A Formal Framework of Shielding Systems by Stepwise Refinement

Jiabin Zhu[1], Wenchao Huang[1], Fuyou Miao[1], Cheng Su[1], Baohua Zhao[2], and Yan Xiong[1]
(Corresponding author: Wenchao Huang)

School of Computer Science and Technology, University of Science and Technology of China[1]
The Library of West Campus of USTC, Huang Shan Road, Hefei, Anhui Province, China
Global Energy Interconnection Research Institute[2]
Global Energy Interconnection Research Institute, Binhe Road, Beijing, China
(Email: huangwc@ustc.edu.cn)

(Received May 31, 2018; Revised and Accepted Oct. 18, 2018; First Online July 16, 2019)

Abstract

The shielding systems, *e.g.*, special-purpose hypervisor, provide more secure environments for security-critical applications (SCAs), compared with traditional computer systems. In this paper, we propose a general framework of formally modeling and verifying the shielding systems for enhancing the security. The framework supports multiples types of shielding systems based on different technologies, such as Intel TXT or TrustZone. It is implemented by stepwise refinement, in which the early steps model the common states, events and security properties among the systems. Then the shielding systems are modeled in latter steps, where all the events are refined from the ones in the previous steps without the requirement of reproving soundness of security properties, *e.g.*, memory isolation, data confidentiality, upon the occurrence of each event. Therefore, the complexity of formally verifying new shielding systems is reduced. We implement the framework in the Coq proof assistant, and find potential security threats in using the shielding systems.

Keywords: Formal Methods; Framework; Security Analysis; Shielding Systems

1 Introduction

Recently, many shielding systems [5, 10, 11, 14, 15, 18, 20, 21], *e.g.*, special-purpose hypervisor, have been proposed to enhance security upon traditional computer systems. It is achieved by leveraging instructions of modern CPUs, *e.g.*, Intel TXT [17], TrustZone [1], for launching security-critical applications (SCAs), which run in isolation with legacy OS. For instance, a shielding system may keep the OS from accessing memory being used by SCAs. As a result, the security of SCAs no longer relies on the OS, but only the shielding system. Since the OS is error-prone in design due to its large size, it greatly reduces risks of security breaches by adopting a much smaller shielding system as the substitute of Trusted Computing Base (TCB).

It also requires to formally prove that a shielding system satisfies the security properties, *e.g.*, memory isolation and confidentiality, to achieve enhanced security. As the shielding system provides calling interfaces to the OS and SCAs, it should formally guarantee that both the caller, *i.e.*, SCA or OS, and the callee, *i.e.*, shielding system, run at expected states without any security breaches. The designers of SCAs may be unfamiliar with the shielding system, thus the properties may be broken when the interfaces are called at inappropriate states. On the other hand, since few source codes of shielding systems can be obtained online along with the literatures, it is urgent to re-implement the systems without flaws.

We propose and implement a general framework for formally verifying shielding systems. The framework supports various types of shielding systems, which may use different techniques, *e.g.*, hypervisor or TrustZone. To reduce the complexity of developing different systems, we model and verify the systems by stepwise refinement. Each refinement step is composed of states, events with action and guard, and invariants. The invariants should be preserved whenever any event occurs. In early steps of refinement, we model the internal processes of shielding systems, and prove the invariants of security properties, *e.g.*, memory isolation and data confidentiality. In latter steps, the processes are refined and merged into interfaces provided to the SCAs and the OS. The properties in the steps are still preserved by proving that the interfaces are correctly refined from previous processes. Therefore, the early steps can be reused when verifying new shielding systems, which reduce the workload of modeling and verifying.

The framework contains the following refinement steps as the general model for all shielding systems:

S0 *Abstract specification of memory isolation*: The model's state only contains sufficient structure for proving memory isolation in shielding systems.

S1 *Memory isolation with multi-core support*: The model considers the case that multiple entities, *e.g.*, SCAs, use different cores of CPU simultaneously. We prove the soundness of memory isolation in this case.

S2 *Data confidentiality*: The states and events are refined for proving data confidentiality. Besides the addition of relevant states and events, we introduce an adversary model in Dolev-Yao style [12], and then analyze whether the private data may be leaked under the model.

