IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 28, 2020, accepted February 13, 2020, date of publication February 17, 2020, date of current version February 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2974487

A Design and Verification Methodology for a TrustZone Trusted Execution Environment

HAIYONG SUN AND HANG LEI
School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China
Corresponding author: Haiyong Sun (haiyong.sun@foxmail.com)

ABSTRACT Hardware support for isolated execution (e.g., ARM TrustZone) enables the development of a trusted execution environment (TEE) that ensures the security of the code and data while communicating with a compromised rich execution environment (REE). The ability to satisfy various security services is complicated and usually consists of trusted applications, a trusted kernel and a secure monitor. However, formally verifying the security of an entire TEE security remains challenging. We present a methodology for designing a TEE in a way that enables verification of its security properties. Our methodology consists of forcing a trusted application and kernel to communicate with an REE via a narrow interface and compile and link them with a small secure monitor that implements the interface and runs at the highest privilege level. We provide functional verification of the secure monitor to ensure that it correctly switches the TEE/REE, communicates with the REE at a pre-defined memory space and has no integer overflow vulnerability. We also perform a verification of the secure monitor's scheduler to ensure that it satisfies information flow noninterference. We present a modular verification framework that can prove an end-to-end security property for cross-language programmes (e.g., C and assembly languages). Our evaluation suggests that the methodology scales to real-world TEE applications.

INDEX TERMS Formal verification, information flow noninterference, trusted execution environment, TrustZone.

I. INTRODUCTION

The rapid development and extensive application of the mobile internet offers substantial convenience to people's lives. With an increase in the number of cyber-attacks, users' information security is facing an increasing number of serious threats. A user's private information can be compromised due to a host of reasons, including vulnerabilities in the OS, hypervisor, and insufficient access control. Recognizing this problem, the ARM TrustZone [1] provides an isolated TEE and REE (rich execution environment) for the upper software system by securely extending the processor, memory and peripherals. Users' security-sensitive information can be separated from the compromised REE (e.g., iOS and Android). To minimize the trusted computing base (TCB), a TEE usually does not implement complex software modules, such as a file system and network protocol stack, but instead relies on the compromised REE for basic services, such as storage and network communication; the burden of correctly programming TEE software and ensuring security remains with the programmer.

Unfortunately, many security vulnerabilities exist in the TEE, such as the Qsee TrustZone kernel integer overflow vulnerability [2] and the HTC getting caught storing fingerprint data in unencrypted plain text [3]. To guarantee a strong security of the TEE, we analyse the attack model of the TEE and propose a verification methodology to verify that the TEE preserves security properties.

A. THREAT MODEL

The illustration of our threat model in Fig. 1 lists all the system components that are under the attacker's control. The adversary may fully control all of the hardware in the ARM system-on-chip board, including solid state disks (SSD), RAM chips, and network cards in the system, except the CPU and TrustZone-protected RAM (TZRAM). The adversary may record, replay and modify network packets or files, as well as read or modify data after it leaves the TEE using physical probing or similar techniques. We assume that the adversary cannot physically attack the CPU or TZRAM to

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

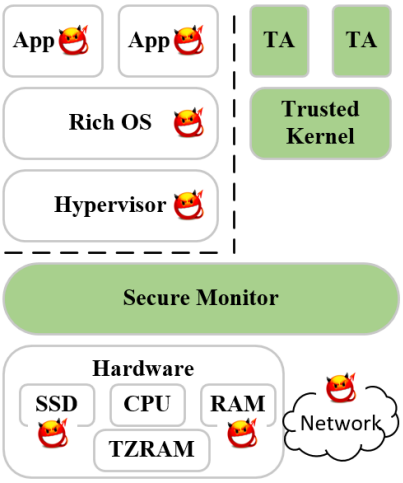

FIGURE 1. Threat Model: The adversary controls the rich OS, hypervisor, and any hardware beyond the CPU package and TrustZone-protected RAM, which may include both unprotected RAM chips and storage devices. The adversary also controls the network. The TA, Trusted Kernel and Secure Monitor are the only trusted software components.

extracts secrets. The adversary may also control all of the system software in the REE, including the rich OS and the hypervisor. This adversary is general enough to model privileged malware running in the REE system software to access data by inspecting disks and memory.

We assume a user that wishes to protect the security-sensitive information processed by the TEE. We assume that the TEE is not designed to write security-sensitive information outside the TrustZone-protected area. However, the TEE may have bugs such as accidental writes of security-sensitive information to TrustZone-unprotected RAM, as well as exploitable bugs, such as integer overflow, buffer overflow and dereferences of uninitialized or corrupted pointers, which could result in security-sensitive information leaking out of the TEE.

Therefore, we design and verify that the TEE cannot leak secrets even in the presence of a powerful adversary. However, DOS attacks, covert channel and side-channel attacks are outside the scope of this paper.

B. DESIGN AND VERIFICATION METHODOLOGY

Our approach is based on multi-level architecture: the trusted application (TA) level that contains the application logic, the trusted kernel level that provides core APIs (e.g., memory management) and the secure monitor level that provides scheduling between a TEE and an REE and encrypted channels for communication. We limit the memory access permissions of the TA and trusted kernel to the secure zone by establishing a TrustZone address space controller. As a result, we restrict the interaction between the TEE and the REE to the narrow interface implemented by the secure monitor. For the secure monitor, we need to prove the following tasks: the secure channel always reads/writes data in a pre-defined memory space; the scheduler always saves registers into and loads new values from proper places and then switches to a proper execution environment; and the secure monitor's implementation satisfies the information flow noninterference property. To complete these verification tasks, the following challenges need to be addressed:

- Cross-Language Specification—Some parts of software modules must be written in both C and assembly for various reasons. For example, the secure monitor's secure channel is written in C, but its scheduler is written in ARM assembly. How do we specify clear and precise behaviour semantics for both C and assembly language in a unified verification framework?
- Security Property modelling—How do we model a clear and precise security property for different domains? If we express the property in terms of the abstraction level specification, then what will this task imply for the programme implementation level? We need to model properties at different levels of abstraction and translate between or link separate properties.
- Proving Security—Research [4], [5] indicates that refinements may not propagate security guarantees. In addition, some abstraction level specifications may atomically preserve security properties. However, its non-atomic implementation may temporarily break the properties during intermediate states. Thus, we must strengthen the refinement relation and carefully formulate the security of an API's implementation.

The modular verification framework that we proposed can be used to verify the end-to-end security of a system software and adequately address these challenges. First, a security property is proven at the abstraction level functional specification. Second, we apply bi-simulation techniques to automatically obtain a sound security guarantee for the C and assembly implementation level. We demonstrate the efficacy of our verification framework by applying it to complete the secure monitor's verification tasks.

The primary contributions of this work are listed as follows:

- A design methodology to build a TEE using a narrow interface to communicate with the REE.
- A modular verification framework for end-to-end security verification of software systems or critical modules written in both C and assembly.
- A security proof of the TEE critical module—secure monitor, which is completely formalized in the Coq proof assistant [6].

The remainder of this paper is organized as follows. Section II introduces the TEE design architecture and analysis of impact of different TEE architectures on formal verification tasks. Section III discusses the challenges when verifying the security of the secure monitor. Section IV describes our modular verification framework and its implementation in Coq. Section V describes the security verification of the secure monitor using our modular verification framework. Section VI provides a performance evaluation

of our TrustZone-based TEE and an analysis of its security. Section VII and Section VIII discuss related work and the conclusions, respectively.

II. TEE DESIGN BASED ON TRUSTZONE

In this section, first, we introduce the design architecture of the TEE. Second, we introduce the design of a secure communication channel. Last, by comparing the multi-level architecture and traditional monolithic architecture, we illustrate the advantages of the multi-level architecture proposed in this paper.

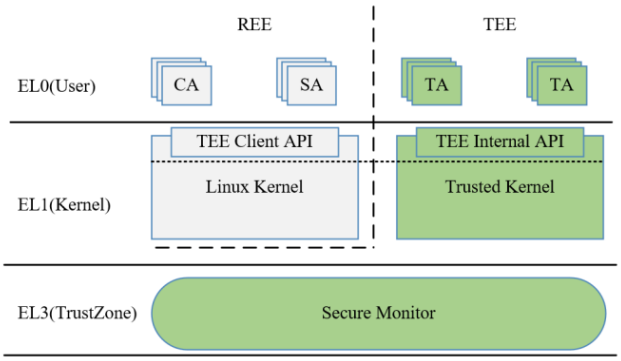

FIGURE 2. TEE multi-level architecture, where the privilege level increased from EL0 to EL3.

A. TEE MULTI-LEVEL ARCHITECTURE

From top to bottom (Fig. 2), the entire TEE software stack is divided into user level, kernel level and TrustZone level.

At the user level, applications in REE that can request security services are named secure applications (SAs); the remainder of the applications are named client applications (CAs), and applications in TEE are named trusted applications (TAs). The security service request from SA to TA will involve cross-environment communication and context switching. To ensure that the interaction details are transparent to application developers, both the REE and the TEE encapsulate the TEE client API and the TEE internal API, respectively.

At the kernel level, the Linux kernel provides more complete system services, such as memory management, process management, network communication, and file system management, while the trusted kernel provides memory management and process management. Although we chose the Linux kernel as the REE kernel, developers can choose other common kernels.

