



Refinement-based Modeling and Formal Verification for Multiple Secure Partitions of TrustZone

Fanlang Zeng (曾凡浪)^{1,2}, Rui Chang (常瑞)^{1,3}, Hao Xu (许浩)^{1,2}, Shaoping Pan (潘少平)^{1,2}, Yongwang Zhao (赵永望)^{1,3}

¹ (School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

² (ZJU Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311200, China)

³ (Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, Zhejiang University, Hangzhou 310027, China)

Corresponding author: Rui Chang, crlx1021@zju.edu.cn

Abstract As a trusted execution environment technology on ARM processors, TrustZone provides an isolated and independent execution environment for security-sensitive programs and data on the device. However, running the trusted OS and all the trusted applications in the same environment may cause problems—The exploitation of vulnerabilities on any component may affect the others in the system. Although ARM proposed the S-EL2 virtualization technology, which supports multiple isolated partitions in the secure world to alleviate this problem, there may still be security threats such as information leakage between partitions in the real-world partition manager. Current secure partition manager designs and implementations lack rigorous mathematical proofs to guarantee the security of isolated partitions. This study analyzes the multiple secure partitions architecture of ARM TrustZone in detail, proposes a refinement-based modeling and security analysis method for multiple secure partitions of TrustZone, and completes the modeling and formal verification of the secure partition manager in the theorem prover Isabelle/HOL. First, we build a multiple secure partitions model named RMTEE based on refinement: an abstract state machine is used to describe the system running process and security policy requirements, forming the abstract model. Then the abstract model is instantiated into the concrete model, in which the event specification is implemented following the FF-A specification. Second, to address the problem that the existing partition manager design cannot meet the goal of information flow security verification, we design a DAC-based inter-partition communication access control and apply it to the modeling and verification of RMTEE. Lastly, we prove the refinement between the concrete model and the abstract model, and the correctness and security of the event specification in the concrete model. The formalization and verification consist of 137 definitions and 201 lemmas (more than 11,000 lines of Isabelle/HOL code). The results show that the model satisfies confidentiality and integrity, and can effectively defend against malicious attacks on partitions.

Keywords trusted execution environment; secure partition; theorem proving; refinement; security analysis

This is the English version of the Chinese article “基于精化的 TrustZone 多安全分区建模与形式化验证. 软件学报, 2023, 34(8):3507–3526. DOI: 10.13328/j.cnki.jos.006866”

Funding items: Key Research and Development Program of Zhejiang Province (2022C01165); National Natural Science Foundation of China (62132014); Zhejiang Province Spearhead Plan (2022C01045); Special Fund of Fundamental Research Funds for the Central Universities of China (NGICS)

Received 2022-09-04; Revised 2022-10-13; Accepted 2022-12-14; IJSI published online 2023-09-27

Citation Zeng FL, Chang R, Xu H, Pan SP, Zhao YW. Refinement-based modeling and formal verification for multiple secure partitions of TrustZone. *International Journal of Software and Informatics*, 2023, 13(3): 297–321. <http://www.ijsi.org/1673-7288/301.htm>

Computing systems have become increasingly complex with the wide application and diversification of computing devices, providing a larger attack surface for virus software and malicious attacks. Software providers and users require reliable defense mechanisms to protect the security of their code and data. The ARM TrustZone^[1] based Trusted Execution Environment (TEE)^[2] protects secrets with hardware isolation mechanisms. However, there are still threats in practical applications. Pinto *et al.*^[3,4] analyzed the security issues existing in the major commercial TrustZone systems and found a considerable number of vulnerabilities and bugs in their security monitors, trusted Operating Systems (OSs), and Trusted Applications (TAs). The Common Vulnerabilities and Exposures (CVE) database also continuously reveals the problems in TrustZone. In CVE-2021-34389^[5], the incorrect bounds check in the TA of Nvidia TEE can allow a local user through a malicious client to access the heap memory in the TrustZone. A vulnerability found in the CSU driver of OPTEE-OS in CVE-2021-44149^[6] allows the normal world to bypass TrustZone and arbitrarily read and write secure world memory.

ARMv8.4 has proposed the S-EL2 virtualization architecture^[7] and the Application Binary Interface (ABI) specifications for partition manager in Firmware Framework for ARM A-profile (FF-A)^[8] to alleviate such problems. Specifically, the secure world is divided into multiple isolated Secure Partitions (SPs), with Trusted OSs and TAs running in different partitions and managed by Secure Partition Managers (SPMs) at the higher privilege level. In this architecture, the SPM and the secure monitor are the Trusted Computing Base (TCB). The partition manager provides message communication interfaces between partitions to support functional invocation and data transmission between partitions.

However, this architecture lacks strict mathematical proofs to ensure the consistency of design and implementation, and there are still security threats, such as illegal memory access and malicious interface calls. The specific issues are as follows.

- (1) The partition manager is a key component to ensure the secure isolation of partitions. An improperly implemented SPM may be compromised by malicious programs, leading to security risks for the system. For example, CVE-2021-30278^[9] reveals that improper input validation of the memory transfer interface in Qualcomm TrustZone can lead to information disclosure in several system applications.
- (2) In the multiple partitions system design of ARM, the communication between partitions is unlimited, which means that information flows arbitrarily between partitions. This weakens the effect of partition isolation and fails to meet the requirements of formal verification of information flow security.

To address the above problems, we propose a refinement-based modeling and verification method for multiple secure partitions of TrustZone, design the model of Refinement-based Multiple secure partitions Trusted Execution Environment (RMTEE), and verify the functional correctness and information flow security with machine-checkable proofs in the theorem prover Isabelle/HOL. Inspired by the refinement method in program development^[10], we adopt the refinement calculus^[11] in the formal model. Starting from the high-level abstract specification, the details of data and functions are gradually increased. Each step of refinement gets a specification closer to the underlying implementation and eventually builds the specification of the implementation. Furthermore, we propose the Inter-Partition Communication (IPC) access control based on Discretionary Access Control (DAC) to enforce inter-partition communication security. We define an Access Control Matrix (ACM) in the system configuration, which is

queried during ABI calls to validate the legitimacy of the calls, ensuring that the system meets information flow security requirements.

The main contributions of this paper are as follows.

- (1) A refinement-based modeling method for multiple secure partitions of TrustZone. We build the RMTEE model, which includes an abstract model and a concrete model. The abstract model consists of an abstract state machine and security properties, and the concrete model includes the execution model and event specification. The execution model instantiates the abstract model using concrete variables and functions, and the event specification implements the concrete event interface according to the partition manager standard from the ARM FF-A specification. The RMTEE model contains 137 definitions and 1,462 LOC in Isabelle/HOL.
- (2) A security enhancement mechanism of inter-partition communication. We discover the potential threat of information flow in the FF-A specification through formal verification. To address this deficiency, we propose the DAC-based security enhancement mechanism for inter-partition calls. It is applied to the formalization and verification, and we analyze two types of attacks that the mechanism can defend against.
- (3) The correctness and security verification of RMTEE. By means of the theorem proving method, we verify the refinement between the abstract model and the concrete model of RMTEE, as well as the correctness of the event interface and information flow security in the concrete model. The proofs are carried out with 201 theorems and 9,715 LOC in Isar. The results prove that RMTEE conforms to confidentiality and integrity.

Section 1 of this paper introduces the research progress in the field of this work. Section 2 analyzes the threat model and security assumptions. Section 3 presents the formal modeling and verification method of this work. Sections 4 to 6 describe the abstract model, the concrete model specifications, and the concrete model proofs, respectively. Section 7 conducts a security analysis and evaluation of this work. Section 8 provides a summary of this work.

1 Background and Related Work

1.1 Multiple secure partitions of TrustZone

There are efforts in academia and industry to establish secure partitions using the isolation feature of TrustZone. In some work, partitions are also known as Virtual Machines (VMs). TZ-RKP^[12] runs Hypervisor in the secure world to provide high privilege levels and isolation. It intercepts and checks system calls from the normal world at runtime, preventing the target system from running unauthorized privileged code. TEEv^[13] designs a method to run multiple VMs in the secure world. Since the secure world does not have a dedicated privilege level for running Hypervisor, it runs the specially implemented Hypervisor at the same privilege level (S-EL1) as the VMs. To ensure the higher privilege of the Hypervisor to manage VMs, TEEv requires modifying the VM kernel to restrict its access to certain privileged instructions. Without hardware support, the previous work requires scanning binary files or software interception to check or modify VM instructions, causing implementation or running costs. ReZone^[14] utilizes the auxiliary hardware available on ARM devices to divide the secure world into isolated partitions, and the memory access control guaranteed by hardware limits the access permissions of the partitions to other partitions and the normal world. Such a design reduces the unnecessary privileges of S-EL1 and alleviates the system threats caused by S-EL1 hijacking. ARMv8.4 introduces the S-EL2 hardware isolation mechanism^[7], isolating multiple partitions in the secure world, and reducing the impact of vulnerability exploits in a single partition. TwinVisor^[15] utilizes the S-EL2 extension to implement isolated partitions in the secure world. Different

from the ARMv8.4 specification, TwinVisor runs a simple Hypervisor in the secure world to ensure VM security, while reusing the normal world Hypervisor to manage hardware resources. TwinVisor has designed solutions to support the collaborative management of VM resources by secure world Hypervisors and normal world Hypervisors and reduce system overhead, while it does not provide communication mechanisms between VMs like the ARM standard design. Table 1 compares the mechanisms of multiple secure partitions of TrustZone.