In the case studies, we model two typical shielding systems, TrustVisor [21] and OSP [11], which use different technologies, *i.e.*, virtualization extensions and TrustZone, respectively. Moreover, TrustVisor only uses a single CPU core, while OSP may use several CPU cores. Faced with the differences, both systems can be successfully refined from our general framework. Specifically, as the shielding systems provide calling interfaces, each callee is divided into several events, which are refined from the events in **S2**.

We implement the framework and verify the cases by the Coq proof assistant [13] with the theory of refinement that borrows elements from [3, 7]. The results show that by using the framework, the complexity in modeling and verifying shielding systems is reduced. We also find potential security threats in using TrustVisor and OSP.

The paper is organized as follows. We provide preliminaries in Section 2. We introduce the refinement framework in Section 3. We propose the general model of the framework in Section 4, and discuss how to use the model to analyze different shielding systems in Section 5. We review some related work in Section 6. Finally, we conclude the paper in Section 7.

2 Preliminaries

2.1 Virtualization Extensions and TrustZone

We summarize recent technologies used for memory isolation, including virtualization extensions [29, 30] and the TrustZone [1], which are modeled and verified in our refinement.

2.1.1 Virtualization Extensions

Virtualization extensions enable an OS to execute in a virtual machine which offers virtual system resources. Specifically, the OS in a virtual machine executes on a virtual version (called host virtual memory) of real physical memory. Hardware components called MMUs in the CPUs that support the extensions can be configured to provide host virtual memory, and data structures called nested page tables are used in the MMUs to translate the host virtual memory addresses to host physical memory addresses.

2.1.2 TrustZone

ARM TrustZone technology [1] offers an isolated execution environment for security programs. Specifically, it partitions system resources into a normal world and a secure world, and prevents programs, *e.g.*, the untrusted OS, executed in the normal world to access the resources in the secure world. For partitioning memory, which is one of the resources, ARM TrustZone provides SoC peripherals called TrustZone Address Space Controllers (TZASCs). Configurations of TZASCs determine memory regions in the secure world, and the configurations can be changed by privileged software executed in the secure world.

2.2 Notations

We use standard notation for equality and logical connectives [3]. We extensively use record types and enumerated types. Record types are defined with the form $rec \overset{def}{=} \{l_1\colon T_1, \ldots, l_n\colon T_n\}$, and therefore elements of the types are of the form $\langle t_1, \ldots, t_n \rangle$. On extending *rec* with a component $l_{n+1} : T_{n+1}$, we define $erec = rec + l_{n+1} : T_{n+1} = \{l_1\colon T_1, \ldots, l_{n+1}\colon T_{n+1}\}$. Accordingly, on reducing *rec* with the component l_k, we define $rrec = rec - l_k$ $(1 \le k \le n)$. Enumerated types are defined by using Haskell-like notation; for example, we define for every type T the type $option\ T \overset{def}{=} NONE \mid Some\ (t\colon T)$. The $option\ T$ is the extension of T with an element *None*. Note that *NONE* has a polymorphic type, and a detailed explanation can be found in the manual of Coq [13] for polymorphic type. We define $T\ set$ as the type of sets over T. Then, we make an extensive use of maps: the type of maps from objects of type A into objects of type B is written $A \mapsto B$. Application of a map m on an object of type a is denoted as $m(a)$ and map update is written as $m(a) := b$, where b overwrites the value associated to a. Finally, we use the notation let $a = b$ in c to simplify our expressions of events, invariants, and so on. For example, the $(s.k).t1 + (s.k).t2$ may be expressed as let $b = (s.k)$ in $b.t1 + b.t2$.

3 Refinement in Coq

Before we introduce the refinement framework, we summarize our theory of refinement that we developed in the Coq proof assistant [13]. The theory borrows elements of refinement from Event-B [7].

3.1 Models for Refinement

Our models are state machines which consist of states and events that result in the transition of states. The goal of machines is to prove soundness of invariants, where the invariants preserve security properties for all states in the machines.

3.1.1 States

We model states of a machine by a record, which is a tuple of state variables. The machine starts at an initial state, and the state changes when an event occurs.