At the TrustZone level, the secure monitor provides the low-level TrustZone mechanisms that are responsible for execution environment scheduling, communication and protection.

B. SECURE CHANNEL

To serve its memory needs, the TEE requires physical RAM dedicated to the TAs, trusted kernel and secure monitor. The TrustZone ensures that the REE cannot address or access any TrustZone-protected RAM, TZRAM (TEE RAM). To serve a secure monitor's communication, a piece of unprotected RAM is used as shared memory between the TEE and the REE.

To protect user private data, a secure channel is implemented in the secure monitor and consists of a *channel_recv* API and a *channel_send* API. Specifically, the *channel_recv* copies encrypted data from the shared memory to a TEE buffer and then decrypts the data, and *channel_send* encrypts data in the TEE buffer and then copies the encrypted data to the shared memory.

C. IMPACT OF DIFFERENT TEE DESIGN ON FORMAL VERIFICATION TASKS

The design of TEE is flexible: it can be either a monolithic architecture or a multi-level architecture. For the former, TA has the authority of key management and can communicate directly with the REE. For the latter, the key management and communication are hosted by the secure monitor. The following section provides a more detailed comparative analysis of a secure storage method with different architectures.

```
1.  void SeStore (BYTE *dataEnc, BYTE *outputEnc) {
2.    char dataBuf[DATA_SIZE];
3.    Decrypt (dataEnc, dataBuf, DATA_SIZE);
4.    // do some computing work
5.    // get computation results – comRes
6.    char cleartext[BUF_SIZE];
7.    memcpy(cleartext, comRes, BUF_SIZE);
8.    Encrypt (cleartext, outputEnc, BUF_SIZE);
9.  }
```

(a) Sample secure storage method

```
1.  void SeStore(void){
2.    char dataBuf[DATA_SIZE];
3.    channel_recv (dataBuf, DATA_SIZE);
4.    // do some computing work
5.    // get computation results -- comRes
6.    char cleartext [BUF_SIZE];
7.    memcpy (cleartext, comRes, BUF_SIZE);
8.    channel_send (cleartext, BUF_SIZE);
9.  }
```

(b) Secure storage method using secure channels

FIGURE 3. Secure storage method that illustrates the impact of different TEE design architectures on formal verification tasks.

Consider the *SeStore* method in Fig. 3(a), which acts as a secure storage function. The *SeStore* receives encrypted data from the REE. The method decrypts the encrypted data, performs some computing work, and writes the computation results to a buffer. The buffer is encrypted and written back to the REE. Proving security of this *SeStore* method is challenging. The code writes the results of the computation to a stack-allocated buffer without checking the size of the inputs. This step may produce a vulnerability that can be exploited to overwrite the return address, execute arbitrary code and leak secrets. Therefore, the proof must show that the result does not exceed the buffer size. *SeStore* directly writes to a location outside the TEE. The proof must show that the

written data are either encrypted or independent of secrets. The latter requires the tracking of secrets in the entire TEE memory. Thus, building a scalable verification framework is challenging for arbitrary user code.

Fig. 3(b) shows the *SeStore* method that has been rewritten to use our secure channel. The method calls *channel_recv* to retrieve data from the channel, which reads and decrypts encrypted data from the REE. After computing the results, the method calls *channel_send* to write data to the channel, which internally encrypts the data and writes to it back to the REE. Observe that there are no direct communications to nonsecure memory for this *SeStore* method. Thus, we only need to track secrets in the secure monitor execution space instead of the entire TEE memory.

As previously mentioned, the formal verification task with the monolithic architecture needs to consider the implementation details of each single TA, which will increase the difficulty of formal verification and reduce the scalability of verification due to the increase in the number of TAs or the complexity of programme logic. The formal verification task with our multi-level architecture will not be affected by the implementation details of each single TA. With TrustZone hardware protection, only one-time verification of the secure monitor module can guarantee the security of the TEE, which reduces the difficulty of formal verification. We divide the verification task of TEE security into two subtasks: 1) verify the functional correctness of the secure monitor; and 2) verify that the secure monitor satisfies information flow noninterference.

III. VERIFICATION CHALLENGES

We discuss key challenges when verifying the functional correctness and noninterference property of the secure monitor.

A. VERIFICATION OF THE SCHEDULER

The scheduler of the secure monitor is an indispensable component of the TrustZone-based TEE. Verification of its implementation is essential in building trusted software systems. However, implementation of the scheduler requires low-level operations, such as manipulation of stack pointers and return code pointers, which prevents the accurate specification of the code behaviours.

As shown in Fig. 4, the scheduler implements scheduling by detecting the direction of the schedule request (lines 2-5 of (a)), toggling the processor mode bit (lines 6-7 of (a)), saving and restoring the processor state (by calling the *TEE_switchctxt* subroutine, line 8 of (a)) from the corresponding execution environment context stack, and then switching to a target environment (line 19 of (b)). We explain the challenges of verifying the scheduler code using a small example with an SA process and a TA process (Fig. 5).

1) MANIPULATION OF RETURN CODE POINTERS

When a function returns, it needs the value stored on top of the stack as the return address. The return address of a normal function is the same address passed to it by the caller.

```
1.  TEE_sched:
2.  READ_SCR   r2
3.  TST   r2, #NS_BIT
4.  MOVEQ   r0, #0
5.  MOVNE   r1, #1
6.  EOR   r2, r2, #NS_BIT
7.  WRITE_SCR  r2
8.  B   TEE_switchctxt
```

(a) Secure monitor scheduler

```
1.  TEE_switchctxt:
2.  ; save old ctxt
3.  LDR r2, =CtxtPool
4.  MOV r3, #SM_CTX_SEC
5.  MLA r0, r0, r3, r2
6.  STR r8, [r0]
7.  STR r9, [r0, #4]
8.  ...
9.  STR sp, [r0, #20]
10. STR lr, [r0, #24]
11.
12. ; restore new ctxt
13. MLA r1, r1, r3, r2
14. LDR r8, [r1]
15. LDR r9, [r1, #4]
16. ...
17. LDR sp, [r1, #20]
18. LDR lr, [r1, #24]
19. MOV pc, lr
```

(b) TEE context switching

FIGURE 4. Scheduling example between TEE and REE.

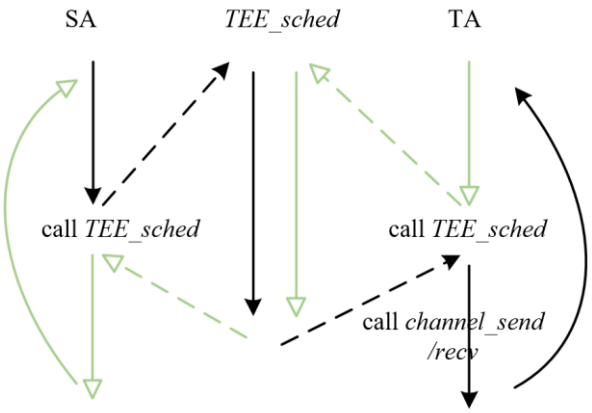

FIGURE 5. Two simple processes.

However, *TEE_sched* changes the stack pointer. When the TA process refers to *TEE_sched*, *TEE_sched* does not return to TA. Instead, it returns to the SA process shown in Fig. 5. To verify *TEE_sched*, we need to prove that it always uses a valid return address.

2) SPECIFYING THE SCHEDULER

As shown in Fig. 5, before *TEE_sched* returns to the calling process (e.g., process SA), the control has been transferred to a different process (process TA). This approach presents issues: Should the *channel_send*/*recv* code (parts of behaviour of the process TA) be considered part of the scheduler? How can *TEE_sched* be specified without knowing in advance whether the *channel_send*/*recv* will be called?

B. END-TO-END SECURITY VERIFICATION

To prove the security properties from the abstraction end to the implementation end, we usually automatically derive the implementation level security property from a refinement relation and a proof of the abstract security property. As discussed in the following section, security is not automatically preserved across a refinement relation.

1) NONDETERMINISTIC SECURE PROGRAMME

A potential issue is that a nondeterministic secure programme can be refined into a more deterministic but insecure programme, which is known as the refinement paradox. For example, assume two programmes $\mathcal{P}$ and $\mathcal{Q}$ and a secret Boolean value stored in §, where $\mathcal{Q}$ is a refinement of $\mathcal{P}$. The $\mathcal{P}$ randomly prints either true or false but the $\mathcal{Q}$ directly prints the value of χ. Obviously, $\mathcal{P}$ is secure since its outputs have no dependency on the secret value, but $\mathcal{Q}$ is insecure. In the operating system verification community [7], [8], this issue is avoided by disallowing specifications from exhibiting any domain-visible nondeterministic programmes. More formally, using an unwinding condition [9], [10] to define a noninterference property, for any domain d and state transition machine $\mathcal{M}$ with single-step transition semantics $\mathcal{T}_{\mathcal{M}}$, we say that $\mathcal{M}$ is secure for if the following property holds for all states $s_1, s_2, s_1{}', \textit{ and } s_2{}'$:

$$\begin{aligned}\mathcal{O}_d(s_1) \;&=\; \mathcal{O}_d(s_2) \wedge (s_1, s_1') \in \mathcal{T}_{\mathcal{M}} \wedge (s_2, s_2') \in \mathcal{T}_{\mathcal{M}} \\ &\Longrightarrow \mathcal{O}_d(s_1') = \mathcal{O}_d(s_2'),\end{aligned}$$

where $\mathcal{O}_d(s)$ is a domain-visible function that is defined in terms of the programme state. Note that despite that $\mathcal{P}$ is obviously secure, it does not actually satisfy the unwinding condition defined above (with the same inputs, its outputs may be different) and thus is not provably secure. This means that we must prohibit programmes such as $\mathcal{P}$ above.

