements of firmware components and meta-data about those components. In many implementations, HES will trust the bootloader to correctly compute the measurements and pass that state to the HES. However, some HES implementations may perform the measurement computations themselves based on a memory range passed by the bootloader.

2) REALM ATTESTER

As shown in Fig. 2, the Realm Attester consists of RMM as the Attesting Environment and an instance of Realm as the Target Environment. The Claims in Realm Evidence include the measured state at the boot of the Realm requesting the attestation – named the *Realm Initial Measurement* (RIM) – and a bank of *Realm Extended Measurements* (REM). RIM is calculated during Realm preparation. As memory is added to the Realm, it is hashed, and the resulting values are extended into the RIM. Once the Realm is activated, the RIM becomes immutable. Then, Realm software can use the REM bank for storing any kind of run-time state that may be relevant to a relying party interacting with the application. Measurements are applied to a REM in an extend operation available via an ABI exposed by the RMM to Realm software. Each Realm has a bank of 4 REMs in the CCA reference implementation.

B. FORMALIZATION OF PROVISIONING PHASE

As introduced in Section II-B, Provisioning encompasses Attester Provisioning as well as Verifier Provisioning. In this section, we explain the formalization of both for Arm CCA.

1) ATTESTER PROVISIONING

The lifecycle state of the CCA platform, *pLifeCycleState*, can be one of the seven possible values in Fig. 3 [10]. During manufacturing, in the *Platform Root of Trust (RoT) Provisioning* lifecycle state, a set of immutable parameters will be provisioned to a shielded location (i.e., on-chip tamper-resistant, non-volatile storage) only available to HES. For attestation, this includes:

1) the *Group Unique Key* (GUK): a random seed shared by a group of hardware-equivalent CCA instances and
2) a *Hardware Unique Key* (HUK): a randomly unique seed for an instance.

These parameters remain immutable for a CCA system in the *secured* lifecycle state. In our formalization, CCA Attester provisioning is modelled as freshly generated keys *guk* and *huk*, representing GUK and HUK, respectively. These keys are then passed only to the *Hes* subprocess, representing that the adversary and other subprocesses cannot access these keys.

2) VERIFIER PROVISIONING

As a prerequisite for verification of the Platform Evidence, the supply chain that builds the CCA implementation needs to convey, over a mutually authenticated channel, Reference Values and Endorsements to the Platform Verifier. The Reference Values include: 1) platform configuration *pConfig* and 2) measurement value for the platform boot state pSwCompMeasRef. The foundational Endorsement of any CCA platform is the public part *pubCpak* of the derived CPAK (see Section III-C1). Other endorsed values that could be linked to a CCA platform and/or Realm include security certifications, controls and benchmarks. These are typically associated with specific states of the installed firmware, software and related configuration, which are discovered through Evidence verification.

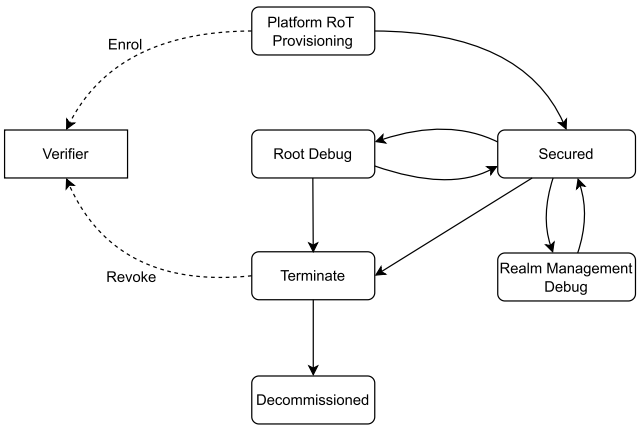

FIGURE 3. Arm CCA security life cycle [35] where boxes with rounded edges represent lifecycle states. The seventh state *unknown* is not shown.

Arm does not specify or provide any recommendation on how this mutually authenticated channel is created. So we abstract this and formally model the Reference Values as free names in ProVerif, so that *Verifier* subprocess can access these values. For Endorsements, we assume that the Verifier is already provisioned with *pubCpak*.

C. FORMALIZATION OF INITIALIZATION PHASE

The initialization phase basically provisions RMM with its signing key RAK. This section describes the derivation of keys. The formal model of the initialization phase in ProVerif consists of a parallel composition of 3 subprocesses:

$$Hes\,(privCpak, guk, pLifeCycleState) \mid RmmInit() \mid \mathbf{in}(\mathsf{cRmmInitMain}, (privRak, pubRak, pEvidence)); \quad (1)$$

where first subprocess *Hes* represents HES and | represents the parallel composition of subprocesses. To faithfully represent the protocol, we use a second subprocess *RmmInit* to represent the initialization part of RMM. However, this raises a problem of how to transport the parameters *privRak*, *pubRak*, and *pEvidence* to the RMM in the attestation protocol phase. We use the third subprocess to transport these parameters to the main process via a secure channel cRmmInitMain. The ProVerif keyword **in** represents message input.

1) CPAK DERIVATION

At boot, HES derives the CPAK, an asymmetric key pair that provides the Platform RoT identity. An important property of the CPAK is that its private part is guaranteed never to leave the HES. At a minimum, CPAK is derived from the following contributions:

1) The GUK (as explained in Section III-B). Note that implementations may choose alternative schemes for the derivation, for example, using device-unique parameters.
2) Current security lifecycle state of the platform. The security lifecycle is managed entirely by HES. Most systems will remain in the *secured* state.

In our formalization, the private part of CPAK is derived via a constructor kdfCpak with parameters *guk* and *pLifeCycleState*. This key derivation is done in the main process so that public part of CPAK can be directly passed to *Verifier* subprocess. The private part is passed only to *Hes* subprocess (as shown in Eq. (1)). The KDF parameters can be updated accordingly to formally verify a specific implementation.

A direct consequence of the key derivation strategy described above is that, if desired, the same CPAK can be shared by multiple systems with the same CCA hardware implementation. Also, because of the dependency of CPAK on the security lifecycle state, a different CPAK is derived if the state is not *secured*. This has an important consequence on the appraisal phase because only an attestation generated by a device in the *secured* state can be considered trustworthy.

2) RAK DERIVATION AND ATTESTATION

Fig. 4 describes the process by which, at boot, the RMM obtains its RAK. RMM starts by requesting HES via the Monitor, which will only accept this request from Realm world EL2. HES typically derives the RAK using the following contributions (step 1 in Fig. 4):

1) GUK;
2) Boot state of the system;
3) Current security lifecycle state of the platform.

A different RAK is therefore derived if the boot state of CCA platform firmware is updated or if the security lifecycle state is not *secured*. Formally, the private part of RAK is derived inside the *Hes* subprocess via a constructor kdfRak with parameters *guk*, *pSwComp* and *pLifeCycleState*. The KDF parameters can be updated accordingly to formally verify a specific implementation.

After deriving the RAK, HES queries data from the Target Environment – Monitor Security Domain as shown in Fig. 2 – and populates the corresponding *Entity Attestation Token* (EAT) [36] Claims. The format used is a variant of the PSA token [37] containing the platform boot state *pSwComp*, the security lifecycle state *pLifeCycleState*, as well the CCA platform implementation identifier(s) *pImpId*. Formally, these values are obtained as input in the *Hes* subprocess. In addition, the RAK is attested by hashing its public part and storing it into the *nonce* Claim *pChallenge*. In our formalization, the platform Claims *pClaims* is represented as:

$$\begin{aligned} pClaims = {} & pProfile \,||\, pChallenge \,||\, pImpId \,||\\ & pInstId \,||\, pConfig \,||\, pLifeCycle \,||\\ & pSwComp \,||\, pVer \,||\, pHashId \end{aligned}$$

where || represents concatenation and other symbols are as defined in Table 2.

The Claims are then signed with the CPAK private key (step 2 in Fig. 4) and wrapped in a *CBOR Web Token*

TABLE 2. CCA platform claims.

Symbol	Description
pProfile	Profile claim
pChallenge	Challenge claim
pImpId	Implementation ID claim
pInstId	Instance ID claim
pConfig	Config claim
pLifeCycle	Lifecycle claim
pSwComp	Software components claim
pVer	Verification service claim (optional)
pHashId	Hash algorithm ID claim

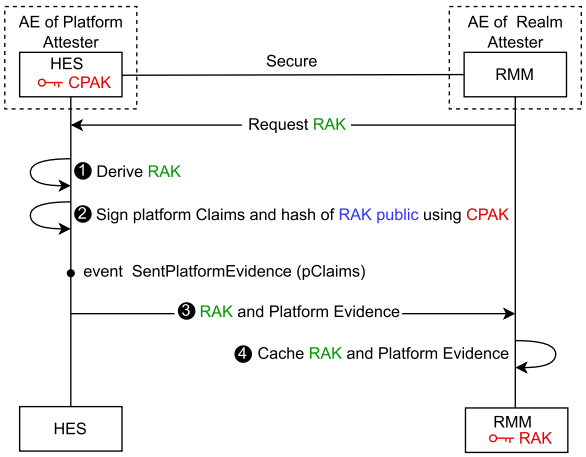

FIGURE 4. RAK derivation and attestation in the initialization phase. Keys within the rectangles at the top and bottom represent initial and final knowledge, respectively. RMM is provisioned with a signing key at the end of the protocol. AE represents attesting environment.

(CWT) [38] that serializes the combined Platform and RAK attestation. Finally, the computed Evidence is returned to RMM along with the RAK (step 3 in Fig. 4). RMM caches the RAK and the Platform Evidence for subsequent use (step 4 in Fig. 4). In our formalization, the steps of HES and RMM are modeled in the subprocesses *Hes* and *RmmInit*, respectively.

Note that because the RAK is exported from the HES, its private part could be exposed by a side-channel in the *application processor* or a vulnerability in the Monitor and/or RMM firmware. Therefore, CPAK private key, guaranteed never to leave the separated HES, is generally more secure than RAK private key. The delegated design trades a slight decrement in security with an increase in performance by signing directly at RMM without requiring a roundtrip to HES [35]. This is useful in deployments – especially in a heavily multi-tenant scenario, e.g., a cloud provider data centre – where high contention on the HES would create a system bottleneck. An implementation may choose to refresh the RAK on some event other than boot or use a design that obtains Platform Evidence directly from the HES for every Realm attestation request.