3.1.2 Events

We define an event by using 2 propositions: A conjunction of guards and an action.

$$\mathbf{Grds}_e(p_e, s) \wedge \mathbf{Act}_e(p_e, s, s') \tag{1}$$

Here, the state is transformed from s to s', if the event occurs. Denote p_e as the set of parameters of the event. The event can occur if and only if the conjunction $\mathbf{Grds}_e$ is true. The action $\mathbf{Act}_e$ describes the relation between s and s'. Hence, we can use $\mathbf{Act}_e$ to indicate the changes of states after the occurrence of the event, *e.g.*, changes of the value of a state variable. For convenience, we use the notation $s \sim_{c_1,\ldots,c_n=v_1,\ldots,v_n} s'$ for indicating that values of state variables $c_1, c_2, \ldots, c_n$ of s' is $v_1, v_2, \ldots, v_n$ respectively, while values of other state variables in s' remain unchanged.

3.1.3 Invariants

Invariants are propositions for states of a model. For each event which incurs transitions of states, the invariants should still be preserved. The property can be defined as follows:

$$\forall s\ s'\ p_e.\mathrm{I}(s) \wedge \mathbf{Grds}_e(p_e, s) \wedge \mathbf{Act}_e(p_e, s, s') \rightarrow \mathrm{I}(s'). \tag{2}$$

Here, $\mathrm{I}(s)$ is the conjunction of all invariants of a model at a state s. The property states that for an event e whose parameters are p_e and that leads a state s to a state s', if the invariants are valid in the state s, the invariants are valid in the state s'.

3.2 Refinement Method

The refinement is a process of constructing a more concrete model from the previous abstract model. The concrete model should preserve invariants of the abstract model so that we can model a complex object, *e.g.*, a shielding system, by using several models and each of the model is constructed by focusing on individual design aspects. In practice, the concrete model preserves a refined form of the invariants. The refined form and the conjunction of invariants are mutually implicated, and therefore we should prove the following theorem.

$$\forall s_a\ s_c.\mathrm{I}_a(s_a) \wedge \mathrm{SR}^{ac}(s_a, s_c) \leftrightarrow \mathrm{I}_{r,c}(s_c) \wedge \mathrm{SR}^{ac}(s_a, s_c). \tag{3}$$

I_a is the conjunction of all invariants of the abstract model and $\mathrm{I}_{r,c}$ is the refined form of I_a. SR^{ac} is the conjunction of all propositions that specify relations between values of state variables of states of the concrete model and the abstract model. The theorem specifies $\mathrm{I}_{r,c}$ is the refined form of I_a. Now, we prove that $\mathrm{I}_{r,c}$ is preserved in the concrete model by using the preservation of I_a in the abstract model for reducing workload in auditing each event in the concrete model for $\mathrm{I}_{r,c}$. Specifically, for each event e_c in the concrete model, we first manually identify a corresponding abstract event e_a in the abstract model. We assume e_c leads a state s_c to a state s'_c, and e_a leads a state s_a to a state s'_a. Then, we first prove the following theorem.

$$\forall s_c\ p_{e_c}.\mathrm{I}_c(s_c) \wedge \mathbf{Grds}_{e_c}(p_{e_c}, s_c) \rightarrow\ \exists p_{e_a}\ \exists s_a.\mathrm{SR}^{ac}(s_a, s_c) \wedge \mathbf{Grds}_{e_a}(p_{e_a}, s_a). \tag{4}$$

The theorem that states that a concrete event can only occur when the corresponding abstract event occurs. Then, we prove the following theorem.

$$\begin{aligned} &\forall s_c\ s'_c\ p_{e_c}.\mathrm{I}_c(s_c) \wedge \mathbf{Act}_{e_c}(p_{e_c}, s_c, s'_c) \rightarrow \\ &\exists p_{e_a}\ \exists s_a\ \exists s'_a.\mathrm{SR}^{ac}(s_a, s_c) \\ &\wedge \mathrm{SR}^{ac}(s'_a, s'_c) \wedge \mathbf{Act}_{e_a}(p_{e_a}, s_a, s'_a). \end{aligned} \tag{5}$$

The theorem that states that if a concrete event occurs, the abstract event can occur in such a way that the resulting states correspond again. With the two theorems, we can ensure the preservation of $\mathrm{I}_{r,c}$ on each concrete event by using the preservation of I_a on the corresponding abstract event.