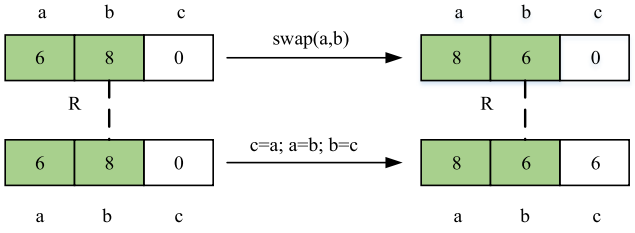

FIGURE 6. Security violating refinement.

2) INCOMPATIBILITY BETWEEN REFINEMENT RELATIONS AND DOMAIN-VISIBLE FUNCTIONS

A new issue will be produced when we prove end-to-end security properties based on refinement relations and domain-visible functions. The issue arises from the fact that both refinement relations and domain-visible functions are defined in terms of the programme state, and they are both arbitrarily general. This means that certain refinement relations may behave poorly with respect to the domain-visible function. Fig. 6 illustrates an example. Assume the programme state consists of three variables a, b, *and* c at both levels. The domain-visible function is the same at both levels: a and b are invisible while c is visible. Suppose there is an abstract *swap* function saying that the values of a and b are swapped, and the value of c is unchanged. Additionally, suppose there is a refinement relation $\mathcal{R}$ that relates two states where a and b have the same values, but c may have different values. Using this refinement relation, we can prove that the implementation end swap follows the abstraction end swap specification. Since the *swap* specification is deterministic and holds the unwinding condition, it is a secure programme. Nevertheless, this example fails to preserve security across refinement, as the implementation end leaks the secret value of a into the visible variable c. Indeed, the root cause of this issue is that the domain-visible state (e.g., variable c) is not contained by the refinement relation $\mathcal{R}$. Thus, one solution to this issue is to restrict refinements to ensure that security properties are preserved. More formally, given the domain d, to show that machine m refinements $\mathcal{M}$ under the refinement relation $\mathcal{R}$, the following property must be proved for all domain-visible states σ_1, σ_2 of $\mathcal{M}$ and all domain-visible states s_1, s_2 of m:

$$\begin{aligned}\mathcal{O}_{\mathcal{M};d}(\sigma_1) \;&=\; \mathcal{O}_{\mathcal{M};d}(\sigma_2) \wedge (\sigma_1, s_1) \in \mathcal{R} \wedge (, \sigma_2, s_2) \in \mathcal{R} \\ &\Longrightarrow \mathcal{O}_{m;d}(s_1) = \mathcal{O}_{m;d}(s_2)\end{aligned}$$

IV. MODULAR VERIFICATION FRAMEWORK

In this section, we present the basic model of our modular verification framework and the notion of end-to-end security, which can address the challenges mentioned in section III.

A. BASIC MODEL

Our framework consists of two parts: i) an atomic module verification framework that will enable us to rapidly and easily define as many atomic modules for verification as we need; and ii) a compositional verification technique that will enable us to link all verified atomic modules. We provide a formalization of atomic modules and their composition.

Definition 1 (Atomic Module): An atomic module is a state transition system that is extended with predicates $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \Psi)$, where

1) $(\mathcal{S}, \mathcal{T})$ is a state transition system, that is,
   - $\mathcal{S}$ is a set of programme states (consists of global abstract states and function input/output arguments),
   - $\mathcal{T}$ is the transition relation $\mathcal{T}$ of type $\mathcal{P}(\mathcal{S} \times \mathcal{S})$, ($\mathcal{P}$ is a power set operator),
2) Ψ is the interface specifications of $\mathcal{M}$, that is,
   - For each $t \in \mathcal{T}$, $\mathcal{I}_t$ is an invariant, a predicate on $\mathcal{S}$, and $f_t(\mathcal{S}, \mathcal{S}')$ is an update (step) relation, a predicate on $\mathcal{S}$(pre-) and $\mathcal{S}'$(post-) states.
   - For each $t \in \mathcal{T}$, $\psi_t = \mathcal{I}_t \cup f_t$ and $\psi_t \in \Psi$.

Note that our definition is slightly different from most traditional definitions of automata as we move concepts in traditional definitions of automata (e.g., input/output events) into the programme state, which can simplify the formal specifications and proofs.

We define the logic from verifying an atomic module to composing all verified atomic modules.

Definition 2 (Verified Atomic Module): Given the atomic module $\mathcal{M}$, we define a verified atomic module as $\mathcal{M}, \mathcal{L} \vdash_{\mathcal{R}} \mathbb{C} : \Psi$, where

- $\mathbb{C}$ is the interface procedures of $\mathcal{M}$, for each $t \in \mathcal{T}$, $c_t \in \mathbb{C}$,
- $\mathcal{L}$ represents the interface specifications that underlay the called interface procedures,
- $\mathcal{R}$ is the refinement relation between $\mathbb{C}$ *and* Ψ.

In addition to $\mathcal{M}$'s underlay specifications $\mathcal{L}$, a verified atomic module means that every interface procedure in $\mathbb{C}$ strictly follows its specification in Ψ. All the verification processes of an atomic module follow the same pattern:

$$\frac{\mathcal{M}, \mathcal{L} \vdash_{id} \mathbb{C} : \Sigma \; \Sigma \vdash_{\mathcal{R}} \emptyset : \Psi}{\mathcal{M}, \mathcal{L} \vdash_{\mathcal{R}} \mathbb{C} : \Psi}$$

i) proofs of correctness for the C or assembly code, which can be stated as $\mathcal{M}, \mathcal{L} \vdash_{id} \mathbb{C} : \Sigma$ (where Σ is the separation logic [11] specification and *id* means no abstraction exists between $\mathbb{C}$ *and* Σ);

ii) proofs of the $\mathbb{C}$ code specifications refine Ψ, which can be formulated as $\Sigma \vdash_{\mathcal{R}} \emptyset : \Psi$.

When we verify using modules, the module will commonly require the interface procedures of another module. However, arbitrary module interaction or dependencies may create an issue in that the invariant held in one function can be easily broken when it calls a function defined in another module. To avoid this problem, we divide the atomic modules into different layers according to their abstraction levels. Only modules with identical state views (i.e., at the same abstraction level) can be composed at the same layer, which eliminates most unwanted interaction and dependencies when we reason about one small atomic module. Inspired from compositional verification technology [12], [13], we list our composition rules as follows:

Definition 3 (Parallel Composition): Given the two verified atomic modules $\mathcal{M}_1, \mathcal{L} \vdash_{\mathcal{R}} \mathbb{C}_1 : \Psi_1$ and $\mathcal{M}_2, \mathcal{L} \vdash_{\mathcal{R}} \mathbb{C}_2 : \Psi_2$, we define $\mathcal{M} = \mathcal{M}_1 \oplus \mathcal{M}_2, \mathcal{L} \vdash_{\mathcal{R}} \mathbb{C}_1 \oplus \mathbb{C}_2 : \Psi_1 \oplus \Psi_2$, where $\mathcal{M}_1$ and $\mathcal{M}_2$ have identical state views.

Using parallel composition, we can link atomic modules at the same abstraction level and obtain the entire verified layer. Next, we present the definition of vertical composition, which can link atomic modules at different abstraction levels and obtain a fully verified large module or system.

Definition 4 (Vertical Composition): Given two verified atomic modules $\mathcal{M}_1, \mathcal{L}_1 \vdash_{\mathcal{R}} \mathbb{C}_1 : \mathcal{L}_2$ and $\mathcal{M}_2, \mathcal{L}_2 \vdash_{\mathcal{S}} \mathbb{C}_2 : \Psi_2$, we define $\mathcal{M} = \mathcal{M}_1 \oplus \mathcal{M}_2, \mathcal{L}_1 \vdash_{\mathcal{R} \circ \mathcal{S}} \mathbb{C}_1 \oplus \mathbb{C}_2 : \Psi_2$, where $\mathcal{M}_1$ and $\mathcal{M}_2$ exist at different abstraction levels.

B. END-TO-END SECURITY

As presented in section A of IV, to build a verified atomic module, we introduce an intermediate specification Σ, which seems to define a new "real machine" m that will mimic real programme implementations in terms of domain-visible states and logics while mimicking $\mathcal{M}$ in terms of programme states and transitions. Our end-to-end security is proven in two steps: first, we prove the security properties for the abstract machine $\mathcal{M}$; and second, similar properties can be proven for m using a refinement relation and the security properties of $\mathcal{M}$. However, the state transitions and state types of two different machines may differ. We assume that all machines under consideration use the same type for domain-visible states. In practice, our secure monitor proves that the use of a bi-simulation relation [14] between the m and $\mathcal{M}$ obeys this assumption.