Table 1 Comparison of the mechanisms of multiple secure partitions of TrustZone

Mechanism of multiple secure partitions	Hardware support	Without system modification	Inter-partition communication	Performance overhead
TZ-RKP	×	✓	×	Intercepting and checking partition instructions at runtime, with high overhead
TEEv	×	×	✓	Pure software solution with high overhead
ReZone	✓	✓	×	When a secure partition runs, other cores enter an idle state, resulting in high overhead
TwinVisor	✓	✓	×	Requiring collaboration between Hypervisors in two worlds, with moderate cost
ARM S-EL2	✓	✓	✓	Hardware-supported virtualization, with low overhead

ARMv9 proposes the Confidential Compute Architecture (CCA)^[16-18], enhancing ARM's support for confidential computing to cope with compute-intensive tasks. Based on the Realm Management Extension (RME) and the Realm Management Monitor (RMM), CCA builds the concept of realm on the security foundation of TrustZone. A realm is a dynamically created secure partition managed by RMM. In addition, RME supports dynamic allocation and recycling of physical memory to the secure world. As ARM has not yet released detailed specifications for CCA, its commercial implementation will take some time.

1.2 Formal verification of partition isolation system

Remarkable achievements have been obtained in the formal verification of partition isolation systems. The seL4 project^[19] launches a comprehensive formal verification for a high-performance operating system. The seL4 microkernel is the first formally verified operating system kernel and remains the only fine-grained, capability-based, high-security, and high-performance operating system that has been verified. This work proposes a layer-by-layer refinement modeling method, which has been leveraged in subsequent extensive formal verification work. Our work is also inspired by this method. The CertiKOS project^[20] hosted by Shao's team at Yale University verifies an operating system running on multi-core processors. Traditional OS kernels are intricate, and a vulnerability in a single component will affect the security of the entire system, which is difficult to design and verify. Shao's team innovatively proposes compositional specifications at different abstraction layers, which tackle the challenges of verifying concurrent kernels while improving extensibility. Zhao *et al.*^[21, 22] implement a top-level specification in Isabelle/HOL that conforms to the ARINC 653 isolation kernel. Then they formally model the partition management, partition scheduling, and communication services for ARINC 653, and verify an industrial implementation and two open-source implementations of the ARINC 653 isolation kernel. They identify six security flaws that may lead to information leakage in the ARINC 653 standard and its implementation. The modeling and verification in this paper mainly refer to the framework of their work. SeKVM^[23, 24] conducts formal verification for the first time on the commercial Hypervisor KVM, by breaking it down into a core part and a set of untrusted services. It uses Coq to prove the confidentiality and integrity

of the KVM core through layer-by-layer refinement and verify its concurrent security using the rely-guarantee framework. This is a reference method for verifying the concurrent security of TrustZone multiple secure partitions in future work. Most studies focusing on the formal verification of partition kernels aim to prove data isolation and information flow security with theorem proving and refinement methods.

1.3 Formal verification of TEE

Few studies have been conducted in terms of secure isolation verification for TEE. Jin *et al.*^[25, 26] propose a refinement-based verification method for the security of the memory isolation mechanism for TrustZone. They model the key hardware and software of the memory isolation mechanism of TrustZone, including the address space controller, MMU, TLB, and secure monitor. The model is verified in the theorem prover Isabelle/HOL to satisfy the information flow security properties, i.e., noninterference, nonleakage, and noninfluence. Their work mainly focuses on verifying the isolation between the normal world and the secure world under the conventional TrustZone architecture. Our study verifies the isolation and secure communication between multiple partitions in the secure world as in the ARMv8.4 and later architectures. Miao *et al.*^[27] propose a general formal model framework for the fine-grained memory isolation and access control mechanism implemented in TEE using memory tag technology and provide a security analysis method based on model checking. Moreover, they design and implement the abstract machine model of the framework using the formal language B and verify the correctness and security of the access control mechanism in TEE. The memory tag isolation technology used by TIMBER-V (targeted at RISC-V) in the work is quite different from the ARM S-EL2 isolation mechanism verified in this paper, and the model checking method used is also different from the theorem proving method we use. To the best of our knowledge, RMTEE is the first formal modeling and verification work for multiple secure partitions in the secure world of TrustZone.

2 Threat Model and Assumptions

This paper focuses on the partitions in the secure world, without considering the isolation between the normal world and the secure world. As shown in Figure 1, the partitions in the secure world are not trusted, as there are numerous vulnerabilities in the TAs and OS kernel and they may be exploited by malicious users and become malicious programs. Partitions are responsible for their vulnerabilities. The formally verified partition manager ensures that there are no exploitable vulnerabilities in the interfaces provided to the partitions, and the secret data in one partition is not stolen or tampered with by other malicious partitions. This paper does not consider side-channel attacks or complex physical attacks.

In addition, this work is based on the following assumptions.

- (1) Hardware and the secure monitor are trustworthy.
- (2) The images of the partitions and partition manager in the persistent storage are not tampered with (their integrity is enforced by secure boot).

Attackers can steal or tamper with sensitive data from other partitions by reading or writing arbitrary memory addresses and initiating malicious inter-partition calls. The threat model is formally defined as follows.

Definition 1. A *threat model* is a pair $T = \langle S, E \rangle$, where

- S is the set of system states;
- E is the set of adversary event. $E = \{mem_read, mem_write, ipc_call\}$, where *mem_read* represents a memory read operation; *mem_write* represents a memory write operation; and *ipc_call* represents a call between partitions. The events contained in E

do not comply with system security constraints, that is,

$$\forall e \in E. (e \in \{mem_read(s, mem, p), mem_write(s, mem, p)\} \wedge mem.owner \neq p) \vee (e \in \{ipc_call(s, p1, p2)\} \wedge (p1, p2, ipc_call) \notin ipc_acm)$$

where s represents a system state; mem represents a memory block; p represents a partition; $mem_read(s, mem, p)$ indicates that partition p is reading a certain memory block mem under state s ; $mem.owner \neq p$ indicates that partition p is not the owner of the memory mem ; $ipc_call(s, p1, p2)$ indicates that partition $p1$ initiates an interface call to partition $p2$ under state s ; ipc_acm is the access control matrix for inter-partition communication; and $(p1, p2, ipc_call) \notin ipc_acm$ indicates that the access control configuration prevents partition $p1$ from invoking partition $p2$.

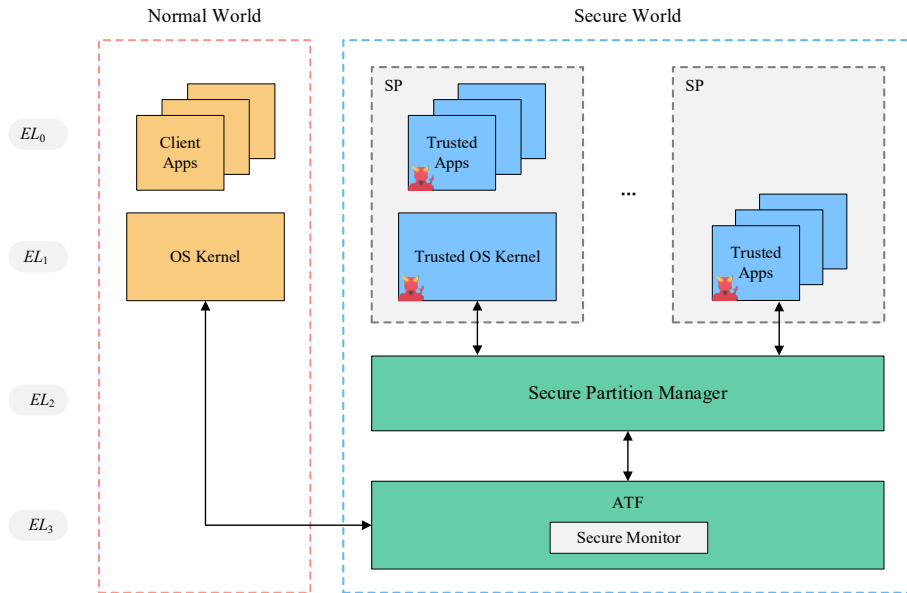


Figure 1 Threat model of RMTEE

3 Method Overview

Given the problem of state-space explosion in the model checking approach, it can hardly be applied to analyze the security of information flow in partitioned systems. Therefore, this work employs the theorem proving method, precisely, the Isabelle/HOL theorem prover. The simple and powerful functional programming language in Isabelle enables us to develop the formal specification. Mathematical calculation, mathematical theorem proving, and computer software and hardware systems can be described and implemented through this functional language. Isabelle's built-in logic system also provides strict reasoning and proof capabilities for formal verification. Isabelle/HOL is able to deal with large-scale formal verification of real-world commercial systems due to its high expressiveness, high degree of proof automation, and powerful library. In addition, most partitioned systems are successfully verified in Isabelle/HOL, such as seL4 and PikeOS, which is also a factor supporting the use of Isabelle/HOL in this work.

Figure 2 presents the modeling and verification framework of this paper. First, we define the security assets and security policies of the TrustZone secure partitions and the *interference*

relationship between partitions in an abstract way. The security properties of the information flow, i.e., *confidentiality* and *integrity*, are defined based on *interference*. A parameterized abstract state machine is used to describe the abstract model. The state parameters and functions defined in the state machine use abstract types, independent of the implementation of the system. Then, the abstract model is instantiated into the concrete model, which further composes the concrete specification. In this way, the parameters, constraints, and security properties defined in the abstract model can be preserved and reused in the concrete model. The concrete specification includes two parts: the execution model; the event specification. The execution model is an instance of the abstract model, and the event specification defines the partition lifecycle management interfaces, memory management interfaces, and inter-partition communication interfaces defined in the ARM FF-A specification. The concrete interfaces in the event specification are invoked by the execution model. Then, we prove the refinement relationship between the concrete model and the abstract model and the correctness of all the event interfaces in the concrete model, and that all the event interfaces satisfy the security properties defined in the abstract model. Finally, the attack of arbitrary address mapping and unauthorized inter-partition communication are modeled, verifying that the model can defend against attacks from partitions.