4 A General Model of Shielding Systems

In this section, we construct a general model of shielding systems by stepwise refinement. The model contains general operations used for allocating and deallocating storage of the OS and SCAs and for managing data of the OS and SCAs.

4.1 Requirements and Assumptions

We start by specifying general requirements of shielding systems and by making our assumptions about the system explicit.

Requirement 1 (Shielding System). A shielding system, *e.g.*, a hypervisor, executes in a higher privilege mode against **guests**, *i.e.*, OS or SCAs.

Requirement 2 (Memory Isolation). Memory that can be accessed by each guest is pairwise disjoint.

Requirement 3 (Data Confidentiality). Critical data, *e.g.*, private keys, of an SCA cannot be leaked to other guests.

The compromised guests may operate in following ways.

Assumption 1 (Dolev-Yao Adversary). Similar to the Dolev-Yao adversary model [12], the guests may perform data analyzing, (*e.g.*, decomposing data, decrypting data using obtained keys), and data synthesizing according to the analyzed result. However, the adversary cannot perform any crypto-analysis.

Assumption 2 (Calls). A guest may call any interface of the shielding system exposed to the guest when the guest is executing. Moreover, the guest may call the interfaces with arbitrary parameter values.

4.2 Memory Isolation (S0)

The abstract machine (*S0*) models the property of memory isolation.

States S^0: Denote S^i as the record for states at the ith refinement. We define S^0 in *S0* as follows:

$$S^0 \overset{def}{=} \{loc : tguests \mapsto tloc\ set\}. \qquad (6)$$

Here, $loc(id)$ represents memory locations that can be accessed by a guest whose identifier is id. Generally, we define $t*$ as the type of some variables. For example, *tloc*, *tguests* is the type of memory locations and identifiers of guests respectively. $tguests \overset{def}{=} Tisca(id : tident_{sca})|OSID$. Therefore, a guest may be an SCA with the identifier id or an OS with the identifier *OSID*.

Invariant: We model Memory Isolation (*i.e.*, Requirement 2) by the following invariant.

$$\text{Isolation}^0_{mem}(s) \overset{def}{=} (\forall\ i\ j.i \neq j \rightarrow (s.loc(i) \cap s.loc(j)) = \phi). \qquad (7)$$

Events: According to the invariant Isolation^0_{mem}, we model an event ChLoc^0 in which the memory that can be accessed by a guest id is changed to l. Specifically,

$$\begin{aligned}&\mathbf{Grd}_{\mathsf{ChLoc}^0}\ l\ i\ s \overset{def}{=}\\ &\quad \textbf{(i)}\ (\forall j, j \neq i \rightarrow l \cap s.loc(j) = \phi)\\ &\mathbf{Act}_{\mathsf{ChLoc}^0}\ l\ i\ s\ s' \overset{def}{=}\\ &\quad newl = (s.loc(i) := l) \wedge s \sim_{loc=newl} s'.\end{aligned} \qquad (8)$$

In the guard, we require that l should not intersect with memory that can currently be accessed by other guests, including the OS and other SCAs. In the action, the state variable *loc* is changed into l, where l is built by leveraging *loc* in original state s, *i.e.*, $s.loc$. Hence, we can prove that the invariant $\text{Isolation}^0_{mem}(s)$ is preserved after the occurrence of the event.

4.3 Memory Isolation with Changing Modes (S1)

In the first refinement (*S1*), we additionally model events when guests are changing their internal modes. Specifically, the shielding system determines whether a guest can execute or not, which correspond to executing mode and suspending mode. Therefore, for any guest i, $loc(i)$ turns into ϕ when guest i is switched into suspending mode. Moreover, the shielding system may allocate new memory of a guest when a guest is in suspending mode, and the value of $loc(i)$ becomes more complicated when the guest is switched back to executing mode. The above situations motivate us to model new events and prove memory isolation in *S1*.