D. FORMALIZATION OF ATTESTATION PROTOCOL PHASE

This phase assumes the initialization described in Section III-C2 to be successfully completed so that the Platform Evidence is readily available to the RMM, along with the signing key RAK. The formal model in ProVerif consists of a parallel composition of three subprocesses *Rmm*, *Realm* and *Verifier* corresponding to the principals RMM, Realm and Verifier, respectively, as shown in Fig. 5, i.e.,

$$!(Rmm(privRak, pubRak, pEvidence) \mid Realm()) \mid (!Verifier(pubCpak))$$

where ! denotes unbounded replication, *privRak* represents private part of RAK, *pubRak* represents public part of RAK, *pEvidence* represents platform Evidence, and *pubCpak* represents public part of CPAK. We faithfully model all steps shown in the figure, as described below.

TABLE 3. Realm claims.

Symbol	Description
rChallenge	Challenge claim
rpv	Personalization Value
rim	Initial Measurement claim
rem	Extensible Measurements claim
rHashAlgoId	Hash algorithm ID claim
pubRak	Public key claim
rRakHashAlgoId	Public key hash algorithm identifier claim

1) GENERATION OF EVIDENCE

The first six steps in Fig. 5 describe the sequence of events involved in Evidence generation in a CCA system. We assume a challenge-response interaction between a Verifier (the challenger) and the Arm CCA Attester (the responder) that triggers the attestation request from the Realm to the RMM. The Verifier sends a nonce as a challenge to the Realm (step 1 in Fig. 5). The Realm to RMM attestation request (step 2 in Fig. 5) is made via an ABI call, assumed to be immune to interposition. This assumption is formally modelled as a secure channel between RMM and Realm.

When requested to do an attestation, the RMM queries its Target Environment and populates the corresponding EAT Claims with the sampled values. These include the Realm's boot state *rim*, the Realm's extended measurements *rem* and the RAK public key *pubRak*. In addition, the challenge coming from the Verifier via Realm is stored in the Realm challenge claim *rChallenge*. The Realm Claims *rClaims* is modelled as:

$$rClaims = rChallenge \mid\mid rpv \mid\mid rim \mid\mid rem \mid\mid rHashAlgoId \mid\mid pubRak \mid\mid rRakHashAlgoId$$

where symbols are as described in Table 3. In addition, the challenge coming from the Verifier is stored in the *nonce* claim.

The Realm Claims are then signed with the RAK private key (step 4 in Fig. 5) and wrapped in a CWT that serializes the Realm Evidence. Subsequently, the Platform Evidence previously computed together with the newly created Realm Evidence are assembled into an EAT Collection payload [39]. The bundled Evidence is returned to the requesting Realm

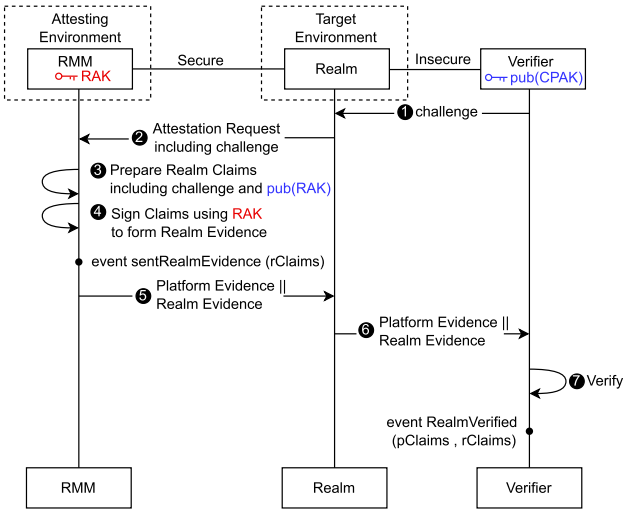

FIGURE 5. Arm CCA evidence generation and appraisal.

(step 5 in Fig. 5), which can then forward it to the Verifier (step 6 in Fig. 5).

2) APPRAISAL OF EVIDENCE

None of the Arm specifications describes the Appraisal of Evidence (step 7 in Fig. 5). Hence, we explain it in detail. Successful appraisal of CCA Evidence allows the Verifier to gain confidence that the Attester implements the *Arm CCA security guarantee* [35], i.e., that the Realm's memory contents and execution context can neither be accessed nor modified by other Realms or any non-CCA software or hardware. Verifiers can be split or integrated. It can be seen as integrated from a high level, so we formalize integrated verifiers.

a: SPLIT VERIFIERS

Typically, two separate supply chains are involved in the appraisal of CCA Evidence: one associated with manufacturing the CCA system and another with developing the Realm application. We separate the verification procedure along these two axes, which also map naturally to the split between the Platform and Realm parts of the CCA Evidence.

Platform Evidence Verification: The procedure to verify CCA Platform Evidence is as follows:

1) Extract CCA platform implementation identifier *pImpId* from the Platform Claims (note that the use of a not-yet-authenticated claim is not a problem in this case);
2) Extract CCA platform instance identifier *pInstId* from the Platform Claims;
3) Use the CCA platform implementation identifier *pImpId* and instance identifier *pInstId* to look up the associated CPAK public key *pubCpak*;
4) Use the found CPAK public key *pubCpak* to verify the signature over the Platform Claims;
5) Extract the lifecycle state claim *pLifeCycleState* and check its value is *secured* (0×30);
6) Use the CCA platform implementation identifier *pImpId* to look up the Reference Values for measurement value pSwCompMeasRef[2] and signer ID pSwCompSignerIdRef for the platform boot state;
7) Extract the platform boot state *pSwCompMeas* and *pSwCompSignerId* from the Platform Claims and check its value matches the corresponding Reference Values pSwCompMeasRef and pSwCompSignerIdRef;
8) Extract the platform configuration *pConfig* from the Platform Claims and check its value matches the Reference Value(s) pConfigRef.

Since concrete values (such as 0×30 in step 5) cannot be modelled in symbolic analysis, we abstract such values via free names, e.g., pLifeCycleStateRef for step 5. The *Verifier* subprocess can then compare the received Claim with pLifeCycleStateRef for an exact match. Steps 6, 7 and 8 are implemented similarly. However, because of an absence of clear specifications, we do not model the precise association of CCA platform implementation identifier *pImpId* and instance identifier *pInstId* with *pubCpak* (step 3) and Reference Values (step 6).

The Platform Verifier is also responsible for checking the correctness of the binding between Realm and Platform Evidence. The procedure to verify the binding is as follows:

1) Extract the RAK public key *pubRak* from the Realm Claims;
2) Extract the *nonce* claim *pChallenge* from the Platform Claims;
3) Hash the RAK public key and ensure it matches the *nonce* claim value *pChallenge* from step 2;
4) Use the RAK public key *pubRak* to verify the CWT signature over the Realm Claims.

If any of the steps above – platform verification and binding check – fail, the Platform Evidence is not verified, and the system must not be considered trustworthy. Otherwise, the Platform Evidence is verified, and the Verifier can trust the system to implement the CCA security guarantee, particularly for the Realm it interacts with. The Verifier can then continue with a further appraisal of the Realm Evidence.

Realm Evidence Verification: To fully verify the Realm Evidence, the organization that develops the Realm application needs to convey, over a mutually authenticated channel, the following information to the Realm Verifier: 1) Reference Value(s) for the Realm boot state, in particular, the RIM rimRef, and 2) (optionally) Reference Values remRef for the REMs. The procedure to verify Realm Evidence is as follows:

1) Extract the RAK public key *pubRak* from the Realm Claims;
2) Use the RAK public key *pubRak* to verify the CWT signature over the Realm Claims;
3) Check freshness (received challenge *rChallenge* matches the one sent by the Verifier);

[2]We use the standard ProVerif fonts for representation of constants, channels and Reference Values to distinguish them from variables.

4) Extract the Realm initial measurements *rim* from the Realm Claims and check its value matches the Reference Value(s) rimRef;
5) Optionally based on policy, extract the Realm extended measurements *rem* from the Realm Claims and check their value match the Reference Value(s) remRef;
6) Optionally based on policy, extract the Realm personalization value *rpv* from the Realm Claims and check their value match the Reference Value(s) rpvRef.

Note that steps 1 and 2 are also part of the binding checks made by the Platform Verifier and can be avoided here if the Realm and Platform Verifiers are integrated (see Section III-D2b). In some circumstances, the guest OS deployed within a Realm may be sourced from a different organization than the Realm application. This must be considered when sourcing the appropriate Reference Values.

b: INTEGRATED VERIFIERS

We present the algorithm when the Realm and Platform Verifiers are integrated. We use the symbols described in Tables 2 and 3. The suffix *_Ver* represents the view of *Verifier* subprocess of a variable, and the suffix Ref represents the Reference Value of the variable. We categorize all the checks into four groups. The first group validates cryptography:

$$\begin{aligned}&(pImpId\_Ver = \mathsf{pImpIdRef}) \wedge (pInstId\_Ver = \mathsf{pInstIdRef})\\ &\wedge \mathsf{verifySign}(pubCpak, pClaims\_Ver, cpakSig\_Ver) = \mathsf{true}\\ &\wedge \mathsf{verifySign}(pubRak\_Ver, rClaims\_Ver, rakSig\_Ver) = \mathsf{true}\\ &\wedge pChallenge\_Ver = \mathsf{sha256}(pubRak\_Ver)\\ &\wedge rChallenge\_Ver = challenge\end{aligned}$$

The second group is a check on the security lifecycle state:

$$pLifeCycleState\_Ver = \mathsf{pLifeCycleStateRef}$$

The third group checks five mandatory Reference Values:

$$\begin{aligned}&(pSwCompMeas\_Ver = \mathsf{pSwCompMeasRef})\\ &\quad \wedge pSwCompSignerId\_Ver = \mathsf{pSwCompSignerIdRef}\\ &\quad \wedge pConfig\_Ver = \mathsf{pConfigRef}\\ &\quad \wedge pProfile\_Ver = \mathsf{pProfileRef}\\ &\quad \wedge rim\_Ver = \mathsf{rimRef}\end{aligned}$$

The last group checks the optional Reference Values and dependent on a policy:

$$\begin{aligned}&(rpv\_Ver = \mathsf{rpvRef}) \wedge (rem\_Ver = \mathsf{remRef})\\ &\qquad\qquad \wedge (pVer\_Ver = \mathsf{pVerRef})\end{aligned}$$

If all the checks pass, we define an event:

$$\textbf{event}\ \mathsf{RealmVerified}(pClaims\_Ver, rClaims\_Ver)$$

which is a key event for checking security properties, such as integrity.