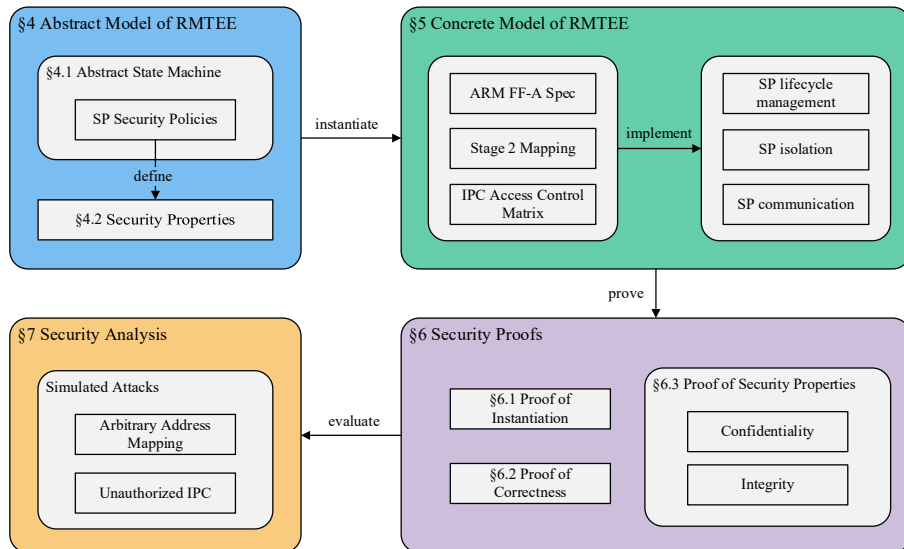


Figure 2 Refinement-based modeling and verification of multiple secure partitions of TrustZone

4 Abstract Model of RMTEE

This section introduces the abstract model of RMTEE which is composed of a state machine, security properties, and the inference relationship between security properties. In the state machine, state variables represent the system state, and the transition function and auxiliary functions are used to describe the changes of the state variables. The state transition function is an abstraction of the exposed interfaces of the system. Security properties are a formal description of the system security requirements. The abstract model is defined using Isabelle's *locale* keyword, in which the parameters are abstract and do not involve concrete data structures or function implementations to focus on the description of system properties. These abstract parameters are afterward refined into concrete implementations in the concrete model.

4.1 Abstract state machine

The state machine includes basic elements (variables, functions, operation rules, etc.), secure initial state, and security policies. The system state machine is a tuple composed of abstract data types, defined as follows.

Definition 2. A *system state machine* is a tuple $M = \langle \mathcal{S}, \mathcal{D}, \mathcal{E}, \varphi, \rightsquigarrow, \sim, \mathbb{D}, s_0, \text{SPM} \rangle$, where

- \mathcal{S} represents the set of the system states;
- \mathcal{D} represents the set of all secure domains. Each partition is a secure domain, and also the partition manager is a secure domain;
- \mathcal{E} represents the finite set of events. Events are abstractions of system interfaces. Since the system provides a certain number of interfaces, the set of events is finite;
- φ is the state transition function, $\varphi(e) \in \mathcal{S} \times \mathcal{S}$ indicates that the system reaches a new state after executing a specific event e under one state;
- \rightsquigarrow represents the interference relationship between security domains, $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$, $d1 \rightsquigarrow d2$ indicates that information is allowed to flow from partition $d1$ to partition $d2$. Noninterference is denoted as $\not\rightsquigarrow$;
- \sim represents state equivalence. $s \sim d \sim t$ means that two states s and t are equivalent for security domain d , that is, the private data of the security domain d is identical in states s and t . In some papers, equivalence is expressed as indistinguishable, meaning that a partition cannot distinguish between two states;
- \mathbb{D} represents the execution domain of an event, which is generated by an interface call initiated by the partition. An event e generated under state s is associated with an execution partition, denoted as $\mathbb{D}(s, e) \in \mathcal{D}$;
- $s_0 \in \mathcal{S}$ represents the initial state of the system; and
- SPM represents the partition manager, $\text{SPM} \in \mathcal{D}$. SPM has the highest permission, which is unique from normal partitions.

The interference relationship describes the information flow allowed by the system, representing the granting of data read permission to the latter partition and data write permission to the former partition. The interference relationship and state equivalence are the basis for describing the characteristics of system state changes. Based on their definitions and system security policies, the following six security policy specifications are defined:

- (1) $\forall s, d. (s \sim d \sim s)$
- (2) $\forall s, t, d. (s \sim d \sim t) \longrightarrow (t \sim d \sim s)$
- (3) $\forall s, t, r, d. (s \sim d \sim t) \wedge (t \sim d \sim r) \longrightarrow (s \sim d \sim r)$
- (4) $\forall d \in \mathcal{D}. (d \rightsquigarrow d)$
- (5) $\forall d \in \mathcal{D}. (\text{SPM} \rightsquigarrow d)$
- (6) $\forall d \in \mathcal{D}. (d \rightsquigarrow \text{SPM}) \longrightarrow d = \text{SPM}$

The policies (1)–(3) describe the reflexivity, symmetry, and transitivity of the the state equivalence, respectively, while the policies (4)–(6) describe the system security policy, which is the rule of information flow between partitions. That is, partitions can interfere with themselves, the partition manager can interfere with any partition, and normal partitions cannot interfere with the partition manager. As the partition manager has the highest permission, information from the partition manager is allowed to flow to any partition.

4.2 Security properties

The system security properties are defined based on the state machine model. Previous work has defined information flow security as noninterference and nonleakage. **Noninterference**^[28] specifies that if a partition cannot interfere with partition d , then events generated by the partition

do not affect the state of d . $\text{Nonleakage}^{[29]}$ means no unexpected information disclosure to other partitions during the execution of events. Ref. [29] has proven that the combination of noninterference and nonleakage is equivalent to noninfluence. Ref. [22] extends these security properties and proves the implication relationship between all these properties. Then, it is demonstrated that the noninfluence property can be inferred by a set of unwinding conditions. This paper follows the inference framework of Ref. [22] and defines *confidentiality* and *integrity* in the abstract model as the information flow security properties.

Definition 3. The *confidentiality* of an event e is defined as

$$\begin{aligned} \text{confidentiality}(e) &\equiv \forall d, s, t, s', t'. \\ &(s \sim d \sim t) \wedge \\ &(\mathbb{D}(s, e) = \mathbb{D}(t, e)) \wedge \\ &((\mathbb{D}(s, e) \rightsquigarrow d) \longrightarrow (s \sim \mathbb{D}(s, e) \sim t)) \wedge \\ &(s, s') \in \varphi(e) \wedge (t, t') \in \varphi(e) \longrightarrow \\ &(s' \sim d \sim t') \end{aligned}$$

Confidentiality ensures that there is no unexpected information leakage. A partition invokes the system interface to communicate with other partitions, generating information flow between partitions. The definition of confidentiality requires that under any two states s and t , the impacts on other partitions are the same after executing the same event. That is, the information passed to other partitions is the same, which ensures that the partition does not leak unexpected information. If an event execution produces different results according to data from other partitions, it will result in unequal result states s' and t' , thus leaking data from other partitions.

Definition 4. The *integrity* of an event e is defined as

$$\begin{aligned} \text{integrity}(e) &\equiv \forall d, s, s'. \\ &(\mathbb{D}(s, e) \not\rightsquigarrow d) \wedge \\ &(s, s') \in \varphi(e) \longrightarrow \\ &(s \sim d \sim s') \end{aligned}$$

Integrity prevents unauthorized data tampering. During the execution of any event by any partition, it is not allowed to modify the private data of other partitions defined in the system security policy that cannot be interfered with by that partition. The states before and after the event execution are equivalent to those partitions that cannot be interfered with by that partition.

In summary, confidentiality defines read access control for partitions, and integrity defines write access control for partitions. Based on the confidentiality and integrity of a single event, the confidentiality and integrity of the system are defined as $\text{confidentiality} \equiv \forall e. \text{confidentiality}(e)$ and $\text{integrity} \equiv \forall e. \text{integrity}(e)$.

5 Concrete Model of RMTEE

The RMTEE concrete model consists of an execution model and an event specification. The execution model is an instance of the abstract model, using *interpretation* to instantiate the abstract parameters and functions in the *local* into concrete implementations. The event specification concretizes events into a set of FF-A interface functions, which are used to model the functionalities and services of the target modules. *Locale* is similar to interface in an object-oriented language, providing support for the refinement from the abstract model to the concrete

model in this paper. *Interpretation* corresponds to the implementation of the interface, using specific parameters to interpret the content in the *local* into the current context. Meanwhile, we utilize Isabelle's automatic theorem prover to prove that the concrete model satisfies the specifications defined in the abstract model.

5.1 DAC

ARM S-EL2 defines standard interfaces for communication and invocation between partitions, excluding specific message content. Communication protocols are implementation defined and various services can be invoked with the same standard interface by offering different message payloads. However, this loose design carries potential risks. (1) If the implementation of a specific partition does not perform legitimacy checks on the input parameters in the message payload, errors may occur, such as division by zero, integer overflow, and code injection. (2) A malicious partition can send unlimited requests to a specific partition to fill its *vCPU*, thus achieving a denial of service attack. It is not practical to check message content in the partition manager since the partition manager is not aware of the structure and content of the message.

Therefore, we design DAC for inter-partition communication. Partitions specify the trusted partitions and the allowed interface invocations in their manifests. The DAC-based information flow security policies increase security assurance to the partition and mitigate the risks of vulnerability utilization and attacks. Logically, the access control for IPC can be expressed as an access control matrix as follows.