States S^1: The record for states in *S1* is shown in the following.

$$\begin{aligned}S^1 \overset{def}{=}\ &\{mode : tguests \mapsto tmode,\\ &mem : tguests \mapsto tloc\ set\}.\end{aligned} \qquad (9)$$

The internal mode of a guest i is denoted as $mode(i)$. $mode(i) = EXE$ or SUS, when i is executing or suspending respectively. Denote $mem(i)$ as locations of memory allocated for guest i by the shielding system. Then, we define the invariant for modeling the relation between S^0 and S^1:

$$\begin{aligned}&\text{SR}^{01}(s0, s1) \overset{def}{=} (\forall i.s1.mode(i) = EXE \rightarrow\\ &s0.loc(i) = s1.mem(i)) \wedge\\ &(\forall i.s1.mode(i) = SUS \rightarrow s0.loc(i) = \phi).\end{aligned} \qquad (10)$$

The invariant means that for any guest i, if it is in executing mode, then the memory that can be accessed by i, *i.e.*, $s0.loc(i)$, is $s1.mem_{aloc}(i)$. Otherwise, if i is in suspending mode, no memory can be accessed by i.

Invariant: Based on the refined state S^1, we refine $\text{Isolation}^0_{mem}(s)$ by the following invariant.

$$\begin{aligned}&\text{Isolation}^1_{mem}(s) \overset{def}{=} \forall i\ j.i \neq j \rightarrow\\ &s.mode(i) = EXE \rightarrow s.mode(j) = EXE \rightarrow\\ &s.mem(i) \cap s.mem(j) = \phi\end{aligned} \qquad (11)$$

The invariant states that there is no shared memory between memory that can be accessed by two guests that are executing simultaneously. We notice the invariant because we use refinements of *S1* to model shielding systems [11, 18] which support multi-core CPU. In this case, several SCAs may run on different cores simultaneously.

Events: There are 2 types of events in *S1*:

1) The event when a guest is in suspending mode. Without loss of generality, the action in this case is modeled as arbitrary changes of *mem*. We further refine the action according to different implementations in the case studies.

2) Transition of the modes. The events used for executing a guest or suspending a guest, and the events are refined from ChLoc^0_{guest}. For example, we model an event in the following:

$$\begin{aligned}
&\mathbf{Grd}_{\mathsf{StoE}^1}\ i\ s \overset{def}{=} \\
&\quad \textbf{(i)}\ mode(i) = SUS \wedge \\
&\quad \textbf{(ii)}\ (\forall j. j \neq i \wedge mode(j) = EXE \rightarrow \\
&\quad s.mem(i) \cap s.mem(j) = \phi) \\
&\mathbf{Act}_{\mathsf{StoE}^1}\ \ i\ s\ s' \overset{def}{=} \\
&\quad newm = (s.mode(i) := EXE) \wedge s \sim_{mode=newm} s'.
\end{aligned} \tag{12}$$

In the event, the mode of the guest i is transited from *SUS* to *EXE*. Specially, the guard requires that $mem(i)$ does not intersect with memory that can be accessed by other guests. The guard helps proving the theorem of refinement (4). Note that we do not model the event when a guest is executing, since *mem* remains unchanged according to current systems.

4.4 Data Confidentiality (*S2*)

The goal of *S2* is to prove data confidentiality as illustrated in Requirement 3. We leverage property of memory isolation in *S1*, and model more state variables that may affect the goal. For example, a potential attack occurs when a guest is switched into suspending mode. In this case, the shielding system may save the guest's data on the memory and registers, and then allocates the memory to other guests. If the memory or registers are not cleared before they are allocated, the data may be leaked since other guests are assumed to be adversarial. Hence, we formulate the data stored on the memory and refine the events in *S1*.

$tdata \overset{def}{=}$	*Hash tdata*	Hashes of data
$\mid$	*Key tkey*	Keys
$\mid$	*Enc tkey tdata*	Ciphertexts
$\mid$	*Id tguest*	Identifiers of guests
$\mid$	*Cons tdata tdata*	Concatenation of data
$\mid$	*Others*	Other types of data

Figure 1: The sub-types of data defined in *S2*

For preventing more complicated attacks, we model the adversary in Dolev-Yao style. The adversary may perform analysis on obtained data, *e.g.*, decrypting a cypher using obtained encryption key, and then forge data. In Figure 1, we firstly define the type *tdata* for formulating data, and divide *tdata* into several sub-types. Then, we model the ability of adversary by defining functions *analz* and *synth* and the corresponding axioms [6]. Both functions share the same declarations:

$$analz, synth : tdata\ set \mapsto tdata\ set$$

The function *analz* outputs the set of data that can be analyzed from the input. For example, if $Key(invKey(k)) \in s$ and $Enc(k,m) \in s$, then $m \in analz(s)$. *invKey* is a function that leaves a key unaltered if the key is symmetric, or turns a key into its corresponding asymmetric half if the key is asymmetric. The function *synth* outputs the set of data that can be composed by using the input. For example, if $m, k \in s$, then $Enc(k, m) \in synth(s)$. The properties in both examples can be proved by using the defined axioms. For simplicity, we use the notation δ that $\delta(d) = synth(analz(d))$.