1) ABSTRACT MACHINE $\mathcal{M}$ SECURITY

As described in section B of III, $\mathcal{M}$ security indicates that each individual transition preserves the noninterference property and requires a functional correctness proof as a precondition.

Definition 5 (Functional Correctness): We state that the machine $\mathcal{M}$ is functional correct with the specifications Ψ, which is written Correct($\mathcal{M}$, Ψ), when each transition $t (t \in \mathcal{T})$ of the machine follows its corresponding specification $\psi (\psi \in \Psi)$.

Definition 6 (Security): We state that the machine $\mathcal{M}$ is secure for domain $\mathcal{d}$ with the specifications Ψ when

Lemma 1: Correct($\mathcal{M}$, Ψ)

Lemma 2:

$$\begin{aligned} \forall s_1, s_2 \;\; &\in \;\; S_{\mathcal{M}}, s_1^{'}, s_2^{'}. \mathcal{O}_{\mathcal{d}}(s_1) = \mathcal{O}_{\mathcal{d}}(s_2) \wedge \\ (s_1, s_1{}') \;\; &\in \;\; \mathcal{T}_{\mathcal{M}} \wedge (s_2, s_2{}') \in \mathcal{T}_{\mathcal{M}} \\ &\Longrightarrow \mathcal{O}_{\mathcal{d}}\left(s_1^{'}\right) = \mathcal{O}_{\mathcal{d}}\left(s_2^{'}\right) \end{aligned}$$

Lemma 2 is a noninterference property that guarantees that a transition for which $\mathcal{d}$ is active in its target state depends only on the domain-visible states. In particular, a transition that ends in a state where the domain $\mathcal{d}$ is active is not influenced by data owned by the other domain.

2) REAL MACHINE m SECURITY

Similar properties can be proven for the real machine using the bi-simulation relation and the security properties of the abstract machine $\mathfrak{M}$. The following corollary states the noninterference property for the real machine.

Corollary 1: Let $\Sigma_m = \{\sigma | \exists s. \sigma \overset{bisim}{\Longleftrightarrow} s\}$ be the concrete states with the bi-simulation relation.

$$\begin{aligned} \forall \sigma_1, \sigma_2 \;\; &\in \;\; \Sigma_m, \sigma_1', \sigma_2'. \\ \mathcal{O}_{\mathcal{d}}(\sigma_1) \;\; &= \;\; \mathcal{O}_{\mathcal{d}}(\sigma_2) \wedge (\sigma_1, \sigma_1') \in \mathcal{T}_m \wedge (\sigma_2, \sigma_2') \in \mathcal{T}_m \\ &\Longrightarrow \mathcal{O}_{\mathcal{d}}(\sigma_1') = \mathcal{O}_{\mathcal{d}}(\sigma_2') \end{aligned}$$

Proof Sketch: Since σ_1 *and* σ_2 are in Σ_m, then $s_1 and s_2$ exist such that $\sigma_1 \overset{bisim}{\Longleftrightarrow} s_1$ and $\sigma_2 \overset{bisim}{\Longleftrightarrow} s_2$. We follow the assumptions and the definition of "$\overset{bisim}{\Longleftrightarrow}$" that $\mathcal{O}_{\mathcal{d}}(s_1) = \mathcal{O}_{\mathcal{d}}(s_2)$. According to the relation of bi-simulation, for $i = 1, 2, s_i{}'$ exists such that $(s_i, s_i{}') \in \mathcal{T}_{\mathcal{M}}$ and $\sigma_i{}' \overset{bisim}{\Longleftrightarrow} s_i{}'$. Thus, $\mathcal{O}_{\mathcal{d}}\left(\sigma_i{}'\right) = \mathcal{O}_{\mathcal{d}}\left(s_i{}'\right)$. We conclude the proof by showing that $\mathcal{O}_{\mathcal{d}}\left(\sigma_1{}'\right) = \mathcal{O}_{\mathrm{d}}\left(s_2^{'}\right)$ according to lemma 2.

C. IMPLEMENTATION

This framework has been implemented in several modelling languages and formal verification tools. To integrate all modelling language and tools and minimize the semantic gaps at specification interfaces, we provide an integrated verification

platform (Fig. 7) such that all verifications can be performed in Coq. The platform integrates C code verification, assembly code verification, programme abstraction, and the end-to-end property proofs. We have discussed the end-to-end security and its proofs. Next, we will provide additional implementation details about the C/assembly code verification and programme abstraction.

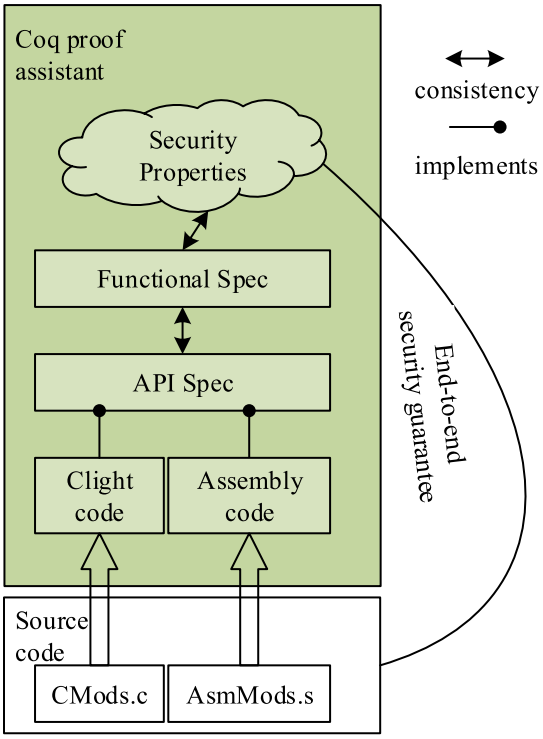

FIGURE 7. Integrated verification platform where the clight code can be generated by clightgen parser and the assembly code can be modeled according to our ARM assembly modelling language.

1) C CODE VERIFICATION

Since Verifiable C [15] is a separation logic that applies to the real C language and works in the Coq proof assistant environment, we use Verifiable C to specify the API of each function and prove that each function's body satisfies its API specification. To specify the API of a C function in Verifiable C, a person writes

DECLARE f WITH v
PRE [*params*]
PROP (𝒫) *LOCAL* (𝒬) *SEP* (ℛ)
POST [*ret*]
PROP (𝒫) *LOCAL* (𝒬) *SEP* (ℛ).

where f is the name of the function, *params* are the formal parameters, and *ret* is the return type. The separation's precondition and postcondition have the form *PROP* (𝒫) *LOCAL* (𝒬) *SEP* (ℛ), where $\mathcal{P}$ is a list of pure propositions, $\mathcal{Q}$ is a list of local/global variable bindings, and $\mathcal{R}$ is a list of separation logic predicates that describe the contents of memory. The *WITH* clause describes the abstract values v that can be referred to anywhere in the precondition and postcondition. Additional usage information of Verifiable C can be checked in [16].

To prove the functional correctness of each function, based on preconditions (input data structures available in parameters and global variables), we should prove that the function's return value and changes to the data structures follows its predicates in the postconditions.

2) ASSEMBLY CODE VERIFICATION

We model the ARM assembly code as a state machine with a register set and a memory state. An assembly atomic module can be considered a set of functions, where each function is the sequence of statements $s_0; s_1; \ldots; s_n$.

v ∈ Vars
c ∈ Constants
i ∈ Instructions ≔ add|mov|tst|eor| ...
e ∈ Expr ≔ v|c|i(e, ...)
s ∈ Stmt ≔ str(e, e)|v ≔ ldr(e)|
v ≔ e|b e|bl e

FIGURE 8. Syntax of ARM assembly code.

The syntax of the ARM assembly is shown in Fig. 8. Variables in *Vars* consist of machine registers (e.g., r0, r8, sp, lr, and pc), CPU flags (e.g., N, Z, C, and V) and memory. Memory is modelled as a map: $addr \rightarrow val$, where *addr* is an integer type that denotes the memory address and *val* is designed as a Coq inductive type as follows:

Inductive val : Type :=
|Vundef:val
|Vint:int ⟶ val
|Vptr:Positive ⟶ int ⟶ val.

where *val* is either a machine integer—a pointer: a pair of a memory addresses and an integer offset with respect to this address—or a *Vundef* value that denotes an arbitrary bit pattern, such as the value of an uninitialized variable.

Memory accesses are encoded using *mem.load* (*a*:*addr*) : *optionval* and *mem.store* (*a*:*addr*) (*va*:*val*) :*option mem*.

Assignment statements can assume one of following two forms: (1) $v{:=}e$ sets $v \in Vars$ to the value of expression e, and (2) $reg{:=}load(e)$ sets $reg \in regs$ to the value of the memory at address e.

The control flow changes with *b* and *bl* statements, which override the value of *pc*. *be* encodes a jump to an arbitrary location, either in the current function or the beginning of a function. *ble* is semantically equivalent to the ARM *bl* instruction. *bl* puts the address of the next instruction into *lr*, jumps to the instruction pointed by the label address (e) to execution and eventually returns to the caller by overriding *pc* using *lr*.