Definition 5. The *access control matrix for IPC* is a triplet $IPC_ACM = \langle S, O, A \rangle$, where

- $S = \mathcal{D}$, representing the set of subjects, i.e., partitions that initiate the IPC;
- $O = \mathcal{D}$, representing the set of objects, i.e., partitions that receive the IPC; and
- A is the access control matrix, $A[s][o] \in P(\mathcal{E})$, with $s \in S$ and $o \in O$, where P is the power set operation and $A[s][o]$ is a subset of the event set, representing allowed interface invocations from the subject partition s to the object partition o . Table 2 illustrates an example of the access control matrix, where \mathcal{E} represents the full set of events and \emptyset represents the empty set.

Table 2 Example of access control matrix for IPC

Partition	P1	P2	P3	P4
P1	\mathcal{E}	{FFA_MSG_SEND_ DIRECT_RESP}	{FFA_MSG_ SEND2}	{FFA_RUN}
P2	{FFA_MSG_SEND_ DIRECT_REQ}	\mathcal{E}	\emptyset	{FFA_MEM_ RELINQUISH}
P3	{FFA_MSG_SEND2}	\emptyset	\mathcal{E}	\emptyset
P4	\emptyset	{FFA_MEM_DONATE, FFA_MEM_LEND, FFA_MEM_SHARE}	\emptyset	\mathcal{E}

Table 2 specifies that partition P2 can initiate a direct messaging to P1, and P1 can respond to P2; indirect messaging is allowed between P1 and P3; P1 can run P4; P4 can transfer or share memory blocks to P2; and P2 can return the memory access permissions shared by P4.

5.2 Execution model

The execution model assigns concrete parameters and functions to the abstract model. The concrete parameters include domain (d), state (s), and event (e). The concrete functions include interference (\rightsquigarrow), state equivalence (\sim), and execution domain of events (\mathbb{D}). Their refined implementation in the concrete model is shown in Table 3.

Table 3 Refined implementation of abstract parameters in the state machine

State machine parameter	Abstract model	Concrete model
Domain	d	<i>record</i> Domain
State	s	<i>record</i> State
Event	e	<i>datatype</i> Event
State transition	φ	<i>definition fn_step</i>
Interference	\rightsquigarrow	<i>definition fn_interference</i>
State equivalence	\sim	<i>definition fn_equiv</i>
Execution domain of Event	\mathbb{D}	<i>definition fn_kdom</i>

Domain and *State* are defined by Isabelle's *record* with several fields, which is similar to the structure type. We further define the *Partition* and *SPM* types through the *record* extension syntax (similar to inheritance in object-oriented languages). They inherit the fields from *Domain* and contain their own unique extension fields. *State* describes the data of the global system, including the runtime state data for partitions and SPM. *Event* are defined via *datatype*, similar to a union. The data structures of the concrete parameters are shown in Table 4.

Table 4 Refined parameter data structures in the concrete model

Parameter	Field	Field type	Description
<i>Domain</i>	<i>id</i>	<i>Domain_ID</i>	Unique identifier of the partition
	<i>version</i>	$Major_Version \times Minor_Version$	FF-A version implemented by the partition, composed of tuples of major and minor version numbers
	<i>vcpu_ids</i>	<i>vCPU_ID</i> set	The set of <i>vcpu</i> ids of the partition
<i>Partition</i>	<i>boot_order</i>	<i>nat</i>	Partition boot order, the partition manager boots each partition in a priority order during initialization
	<i>rx_buffer</i>	<i>IPA</i>	The base virtual address and size of the RX buffer
	<i>tx_buffer</i>	<i>IPA</i>	The base virtual address and size of the TX buffer
	<i>reserved</i>	<i>nat</i>	Reserved field. No additional configurations for <i>SPM</i>
<i>State</i>	<i>currents</i>	$CPU_ID \rightarrow Domain_ID \times vCPU_ID$	Which <i>vCPU</i> is running on each physical CPU
	<i>partitions_state</i>	$Domain_ID \rightarrow Partition_State$	Runtime data for each partition
	<i>buffers</i>	$Domain_ID \rightarrow RXTXBuffer$	RX/TX Buffer data for each partition
	<i>registers</i>	$Domain_ID \times vCPU_ID \rightarrow Register$	Register data in each <i>vCPU</i> of each partition
	<i>memories</i>	<i>Mem_Block</i> set	The set of all the memory blocks
<i>Partition_State</i>	<i>vcpus</i>	<i>vCPU</i> set	The set of <i>vCPUs</i> of the partition
	<i>vmmu</i>	$IPA \rightarrow PA$	The page table of the partition
<i>Event</i>	<i>hyperc</i>	<i>Hypercall CPU_ID</i>	System call interface provided to partitions
	<i>intlc</i>	<i>Internal_Call CPU_ID</i>	Internal functions of the partition manager
	<i>spm</i>	<i>Domain_ID</i>	<i>ID</i> of the partition manager
<i>Sys_Conf</i>	<i>partitions_conf</i>	$Domain_ID \rightarrow Partition$	Manifests of each partition
	<i>cpus</i>	<i>CPU_ID</i> set	The set of physical <i>CPU_IDs</i>
	<i>memories</i>	<i>Mem_Block</i> set	The set of initial physical memory blocks, representing the granularity size (such as 4 KB)
	<i>ipc_acm</i>	$(Domain_ID \times Domain_ID \times Event)$ set	Access control matrix for IPC

Domain contains the unique identifier of the partition and records the FF-A version information implemented by the partition. *Partition* is a static manifest used for partition initialization, where *IPA* represents a virtual address space composed of base address and size. *State* holds the runtime data of all the partitions, including the RX/TX Buffer of the partition, memory blocks allocated to the partition, *vCPUs* of the partition, page table of the partition, and register data of each *vCPU* in the partition. The field “*memories*” represents the set of all the memory blocks in the system, and the field “*owner*” in the *Mem_Block* identifies which partition the memory block belongs to. The field “*currents*” in *State* identifies the *vCPU* running on each physical CPU in the current state. The fields in the *record* variable are retrieved in the form of (*field_name record_name*). The symbol \rightarrow in the table represents a partial function, which may not have corresponding values on some values of the domain of definition. For example, if the *cpu0* is idle in the state *s*, the value of (*currents s cpu0*) is *None*. The symbol \times represents a 2-tuple, e.g., if the value of the field “*currents*” exists, it is a tuple of partition id and *vCPU id*. Events are classified into system calls provided to partitions (*Hypercall*) and partition manager internal functions (*Internal_Call*). These two types of events are further enumerated as a series of concrete event interfaces, which are mapped to the standard interfaces defined by the FF-A specification in *fn_step*. The state transition function *fn_step* uses concrete state and event types to refine event execution. The pseudo code is defined as follows.

Definition 6. The *state transition fn_step* on event *e* is defined as

```

fn_step  $\equiv$  case e of
  Hypercall.FFA_MSG_SEND2  $\Rightarrow$  (s, ffa_msg_send2(s,  $\dots$ ))
  Hypercall.FFA_MSG_WAIT  $\Rightarrow$  (s, ffa_msg_wait(s,  $\dots$ ))
  ...
  Internal_Call.SP_INIT  $\Rightarrow$  (s, sp_init(s,  $\dots$ ))
  _  $\Rightarrow$  (s, s)

```

fn_step invokes different event interfaces based on event types and returns a tuple composed of start and end states. Functions such as *ffa_msg_send2* are concrete implementations of the event interfaces, constituting the event specification. They are described in detail in Section 5.3. The instantiated state equivalence function *fn_equiv* specifies that two states are equivalent for SPM if the memory block set in both states is the same, and they are equivalent for a partition if the partition’s RX/TX Buffer, memory blocks, *vCPU*, and registers are equal. *Event* includes a *CPU_ID* field, which identifies the CPU that initiates the event. In combination with the *currents* field in the state, *fn_kdom* finds out which partition is executing on the CPU, that is, the execution domain of the event. The pseudo code of the interference function *fn_interference* in the execution model is defined as follows.

Definition 7. The *interference function fn_interference* on two partitions *d1* and *d2* is defined as

```

fn_interference d1 d2  $\equiv$ 
  if (d1 = d2) then True
  else if is_spm(d1) then True
  else if is_spm(d2) then False
  else if is_valid_ipc(d1, d2) then True
  else False

```

The function **fn_interference** is defined by *definition*, and *d1* and *d2* are *Domain_ID*, representing the source partition and target partition, respectively. First, the function determines whether *d1* is equal to *d2*. If so, the function returns True, indicating that the partition can interfere with itself. *is_spm*(*d1*) determines whether *d1* is the partition manager. If so, the

function returns True, indicating that the partition manager can interfere with any partition. Then, it determines whether $d2$ is the partition manager (with the implicit premise that $d1$ is not a partition manager). If so, the function returns False, indicating that any partition cannot interfere with the partition manager. $is_valid_ipc(d1, d2)$ determines whether the pair $(d1, d2)$ is in the access control matrix, indicating that interference between partitions is determined by the access control matrix for IPC in the system configuration. The last branch states that partitions beyond the system definition cannot interfere with any partition.

The concrete model also defines the system configuration (sys_Conf), which is a *record* structure composed of a series of static variables. The parameters in the system configuration remain unchanged during system runs and are used to initialize the system and verify the legitimacy of event execution. The data structure of the system configuration is shown in Table 4. The field spm specifies the ID of the partition manager; $partitions_conf$ is a mapping from partition id to *Partition*, recording the manifests of all the partitions; $cpus$ is a set of physical processor ids; $memories$ is a collection of initial physical memory blocks; ipc_acm is a triple $(partition, partition, event)$, which is equivalent to the access control matrix for IPC described in Section 5.1. Since the system configuration determines the start state and runtime behavior of the system, the model specifies 5 constraints on the system configuration.