States S^2 : The record for states in *S2* is shown in the following.

$$\begin{aligned}
S^2 \overset{def}{=}\ & S^1 + data_{mem} : tloc \mapsto tdata\ set \\
& + pdata : tguests\ \mapsto tdata\ set \\
& + know : tguests \mapsto tdata\ set \\
& + gset : tdata\ set \\
& - mode \\
& + core : tguests \mapsto tcores\ set \\
& + data_{regs} : tcores \mapsto tdata\ set
\end{aligned} \tag{13}$$

Here, $data_{mem}(i)$ represents the set of data that may currently be stored on memory location i. $pdata(i)$ represents the set of private data owned by guest i. $know(i)$ represents guest i's **knowledge**, *i.e.*, the set of data that may have been obtained by guest i. In our model, the data $d \in know(i)$, if d was in the memory location or register that could be accessed by the guest i. Note that the data d may be divide into multiple blocks, *e.g.*, $\{b_1, b_2, \ldots, b_m\}$, which are stored into multiple locations of memory, *e.g.*, $\{d_1, d_2, ..., d_m\}$, respectively. Obviously, an adversary who only reads a single block of d, *e.g.*, b_i in $\{b_1, b_2, \ldots, b_m\}$, does not obtain d. However, the private information, *e.g.*, the encryption key, may happen to be in b_1, which lead to successful attacks. Hence, we simply assume that the adversary may perform attacks by leveraging d, if the adversary obtains any b_i from the location d_i.

The state variable *gset* represents the set of global data that have been generated by the shielding system and guests, and is used to achieve the assumption of Dolev-Yao adversary. Since the adversary cannot perform crypto-analysis, the possibility that newly generated random data equal any historical randomly generated data is assumed to be 0. Therefore, in our model, the data d should satisfy the guard that ($\forall$ k. $k \in parts(d) \rightarrow k \notin s.gset$), before d is newly generated in the events, in which $parts(d)$ is added into *gset*. *parts* is a function that outputs the set of data that can be extracted from the input. For example, if $Enc(k,m) \in s$, then $m \in parts(s)$. Axioms for *parts* can be found in [6].

For generality, we also model and prove confidentiality of data stored in the registers, besides the memory. Recall that the cores of an CPU may be used by multiple guests simultaneously. It should be proved that the data stored on the registers, which belong to the cores used by a guest, are not leaked to other guests. In S^2, we replace *mode* with *core*, where *core*(i) represents the set of cores currently used by the guest i. Similar to $data_{mem}$, $data_{regs}(i)$ represents the set of data that may currently be stored at the registers of the core i. Therefore, *core* is a refined state of *mode* that a guest is using a core or several cores if it is executing; otherwise, it does not use any cores. Formally, we define the relation as invariant SR^{12} as follows.

$$\begin{aligned}&\mathrm{SR}^{12}(s1, s2) \overset{def}{=}\\ &\quad(\forall i.\ s1.mode(i) = EXE \leftrightarrow s2.core(i) \neq \phi)\wedge\\ &\quad(\forall i.\ s1.mode(i) = SUS \leftrightarrow s2.core(i) = \phi).\end{aligned} \tag{14}$$

Invariant: We prove data confidentiality Conf as follows.

$$\begin{aligned}&oknow(s, i) = \{x|j \in GIDS, j \neq i, x \in s.know(j)\}.\\ &\mathrm{Conf}(s) \overset{def}{=} \forall i. i \in GIDS \wedge i \neq OSID \rightarrow\\ &\delta(oknow(s, i)) \cap s.pdata(i) = \phi.\end{aligned} \tag{15}$$

The invariant states that the private data owned by any SCA, *i.e.*, *pdata*(i), cannot be analyzed or synthesized according to the knowledge of other guests. *oknow*(s,i) represents union of knowledge of guests other than guest i. Here, we assume that the other guests may collude by sharing knowledge with others and perform attacks on guest i. *GIDS* is a constant that represents the set of identifiers of all guests.