E. USE CASE

Architecturally-defined attestation is one of the most fundamental mechanisms of Confidential Computing. Hence, our work serves as the formal foundation for all use cases of Arm CCA. For instance, the presented attestation protocol can be composed with authentication and key exchange protocols. We believe our formalization can be extended to cover a Confidential Computing Consortium project[3] on attested TLS, allowing a generic way of passing Evidence and Attestation Results in the TLS handshake [23]. From the attestation side, it requires only minor changes in Fig. 4 and our artifacts:

1) Before step 1: *Realm* subprocess generates an ephemeral key-pair called Realm application identity key (RIK).
2) In step 2, in addition to the challenge, RIK public part represented by *pubRik* is also sent. The combination is called Key Attestation Token (represented by *kat*), i.e., $kat = challenge \,||\, pubRik$.
3) In step 3, $rChallenge = challenge$ is replaced by $rChallenge = hash(kat)$.
4) In step 6, *kat* is sent as a prefix to Platform Attestation Token (combination of Platform Evidence and Realm Evidence).

F. THREAT MODEL/ADVERSARY CAPABILITIES

Following the outline in Section II-C, the threat model used in ProVerif for attestation in Arm CCA is defined as follows: 1) Entities: The participants HES, RMM, and Verifier are honest, whereas Realm can be both honest and malicious. These are reasonable assumptions because HES is the RoT, which by definition must be trusted, and RMM is part of the system's TCB. The Verifier's honesty must be assumed, given its role as a Trusted Third Party. 2) Channels: All the channels are public apart from the channel between a) HES and RMM (Fig. 4) for transporting the RAK key pair, and b) RMM and Realm (Fig. 5) as ABI calls between RMM and Realm are assumed to be immune to interposition. In both cases, the authentication and confidentiality properties of the channel are provided architecturally. Since the *Secure Monitor Call* (SMC) instruction is guaranteed to trap to a higher Exception level, the SMC that the RMM at EL2 executes to obtain the RAK is guaranteed to be handled by EL3, which the RMM implicitly trusts. The EL3 Monitor then forwards the RMM's request to the HES. Similarly, any SMC (therefore, any *Realm Service Interface* command) executed by a Realm at EL1 is guaranteed to trap to the RMM at EL2. Note, however, that the portion of the channel connecting the EL3 Monitor (i.e., the trusted proxy) and HES is implementation-dependent. 3) Functions: All functions are available to the adversary. 4) Technology-specific capabilities: Realm measurements are taken as input from the adversary to allow her to create a Realm of any desired measurements, i.e., potentially creating fake Realms. 5) Modeling assumptions: Endorsement, namely Public part of CPAK, is pre-configured

[3] https://github.com/ccc-attestation/attested-tls-poc

TABLE 4. Summary of verification results for Arm CCA.

Attester	Integrity	Freshness	Confidentiality	Authentication
Platform	✓	×	✓	-
Realm	✓	✓	✓	×

in the Verifier. Reference Values are modelled as free names (cf. Sections III-B2 and III-D2a).

G. PROPERTIES

In this section, we formalize the four properties outlined in Section II-C for Arm CCA, where a summary is presented in Table 4. The verification uses ProVerif version 2.04 on Ubuntu 20.04 LTS on an Intel Core i7-11800H processor with 64 GB of RAM. The average verification time to prove all the properties in this experimental setup is 111 s. In fact, except for the freshness of Platform Evidence, all other properties are verified within a second.

1) INTEGRITY OF PLATFORM EVIDENCE AND REALM EVIDENCE

Since Arm CCA Attester is composite (Fig. 2), the integrity of Platform Claims *pClaims* as well as Realm Claims *rClaims* must be ensured. This can be captured by two separate queries. We formalize the integrity check for Platform Evidence in ProVerif as follows:

query *pClaims*, *rClaims*: bitstring;
event (RealmVerified(*pClaims*, *rClaims*))
$\Longrightarrow$ **event** (SentPlatformEvidence(*pClaims*)).

where **query**, **event** and bitstring are ProVerif keywords for checking properties, non-injective correspondence assertion and predefined bitstring datatype, respectively. As shown in Fig. 4, event SentPlatformEvidence is placed in the *Hes* subprocess just before sending out the Platform Evidence. Moreover, event RealmVerified is placed in the *Verifier* subprocess after finishing all the verification steps mentioned in Section III-D2 (Fig. 5). Then, the above query asserts whether *pClaims* remain unmodified during transport from the *Hes* subprocess up to the *Verifier* subprocess, i.e., *pClaims* is the variable of agreement. ProVerif confirms that the query holds, meaning that the protocol provides integrity of Platform Claims to the Verifier. We also analyze the reachability of event RealmVerified in ProVerif to ensure it is indeed reachable.

In a similar fashion, we formalize the integrity of Realm Evidence as follows:

query *pClaims*, *rClaims*: bitstring;
event (RealmVerified(*pClaims*, *rClaims*))
$\Longrightarrow$ **event** (SentRealmEvidence(*rClaims*)).

where event SentRealmEvidence is placed in the *Rmm* subprocess just before sending out the Realm Evidence, as shown in Fig. 5. In this query, *rClaims* is the variable of agreement. ProVerif confirms that the query holds, meaning that the protocol provides integrity of Realm Claims to the Verifier.

2) FRESHNESS OF PLATFORM EVIDENCE AND REALM EVIDENCE

Similar to integrity, freshness of Platform and Realm Evidences can be checked separately. The one for the platform is as follows:

query *pClaims*, *rClaims*: bitstring;
event (RealmVerified(*pClaims*, *rClaims*))
$\Longrightarrow$ **inj−event** (SentPlatformEvidence(*pClaims*)).

where **inj−event** is ProVerif keyword for injective correspondence assertion, and all other symbols are as already explained in the integrity check. The property we are checking here is strictly stronger than the one in integrity, that is, whether each RealmVerified event corresponds to a *unique* SentPlatformEvidence event. So, it checks freshness in addition to integrity. ProVerif confirms that the query does not hold, meaning that the protocol does not provide freshness of Platform Claims to the Verifier. This is because, by design, the same Platform Evidence is sent in each attestation request, and there is no fresh value to allow uniqueness. Note that this is not a problem as long as the platform TCB does not change – for example, because of a live update, or if a different security-lifecycle state (e.g., debug) is entered. The current reference version of CCA does not support live update.

Similarly, the freshness check for Realm Evidence is formalized as:

query *pClaims*, *rClaims*: bitstring;
event (RealmVerified(*pClaims*, *rClaims*))
$\Longrightarrow$ **inj−event** (SentRealmEvidence(*rClaims*)).

ProVerif confirms that the query holds, meaning that the protocol provides freshness of Realm Claims to the Verifier.

3) SECRECY

For completeness, we also analyze the secrecy properties for CPAK and RAK private keys, represented by *privCpak* and *privRak*, respectively, by the following two ProVerif queries:

query secret *privCpak*.
query secret *privRak*.

where **secret** is a ProVerif keyword for checking the secrecy of a bound name or variable. Note that the ProVerif keyword **attacker** cannot be used because *privCpak* and *privRak* are not free names in the formal model. The secrecy queries hold trivially as these keys are not sent on a public channel; indeed, ProVerif confirms that.

4) AUTHENTICATION

Finally, we formalize the authentication property for the model with the changes described in Section III-E as

follows:

$$\begin{aligned}&\textbf{query } pubRIK\_Ver\text{: VerifyingKey};\\&(\textbf{event } (\mathsf{VerIdentity}(pubRIK\_Ver))\\&\wedge \textbf{event } (\mathsf{RealmIdentity}(pubRIK)))\\&\Longrightarrow (pubRIK = pubRIK\_Ver).\end{aligned}$$

where the events RealmIdentity and VerIdentity are placed after generation of RIK in *Realm* subprocess, and after complete verification of Remote Evidence in *Verifier* subprocess, respectively. ProVerif confirms that the query does not hold, meaning that the attestation protocol (without authentication and key exchange protocols) does not provide server authentication. The counterexample shows two different sessions of Realm, each generating its own ephemeral key pair RIK. The attacker sends the Evidence from one of the sessions of Realm to the Verifier, and then after the Verification, it causes a mismatch with the public key of other session, failing the property. Hence, when authentication is desired, architecturally-defined attestation must be composed with some form of authentication and key exchange protocol (such as TLS).

H. AMBIGUITIES IN ARM SPECIFICATIONS

While developing this formalization, we also spotted a few ambiguities in the Arm specifications, which we have reported already. Here is a summary:

1) The precise meaning of *boot state of the system* (as used in Sections III-A1 and III-C2 in unclear. This is important because it is one of the inputs to the key derivation function of RAK. We define it as equivalent to *pSwComp*.
2) The precise meaning of *platform boot state* (Sections III-B2,III-C2 and III-D2) is unclear. We define it as equivalent to the combination of mandatory fields – *pSwCompMeas* and *pSwCompSignerId* – in *pSwComp*.
3) The precise meaning of *Realm boot state* (Section III-D2) is unclear. We define it as equivalent to the combination of *rim* and *rpv*.

IV. ATTESTATION IN INTEL TDX

Intel announced new extensions for its Instruction Set Architecture (ISA), namely TDX [11] in 2020. These extensions combine Intel Virtual Machine Extensions (VMX), Intel Multi-Key Total Memory Encryption (MKTME), and Intel Central Processing Unit (CPU)-attested software module. Intel TDX inherits some aspects of the well-known Intel SGX. The primary difference is the granularity of memory protection, i.e., in contrast to the process-based nature of Intel SGX, Intel TDX is a VM-based TEE. Intel TDX architecture provides a new CPU mode, namely Secure-Arbitration Mode (SEAM), in which an isolated software module – TDX Module – at ring -1 (hypervisor level) facilitates the operation and management of the protected VMs, namely Trust Domains (TDs) [40].