- (1) Any normal partition is not the partition manager.
 $\forall p. (sys_conf.partitions_conf[p] \neq None) \longrightarrow (p \neq sys_conf.spm)$
- (2) The partition manager is not a normal partition.
 $\forall p. (sys_conf.spm = p) \longrightarrow (sys_conf.partitions_conf[p] = None)$
- (3) The partition id in the partition configuration is correctly configured.
 $\forall p. (sys_conf.partitions_conf[p] \neq None) \longrightarrow ((sys_conf.partitions_conf[p]).id = p)$
- (4) The addresses of any two physical memory blocks do not overlap with each other.
 $\forall x, y. (x \neq y \wedge x \in sys_conf.memories \wedge y \in sys_conf.memories) \longrightarrow$
 $\neg (is_memory_conflicted(x, y))$
- (5) The partitions in the access control matrix for IPC are all valid.
 $\forall (d1, d2, e) \in sys_conf.ipc_acm. (sys_conf.partitions_conf[d1] \neq None) \wedge$
 $(sys_conf.partitions_conf[d2] \neq None)$

Only system configurations that comply with these constraints are legal, thus ensuring correct system behavior. They are defined with Isabelle's *specification*, which is used to define a series of formulas on specific constants to describe their constraints. We prove the existence of such constants that satisfy the constraints.

5.3 Event specification

As described in Section 5.2, the event specification defines two types of events, system calls and internal functions, which describe the system behaviors and interfaces defined by FF-A. They are invoked as the actual execution entity by the state transition function fn_step . System calls are the system interfaces provided by the partition manager to partitions, used to achieve inter-partition communication. Internal functions implement the internal functionalities of the partition manager and are not exposed to partitions. The event specification defines a total of 36 functions for three modules: partition lifecycle management, partition isolation, and inter-partition communication, as shown in Table 5.

Partition lifecycle management. Partition lifecycle management maintains the operation of partitions, including partition initialization, partition switching, and partition status reporting. Partition images and manifests are stored in persistent storage, and they are signed with certificates fixed in the chip and guaranteed to be intact through secure boot. The initialization function of the partition manager loads partition images and manifests from persistent storage.

Table 5 RMTEE event functions

Interface function	Description
<i>ffa_error</i>	State report interface, return error code on the failure of FF-A interface call
<i>ffa_success</i>	State report interface, return the result of successful FF-A interface call
<i>ffa_interrupt</i>	State report interface, switch to partition for interrupt handling
<i>ffa_version</i>	Query framework version number
<i>ffa_features</i>	Query whether the specified function is implemented
<i>ffa_rxtx_map</i>	Establish the mapping for the RX/TX Buffer of partitions
<i>ffa_partition_info_get</i>	Query information for a specified partition
<i>ffa_id_get</i>	Query partition ID
<i>ffa_spm_id_get</i>	Query partition manager ID
<i>ffa_msg_wait</i>	The partition ends up its execution and enters the READY status
<i>ffa_yield</i>	The partition blocks its execution and enters the BLOCKED status
<i>ffa_run</i>	The partition blocks its execution and wakes up the specified vCPU of the target partition for execution, causing itself to enter the BLOCKED status
<i>ffa_msg_send2</i>	Asynchronous messaging interface, transfer messages from the <i>sender</i> 's TX Buffer to the <i>receiver</i> 's RX Buffer
<i>ffa_msg_send_direct_req</i>	Synchronous messaging interface, where the sender wakes up a vCPU of the receiver to process the message and enters the BLOCKED status
<i>ffa_msg_send_direct_resp</i>	Send a response message for <i>ffa_msg_send_direct_req</i> , and the <i>sender</i> wakes up the target <i>endpoint</i> and enters the READY state
<i>ffa_mem_donate</i>	The partition transfers ownership and access permissions of a memory block to a specified partition
<i>ffa_mem_lend</i>	The partition transfers exclusive access permissions of a memory block to a specified partition
<i>ffa_mem_share</i>	The partition shares the shared access permissions of a memory block with a specified partition
<i>ffa_mem_retrieve_req</i>	The partition requests to complete one of the three transactions: <i>donate</i> , <i>lend</i> , or <i>share</i>
<i>ffa_mem_retrieve_resp</i>	Respond to <i>ffa_mem_retrieve_req</i> request
<i>ffa_mem_relinquish</i>	The partition returns memory block access permissions for <i>lend/share</i>
<i>ffa_mem_reclaim</i>	The partition reclaims exclusive access permissions of a memory block
<i>ffa_mem_frag_rx</i>	The partition requests the callee to transmit the next fragment of the memory transaction descriptor
<i>ffa_mem_frag_tx</i>	Complete and respond to <i>ffa_mem_frag_rx</i> request
<i>mm_map</i>	Establish page table mapping for a specified memory region
<i>mm_unmap</i>	Unmap the page table for a specified memory region
<i>mem_alloc</i>	Memory request function
<i>mem_free</i>	Memory release function
<i>sp_get_count</i>	Query the number of partitions in the system
<i>sp_find</i>	Find the partition by ID
<i>sp_get_vcpu</i>	Obtain the vCPU of the secure partition by vCPU ID
<i>vCPU_on</i>	Update the vCPU status from the initial status OFF to READY
<i>vCPU_run</i>	Update the vCPU status to RUNNING
<i>vcpu_switch</i>	vCPU switch interface, switch to the specified vCPU execution, and enter the BLOCKED status
<i>get_s2_table_address</i>	Obtain the address of the partition's stage 2 page table
<i>sp_init</i>	Partition initialization interface, read partition manifests, allocate memories and RX/TX Buffers for partitions, establish page table mapping, and initialize vCPUs set

Then, it allocates the required resources for the partition, completes partition initialization according to the manifest description, and sets the partition to be in the waiting status. The secure partitions in TrustZone wait for requests rather than actively running. When the partition manager receives an interface call for a specific partition, it switches to the target partition for execution. A partition can hold multiple execution instances, and their runtime context is organized as vCPU data maintained by the partition manager. The states of the vCPU in the

partition can be any of the following: off, on, running, waiting, blocked, or preempted. The transitions between them^[8] are illustrated in Figure 3.

When the partition manager performs the partition *vCPU* switch, it will save the current register information in the current running partition's *vCPU* structure, load the information of the target partition's *vCPU*, and switch to the target partition *vCPU* for execution. The following is the pseudo code of the blocking switch of *vCPU*.

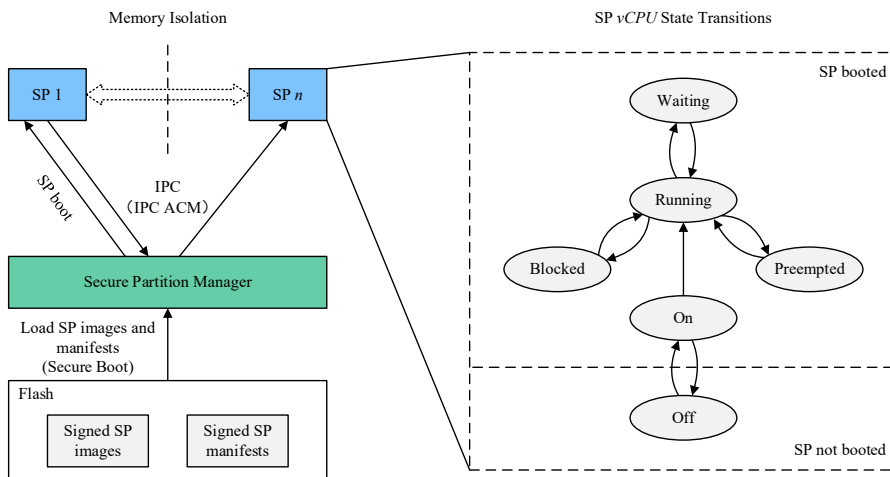


Figure 3 Partition lifecycle management

Definition 8. The *vCPU* switch function `vcpu_switch` is defined as

```

vcpu_switch s cpu_id dst_id dst_vcpu_id ≡
  (src_id, src_vcpu_id) = get_current(s, cpu_id)
  src_vcpu = get_vcpu(s, src_id, src_vcpu_id)
  dst_vcpu = get_vcpu(s, dst_id, dst_vcpu_id)
  s' = update_vcpu_state(s, src_id, src_vcpu, state = BLOCKED)
  s' = update_vcpu_state(s', dst_id, dst_vcpu, state = RUNNING)
  s' = update_current(s', cpu_id, (dst_id, dst_vcpu_id))
  return s'

```

The function `vcpu_switch` is defined by *definition*, and its input parameters include the current states, physical processor id, target partition id, and target *vCPU* id. `get_current` retrieves the partition and *vCPU* id running on the specified physical CPU in the current state; `get_vcpu` obtains the *vCPU* object for the specified partition and *vCPU* id; `update_current` updates the *vCPU* running on the physical CPU; `update_vcpu_state` updates the state of the specified *vCPU*. The function sets the running *vCPU* of the source partition to the blocked state and the target partition *vCPU* to the running state, updates the running *vCPU* of the current physical CPU to the target *vCPU*, and finally returns the updated state.

Partition isolation. Partition isolation is implemented through the partition manager's memory management of partitions, which specifically includes stage 2 page table mapping, memory pool management, and memory sharing. The page table management of conventional operating systems is responsible for converting virtual addresses into physical addresses. The page table records the mapping relationship between virtual addresses and physical addresses. With the expansion of the address space, the first-level page table may expand to a third- or even fourth-level page table. The core functions of the Memory Management Unit (MMU) do

not change. MMU encapsulates the implementation of memory for applications, allowing them to safely apply for and use memory without worrying about memory overruns and overwriting. After the introduction of the partition manager, an Intermediate Physical Address (IPA) is added between the virtual address and the physical address. As shown in Figure 4, the operating system kernel within the partition is responsible for translating virtual addresses to intermediate physical addresses, a process known as stage 1 page table mapping. The partition manager translates intermediate physical addresses into physical addresses, a process known as stage 2 page table mapping^[7]. Through stage 2 page table mapping, the partition manager ensures that each partition can only access its own memory. The operating system in a partition does not have permission to modify the stage 2 page table to map to memory addresses belonging to other partitions. In the view of the partition manager, the IPA of each partition is a virtual address. Therefore, there is no difference between stage 2 address mapping and regular address mapping.