We define the state σ as a valuation of all variables in *Vars*. Let $\sigma(v)$ be the value of the variable $v \in Vars$ in the state σ, and similarly let $\sigma(e)$ be the valuation of expression e in state σ. Let $stmt(\sigma)$ be the statement executed in state σ. The semantics of the statement $s \in Stmt$ is given by the transition relation $\mathcal{T}$ over pairs of pre and post states, where $(\sigma, \sigma') \in \mathcal{T}$ if and only if $s = stmt(\sigma)$ and an execution of s exists starting

$$\langle str(e_a, e_{va}), \sigma\rangle \Downarrow \sigma\big[mem \mapsto \sigma(mem)[\sigma(e_a) := \sigma(e_{va})]\big]$$

$$\langle reg := ldr(e_a), \sigma\rangle \Downarrow \sigma\big[reg \mapsto \sigma(mem)[\sigma(e_a)]\big]$$

$$\langle reg := e, \sigma\rangle \Downarrow \sigma[reg \mapsto \sigma(e)]$$

$$\langle b\ e, \sigma\rangle \Downarrow \sigma[pc \mapsto \sigma(e)]$$

$$\langle bl\ e, \sigma\rangle \Downarrow \sigma[pc \mapsto \sigma(e), lr \mapsto next(\sigma(pc))]$$

FIGURE 9. Operational semantics of $s \in Stmt$: $(\sigma, \sigma') \in \mathcal{T}$ iff $\langle s, \sigma\rangle \Downarrow \sigma'$ and $stmt\ (\sigma) = s.\sigma\ [x \mapsto y]$ denotes a state that is Identical to σ, with the exception that variable x evaluates to y. The memory update expression $mem\ [x := y]$ returns a new memory that is equivalent to mem, with the exception of index x. $next(e)$ is the address of the subsequent instruction in the assembly programme after decoding the instruction starting at address e.

at σ and ending in σ'. We define operational semantics for *Stmt* in Fig. 9 and use standard semantics for the remaining statements. The sequence $\pi = [\sigma_0, \ldots, \sigma_n]$ is referred to as an execution trace if $(\sigma_i, \sigma_{i+1}) \in \mathcal{T}$ for each $i \in \{0, \ldots, n-1\}$. We also use $stmt(\pi)$ to denote the sequence of statements executed in π. The semantics of the function $f \in \mathcal{P}(Stmt)$ is given by the transition relation $\mathcal{T}$ over pairs of pre and post states, where $(\sigma, \sigma') \in \mathcal{T}$ if and only if $f = stmt(\pi)$ and an execution of a sequence of statements $[s_0; s_1; \ldots; s_n]$ starts at σ and ends in σ'.

We study the *TEE_switchctxt* (Fig. 4 (b)) expressed in our ARM assembly modelling language, as shown in Fig. 10. In Fig. 10 (a), we represent an assembly function as a Coq list and the (list instruction) as the contents of a sequence of instructions. The *CtxtPool* represents the location of the saved context of TEE and REE. The (*SAimm*(*Int.repr*20)) is a formalization of immediate number 20. In Fig. 10 (b), we formalize the expected behaviour of the list instruction. Specifically, these instructions will load and restore machine registers from and to a proper memory space. The current execution environment id must be greater than or equal to 0 and less than 2.

Next, we can prove that the *TEE_switchctxt* implements its API specification *TEE_switchctxt_spec_api* using a correctness lemma, as follows.

Lemma Correctness: forall mm′ n n′ rs rs′,
$\left(m', rs'\right)$ = exec_instrsTEE_switchctxtrsm
→ $n = rs\#r0$ → $n' = rs\#r1$
→ TEE_switchctxt _spec_api m (n n' rs) m' (rs').
Proof. . . . **Qed**.

3) PROGRAMME ABSTRACTION

The functional definition of the *TEE_switchctxt* can be formalized in Coq as this mathematical function:

FunctionTEE_switchctxt_functional
(a : Abs)(nn′ : Z) (rs : Regs) : (Abs ∗ Regs) :=
(a{ctxt : ZMap.set n rs(ctxt a)},
ZMap.get n′ (ctxta)).

Definition TEE_switchctxt : list instruction :=
(ldr r2 CtxtPool)::
(mov r3 SM_CTX_SEC):: . . .
(ldr sp r1 (SAimm (Int. repr 20)))::
(ldr lr r1 (SAimm (Int. repr 24)))::
(mov pc lr).

(a) *TEE_switchctxt* assembly code in Coq

Inductive TEE_switchctxt _spec_api :=
|forall m m0 m′ n n′ v0 v1 rs,
m. store CtxtPool (n ∗ SM_CTX_SEC) (r8 rs) = m0
→ . . .
→ m0. store CtxtPool (n ∗ SM_CTX_SEC)(lr rs) = m′
→ m′. load CtxtPool (n′ ∗ SM_CTX_SEC) = v0
→ . . .
→ m′. load CtxtPool (n′ ∗ SM_CTX_SEC + 24) = v1
→ rs#r0 = n → rs#r1 = n′
→ 0 <= n < 2 → 0 <= n′ < 2
→ let (rs′ := rs#r8 ← v0#r9. . . #pc ← v1) in
TEE_switchctxt _spec_api m (n n′ rs) m′ (rs′).

(b) *TEE_switchctxt* API specification in Coq

FIGURE 10. ARM assembly model and API specification of the *TEE_switchctxt* function, where the symbol "rs#pc ← v1" means updating pc's value to v1, where pc is an element of rs.

where Abs is the secure monitor abstract data (additional implementation details in section V). Z is Coq's type for (mathematical) integers. Regs represents machine registers. This functional definition saves the register context of environment n and restores the register context of environment n'. According to the abstract definition, we provide the following functional specification:

Inductive TEE_switchctxt_spec_functional :=
|forall a a' n n' rs rs',
TEE_switchctxt_functional a n n′ rs = (a′, rs′)
→ 0 <= n < 2
→ 0 <= n′ < 2
→ TEE_switchctxt_spec_functionala (n n' rs) a' (rs').

As shown in the following definition, we connect the implementation level memory to the abstract data by an equivalence relationship.

Definition EqRelation (m:mem) (a:abs):Prop :=
m.loadCtxtPool0 = r8(ZMap.get0(ctxta))
∧m.load CtxtPool4 = r9(ZMap.get0(ctxta))
∧. . .
∧m.load CtxtPool20 = sp(ZMap.get0(ctxta))
∧m.load CtxtPool24 = lr(ZMap.get0(ctxta))
∧m.load CtxtPool(SM_CTX_SEC) =
r8(ZMap.get 1(ctxta))
∧m.load CtxtPool(SM_CTX_SEC + 4) =
r9(ZMap.get1(ctxta))
∧. . .
∧m.load CtxtPool(SM_CTX_SEC + 20) =
sp(ZMap.get1(ctxta))
∧m.loadCtxtPool(SM_CTX_SEC + 24) =
lr(ZMap.get1(ctxta)).

We prove that the *TEE_switchctxt* coincides with its functional specifications by a bi-simulation proof. Once we complete the code correctness proof and simulation proof, we can reason the security properties based on the abstract functional specifications without regarding the implementation details of the code.

V. SECURITY VERIFICATION OF SECURE MONITOR

In section B of IV, we provide a security definition, where the noninterference property requires that a transition in a domain cannot be influenced by the data owned by the other domain. However, data may be passed between the REE domain and TEE domain, which can introduce explicit data dependencies. We need to reduce the security condition to noninterference in the special case where no communication occurs, with the exception of the secure channel. This reduced property formalizes the intuition that if the secure channel is removed, then the secure monitor cannot introduce a communication channel via machine registers, a context memory pool or any other way.

At the abstract machine level, we decompose the security verification task into 1) correctness verification of secure monitor's API implementation (e.g., integer overflow checking, and accessible border checking) and 2) noninterference verification of the scheduler. Corollary 1 (section B of IV) guarantees the corresponding security over the real machine level.

A. CORRECTNESS OF SECURE MONITOR'S API

1) ABSTRACT STATE

We formalize the secure monitor's abstract state as a Coq record that consists of 3 separate fields: *shmem*, *cid* and *ctxt*.

Record BufInfo{
addr : Z;
len : Z;
contents : listbyte}
Record Context{
r8 : Z; r9 : Z; . . . ; sp : Z; lr : Z; pc : Z}.
Definition CtxtPool := ZMap.tContext.
Record Abs{
shmem : BufInfo;
cid : Z;
ctxt : CtxtPool}

The *shmem* represents shared memory between the TEE and REE, which is the *BufInfo* type with the buffer's address, length and contents.

The *cid* is an integer type and represents the current execution environment id.

The *ctxt* is the *CtxtPool* type. At the abstract level, the *CtxtPool* is no longer the location of saved machine context; instead, we formalize it as a Coq finite map (ZMap.t).

2) VERIFICATION PLAN

As discussed in section III, instead of treating the entire secure monitor as a single abstraction, we can divide it into individual abstraction levels, such as the *TEE_switchctxt* at a low abstraction level and *channel_send/recv* and *TEE_sched* at a high abstraction level. The correctness proof can be performed in a smaller module, and a larger module can be achieved by cross-abstraction links. The cross-abstraction link means that a high-level programme can directly call the low-level specifications without considering the underlying concrete implementation. Thus, when we consider these simplifications, we establish a verification plan for the secure monitor shown in Fig. 11.