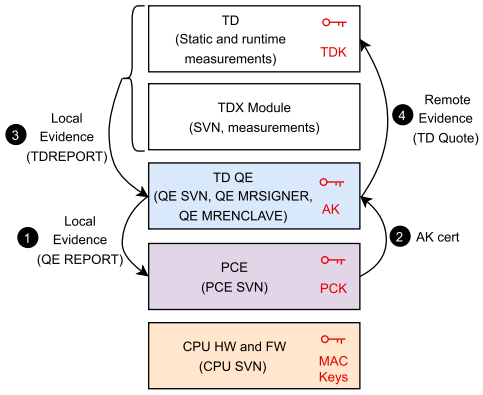

FIGURE 6. Overview of layered attester in Intel TDX showing entities involved, local evidence, important Claims, and cryptographic keys (steps 1 and 2 are part of the initialization phase while steps 3 and 4 are part of the attestation protocol phase).

A. ATTESTATION ARCHITECTURE OVERVIEW

Fig. 6 depicts an overview of Attester in Intel TDX. TD Quoting Enclave (QE) is an architectural enclave responsible for signing the Claims with Attestation Key (AK) and generating Remote Evidence (TD Quote). Provisioning Certification Enclave (PCE) – with its signing key Provisioning Certification Key (PCK) – is another architectural enclave and serves as the local CA of TD QE. CPU hardware (HW) and Firmware (FW) protect MAC keys for local attestation.

In a nutshell, the initialization and the generation of Remote Evidence consist of local attestation. In the initialization phase, TD QE presents its Local Evidence (QE REPORT) to PCE (step 1 in Fig. 6), and PCE verifies the co-location of TD QE resulting in an Attestation Key certificate (AK cert) (step 2). In the generation of Remote Evidence, TD and TDX Module present their Local Evidence (TDREPORT) to TD QE (step 3), and TD QE verifies the co-location of TD and TDX Module, resulting in the TD Quote (step 4), which is the Remote Evidence. The authentication of the Local Evidence in both cases is done via *Message Authentication Code* (MAC). The important Claims, such as measurements and *Security Version Numbers* (SVNs), corresponding to each entity are also shown in Fig. 6. The Remote Evidence sent to the Verifier includes signed Claims and a certificate (cert) chain.

To verify Remote Evidence, the Remote Verifier (cf. Fig. 1) may obtain Endorsements – certs and Certificate Revocation Lists (CRLs) – from the Endorser (Intel), Reference Values – Trusted Computing Base (TCB) Info, QE Identity and TD measurements – from Reference Value Providers (Intel, QE owner and TD owner, respectively), and Appraisal Policy for Remote Evidence from the Verifier Owner (TD owner). The Remote Verifier then appraises the Remote Evidence.

B. FORMALIZATION OF PROVISIONING PHASE

1) ATTESTER PROVISIONING

At the time of writing, the provisioning phase for Intel TDX is not specified by Intel in any of its public specification

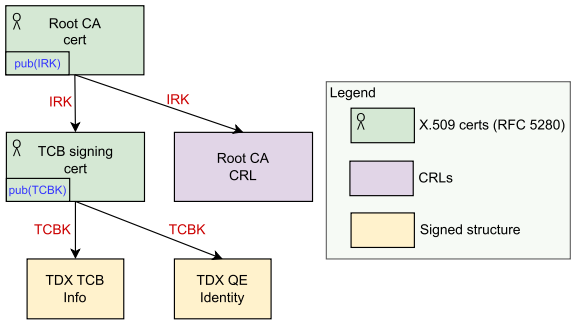

FIGURE 7. Chain of trust for reference values in Intel TDX.

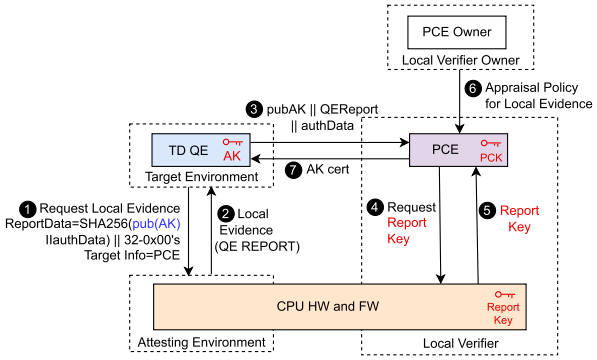

FIGURE 8. SGX local attestation of TD QE and PCE in the initialization phase (Attester consists of attesting environment and target environment).

documents [41]. We believe specification exists internally in Intel.

2) VERIFIER PROVISIONING

Intel has not explicitly presented the complete chain of trust for provisioning Reference Values in Intel TDX. We thoroughly examined the specifications and summarized our understanding in Fig. 7. TDX TCB Info structure contains SVN of SGX TCB components, PCE and TDX TCB components [42]. TDX QE Identity structure contains the enclave identity of TDX QE. The keys on the arrows between the entities (certificates, CRLs and structures) represent the signing keys with which the entity below the arrow is signed. For instance, TDX TCB Info is signed using TCBK. For Endorsements, the Verifier is assumed to be pre-configured with Intel's root key.

C. FORMALIZATION OF INITIALIZATION PHASE

The initialization phase corresponds to steps 1 and 2 in Fig. 6, and more details are given in Fig. 8. This phase aims at issuing an AK cert to TD QE. Our formal model of the initialization phase in ProVerif consists of a parallel composition of four subprocesses:

$$
\begin{aligned}
&PCE(c\_PCE\_CPUinit, PCK) \mid \\
&\quad QEinit(c\_QE\_CPUinit, c\_QE\_main, PCEinfo) \mid \\
&\quad CPUinit(c\_QE\_CPUinit, c\_PCE\_CPUinit, PCEinfo) \mid \\
&\quad \mathbf{in}(c\_QE\_main, (AK\text{: SigningKey}, AKcert\text{: bitstring}));
\end{aligned}
$$

where *PCE*, *QEinit* and *CPUinit* represent PCE, TD QE, and CPU HW and FW, respectively, as shown in Fig. 8. The fourth subprocess enables the transport of parameters *AK* and *AKcert* to the main process for the attestation protocol phase, as explained in Section III-C, using the secure channel c_QE_main. The channel between TD QE and CPU is represented by c_QE_CPUinit while the channel between PCE and CPU is represented by c_PCE_CPUinit. Both these channels are also secure.

As shown in Fig. 8, TD QE sends a Local Evidence request to CPU HW and FW by specifying (i) SHA256 hash of the public part of Attestation Key along with any authentication data as Report data *QErData*, i.e.,

$$QErData = SHA256(pub(AK) \mid\mid authData) \mid\mid zeroPad$$

where *pub*(.) represents the public key, *QErData* represents the additional user-provided authentication data that needs to be certified by the PCE, *zeroPad* represents 32 bytes of zeros, and (ii) PCE as the target enclave (step 1 in Fig. 8). Here, target enclave refers to the verifying enclave for which the Local Evidence is generated. CPU HW and FW collect Claims about TD QE and generate Local Evidence, namely QE Report, targeted for PCE. These Claims are protected via a MAC using Report Key. This Local Evidence is conveyed to TD QE (step 2). TD QE forwards this Local Evidence along with *pubAK* and *QErData* to PCE (step 3). Since the QE Report was targeted for PCE, now PCE can request Report Key from the CPU HW and FW (step 4). On receiving the Report Key (step 5), PCE verifies the Local Evidence based on the Appraisal Policy from the Local Verifier Owner (step 6). The policy typically involves verifying 1) MAC over QE Report Body; 2) hash of *pubAK* and *QErData*; and 3) attribute PROVISIONKEY. If the result of this appraisal is positive (i.e., MAC and hash are correct and PROVISIONKEY attribute is set to true), then the PCE signs the Claims to generate a so-called AK cert and sends it back to the TD QE (step 7).

1) STRUCTURE OF LOCAL EVIDENCE (QE REPORT)

At a high level, a QE Report consists of 1) Report body *QEReportBody*, 2) value for key wear-out protection *keyID*, and 3) MAC *QEmac* over Report body. The Report body *QEReportBody* consists of various fields, notably SVNs (ISVSVN, CPU SVN), enclave identities (MRENCLAVE, MRSIGNER, ISVPRODID), enclave attributes, and Report data *QErData*. SVNs track the security-related – in contrast to functionality-related – updates of the software. More specifically, inside SVNs, the field ISVSVN stands for *Independent Software Vendor SVN* of the enclave (in this case, QE) assigned by the enclave's signer, whereas CPU SVN represents the SVN of the processor. CPU SVN reflects the microcode update version and authenticated code modules supported by the processor. However, Intel does not provide the detailed structure of CPU SVN but only describes that it cannot be mathematically compared [43]. Among the enclave

identities, MRENCLAVE is the overall SHA256 hash of both the enclave code and data as it is loaded. MRSIGNER is the SHA256 hash of enclave's signer public key (corresponding to the 3072 bit RSA key with which enclave is signed) [44]. The signature is stored in the enclave signature structure, named SIGSTRUCT. ISVPRODID in the SIGSTRUCT is the product ID assigned by the enclave's signer. Enclave attributes in *QEReportBody* include the selection of different flags, e.g., DEBUG, which represents whether the enclave is in debug – vs. production – mode, and PROVISIONKEY, which represents whether Provisioning Key is available. The last part, Report data *QErData*, is a user-defined field of 64 bytes that the enclave owner may use to transmit data to other enclaves. Typically, Report data contains the public key or hash of public key appended with authentication data. AES128 CMAC is used for MAC *QEmac*.

2) KDF FOR LOCAL EVIDENCE GENERATION AND VERIFICATION

KDF for Report Key for Local Evidence Generation is based on three major components: 1) secret values from fuses, 2) CPU SVN, and 3) target enclave information. The dependence on secret values from fuses ensures that any other platform cannot generate the Report Key. The derivation based on CPU SVN ensures that if the hardware security version is different at the time of generation and appraisal of Local Evidence, the Report Key will not match at the target enclave. Target enclave information includes measurement (MRENCLAVE) and attributes of the target enclave. The use of a specific target (in this case, PCE) in the key derivation ensures that no other enclave can tamper with the authenticity of the report.

Similar to KDF for Local Evidence Generation, the KDF for Local Evidence Verification is also based on secret values from fuses as well as CPU SVN. The key difference is that instead of the dependence on target enclave information, the CPU HW and FW uses the information of the enclave calling the ENCLU[EGETKEY] instruction. Thus, if the PCE on the same platform (checked via dependence on fuses) calls this instruction and the platform is in the same security state (checked via CPU SVN), PCE would be able to get the symmetric key for MAC verification. Another main difference is that during generation, the key is only accessible by the CPU HW and FW, and not by the Target Environment (TD QE). In contrast, during the verification, the key is accessible by the PCE (cf. step 5 in Fig. 8).