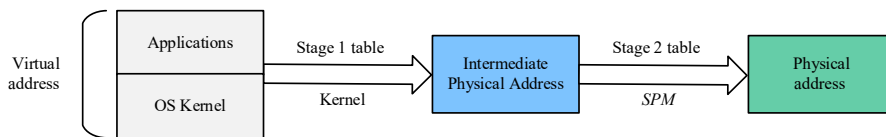


Figure 4 Stage 2 translation

The pseudo code of the page table mapping function is defined as follows.

Definition 9. The page table mapping function **mm_map** is defined as

```

mm_map  $s$  CPU_ID domain_id ipa pa  $\equiv$ 
  if ( $get\_pa\_owner(pa) \neq domain\_id$ ) then
    return  $s$ 
  else
     $vmmu = get\_page\_table(s, domain\_id)$ 
     $vmmu[ipa] = pa$ 
     $s' = update\_page\_table(s, domain\_id, vmmu)$ 
    return  $s'$ 
  
```

The function **mm_map** is defined by *definition*, and its input parameters include the current states, physical processor id, partition id, virtual address *ipa* to be mapped, and physical address *pa*. *mm_map* first obtains the owner of the memory block via *get_pa_owner*, verifies that the physical address to be mapped belongs to the current partition, and then calls *get_page_table* to obtain the stage 2 page table (*vmmu*) of the partition), which is a mapping from IPA to PA. Then, the *ipa*→*pa* mapping is added to *vmmu*, and the partition page table is updated. It finally returns the updated state. Similarly, the unmapping function **mm_unmap** completes the opposite operation.

Another functionality of memory management is memory allocation and recycling. FF-A does not specify how the partition manager implements memory pool management, which is implementation defined. The concrete model of RMTEE uses a bitmap-based dynamic memory allocation algorithm. In the initial state, the partition manager maintains a free memory pool and records the allocation status of each memory block. On partition initialization, memory blocks of appropriate size are allocated for the partition according to the manifest. During the running of partitions, memory can be dynamically applied for and released through the interface provided by the partition manager. The memory allocation function is defined as follows.

Definition 10. The memory allocation function **mem_alloc** is defined as

```

mem_alloc  $s$  sys_conf cpu_id mem_size  $\equiv$ 
   $unalloc\_mem = get\_unallocated\_mem\_bysize(s, mem\_size)$ 
  
```

```

if ( $unalloc\_mem = None$ ) then
  return  $s$ 
else
  ( $caller\_id, \_$ ) =  $get\_current(s, cpu\_id)$ 
   $alloc\_mem = update\_mem(unalloc\_mem, owner = caller\_id,$ 
     $access = \{caller\_id\}, allocated = True)$ 
   $memories = get\_memories(s)$ 
   $memories = insert(memories - \{unalloc\_mem\}, alloc\_mem)$ 
   $s' = update\_memories(s, memories)$ 
if ( $\exists x, y. x \neq y \wedge x \in get\_memories(s') \wedge y \in get\_memories(s') \wedge$ 
   $is\_memory\_conflicted(x, y)$ )
then
  return  $s$ 
else
  return  $s'$ 

```

The function **mem_alloc** is defined by *definition*, and its input parameters include the current state s , system configuration, physical processor id, and requested memory size. The function first confirms the existence of a free memory block $unalloc_mem$ in the system that meets the requested size by $get_unallocated_mem_bysize$, and then modifies its owner to the current partition to make sure the current partition has access permissions ($access = \{caller_id\}$), and updates the memory status to allocated through $update_mem$, obtaining a new memory block object $alloc_mem$. The function $get_memories$ obtains the set of system memory blocks, removes the old memory block $unalloc_mem$ from memories, inserts the new memory block $alloc_mem$, and returns the new system state s' . If the allocation results in the overlapping of any two physical memory block addresses, the allocation is revoked and the system rolls back to the previous state s . Otherwise, the state s' is returned. FF-A defines the interfaces for memory sharing between partitions. A partition can share or temporarily transfer its memory access permissions to another partition, or transfer the memory ownership to another partition. For space limitations, relevant code is not listed here.

Inter-partition communication. The partition manager provides synchronous and asynchronous message communication interfaces. In synchronous messaging, when the sender initiates a message request, it relinquishes its execution control and wakes up the receiver to process the message. After the receiver completes the processing, it returns back the execution control to the sender. Figure 5 shows the sequence of synchronous messaging^[11]. In asynchronous messaging, after the sender initiates a message request, it continues to run. The external scheduler wakes up the receiver for execution to complete message processing. After the receiver completes the request, it returns the response to the sender in the same way, and the sender can accept the response through polling. In both communication modes, the partition manager serves as an intermediary to forward messages. In the concrete model of RMTEE, the partition manager verifies whether both communication parties are valid and in the access control matrix for IPC.

The pseudo code of the message transfer function is defined as follows.

Definition 11. The *message transfer function* **transfer_msg** is defined as

```

transfer_msg  $s\ msg \equiv$ 
   $src\_id = msg.sender\_id$ 
   $dst\_id = msg.receiver\_id$ 
   $s' = update\_buffer(s, src\_id, content = None, owner = src\_id)$ 

```

```

s' = update_buffer(s', dst_id, content = msg, owner = dst_id)
return s'

```

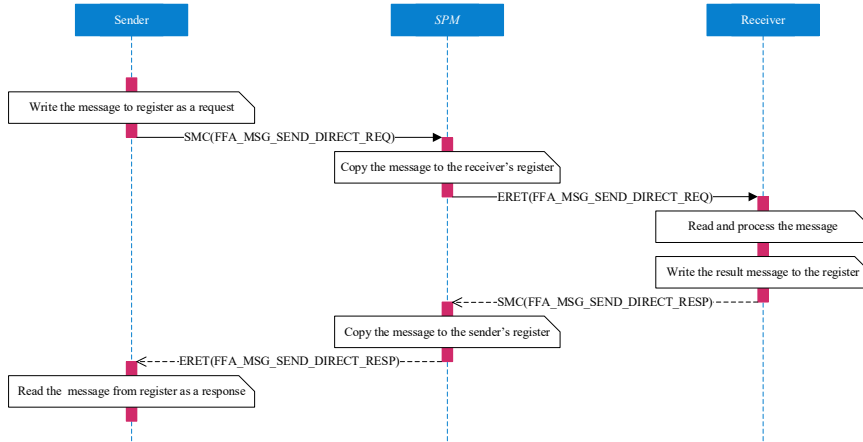


Figure 5 Synchronous messaging

The function `transfer_msg` is defined by *definition*, and its input parameters include the current state s and the message msg to be transmitted. The function `update_buffer` updates the content and owner of the specified partition buffer. The function `transfer_msg` obtains the sender partition src_id and receiver partition dst_id from the message header, copies the message to the receiver's RX Buffer, clears the sender's TX Buffer (set to None), and finally updates the partition buffer data maintained in the system state to return the updated state. The messaging interface (such as `ffa_msg_send2`) will invoke `transfer_msg` to complete the message transfer after validating input parameters and system state.

6 Security Proofs

This section introduces the verification of the RMTEE concrete model. Specifically, instantiated proving of the execution model is carried out to ensure that the concrete model is the correct refinement of the abstract model; the correctness of the event specification is proven to make sure that the concrete events satisfy the functional requirement of the system; all the events are proven to satisfy the confidentiality and integrity properties defined in the abstract model.

6.1 Proofs of refinement

The verification of refinement is to prove that the behaviors of the lower layer are consistent with the upper layer. For the concrete and abstract models in this paper, it is to prove that after the parameters instantiation, the concrete model still satisfies the security policy specifications defined by the abstract model. A series of lemmas are implemented in the RMTEE concrete model to prove that the execution model satisfies the six security policy specifications (reflexivity, symmetry, and transitivity of state equivalence, as well as the constraints for information flow between partitions) described in Section 4.1. For example, normal partitions cannot interfere with the partition manager, which is described in the execution model as follows.

Lemma 1. The fact that partitions cannot interfere with the partition manager is formalized by

$$\mathbf{fn_sp_non_inf_spm} \equiv \forall d. \mathbf{fn_interference}(d, \mathbf{spm}) \longrightarrow (d = \mathbf{spm})$$

Referring to the interference definition $fn_interference$, the above lemma can be proven automatically by the Isabelle theorem prover. Similarly, the instantiated lemmas and their proofs of the other five specifications can be completed, indicating that the execution model of the concrete model is a correct refinement of the abstract model.

6.2 Proofs of correctness

The verification of the correctness of event specification is decomposed into the proving of the correctness of each concrete event in the event specification. We use Hoare logic^[30] to describe the correctness of an event. Hoare logic can be expressed as: $\{P\}C\{Q\}$, where C , P , and Q represent an event, its pre-condition, and its post-condition, respectively. After executing C under the condition P , either C does not terminate or Q is true. As the events are defined with Isabelle's *definition* keyword, the termination of events is automatically ensured. Then the proving of event correctness is transformed into proving the satisfaction of post-condition Q . Taking the $vCPU$ switch function as an example, the correctness is defined as follows.