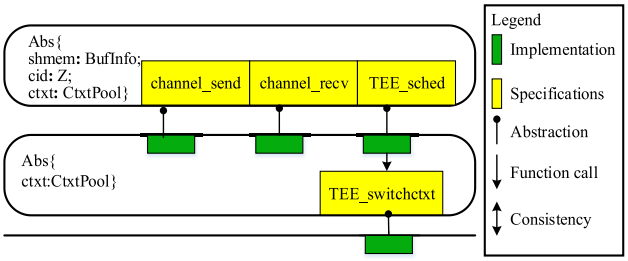

FIGURE 11. Verification plan for the secure monitor.

3) VERIFICATION OF APIS

(a) *channel_recv*(*des*, *size*) must 1) check that the destination buffer is in a user space, and must not 2) modify any memory location outside the destination buffer. We write its functional specification as

Inductive*channel_recv_spec_functional* :=
|forall a d n d′,
channel_recvadn = (a, d′)
→ PM_USRLO < (addrd) ≤ PM_USRHI
→ PM_USRLO < (addrd) + n ≤ PM_USRHI
→ 0 < n < lend
→ (addrd) < (addrd) + n
→ channel_recv_spec_functionala(dn)a(d′).

where a is the *Abs* type, *d* and *d′* are the *BufInfo* type, and *n* is the integer type. The conditions *PM_USRLO* < (*addrd*) <= *PM_USRHI* and *PM_USRLO* < (*addrd*) + *n* <= *PM_USRHI* ensure that the received message is written to a memory region within the user space. The condition 0 < *n* < *lend* ensures that the message is not written to the destination buffer. The condition (*addrd*) < (*addrd*) + *n* ensures that an integer overflow is not exploited to violate secure communication.

(b) *channel_send*(*scr*, *size*) must 1) check that the destination buffer is in a nonsecure shared memory space, and must not 2) modify any memory location outside the shared memory. Formally, we write its functional specification as follows:

Inductivechannel_send_spec_functional :=
|forall a s n s',
channel_sendasn = (a′, s′)
→ (addrs) = PM_NSSHMLO
→ PM_NSSHMLO < (addrs) + n ≤ PM_NSSHMHI
→ 0 < n < lens
→ channel_send_spec_functional a (sn) a′(s′).

The conditions $(addrs) = PM\_NSSHMLO$ and $PM\_NSSHMLO < (addrs) + n \leq PM\_NSSHMHI$ ensure that the message that will be sent is written to a memory region within the shared memory buffer. The condition $0 < n < lens$ ensures that the message is not written out to the shared memory buffer.

(c) The *TEE_sched* ensures execution environment scheduling and that the machine context correctly saves/restores.

As shown in Fig. 11, we separate the *TEE_sched* into two layers. At the low layer, we use an equivalence relationship *EqRelation*(*m*:*mem*)(*a*:*abs*) :*Prop* (section IV) to ensure that the concrete memory *m.CtxtPool* and the abstract state *a.ctxt* contains the same values. Thus, the high layer code of *TEE_sched* can directly manipulate the abstract state *a.ctxt* without considering its underlying concrete implementation. After we abstract the concrete memory and *TEE_switchctxt* operations, we formalize *TEE_sched* on top of the functional specification of *TEE_switchctxt*. The functional formalization of *TEE_sched* is shown as follows:

FunctionTEE_sched_functional
(a : Abs)(rs : Regs) :(Abs ∗ Regs) :=
letcurid := cidain
ifzeqcurid0 then
(a{ctxt : ZMap.setcuridrs(ctxta)}
cid : 1}, ZMap.get1(ctxta))
else
(a{ctxt : ZMap.setcidrs(ctxta)}
{cid : 0, ZMap.get0(ctxta)).

Note that this abstraction function directly manipulates the abstract state of machine context via the Coq ZMap.get/set operators instead of accessing the concrete memory via the *m.store*/*load* operators. By this decomposition, we achieve the following objective: at the high layer, the execution contexts and return code pointers in concrete memory are abstracted away and modelled as mathematical structures that can be accessed only by an abstract environment scheduling operation. At the low layer, we only need to ensure that the machine context switching saves registers into and loads new values from proper memory places.

B. VERIFYING NONINTERFERENCE PROPERTY OF THE SCHEDULER

We consider that each world ID is a distinct domain. The noninterference property is proven by showing that every step of execution preserves an equal relation-: the domain-visible portions of two states are equal. For simplicity, we use ''equal states'' to represent ''the domain-visible portions of two states are equal''.

1) DOMAIN-VISIBLE FUNCTION

We now define the abstract machine level domain-visible function used in our verification. For the given world ID d, the state visibility of s is defined as follows:

- Registers—All machine registers are visible if s is active.
- Active—The machine register is visible regardless of whether *wid* (s) is equal to *d*.
- Register Context—The saved register context of *d* is visible.

Since machine registers are shared by different worlds, to verify noninterference property of the scheduler, the following problem need to be addressed.

Consider any world ID *d*. For any abstract machine level state s, we state that s is ''active'' if *wid* (s) = *d* and ''inactive'' otherwise. If two worlds are isolated, then registers should be visible to d only in active states. What happens if we attempt to prove that an equal relation is preserved when starting from two inactive equal states? Since states are inactive, the registers are invisible, and the registers may have different values in the two states. The two inactive ''equal states'' may execute different instructions, and the resulting active states may not be equal. Thus, the noninterference property will not be preserved in this situation.

The fundamental issue is that when an active world switches to an inactive world, we cannot ensure that the active world's saved register context is not modified.

To address this problem, we divide the proof task into three separate lemmas:

- **Noninterference.** If two active equal states take a step to two inactive states, then these inactive states are equal.
- **Integrity.** If an inactive state takes a step to another inactive state, then these states are equal.
- **Noninterference Restore.**If two inactive equal states take a step to two active states, then these active states are equal.

These lemmas are chained together, as pictured in Fig. 12. The dashed lines indicate that the states are equal. Thus, the noninterference lemma establishes the equal relation of the initial inactive states after world switching; the integrity lemma establishes the equal relation of the inactive states immediately preceding a world switching back to the active world; and the noninterference restore lemma establishes the equal relation of the active states after switching back to the active world.

In the actual proof process, we discover that the world context switch does not save r0-r7 machine registers. After a context switch from one world i to another world j, these registers may contain values that are private to i. We solve this insecurity by clearing all unsaved machine registers to zero before a context switch jump.

C. CODE SIZE AND PROOF EFFORT

Table 1 shows a breakdown of the number of lines of code, excluding comments and whitespace, in Coq. C lines are clight codes generated by the clightgen parser. Assembly lines are ARM assembly instructions written in our ARM model. Specification lines include all machine-checked code: our ARM machine model, API or functional

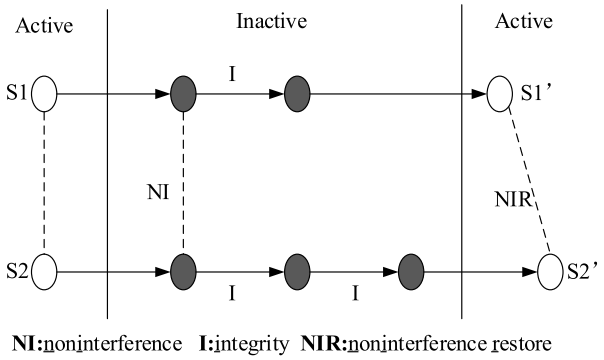

FIGURE 12. Applying the three lemmas to prove the noninterference property of the world scheduler.

TABLE 1. Approximate LOC of proof effort.

Component	C lines	Assembly lines	Spec	Proof
ARM model	×	×	910	87
scheduler	×	109	317	1362
secure channel	179	×	397	3743
noninterference	×	×	162	1009
Total	179	109	1786	6210

specification of the secure monitor, and noninterference properties. Proof lines are proof related codes, such as pre- and postcondition, loop invariants, lemmas and assertions.

We spent a total of approximately 7 person months implementing and specifying the TEE. We first implemented and specified a simplified version of TEE. This work mainly included implementation of the TEE scheduler, specification of the TrustZone-based ARM machine model, functional correctness proofs of the scheduler and refinement proofs. Building this first version took approximately 5 person months, including a steep learning curve for the developers who were unfamiliar with the ARM TrustZone platform and the Verifiable C tool.

We then extended the implementation and specification with the secure channel; this totalled 2 person months of extra work, including 1.5 person months for the functional correctness proofs of the secure channel, refinement proofs and the noninterference proofs.

Thanks to our modular verification framework, we can easily compose the scheduler's functional specifications and the secure channel's functional specifications together, even if they were developed at different stages and using different languages.

VI. EVALUATION

This section presents a performance evaluation of the TrustZone-based TEE and an analysis of its security.