3) STRUCTURE OF AK CERT

Intel's so-called AK cert is basically a cert-like structure identifying the QE and the Attestation Key [45]. Intel does not clearly specify the structure of the AK cert in the public documents. However, Intel mentions that the Quote includes AK cert [46]. Upon careful observation of the Quote, we found that, as shown in Fig. 9, it basically consists of: 1) the AK public part *pubAK*, 2) QE Report body *QEReportBody*, 3) signature *PCKsig* over the QE Report body using Provisioning Certification Key (PCK), and 4) QE authentication data *QErData*.

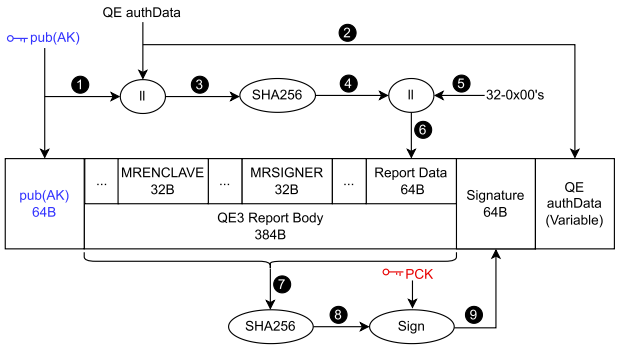

FIGURE 9. AK cert for DCAP (B represents the size in Bytes).

D. FORMALIZATION OF ATTESTATION PROTOCOL PHASE

The formal model of the attestation protocol phase in ProVerif consists of a parallel composition of five subprocesses *QE*, *TD*, *TDXModule*, *CPUHardware* and *Verifier* representing the principals TD QE, TD, TDX Module, CPU hardware and Verifier, respectively:

$$
\begin{aligned}
&!(\ \textbf{new}\ c\_QE\_CPU\colon \mathsf{channel};\ \ \textbf{new}\ c\_TDX\_CPU\colon \mathsf{channel};\\
&\quad \textbf{new}\ c\_TD\_TDX\colon \mathsf{channel};\\
&(QE(c\_QE\_CPU, AK, AKcert, PCKcert, ICAcert,\\
&\quad rootcert)\ |\\
&TD(c\_TD\_TDX)\ |\ TDXModule(c\_TD\_TDX, c\_TDX\_CPU)\ |\\
&CPUHardware(c\_QE\_CPU, c\_TDX\_CPU, MK)))\ |\\
&(!Verifier(\mathsf{pub}(IRK), sgxTcbRef, pceSvnRef,\\
&\quad tdxTcbSvnRef))
\end{aligned}
$$

where `c_QE_CPU` is the channel between QE and CPU, `c_TDX_CPU` is the channel between TDX Module and CPU, and `c_TD_TDX` is the channel between TD and TDX Module. These three channels are secured by the system-on-chip and therefore, these channels are modeled as private in ProVerif. In the formalization, we use `AK` for Attestation Key AK, `AKcert` for the AK cert, `PCKcert` for the PCK cert, `ICAcert` for the intermediate (processor/platform CA) cert, and `rootcert` for the root cert. Free names `sgxTcbRef`, `pceSvnRef`, and `tdxTcbSvnRef` represent Reference Values for SGX TCB components, PCE SVN, and TDX TCB components, respectively.

1) GENERATION OF REMOTE EVIDENCE

As depicted in Fig. 10, TD and TDX Module take on the role of Target Environment. TD sends a Local Evidence request to TDX Module (step 1 in Fig. 10) by specifying the typical usage of report data *rData*:

$$rData = SHA256(challenge)\ ||\ zeroPad \tag{2}$$

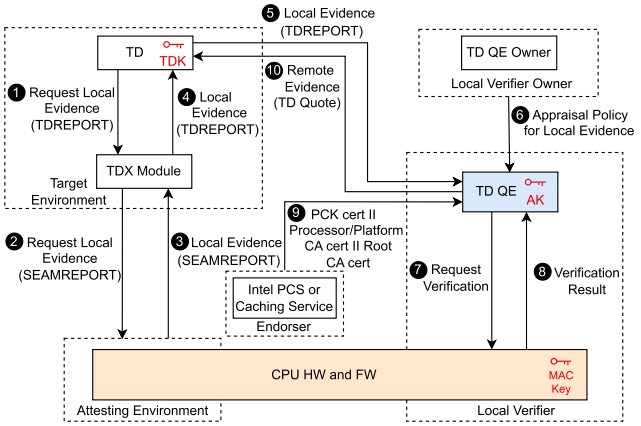

FIGURE 10. Local attestation of TD+TDX module and TD QE.

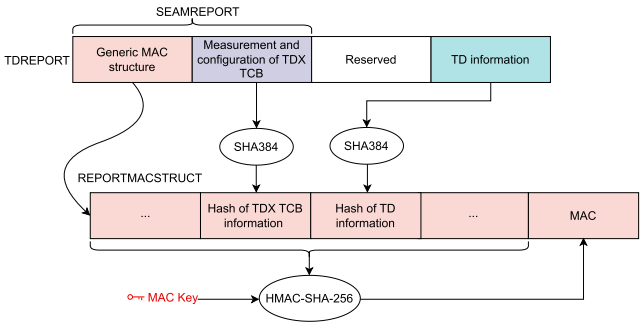

FIGURE 11. Simplified structure of local evidence (TDREPORT).

where *challenge* is the challenge sent by the challenger (Verifier). Other authentication data can be added in the hash as required for the specific use case. TDX Module, in turn, sends a Local Evidence request to CPU HW and FW (step 2). CPU HW and FW collect Claims about TD and TDX Module and generate Local Evidence (namely SEAMREPORT) protected via a MAC. CPU HW and FW send the Local Evidence (SEAMREPORT) to TDX Module (step 3), which appends TD information to form Local Evidence (TDREPORT). TDX Module forwards TDREPORT via TD (step 4) as well as untrusted host VMM (not shown in Fig. 10) to the TD QE (step 5). TD QE, together with CPU HW and FW, takes on the role of the Local Verifier. On getting the Appraisal Policy for Local Evidence (step 6), TD QE ensures that the hashes are correct and then asks the CPU HW and FW (step 7) to check the MAC in the generic MAC structure (REPORTMACSTRUCT). CPU HW and FW send the verification result back to the TD QE (step 8). TD QE fetches the PCK cert chain, i.e., $PCKcert||ICAcert||rootcert$, from Intel Provisioning Certification Service (PCS) or caching service (step 9). Then, the TD QE signs the Claims with AK to form a Remote Evidence (TD Quote). The TD Quote is conveyed via untrusted host VMM (not shown in Fig. 10) to TD (step 10). The TD then conveys the TD Quote as the final Remote Evidence from the platform to a Remote Verifier.

a: STRUCTURE OF LOCAL EVIDENCE (TDREPORT)

Local Evidence, as depicted in Fig. 11, consists of three major data structures: 1) measurement and configuration of TDX TCB *tcbi*, 2) TD information *tdi*, and 3) generic MAC structure *rms*. At the time of writing, the TDX TCB structure contains information about TDX Module only, so important fields of the TDX TCB include SVN *tsvn* and measurement *mrs* of TDX Module. Important fields of TD structure include its static *mrtd* and runtime *rtmr* measurements, TD attributes *tdatt*, and hash *mro* of ID of TD's owner. Important fields of the generic MAC structure include CPU SVN *csvn*, Report data *rdata*, SHA384 hashes of data structures of TDX TCB and TD, and MAC *mac* over all other fields of the generic MAC structure. HMAC-SHA-256 is used for MAC.

b: KDF FOR LOCAL EVIDENCE GENERATION AND VERIFICATION

KDF for the MAC key used in Fig. 11 is unspecified by Intel. We believe specification exists internally in Intel.

c: STRUCTURE OF REMOTE EVIDENCE (TD QUOTE)

At the time of initial writing, we did not find any detailed specification of the structure of Remote Evidence, which is the most important data structure in TD attestation. The only mention of contents of TD Quote in Intel's publicly available specifications is in a figure (cf. Fig. 14.1 in [40]) where TD Quote is shown to consist of type, CPU SVN, hash of TDX Module structure, hash of TD information, Report data, and signature. We have reported this to Intel.

2) APPRAISAL OF REMOTE EVIDENCE

The Remote Verifier may need access to three main artifacts for appraisal: 1) Endorsements, 2) Reference Values, and 3) Appraisal Policy for Remote Evidence.

a: ENDORSEMENTS

Endorsements in Intel TDX are mainly the certificate chain and Certificate Revocation Lists (CRLs). Intel has not publicly presented the complete chain of trust in one place in any of its specification documents. We thoroughly examined the specifications of Intel TDX, and our study result on the chain of trust along with the corresponding entities and CRLs for a single package platform in Intel TDX is presented in Fig. 12 [47]. The keys on the arrows between the certificates and CRLs represent the signing keys with which the certificate/CRL/structure is signed. CRLs consist of Root CA CRL, PCK platform CA CRL and PCK processor CA CRL, issued by Root CA, PCK platform CA and PCK processor CA, respectively.

b: REFERENCE VALUES

Reference Values in Intel TDX are mainly SVNs and measurements. As shown in Fig. 6, SVNs of several entities (including TDX Module, TD QE, PCE, CPU HW and FW)

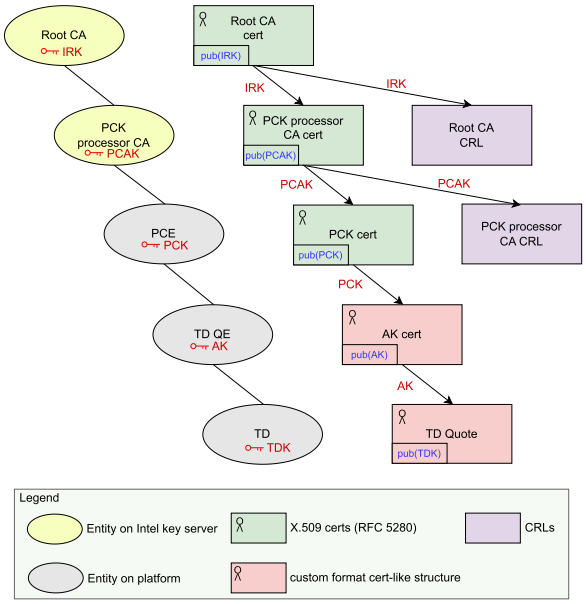

FIGURE 12. Chain of trust for single package platform in Intel TDX. For multi-package platforms, processor CA is replaced by platform CA.