Lemma 2. The fact that partitions cannot interfere with the partition manager is formalized by

$$\begin{aligned} & \text{vcpu_switch_correctness} \equiv \\ & \{ \text{get_vcpu}(s, \text{dst_id}, \text{dst_vcpu_id}) \neq \text{None} \wedge \\ & \text{get_vcpu_state}(s, \text{dst_id}, \text{dst_vcpu_id}) \neq \text{OFF} \wedge \\ & \text{get_vcpu_state}(s, \text{dst_id}, \text{dst_vcpu_id}) \neq \text{RUNNING} \} \\ & r = \text{vcpu_switch}(s, \text{cpu_id}, \text{dst_id}, \text{dst_vcpu_id}) \\ & \{ \text{get_vcpu_state}(r, \text{pid}, \text{dst_vcpu_id}) = \text{RUNNING} \wedge \\ & \text{get_current}(r, \text{cpu_id}) = (\text{dst_id}, \text{dst_vcpu_id}) \wedge \\ & \text{vcpus} - \{\text{vcpu}\} = r_vcpus - \{r_vcpu\} \} \end{aligned}$$

where get_vcpu obtains the specified partition $vcpu$ and get_vcpu_state obtains the running state of $vcpu$. $vcpu$ and $vcpus$ are the specified $vcpu$ and the $vCPU$ s set of the specified partition, respectively, before the event execution; r_vcpu and r_vcpus are the running $vcpu$ and $vCPU$ s set of the partition, respectively, after the event execution. The pre-condition guarantees that the specified $vcpu$ exists, and that it is on and not running. The post-condition indicates that after the function is correctly executed, the specified $vcpu$ is in running status, and other $vcpu$ in the $vCPU$ s set of the partition are not modified (as shown in Figure 6(a)).

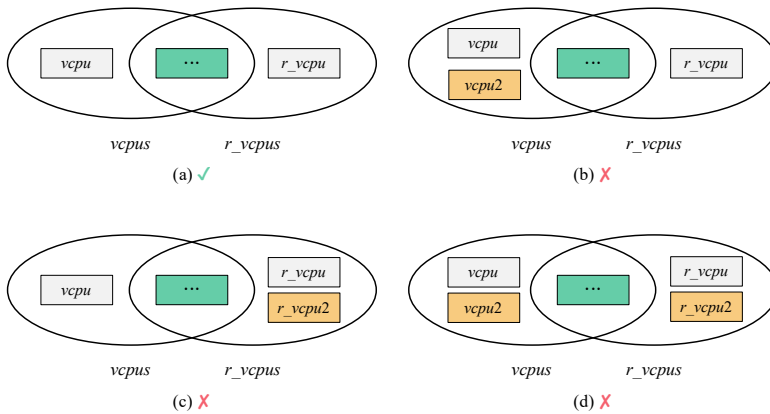


Figure 6 Intersection relationship of the $vCPU$ set before and after modification

If the event modifies other elements in the $vCPU$ set, such as deleting a $vcpu$ (Figure 6(b)),

adding a *vcpu* (Figure 6(c)), or modifying the state of other *vcpus* (Figure 6(d)), then there must be other modified *vcpu* in $vcpus-\{vcpu\}$ or $r_vcpus-\{r_vcpu\}$, which is not equal to the intersection of the two sets; as a result, the assertion of $vcpus-\{vcpu\} = r_vcpus-\{r_vcpu\}$ in the post-condition is not satisfied.

6.3 Proofs of security properties

The concrete model inherits the confidentiality and integrity defined in the abstract model and proves that each concrete event satisfies these two security properties. To simplify the proving of concrete event security, we define the weak confidentiality of event.

Definition 12. The *weak confidentiality* of an event e is defined as

weak_confidentiality(e) $\equiv \forall d, s, t, s', t'.$

$$\begin{aligned} & (s \sim d \sim t) \wedge \\ & (\mathbb{D}(s, e) = \mathbb{D}(t, e)) \wedge \\ & (\mathbb{D}(s, e) \rightsquigarrow d) \wedge \\ & (s \sim \mathbb{D}(s, e) \sim t) \wedge \\ & (s, s') \in \varphi(e) \wedge (t, t') \in \varphi(e) \longrightarrow \\ & (s' \sim d \sim t') \end{aligned}$$

Weak confidentiality assumes that the premise $\mathbb{D}(s, e) \rightsquigarrow d$ in the definition of confidentiality holds, thus obtaining that $s \sim (s, e) \sim t$ also holds. For the case where $\mathbb{D}(s, e) \rightsquigarrow d$ does not hold, the conclusion can be derived through integrity definition. That is, the security properties follow the inference relationship: $\llbracket \text{weak_confidentiality}; \text{integrity} \rrbracket \implies \text{confidentiality}$. Therefore, in the concrete model, we only need to prove the weak confidentiality and integrity of the events, which reduces many repetitive efforts compared with proving confidentiality and integrity.

The security of each concrete event can be defined by replacing the event in the security properties of the abstract model with the event. Taking asynchronous messaging interface *ffa_msg_send2* as an example, its weak confidentiality is defined as follows.

Lemma 3. The weak confidentiality of *ffa_msg_send2* is formalized by

ffa_msg_send2_weak_confidentiality $\equiv \forall d, s, t.$

$$\begin{aligned} & e = \text{Hypercall.FFA_MSG_SEND2} \wedge \\ & (s \sim d \sim t) \wedge \\ & (\mathbb{D}(s, e) = \mathbb{D}(t, e)) \wedge \\ & (\mathbb{D}(s, e) \rightsquigarrow d) \wedge \\ & (s \sim \mathbb{D}(s, e) \sim t) \wedge \\ & s' = \text{ffa_msg_send2}(s, \text{cpu_id}) \wedge \\ & t' = \text{ffa_msg_send2}(t, \text{cpu_id}) \longrightarrow \\ & (s' \sim d \sim t') \end{aligned}$$

The integrity definition (*ffa_msg_send2_integrity*) similarly replaces the events in the integrity definition with *ffa_msg_send2*. In the proofs of the security properties of a concrete interface, the resulting state is expanded to explicitly prompt the prover for the modified state variable of the interface, which, combined with Isabelle's automatic derivation ability, can prove the equivalence of the resulting state. After proving the confidentiality and integrity of all the events, we apply induction rules to prove the security properties of the entire event specification, thereby proving that the concrete model satisfies the requirements of information flow security.

7 Analysis and Evaluation

This section comprehensively evaluates the RMTEE model. Firstly, we analyze the security of the formal model. Secondly, the workload of the formal specification and verification is estimated. Finally, we discuss the scalability of the model.

7.1 Security analysis

This paper assumes that malicious codes or exploitable vulnerabilities exist in the operating system or applications in the partitions, resulting in malicious partitions, which will attempt to steal or tamper with runtime data in other partitions or the partition manager. The RMTEE model verified in this work can resist this malicious behavior. To prove the security of the RMTEE model, this paper models two attack scenarios: arbitrary address mapping and unauthorized inter-partition communication. The memory addresses accessed by each partition need to be translated from IPA to PA through the stage 2 page table of the partition manager. Suppose the partition wants to maliciously access the memory belonging to other partitions, it must invoke the memory mapping function of the partition manager to create a page table entry mapped to other partition PA in its own stage 2 page table. The formal definition of arbitrary address mapping is as follows.

Definition 13. The *arbitrary address mapping* is defined as

arbitrary_addr_map $\equiv \exists s, cpu_id, p1, ipa, mem.$

$mem.owner \neq p1 \wedge$

$mm_map(s, cpu_id, p1, ipa, mem) \neq s$

It indicates that the malicious partition $p1$ maps a physical memory address that does not belong to itself in its own stage 2 table and the mapping function mm_map returns a status other than s , indicating that the invocation is successful and the system state has been modified. Afterward, partition $p1$ can use ipa to access this physical address. The unauthorized inter-partition communication is defined as follows.

Definition 14. The *unauthorized inter-partition communication* is defined as

unauthorized_ipc $\equiv \exists s, cpu_id, p1, p2, msg.$

$(p1, p2, FFA_MSG_SEND2) \notin (sys_conf.ipc_acm) \wedge$

$msg = get_buffer(s, p1).content \wedge$

$msg.receiver_id = p2 \wedge$

$ffa_msg_send2(s, cpu_id) \neq s$

The access control matrix for IPC configured by the system does not allow $p1$ to initiate asynchronous messaging to partition $p2$, while malicious partition $p1$ communicates with partition $p2$ by calling the asynchronous messaging interface successfully. By employing the condition-conjunction equivalence $p \wedge q \equiv \neg(p \rightarrow \neg q)$, the propositions in the above attack model can be converted into implication. For example, *arbitrary_addr_map* is converted to:

$\exists s, cpu_id, p1, ipa mem. \neg ((mem.owner \neq p1) \rightarrow (mm_map(s, cpu_id, p1, ipa, mem) = s))$

By falsifying it in the model, i.e., its negation as true, it can be proven that the model can defend against arbitrary address mapping attack. Its negation is:

$\forall s, cpu_id, p1, ipa mem. ((mem.owner \neq p1) \rightarrow (mm_map(s, cpu_id, p1, ipa, mem) = s))$

The lemma for correctness proof of mm_map is referred to, and the Isabelle theorem prover can automatically deduce that the above proposition is true. Performing a similar verification process on **unauthorized_ipc** can prove that the negation of **unauthorized_ipc** is true, which is defined as:

$$\begin{aligned} &\forall s, \text{cpu_id}, p1, p2, \text{msg}. \\ & (p1, p2, \text{FFA_MSG_SEND2}) \notin (\text{sys_conf.ipc_acm}) \wedge \\ & \text{msg} = \text{get_tx_buffer}(s, p1).\text{content} \wedge \\ & \text{msg.receiver_id} = p2 \longrightarrow \\ & \text{ffa_msg_send2}(s, \text{cpu_id}) = s \end{aligned}$$

It means that for any messaging call that is not in the access control matrix, the system state remains unchanged. Therefore, the inter-partition communication interfaces based on the access control matrix for IPC can defend against unauthorized partition invocations.

7.2 Code size and efforts

We use Isabelle/HOL to implement the specification and verification of multiple secure partitions of TrustZone. The information flow security of the model is proven with Isabelle's structured proof language Isar, allowing for the proof code to be understood by both humans and computers. The proof derivation is completed by the Isabelle theorem prover. As shown in Table 6, the formal model implemented in this paper uses 137 definitions and 1,462 lines of code in Isabelle to establish the RMTEE model. The correctness and security of the model are proven with 201 theorems and 9,715 lines of code in Isar. The entire work takes approximately 18 person months.