A. PERFORMANCE

The software stack that we evaluated consists of the verified secure monitor and the Trust-E [17] (a trusted kernel and several TAs) developed by our team. For brevity, we call the software stack as VTEE for short. We evaluate two sources of performance overhead for TAs when compared to standard Linux applications and the OPTEE [18] applications: (i) the overhead due to executing cross-environment switching, and (ii) the overhead due to executing the secure channel send and receive primitives.

1) METHODOLOGY

To evaluate the performance of the VTEE, we run multiple experiments on the ARM HiKey 960 hardware platform.

We use our two TAs: SeStorage and SePayment. The SeStorage is to store a personal clinical history on the mobile platform and to give patients secure access to this information during patient visits. Our SeStorage application involves three actors: the hospital, the patients, and the SeStorage TA. The hospital encrypts the clinical history records and access control policies to the SeStorage TA. When a patient asks for a record, the request is encrypted to the TA. The TA decrypts it, checks whether the provider's permissions meet the access control policy, and returns the relevant information if the policy is met. The SePayment enables convenient mobile payment. Our SePayment scenario involves three actors: a bank, which issues credit card information, the SePayment TA, which keeps track of the credit card details, and the shopping website. To perform a transaction, the shopping website issues an encrypted challenge to the TA that includes the transaction amount. If the user authorizes the transaction, the TA answers the challenge; otherwise, it aborts. Next, the TA communicates with the banks to record the transactions.

The use case prototypes allow us to measure the VTEE performance with realistic applications. In total, we evaluate 8 methods: four for SeStorage and four for SePayment. We run each test on the VTEE, the OPTEE and on a standard Linux and then compute the difference. We have conducted each test 200 times and report the average value here.

2) PERFORMANCE OF TA CODE EXECUTION

Fig. 13 plots the evaluation results of our use case prototypes showing the execution time of all TA methods on the VTEE, OPTEE and on Linux. The results show that Linux slightly outperforms the OPTEE and the OPTEE slightly outperforms the VTEE. The results are in line with our expectations since we infer that, in Linux, the application for evaluation has no overhead of the cross-environment switching and the secure communication. In OPTEE, the application for evaluation has no overhead of the secure communication but needs an additional overhead of the cross-environment switching. However, in VTEE, the TA needs both overheads. Because these overheads are fixed, the longer the execution time of TA methods, the closer the performance of different systems will be. Thus, as shown in Fig. 13, the performance of Linux, OPTEE and VTEE are similar except M2 and M6.

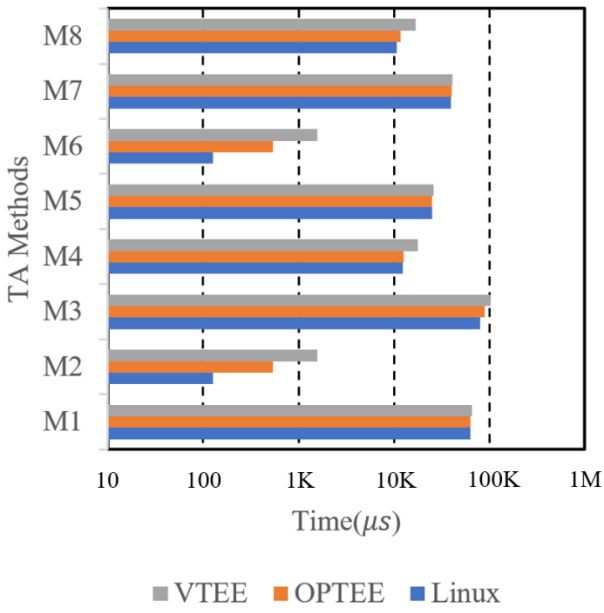

FIGURE 13. Execution time of TA methods from our use case prototypes. These methods from M1 to M8 are SeStorage-QueryRecords, SeStorage-InitQuery, SeStorage-SetRecords, SeStorage-Encrypt, SePayment-Pay, SePayment-InitPay, SePayment-SetCard, and SePayment-Encrypt, respectively.

3) PERFORMANCE OF THE VTEE PRIMITIVES

To further evaluate the concrete overhead of our VTEE primitives, we ran some additional experiments. We first evaluated the switch time between the REE and the TEE, which is performed by running an empty service in the VTEE and OPTEE, respectively. The result shows that the switch time cost is approximately 0.63 ms in VTEE and 0.71 ms in OPTEE. The switch time is very small and can be well accepted.

We then measure the performance overhead of the secure channel communication in VTEE. Because the encryption and decryption durations depend on their parameter size, we further investigate the factors responsible for such variation. Fig. 14 plots our evaluation results of send and receive primitives as we vary the size of the data to be encrypted and decrypted, respectively. Encrypting 1KB takes 13.6 ms and decrypting the same amount of data takes 123.4 ms. The send/receive time can be well accepted because the size of the security-sensitive communication data is usually less than 1 KB in embedded platform.

B. SECURITY ANALYSIS

The attack surface of our TrustZone-based TEE is the secure monitor interface exposed to the REE. We designed a relatively narrow interface, which limits the exposure of code vulnerabilities. Moreover, all of the security-sensitive information is encrypted by the secure channel after it leaves the TEE. Thus, even a powerful adversary who controls the hardware and system software in the REE cannot obtain useful information. Our functional correctness proof ensures

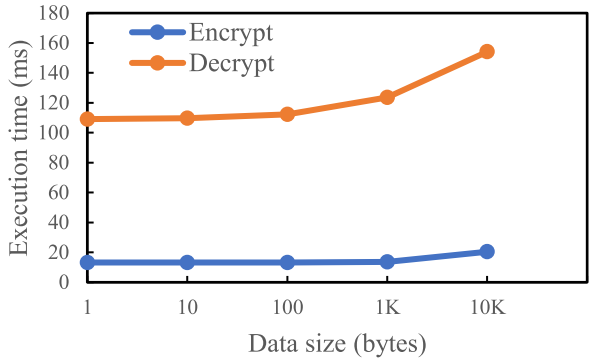

FIGURE 14. Performance of the secure channel send and receive primitives varying the size of encrypted and decrypted data, respectively.

that the TEE is immune to common software attacks such as integer overflow, buffer overflow and code logic errors. Our noninterference proof ensures that the TEE secrets do not leak to and that TEE cannot be influenced by REE.

The TrustZone-based TEE can only provide limited protection against physical attacks: an attacker with the capability of tampering with the hardware can disable the TrustZone protections and attack the TEE, for example, the covert channel attack that is based on time or energy consumption. However, this kind of attacks require a high degree of sophistication.

VII. RELATED WORK

To solve information security problems, building a trusted-hardware-based TEE has become a research hotspot in colleges and enterprises. On the x86 platform, [19] built a TEE based on a trusted platform module (TPM) and trusted kernel. The TPM is used for a trusted boot of the trusted kernel, which provides a runtime protection for security applications. Since this kind of trusted kernel must provide functions such as a network protocol stack and file system, the size of the trusted computing base (TCB) becomes very complicated and guaranteeing security is difficult. To reduce the complexity of the TCB, Intel SGX [20] was developed. SGX provides a TEE for security-critical applications by extending the x86 instruction set to isolate applications from complex system software, such as Linux, Windows, and hypervisors. Reference [21] formalized the information flow property for a single trusted application that runs in the SGX secure container. However, all verification work is at the machine code level with high verification complexity. On the ARM platform, TrustZone provides similar hardware protection. Reference [22] builds a software system based on TrustZone that can execute Linux applications in a TEE; however, the system has a large TCB and does not use any encryption/decryption scheme to ensure the private data. Reference [23] builds a TEE for mobile applications based on the ARM TrustZone and provides secure protocols to ensure confidentiality and integrity of sensitive data. In their design framework, each trusted application can independently communicate with the outside world. If we use formal methods to verify security properties of trusted applications, we need

to separately verify each application, which significantly increases the amount of verification work.

The most important security property for isolated domains is information flow security. Information flow verification involves tracking the flow of confidential information in the programme and checking whether the confidential information directly or indirectly flows to the data objects that are visible to the adversary [24]–[26]. [27]–[29] propose a type system to mark the variables that hold private information and check whether the programme has covert channels that cause information leakage. References [30], [31] provides a language-based information flow verification strategy that requires a specific language and introduces many annotation variables that increase the burden on software developers. In addition, this approach uses complex system software (e.g., OS, hypervisor) as part of the TCB. In comparison, the design of this article excludes complex system software, and the TCB only includes trusted hardware and TEE software. Reference [32] proves the isolation property of a simple separation kernel that runs on physical-separated machines. Instead of using standard information flow analysis techniques (e.g., noninterference), they prove a property by stating that the machine execution is trace-equivalent to execution over an idealized model. Their methodology is fairly different from ours, as we verify the correctness and security properties of a secure channel and then separate it from a secure monitor and prove noninterference.