are part of the Remote Evidence. Therefore, the Remote Verifier needs Reference Values corresponding to each of these SVNs. Reference Values for CPU SVN, PCE SVN and TDX Module SVN are obtained from Intel as explained in Section IV-B (cf. Fig. 7). QE SVN is provided by the TD QE Owner, which is Intel if Intel-provided QE is used or otherwise some non-Intel entity in the ecosystem that authored the QE. Similarly, the Remote Verifier may need Reference Values of measurements of TDX Module, TD QE and TD. They are provided by Intel, TD QE Owner and TD Owner, respectively.

c: APPRAISAL POLICY FOR REMOTE EVIDENCE

It consists of a set of rules with which the Remote Verifier can appraise the Remote Evidence. Specifically, it could define rules for comparison of the current state of the Attester – available as Claims in Evidence – to the reference state available as Reference Values. Examples include allow list, block list as well as upper and lower bounds. For TDX, a policy could require that SVNs in the Remote Evidence (TD Quote) are greater than or equal to the corresponding SVNs in the Reference Values, and that each measurement belongs to the corresponding set of accepted values. Additionally, the policy must ensure verification of the signature chain from Intel's root cert up to the Quote and check each cert against the list of revoked certificates – CRLs.

From a formalization perspective, a new aspect in Intel TDX compared to Arm CCA is that in addition to exact match, there are non-exact comparisons, i.e., greater than or equal to, for SVNs. We formally model these SVNs as natural numbers and obtain the Reference Values as input. The *Verifier* subprocess then checks $SVN >= \mathsf{SVNRef}$ where *SVN* represents the value in Claims and SVNRef represents its corresponding Reference Value.

E. USE CASE

Our work for Intel TDX serves as the formal foundation for all use cases of Intel TDX. For instance, the presented attestation protocol can be composed with authentication and key exchange protocols. Specifically, Intel is using our artifacts for the formal specification and verification of TDX boot flow for *virtual Trust Platform Module* (vTPM) TD using *Security Protocol and Data Model* (SPDM) protocol [48]. From the attestation side, it requires only minor changes in Fig. 10 and therefore our artifacts:

1) Before step 1: *TD* subprocess generates an ephemeral key-pair represented by TDK.
2) In step 1, in addition to the challenge, TDK public part represented by *pubTDK* is also sent, i.e., in Eq. 2, *challenge* is replaced by $pubTDK \| challenge$.

The complete flow, including both initialization and attestation protocol phases, is shown in Fig. 13.

F. THREAT MODEL/ADVERSARY CAPABILITIES

Following the outline in Section II-C, the threat model used in ProVerif for attestation in Intel TDX is defined as follows: 1) Entities: In our formalization, we assume that the principals TD QE, CPU HW, TDX Module and Verifier are honest. TD can be both honest and malicious. We also model PCE as an honest principal. To evaluate Intel's claim that PCE is out of TCB, we purposely leak out the corresponding key *PCK*. This modeling choice provides a neat way to evaluate all properties for both cases – when PCE is trusted and untrusted – simply by uncommenting and commenting that line of code representing key leakage to the adversary. 2) Channels: The following channels are assumed to be secure because the System on Chip (SoC) ensures this: (i) TD and TDX Module, (ii) TDX Module and CPU Hardware, (iii) TD QE and CPU Hardware, and (iv) PCE and CPU Hardware. All other channels are public. 3) Functions: All functions are available to the adversary. 4) Technology-specific capabilities: TD measurements are taken as input from the adversary to allow the adversary to create a TD of any desired measurements, i.e., to potentially create fake TDs. 5) Modeling assumptions: Public part of Intel's root key is pre-configured in the Verifier.

G. PROPERTIES

At the time of this research, Intel used to claim in its white paper [11] that the *only* components trusted by the TD are: 1) Intel TDX Module, 2) Intel Authenticated Code Modules (ACM), 3) TD QE, and 4) Intel CPU hardware, by explicitly mentioning that *all other software* is untrusted by TD (cf. Fig. 5.1 in [11]). Since remote attestation is the fundamental characteristic of a TEE and one of the five major capabilities of TDX mentioned in the white paper [11], Intel's claim of TCB must be satisfied for the attestation mechanism.

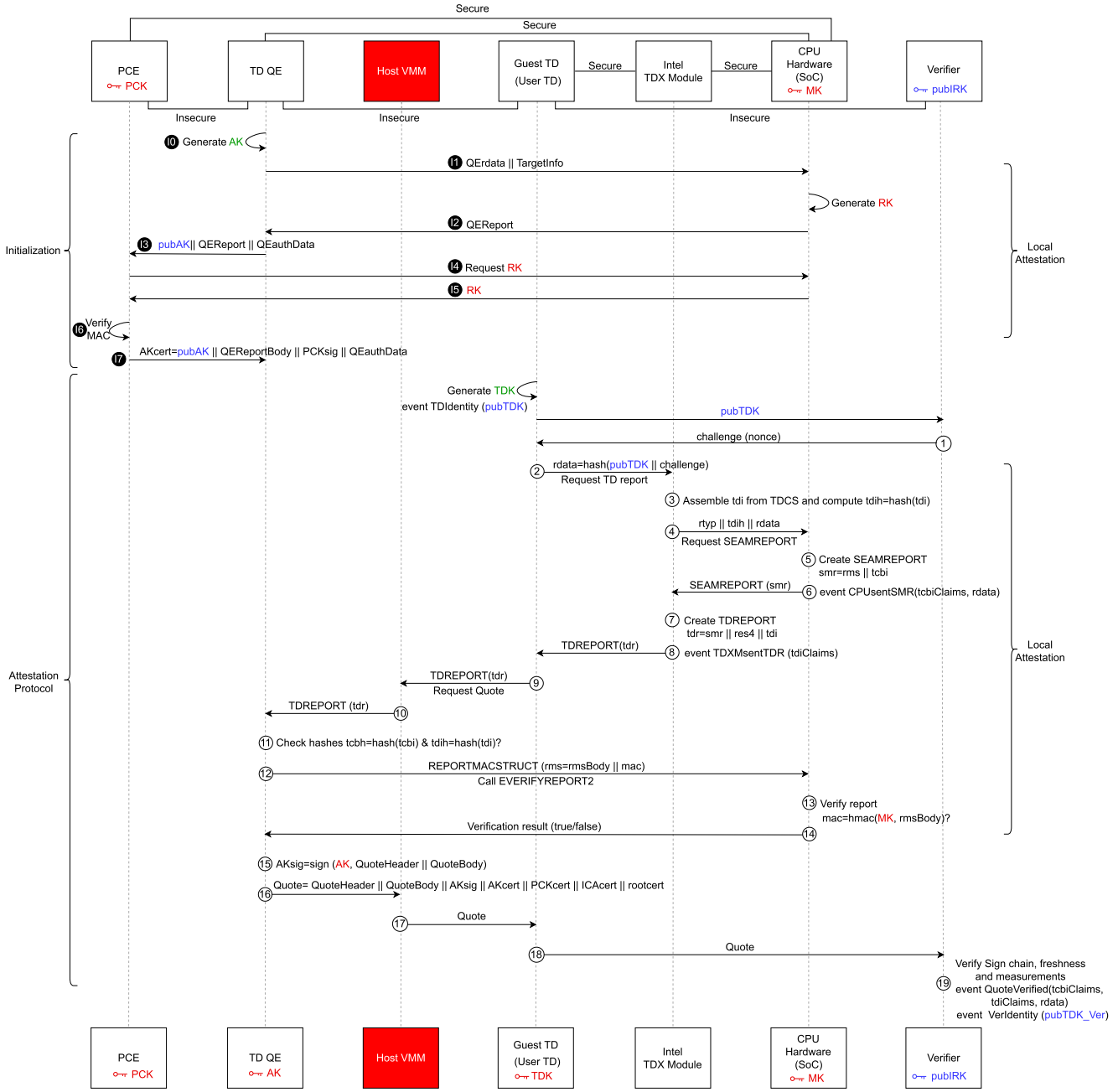

FIGURE 13. Complete flow of Intel TDX attestation.

We evaluate this claim formally in ProVerif by analyzing the four properties outlined in Section II-C. As summarized in the first row in Table 5, none of the properties hold for Intel's claimed TCB. We also analyze the properties for our proposed TCB, i.e., by including PCE as a trusted entity, and present a summary of results in the second row in Table 5. The average verification time to prove all the properties in the experimental setup mentioned in Section III-G is 55 s. We reported our findings to Intel in January'23. In response to the findings described in this section, Intel updated the white paper by replacing TD QE with TD attestation software in the TCB in February'23.

1) INTEGRITY

Events, namely CPUsentSMR and TDXMsentTDR, are placed just before sending *smr* and *tdr* (step 3 and 4 in Fig. 10), respectively. Event QuoteVerified is placed after all the verification steps are completed. The integrity of *tcbiClaims* and *rdata* is then formalized in ProVerif as follows:

query *tcbiClaims*, *tdiClaims*, *rdata*: bitstring;
event (QuoteVerified(*tcbiClaims*, *tdiClaims*, *rdata*))
$\Longrightarrow$ **event** (CPUsentSMR(*tcbiClaims*, *rdata*)).

In this query, *tcbiClaims* and *rdata* are the variables of agreement. Under Intel's claimed TCB, ProVerif confirms that the query does not hold, meaning that the protocol does not provide integrity of TDX TCB Claims to the Verifier. However, switching back PCE to a trusted entity, ProVerif confirms that the query holds. We also analyze the reachability of event QuoteVerified in ProVerif to ensure it is indeed reachable.

Similarly, the integrity of *tdiClaims* is formalized in ProVerif as follows:

query *tcbiClaims*, *tdiClaims*, *rdata*: bitstring;
event (QuoteVerified(*tcbiClaims*, *tdiClaims*, *rdata*))
$\Longrightarrow$ **event** (TDXMsentTDR(*tdiClaims*)).