Table 6 Specification and verification statistics

RMTEE	Specification		Proof		Total (LOC)
	locale/definition	LOC	lemma/theorem	LOC	
Abstract model	14	108	12	117	
Concrete model	123	1,354	189	9,598	11,177
Total	137	1,462	201	9,715	

Specifically, the RMTEE abstract model is defined with Isabelle's parameter theory *locale*, and then the parameter model is instantiated by *interpretation* to build the RMTEE concrete model and the correctness of the instantiation is proven at the same time. In addition, Hoare logic is adopted to prove the correctness of events in the concrete model, and the semi-automatic theorem prover is used to verify their confidentiality and integrity. Since all the code in Isabelle is machine-checkable, the correctness of the model specification and proofs is ensured.

7.3 Discussion

We establish a refinement-based TrustZone multiple secure partitions model. With the modeling and verification framework proposed in this paper, the concrete model can be further refined layer-by-layer, and the detailed design and implementation of the partition manager can be verified. For different partition manager designs and implementations, there is no need to modify the upper model, but only to update the state variables and event specifications in the lower model, and verify the correctness and security of the newly introduced state variables and events. Therefore, the RMTEE model can be extended to different partition manager implementations. For other architectures, the refinement framework in this paper is still applicable, while the concrete model may need to be modified according to the architecture design. Taking the ARMv9 confidential computing architecture as an example, it provides an execution environment for running a confidential virtual machine. As for the concrete model, we need to expand the partitions of the execution model into secure world partitions and confidential VMs and define the interference relationship between them according to CCA specifications. Meanwhile, it is necessary to modify the event specification and relevant correctness and security proofs based on the interface provided by the confidential computing architecture.

8 Conclusion and Future Work

This paper implemented the formal modeling and verification of ARM TrustZone's multiple secure partitions architecture for the first time. We proposed the formal model RMTEE based on the refinement method, including an abstract model and a concrete model. Concrete event specification is provided for the interfaces of three modules: partition lifecycle management, partition isolation, and inter-partition communication. Furthermore, the refinement between the concrete model and the abstract model is proved, as well as the correctness and security of all the events in the event specification. The model is proven to satisfy confidentiality and integrity. Finally, we analyzed the security of the RMTEE model through two simulated attacks of the threat model. For arbitrary address mapping, this illegal operation can be restricted through the address owner check of the stage 2 page table mapping. For unauthorized inter-partition communication, the access control matrix for IPC can check and prevent unauthorized calls.

The concrete model of this paper includes partial modules in the FF-A specification. In future work, we will cover other modules in the FF-A specification, including notification and interrupt management modules. In addition, we plan to model and verify the detailed design and implementation of real-world partition managers. It is promising to extend the RMTEE model to the other multiple secure partitions mechanisms, such as ARMv9 CCA.

References

- [1] Ngabonziza B, Martin D, Bailey A, et al. TrustZone explained: Architectural features and use cases. Proc. of the 2nd Int'l Conf. on Collaboration and Internet Computing. Pittsburgh: IEEE, 2016. 445–451. [doi: 10.1109/CIC.2016.065]
- [2] GlobalPlatform. Introduction to trusted execution environments. 2018. <https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf>
- [3] Pinto S, Santos N. Demystify ARM trustzone: A comprehensive survey. ACM Computing Surveys (CSUR), 2019, 51(6): 1–36. [doi: 10.1145/3291047]
- [4] Cerdeira D, Santos N, Fonseca P, Pinto S. Sok: Understanding the privacy security vulnerability in trustzone assisted tee systems. Proc. of the 41st IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1416–1432. [doi: 10.1109/SP40000.2020.00061]
- [5] VulDB NVIDIA. Jetson OTE Protocol Message Parser memory leak. 2021. <https://vuldb.com/?id.177340>
- [6] F-Secure Security advisory: OP-TEE TrustZone bypass at wakeup on NXP i.MX6UL. 2021. https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_FSC-HWSEC-VR2021-0002-OP-TEE_TrustZone_bypass_at_wakeup.txt
- [7] ARM. Learn the architecture—AArch64 virtualization. <https://developer.arm.com/documentation/102142/0100>
- [8] ARM. Arm firmware framework for ARM a-profile. <https://developer.arm.com/documentation/den0077>.
- [9] VulDB. Qualcomm snapdragon auto TrustZone memory transfer interface information disclosure. 2021. <https://vuldb.com/?id.189552>.
- [10] Wirth N. Program development by stepwise refinement. Communications of the ACM, 1983, 26(1): 70–74. [doi: 10.1145/357980.358010]
- [11] Back RJR, Sere K. Stepwise refinement of action systems. Proc. of the Int'l Conf. on Mathematics of Program Construction. Berlin, Heidelberg: Springer, 1989. 115–138. [doi: 10.1007/3-540-51305-1_7]
- [12] Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G. Hypervision across worlds: Real time kernel protection from the ARM trustzone secure world. Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security. New York: Association for Computing Machinery, 2014. 90–102. [doi: 10.1145/2660267.2660350]
- [13] Li W, Xia Y, Lu L, Chen H, Zang B. TEEv: Virtualizing trusted execution environments on mobile platforms. Proc. of the 15th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. New York: Association for Computing Machinery, 2019. 2–16. [doi: 10.1145/3313808.3313810]

- [14] Cerdeira D, Martins J, Santos N, Pinto S. ReZone: Disarming TrustZone with TEE Privilege Reduction. Proc. of the 31st USENIX Security Symp. (USENIX Security 2022). Boston: USENIX Association, 2022. 2261–2279.
- [15] Li D, Mi Z, Xia Y, Zang B, Chen H, Guan H. TwinVisor: Hardware isolated Confidential Virtual Machines for ARM. Proc. of the 28th SIGOPS Symp. on Operating Systems Principles. New York: Association for Computing Machinery, 2021. 638–654. [doi: 10.1145/3477132.3483554]
- [16] ARM. Introducing ARM confidential compute architecture. 2022. <https://developer.arm.com/documentation/den0125/latest>
- [17] Mulligan DP, Petri G, Spinale N, Stockwell G, Vincent HJM. Confidential computing—A brave new world. Proc. of the 2021 Int'l Symp. on Secure and Private Execution Environment Design (SEED). Washington: IEEE, 2021. 132–138. [doi: 10.1109/SEED51797.2021.0025]
- [18] Li X, Li X, Dall C, Gu R, Nieh J, Say Y, Stockwell G. Design and verification of the ARM confidential compute architecture. Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2022). Carlsbad: USENIX Association, 2022. 465–484.
- [19] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. New York: Association for Computing Machinery, 2009. 207–220. [doi: 10.1145/1629575.1629596]
- [20] Gu R, Shao Z, Chen H, Wu XN, Kim J, Sjöberg V, Costanzo D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). Savannah: USENIX Association, 2016. 653–669.
- [21] Zhao Y, Sanán D, Zhang F, Liu Y. Reasoning about information flow security of separation kernels with channel-based communication. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2016. 791–810. [doi: 10.1007/978-3-662-49674-9_50]
- [22] Zhao Y, Sanán D, Zhang F, Liu Y. Refinement based specification and security analysis of separation kernels. IEEE Trans. on Dependable and Secure Computing, 2017, 16(1): 127–141. [doi: 10.1109/TDSC.2017.2672983]
- [23] Li SW, Li X, Gu R, Nieh J, Hui JZ. A secure and formally verified Linux KVM hypervisor. Proc. of the 2021 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2021. 1782–1799. [doi: 10.1109/SP40001.2021.0049]
- [24] Li SW, Li X, Gu R, Nieh J, Hui JZ. Formally verified memory protection for a community multiprocessor hypervisor. Proc. of the 30th USENIX Security Symp. (USENIX Security 2021). USENIX Association, 2021. 3953–3970.
- [25] Ma Y, Zhang Q, Zhao S, Wang G, Li X, Shi Z. Formal verification of memory isolation for the trustzone based TEE. Proc. of the 27th Asia-Pacific Software Engineering Conf. Singapore: IEEE, 2020. 149–158. [doi: 10.1109/APSEC51365.2020.00023]
- [26] Jin CZ, Zhang QY, Ma YW, Li XM, Wang GH, Shi ZP, Guan Y. Refinement-based verification of memory isolation mechanism for trusted execution environment. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2189–2207 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/6577.htm>[doi: 10.13328/j.cnki.jos.006577]
- [27] Miao XL, Chang R, Pan SP, Zhao YW, Jiang LH. Modeling and Security Analysis of Access Control in Trusted Execution Environment. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3637-3658 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/6612.htm>[doi: 10.13328/j.cnki.jos.006612]
- [28] Haigh JT, Young WD. Extending the non interface version of MLS for SAT. IEEE Trans. on Software Engineering, 1987(2): 141–150 [doi: 10.1109/TSE.1987.226478]
- [29] Oheimb D. Information flow control revised: Non disclosure=non interface+non package. Proc of the 2004 European Symbol on Research in Computer Security. Berlin: Springer, 2004. 225–243. [doi: 10.1007/978-3-540-30108-0_14]
- [30] Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576–580. [doi: 10.1145/363235.363259]



Fanlang Zeng, Ph.D. candidate. His research interests include trusted execution environment and formal methods.



Shaoping Pan, master. His research interests include trusted execution environment and formal verification.



Rui Chang, Ph.D., associate professor, Ph.D. supervisor. Her research interests include hardware-assisted security, formal verification, and program analysis.



Yongwang Zhao, Ph.D., professor, Ph.D. supervisor. His research interests include formalization, operating systems and security, programming language and compiling, and security certification.



Hao Xu, master. His research interests include trusted execution environment and formal methods.