An important work in the area of formal foundational software systems is the certified CertiKOS kernel [33], [34]. There are some similarities between the noninterference proof of CertiKOS and that of our secure monitor, as both noninterference properties are defined following the unwinding condition theory using a similar observation function or domain-visible function. Moreover, both security proofs are conducted over an abstract specification and then propagated down to a concrete C or assembly implementation. Our work, however, has three important differences from the CertiKOS work. First, the assembly model of CertiKOS is mainly built for the x86 architecture, whereas our assembly model is built for the TrustZone-based ARM architecture. Second, the CertiKOS extends CompCert [35] for low-level reasoning and uses less abstract low-level specifications to simulate the realistic calculations and memory operations of a programme. However, this kind of low-level specification cannot handle complicated pointer data structures well. For security and verification purposes, CertiKOS uses bounded arrays instead of traditional pointer-linked lists in their code implementations. In contrast, our API specification uses separation logic to specify efficient embedded programmes in which there exist considerable pointer data structures and relevant operations. Third, to solve the incompatibility between refinement relations and propagate high-level security properties down to a concrete implementation correctly, CertiKOS uses a relatively complex predicate to build their simulation relations between the high-level and low-level specifications. Our API specifications are more abstract than their low-level specifications, which allows us to achieve the following two objectives: the abstract state in the functional specification is very similar to the abstract state in the API specification, such that we can establish a simple equivalent relationship instead of a complex predicate relationship, which reduce the bi-simulation proof efforts (Corollary 1, section IV), and the equivalent relationship contains all the domain-visible state that can be used to solve the incompatibility between refinement relations.

The Serval [36] aims for correctness and security verification of a system software, which shares a similar goal to ours: provide an end-to-end security guarantee for critical modules of an embedded system. However, the overall approaches are quite different. Serval uses Rosette [37] and relies on SMT solving, which allows for more automation. Our approach uses Coq, which can express richer properties that Serval cannot. For noninterference, Serval tends to prove Nickel's [38] specification instead of our noninterference specification. However, it is hard to compare the two noninterference specifications since they are written in different logics and verification tools. Another difference is in the treatment of unbounded loops. Serval must retrofit system implementation and require loops to be bounded for automated verification. Contrast this to our approach, where we directly verify correctness of programmes with unbounded loops (e.g., the memory copy and compare functions used in the secure channel) using Coq.

VIII. CONCLUSION

We presented a methodology for designing a TrustZone-based TEE, which enables verification of a TEE's functional correctness and security properties. A TEE multi-level architecture and a narrow communication interface are designed to reduce the difficulty of verification. We also presented a modular verification framework for verifying the end-to-end security of C and ARM assembly programs. The correctness of an environment scheduler is verified for different abstraction levels. A flexible domain-visible function is used to specify the security property, prove noninterference via unwinding, and soundly propagate the security guarantee across bi-simulation. Our evaluation shows that our VTEE achieves high security confidence with an acceptable performance cost.

ACKNOWLEDGMENT

The authors would like to thank article reviewers for their many valuable comments. The authors would also like to thank AJE for its linguistic assistance during the preparation of this manuscript.

REFERENCES

[1] ARM Technical White Paper. *ARM Security Technology-Building a Secure System Using TrustZone Technology*. Accessed: Apr. 1, 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[2] D. Rosenberg. *QSEE TrustZone Kernel Integer Overflow Vulnerability*. Accessed: Jul. 1, 2014. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf

[3] J. Hruska. *HTC Caught Storing Fingerprint Data in Unencrypted Plain Text*. Accessed: Aug. 10, 2015. [Online]. Available: https://www.extremetech.com/mobile/211985-htc-caught-storing-fingerprint-data-in-unencrypted-plain-text

[4] J. Jürjens, "Secrecy-preserving refinement," in *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliveira and P. Zave, Eds. Berlin, Germany: Springer, 2001, pp. 135–152.

[5] C. Morgan, "The shadow knows: Refinement and security in sequential programs," *Sci. Comput. Program.*, vol. 74, no. 8, pp. 629–653, Jun. 2009, doi: 10.1016/j.scico.2007.09.003.

[6] *The Coq Proof Assistant Reference Manual Introduction and Contents*. Accessed: Jul. 23, 2018. [Online]. Available: https://coq.inria.fr/distrib/current/refman/

[7] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski, "SeL4: Formal verification of an OS kernel," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ. (SOSP)*, Big Sky, MT, USA, 2009, pp. 207–220.

[8] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 1–70, Feb. 2014, doi: 10.1145/2560537.

[9] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. Secur. Privacy*, Oakland, CA, USA, Apr. 1982, p. 11.

[10] J. A. Goguen and J. Meseguer, "Unwinding and inference control," in *Proc. IEEE Symp. Secur. Privacy*, Oakland, CA, USA, Apr./May 1984, p. 75.

[11] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th Annu. IEEE Symp. Logic Comput. Sci.*, Copenhagen, Denmark, Jul. 2002, pp. 55–74.

[12] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *Automated Technology for Verification and Analysis*, S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds. Berlin, Germany: Springer, 2008, pp. 64–79.

[13] S. Bensalem, M. Bozga, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, "Component-based verification using incremental design and invariants," *Softw. Syst. Model.*, vol. 15, no. 2, pp. 427–451, May 2016, doi: 10.1007/s10270-014-0410-8.

[14] D. J. Walker, "Bisimulation and divergence," *Inf. Comput.*, vol. 85, no. 2, pp. 202–241, Apr. 1990, doi: 10.1016/0890-5401(90)90048-M.

[15] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "VST-Floyd: A separation logic tool to verify correctness of C programs," *J. Automat. Reasoning*, vol. 61, nos. 1–4, pp. 367–422, Jun. 2018, doi: 10.1007/s10817-018-9457-5.

[16] A. W. Appel, *Program Logics for Certified Compilers*. New York, NY, USA: Cambridge Univ. Press, 2014.

[17] X. Yang, P. Shi, B. Tian, B. Zeng, and W. Xiao, "Trust-E: A trusted embedded operating system based on the ARM trustzone," in *Proc. IEEE 11th Int. Conf. Ubiquitous Intell. Comput., IEEE 11th Int. Conf. Auton. Trusted Comput., IEEE 14th Int. Conf. Scalable Comput. Commun., Assoc. Workshops*, Bali, Indonesia, Dec. 2014, pp. 495–501.

[18] Jforissier. *The Secure Side Implementation of OP-TEE Project*. Accessed: Jan. 11, 2020. [Online]. Available: https://github.com/OP-TEE/optee_os

[19] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 193, Dec. 2003, doi: 10.1145/1165389.945464.

[20] F. McKeen, "Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave," in *Proc. Hardw. Archit. Support Secur. Privacy*, Seoul, South Korea, Jun. 2016, Art. no. 10.

[21] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Denver, CO, USA, Oct. 2015, pp. 1169–1184.

[22] L. Guan, "TrustShadow: Secure execution of unmodified applications with ARM TrustZone," in *Proc. 15th Annu. Int. Conf. Mobile Syst. Appl. Services*, New York, NY, USA, Jun./Jun. 2017, pp. 488–501.

[23] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 67–80, Feb. 2014, doi: 10.1145/2644865.2541949.

[24] J. McLean, "Proving noninterference and functional correctness using traces," *J. Comput. Secur.*, vol. 1, no. 1, pp. 37–57, Jan. 1992.

[25] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *ACM SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, Dec. 1997, doi: 10.1145/269005.266669.

[26] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2011, pp. 165–179.

[27] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *Software Security—Theories and Systems*, K. Futatsugi, F. Mizoguchi, and N. Yonezaki, Eds. Berlin, Germany: Springer, 2004, pp. 174–191.

[28] G. Barthe and L. P. Nieto, "Secure information flow for a concurrent language with scheduling," *J. Comput. Secur.*, vol. 15, no. 6, pp. 647–689, Sep. 2007.

[29] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, nos. 2–3, pp. 167–187, Jan. 1996.

[30] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977, doi: 10.1145/359636.359712.

[31] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003, doi: 10.1109/JSAC.2002.806121.

[32] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal verification of information flow security for a simple arm-based separation kernel," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Berlin, Germany, Nov. 2013, pp. 223–234.

[33] R. Gu, "Deep specifications and certified abstraction layers," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Mumbai, India, Jan. 2015, pp. 595–608.

[34] D. Costanzo, Z. Shao, and R. Gu, "End-to-end verification of information-flow security for C and assembly programs," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Santa Barbara, CA, USA, Jun. 2016, pp. 648–664.

[35] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, doi: 10.1145/1538788.1538814.

[36] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with Serval," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, Huntsville, ON, Canada, Oct. 2019, pp. 225–242.

[37] J. Bornholt and E. Torlak, "Finding code that explodes under symbolic evaluation," *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018, Art. no. 149, doi: 10.1145/3276519.

[38] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang, "Nickel: A framework for design and verification of information flow control systems," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement.*, Carlsbad, CA, USA, Oct. 2018, pp. 287–306.

HAIYONG SUN received the bachelor's degree in software engineering from Hangzhou Dianzi University. He is currently pursuing the Ph.D. degree with the School of Information and Software Engineering, University of Electronic Science and Technology of China. His research is focused on the specification and verification of an embedded operating system kernel.

HANG LEI received the Ph.D. degree in computer science from the University of Electronic Science and Technology of China, China, in 1997. After graduation, he conducted research in the fields of real-time embedded operating systems, operating system security, and program verification, as a Professor with the Department of Computer Science, University of Electronic Science and Technology of China, where he is currently a Professor (a Doctoral Supervisor) with the School of Information and Software Engineering. His research interests include big data analytics, machine learning, and program verification.