Again, under Intel's claimed TCB, ProVerif confirms that the query does not hold, meaning that the protocol does not provide integrity of TD Claims to the Verifier. However, switching back PCE to a trusted entity, ProVerif confirms that the query holds.

2) FRESHNESS

Freshness properties are formalized by replacing **event** with **inj−event**.

query *tcbiClaims*, *tdiClaims*, *rdata*: bitstring;
event (QuoteVerified(*tcbiClaims*, *tdiClaims*, *rdata*))
$\Longrightarrow$ **inj−event** (CPUsentSMR(*tcbiClaims*, *rdata*)).

Since under Intel's claimed TCB, integrity properties do not hold, the freshness properties also do not hold. ProVerif confirms that the above query does not hold for Intel's claimed TCB, while switching PCE to a trusted entity, it holds. Similarly, the freshness of *tdiClaims* is formalized as:

query *tcbiClaims*, *tdiClaims*, *rdata*: bitstring;
event (QuoteVerified(*tcbiClaims*, *tdiClaims*, *rdata*))
$\Longrightarrow$ **inj−event** (TDXMsentTDR(*tdiClaims*)).

Again, under Intel's claimed TCB, ProVerif confirms that the query does not hold, while switching PCE to a trusted entity, it holds.

3) SECRECY

For completeness, we also analyze the secrecy properties for PCK private part, represented by *privPck*, respectively, by using the following ProVerif queries:

query secret *privPck*.

Again, under Intel's claimed TCB, ProVerif confirms that the query does not hold, while switching PCE to a trusted entity, it holds trivially as the keys are not sent on a public channel.

4) AUTHENTICATION

For completeness, we formalize the authentication property, which can be useful for use cases of attestation in combination with key exchange and authentication protocols (such as TLS). For this property, we use the model with the changes described in Section IV-E.

query *pubTDK*, *pubTDK_Ver*: VerifyingKey;
(**event**(VerIdentity(*pubTDK_Ver*))
$\wedge$ **event**(TDidentity(*pubTDK*)))
$\Longrightarrow$ (*pubTDK* = *pubTDK_Ver*).

where the events TDidentity and VerIdentity are placed after generation of TDK in *TD* subprocess, and after complete verification of Remote Evidence in *Verifier* subprocess, respectively. ProVerif confirms that the query does not hold, meaning that the protocol (without TLS) does not provide server authentication.

TABLE 5. Summary of verification results for Intel TDX.

	Integrity	Freshness	Confidentiality	Authentication
Intel's claimed TCB	×	×	×	×
Intel's claimed TCB + PCE	✓	✓	✓	×

H. AMBIGUITIES IN INTEL SPECIFICATIONS

We found several ambiguities about the fields of TD Quote. From a transparency and formalization perspective, a critical issue is that Intel repeatedly updates the specification on the same Uniform Resource Locator (URL), while the older specifications disappear. We reported this to Intel privately and then publicly [49]. Some examples of discrepancies found during the formalization process in the latest versions of the specification are presented in Appendix VI-A.

We summarize the missing specifications from Intel's public documentation at the time of writing, including 1) attester provisioning, 2) structure of AK cert, and 3) KDF for generation and verification of Local Evidence (TDREPORT). Without these important specifications, making a complete security argument is very hard.

V. RELATED WORK

Here we focus on works related to formal specification and verification of Arm CCA and Intel TDX. First, Arm has internally verified the security and safety properties of both the specification and a prototype implementation of the RMM using HOL4 theorem prover and CBMC model checker [50]. However, this work does not cover attestation mechanisms and is thus complementary to our work. Separately, Arm's published specification only describes the attestation report data model, including cryptographic operations and the API/ABI interfaces to obtain the report. In the paper, we provided the instantiation of the *CCA Attester in the TEE-agnostic architecture* (cf. Fig. 2) and the instantiation of the *challenge/response interaction model* (cf. Fig. 4 and 5). Moreover, we presented the details for the Appraisal of Evidence (cf. Section III-D2), which are not covered in any public specifications of Arm.

Regarding Intel TDX, in our previous work [21], we presented a formal analysis of the attestation protocol phase in Intel TDX using ProVerif [27]. However, that work did not specify provisioning, initialization as well as PCE and cert chain in the attestation protocol. Attestation in Intel TDX is based on attestation in Intel SGX. In our previous works, we presented the formal specification and analysis of EPID-based [19] as well as DCAP-based [20] attestation of Intel SGX. Challenges that we have addressed in this work compared to these works [19], [20], [21] include:

1) *Variable* (vs. constant) measurements
2) More *realistic* and *comprehensive* formal model, covering the initialization phase (cf. Section III-C) and certificate chain (cf. Fig. 12) as well as Verifier steps (cf. Section IV-D2). In contrast to [21], the inclusion of the initialization phase in the formal model confirmed the insecurity of TCB claimed by Intel. Moreover, our formal model includes Reference Values and Endorsements. While our previous work [21] assumes the signature verification key is pre-configured in the Verifier, we formally model the complete cert chain in the attestation protocol phase in this work.
3) More *precise* adversary model, allowing it to launch and control arbitrary TEEs

Therefore, this work is a substantial extension compared to our previous works [19], [20], [21].

VI. CONCLUSION

The discovery of subtle design and security issues using our holistic approach demonstrates that a precise specification of all phases of attestation mechanisms can help vendors eliminate several such issues. Since attestation is at the heart of CC, our work serves as a solid formal foundation for many extensions. For example, our verification artifacts, available at [22], provide the building blocks for the composition of attestation in TEEs with transport protocols (e.g., TLS) as well as virtual Trust Platform Module (vTPM) to formally verify the whole stack. A Confidential Computing Consortium project[4] on attested TLS plans to pursue the former, while Intel is pursuing the latter. Moreover, specifically for Intel TDX, it can help design high-confidence solutions for replacing TD QE by Quoting TD, and challenging problems, such as support for third-party TDX Module and third-party provisioning root key. Other interesting future works include a rigorous analysis of attestation in AMD SEV-SNP and complement our design-time verification approach with runtime verification [51] of properties. Since runtime verification analyzes a single execution trace, we expect it will not add much complexity to our approach. Runtime verification is particularly useful for AMD SEV-SNP, which, unlike Intel TDX, currently does not provide runtime measurement registers.

[4]https://github.com/ccc-attestation/attested-tls-poc

APPENDIX

A. INTEL TDX

Here we provide examples of discrepancies found in latest versions of the specification during the process of formalization:

1) AMBIGUOUS (RE-)NAMING/UNDEFINED NAMES

- SEAMINFO data structure (as used in, for example, §20.7.2, p. 153 in [40]) should be the same as TEE_TCB_INFO data structure (`tcbi`).
- The leaf functions EVERIFYTDREPORT2 (as used in, for example, Fig. 2.4, §2.7, p. 23 in [40]) and VERIFYREPORT (as used in, for example, Fig. 8-2, p. 40 in [52]) should be the same as EVERIFYREPORT2 leaf function. This is because EVERIFYREPORT2 leaf function is the only leaf function that is defined for the verification of the report in [53].

2) MISSING FIELDS IN DATA STRUCTURES

- MROWNERCONFIG field (`mroc`) is missing in TDINFO data structure `tdi` in Fig. 11.1, p. 95 of [40]. Although it is not clear whether it is part of the SHA384 hash of the data structure, we find no good reason why this should be excluded from the hash, because there is no other cryptographic protection for this field. Moreover, similar fields MROWNER (`mro`) and MRCONFIGID (`mrc`) are included in the hash. Therefore, we believe hash is also computed over this field.

3) INCONSISTENT INFORMATION

- Fig. 11.1 (p. 95) of [40] shows that the SHA384 hash is taken over four fields of the TEE_TCB_INFO data structure (`tcbi`) in a specific order, whereas the SEAMREPORT leaf operation in [53] shows that the SHA384 hash is taken over the whole data structure (p. 2-9) in a completely different order given in Table 2-3 (p. 2-6). It is worth pointing that order matters for hash computation.
- Fig. 11.1 (p. 95) of [40] shows that the reserved field (`res5`) in the TDINFO data structure (`tdi`) is not included in the SHA384 hash stored in REPORTMACSTRUCT (`rms`) and Quote, whereas for the verification (for reference, §20.7.3, p. 154 in the same document), it is implied that the hash is over the complete TDINFO data structure (`tdi`). Moreover, the order of fields in TDINFO data structure (`tdi`) in Fig. 11.1, p. 95 of [40] is inconsistent with the order in Table 20.20, p. 155 of the same document.

ACKNOWLEDGMENT

The authors would like to thank Nikolaus Thümmel, Ante Derek, and Jiewen Yao, for insightful discussions and helpful feedback. They also thankfully acknowledge the travel funds from sponsors, specially The Blockchain Technology Limited (TBTL) and COST Action EuroProofNet, for various

events, where the discussions and the learnings helped add depth to the article.

REFERENCES

[1] European Commission. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the Protection of Natural Persons with Regard to the Processing of Personal Data and on the Free Movement of Such Data, and Repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA Relevance)*. Accessed: Nov. 4, 2023. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj

[2] D. Evans, V. Kolesnikov, and M. Rosulek, "A pragmatic introduction to secure multi-party computation," *Found. Trends Privacy Secur.*, vol. 2, nos. 2–3, pp. 70–246, Dec. 2018. [Online]. Available: http://www.nowpublishers.com/article/Details/SEC-019

[3] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–35, Jul. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3214303

[4] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM J. Comput.*, vol. 18, no. 1, pp. 186–208, Feb. 1989. [Online]. Available: https://epubs.siam.org/terms-privacy http://epubs.siam.org/doi/10.1137/0218012

[5] M. U. Sardar and C. Fetzer, "Confidential computing and related technologies: A critical review," *Cybersecurity*, vol. 6, no. 1, p. 10, May 2023. [Online]. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-023-00144-1

[6] M. Bartock et al. (Apr. 2022). *Trusted Cloud: Security Practice Guide for VMware Hybrid Cloud Infrastructure as a Service (IaaS) Environments*. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1800-19.pdf

[7] V. Costan and S. Devadas. (2016). *Intel SGX Explained*. [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[8] AMD. (2020). *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. [Online]. Available: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[9] Arm Ltd. (2022). *Arm Realm Management Extension (RME) System Architecture*. [Online]. Available: https://developer.arm.com/documentation/den0129

[10] Arm Ltd. (2022). *Realm Management Monitor Specification*. [Online]. Available: https://developer.arm.com/documentation/den0137

[11] Intel. (Aug. 2021). *Intel ® Trust Domain Extensions*. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/690419

[12] R. Sahita, A. Patra, V. Shanbhogue, S. Ortiz, A. Bresticker, D. Reid, A. Khare, and R. Kanwal, "CoVE: Towards confidential computing on RISC-V platforms," 2023, *arXiv:2304.06167*.

[13] RISC-V AP-TEE TG. (2023). *RISC-V AP-TEE TG Specifications*. [Online]. Available: https://github.com/riscv-non-isa/riscv-ap-tee

[14] G. D. H. Hunt et al., "Confidential computing for OpenPOWER," in *Proc. 16th Eur. Conf. Comput. Syst.* New York, NY, USA: ACM, Apr. 2021, pp. 294–310, doi: 10.1145/3447786.3456243.

[15] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, *Remote ATtestation Procedures (RATS) Architecture*, document RFC 9334, Jan. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9334

[16] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 777–795. [Online]. Available: https://eprint.iacr.org/2019/1393.pdf

[17] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 483–502.

[18] Center for Perspicuous Computing. *E7—Perspicuity and Societal Risk*. Accessed: Nov. 4, 2023. [Online]. Available: https://www.perspicuous-computing.science/projects/e7/

[19] M. U. Sardar, D. L. Quoc, and C. Fetzer, "Towards formalization of enhanced privacy ID (EPID)-based remote attestation in Intel SGX," in *Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2020, pp. 604–607. [Online]. Available: https://ieeexplore.ieee.org/document/9217791/

[20] M. U. Sardar, R. Faqeh, and C. Fetzer, "Formal foundations for Intel SGX data center attestation primitives," in *Formal Methods and Software Engineering*, vol. 12531. Cham, Switzerland: Springer, 2020, pp. 268–283, doi: 10.1007/978-3-030-63406-3_16.

[21] M. U. Sardar, S. Musaev, and C. Fetzer, "Demystifying attestation in Intel trust domain extensions via formal verification," *IEEE Access*, vol. 9, pp. 83067–83079, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9448036/

[22] M. U. Sardar and S. Xiong. (2023). *Verification Artifacts for Intel TDX and Arm CCA*. [Online]. Available: https://github.com/CCC-Attestation/formal-spec-TEE

[23] H. Tschofenig, Y. Sheffer, P. Howard, I. Mihalcea, and Y. Deshpande. (Mar. 2023). *Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/03/

[24] N. P. Smart, *Cryptography Made Simple*, vol. 53, no. 10. Berlin, Germany: Springer, Jun. 2016. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-21936-3

[25] Trusted Computing Group. (2021). *DICE Attestation Architecture*. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf

[26] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proc. 11th ACM Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 132–145, doi: 10.1145/1030083.1030103.

[27] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with lemmas, induction, fast subsumption, and much more," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 69–86. [Online]. Available: https://ieeexplore.ieee.org/document/9833653/

[28] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 2, pp. 198–208, Mar. 1983.

[29] Arm Ltd. (2022). *AArch64 Exception Model*. [Online]. Available: https://developer.arm.com/documentation/102412

[30] Arm Ltd. (2022). *Arm Confidential Compute Architecture Software Stack Guide*. [Online]. Available: https://developer.arm.com/documentation/den0127

[31] TF-RMM Contributors. (2022). *Reference Implementation of Arm-CCA RMM Specification*. [Online]. Available: https://www.trustedfirmware.org/projects/tf-rmm/

[32] TF-RMM Contributors. (2022). *Reference Implementation of Arm-CCA RMM Specification*. [Online]. Available: https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/tree/lib/rmm_el3_ifc/src/rmm_el3_runtime.c

[33] TF-RMM Contributors. (2022). *Reference Implementation of Arm-CCA RMM Specification*. [Online]. Available: https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/tree/lib/attestation/include/attestation.h

[34] TF-RMM Contributors. (2022). *Reference Implementation of Arm-CCA RMM Specification*. [Online]. Available: https://git.trustedfirmware.org/TF-RMM/tf-rmm.git/tree/lib/attestation/src

[35] Arm Ltd. (2022). *Arm CCA Security Model*. [Online]. Available: https://developer.arm.com/documentation/DEN0096

[36] L. Lundblade, G. Mandyam, J. O'Donoghue, and C. Wallace. (Oct. 2022). *The Entity Attestation Token (EAT)*. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-rats-eat/17/

[37] H. Tschofenig, S. Frost, M. Brossard, A. L. Shaw, and T. Fossati. (Sep. 2022). *Arm's Platform Security Architecture (PSA) Attestation Token*. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/draft-tschofenig-rats-psa-token/10/

[38] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, *CBOR Web Token (CWT)*, document RFC 8392, May 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8392

[39] S. Frost. (Dec. 2022). *Entity Attestation Token (EAT) Collection Type*. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/draft-frost-rats-eat-collection/02/

[40] *Architecture Specification: Intel Trust Domain Extensions (Intel® TDX) Module*, document 344425-004US, Intel, Santa Clara, CA, USA, Jun. 2022, pp. 1–316. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/733568

[41] Intel. (2022). *Intel® Trust Domain Extensions (Intel® TDX)*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[42] Intel. (2023). *Intel® Trusted Services API Management Developer Portal*. [Online]. Available: https://api.portal.trustedservices.intel.com/documentation

[43] *Intel® 64 and IA-32 Architectures: Software Developer's Manual*, document 325462-077US, Intel, Santa Clara, CA, USA, Apr. 2022. Accessed: Aug. 24, 2022. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200

[44] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. (2013). *Innovative Technology for CPU Based Attestation and Sealing*. [Online]. Available: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing

[45] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski. (2018). *Supporting Third Party Attestation for Intel®SGX with Intel® Data Center Attestation Primitives*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf

[46] S. Johnson. (2019). *Scaling Towards Confidential Computing*. [Online]. Available: https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf

[47] Intel Corporation. (2022). *Intel® SGX PCK Certificate and Certificate Revocation List Profile Specification, Revision 1.5*. [Online]. Available: https://api.trustedservices.intel.com/documents/Intel_SGX_PCK_Certificate_CRL_Spec-1.5.pdf

[48] Distributed Management Task Force. (2023). *Security Protocol and Data Model (SPDM) Specification*. [Online]. Available: https://www.dmtf.org/dsp/DSP0274

[49] M. U. Sardar. (2023). *Full Transparency of Intel TDX Specifications*. [Online]. Available: https://lists.confidentialcomputing.io/g/attestation/topic/full_transparency_of_intel/99387880

[50] A. C. J. Fox, G. Stockwell, S. Xiong, H. Becker, D. P. Mulligan, G. Petri, and N. Chong, "A verification methodology for the Arm® confidential computing architecture: From a secure specification to safe implementations," *Proc. ACM Program. Lang.*, vol. 7, pp. 376–405, Apr. 2023, doi: 10.1145/3586040.

[51] E. Bartocci and Y. Falcone, *Lectures on Runtime Verification* (Lecture Notes in Computer Science), vol. 10457, E. Bartocci and Y. Falcone, Eds. Cham, Switzerland: Springer, 2018. [Online]. Available: http://link.springer.com/10.1007/978-3-319-75632-5

[52] Intel. (Oct. 2021). *Intel® TDX Virtual Firmware Design Guide*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-virtual-firmware-design-guide-rev-1.01.pdf

[53] Intel. (May 2021). *Intel® Trust Domain CPU Architectural Extensions*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents-tps/intel-tdx-cpu-architectural-specification.pdf

MUHAMMAD USAMA SARDAR (Member, IEEE) received the B.S. degree in electronics engineering from Ghulam Ishaq Khan Institute of Engineering Sciences and Technology, Topi, Pakistan, in 2009. He later did his masters from the School of Electrical Engineering and Computer Sciences (SEECS) at the National University of Sciences and Technology (NUST), Islamabad, Pakistan in 2015 with 2nd position in his batch. His master's dissertation was in collaboration with the Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT), Germany. From 2017 to 2021, he received the prestigious DAAD research grant in Computer Science faculty at the Technical University of Dresden (TU Dresden), Germany.

His work experience includes research as well as teaching. He was a Research Internee with the Chair of Embedded Systems, Karlsruhe Institute of Technology, Germany, and a Research Assistant with the System Analysis and Verification Laboratory, NUST, Pakistan. Since 2021, he has been a Research Associate at TU Dresden, working for the Transregional Collaborative Research Centre 248 "Foundations of Perspicuous Software System" (CPEC). His research work has resulted in publications at top international forums, such as the Journal of Parallel and Distributed Computing, the NASA Formal Methods Symposium, Journal of Automated Reasoning and IEEE Access. His current research focuses on the formal specification and verification of architecturally-defined remote attestation for confidential computing, specifically Intel SGX, TDX and Arm CCA. Since Winter 2019/20, he has been a tutor for the master's courses: Systems Engineering 1, Systems Engineering 2, Principles of Dependable Systems and Software Fault Tolerance at the Technical University of Dresden.

Mr. Sardar contributes to various research and professional networks, such as EuroProofNet (Working Group 3), Méthodes formelles pour la sécurité, Confidential Computing Consortium (CCC) Attestation Special Interest Group (SIG), Internet Engineering Task Force (IETF) Remote ATtestation procedureS (RATS), and Internet Research Task Force (IRTF) Usable Formal Methods Research Group (UFMRG). He is also serving as a volunteer at CAVlinks. He has received the best poster award in Postgraduate category in the International Conference on Digital Futures and Transformative Technologies (ICoDT2) in 2021, South Asia Triple Helix Association (SATHA) Innovation Award in 2018, best researcher of the year award in System Analysis and Verification lab in Pakistan in 2017, and best speaker awards at Workshop on Applications in ASIC Design (December 2016) and Workshop on Recent Trends in Theorem Proving (April 2016).

THOMAS FOSSATI is currently an Engineer with the Linux Kernel Working Group, Linaro.

SIMON FROST is currently a Software Architect with the Architecture and Technology Group, Arm.

SHALE XIONG received the Ph.D. degree in computer science from Imperial College London, in 2019. He is currently a Research Engineer with the Architecture and Technology Group, Arm. His research interests include formal verification and programming languages.