Events: We divide events in $S2$ into 5 parts.

1) Generating private data. In the events, a guest i generates private data d, *e.g.*, private keys, and saves d into memory or registers. For example, when d is saved into memory location ld, the event is modeled as follows:

$$\begin{aligned}&\mathbf{Grd}_{\mathsf{Gen}^2_{mem}}\ i\ d\ ld\ s \overset{def}{=}\\ &\quad\mathbf{(i)}\ i \in GIDS \wedge \mathbf{(ii)}\ s.core(i) \neq \phi\wedge\\ &\quad\mathbf{(iii)}\ ld \subseteq s.mem(i) \wedge \mathbf{(iv)}\ (\forall k. k \in parts(d) \rightarrow k \notin s.gset)\\ &\mathbf{Act}_{\mathsf{Gen}^2_{mem}}\ i\ d\ ld\ s\ s' \overset{def}{=}\\ &\quad newd = update_{mem}(s.data_{mem}, ld, d)\wedge\\ &\quad newp = (s.pdata(i) := s.pdata(i) \cup d)\wedge\\ &\quad newk = (s.know(i) := s.know(i) \cup d)\wedge\\ &\quad newe = (s.gset \cup parts(d))\wedge\\ &\quad s \sim_{data_{mem},pdata,know,gset=newd,newp,newk,newe} s'.\end{aligned} \tag{16}$$

Guard **(ii)**, **(iii)** states that guest i is executing and can access the location ld, respectively. Guard **(iv)** states that the data d is newly generated. In the action, the memory, private data and the knowledge owned by guest i are changed. Here, $(update_{mem}(f, l, d))(x) = \{d, \text{if } x \in l; f(x), \text{if } x \notin l\}$.

2) Malicious operations. The guests may perform operations according to the Dolev-Yao adversary model. Specifically, a guest may write data to the memory or registers that can be accessed by the guest. The following event represents the case when guest writes data to the memory.

$$\begin{aligned}&\mathbf{Grd}_{\mathsf{Mal}^2_{mem}}\ i\ d\ ld\ s \overset{def}{=}\\ &\quad\mathbf{(i)}\ i \in GIDS \wedge \mathbf{(ii)}\ s.core(i) \neq \phi\wedge\\ &\quad\mathbf{(iii)}\ ld \subseteq s.mem(i) \wedge \mathbf{(iv)}\ d \in \delta(s.know(i))\\ &\mathbf{Act}_{\mathsf{Mal}^2_{mem}}\ i\ d\ ld\ s\ s' \overset{def}{=}\\ &\quad newd = update_{mem}(s.data_{mem}, ld, \{d\})\wedge\\ &\quad newk = (s.know(i) := s.know(i) \cup \{d\})\wedge\\ &\quad s \sim_{data_{mem},know=newd,newk} s'.\end{aligned} \tag{17}$$

Guard **(ii)** states that only the executing guest can perform the operations. In the action, data d is written to location ld, where ld is restricted by guard **(iii)**, and d is added to the knowledge of the guest i. Data d is restricted by the guard **(iv)** that the data must be analyzed or synthesized from the knowledge of the guest.

3) Transition of modes. There are two events in this case: (1) StoE^2: the mode of a guest is switched from suspending mode to executing mode, (2) EtoS^2: the mode of a guest is switched to suspending mode. When the mode of guest is switched from executing mode to suspending mode, the guest does not use any core, and vice versa. Therefore, the events are directly refined from events in $S1$. For example, when the guest i starts using with core *idc*, we model the event as follows.

$$\begin{aligned}&\mathbf{Grd}_{\mathsf{StoE}^2}\ i\ idc\ s \overset{def}{=}\\ &\quad\mathbf{(i)}\ i \in GIDS \wedge \mathbf{(ii)}\ s.core(i) = \phi\wedge\\ &\quad\mathbf{(iii)}\ (\forall j.\ j \in GIDS \wedge j \neq i \wedge s.core(j) \neq \phi \rightarrow\\ &\quad s.mem(i) \cap s.mem(j) = \phi)\wedge\\ &\quad\mathbf{(iv)}\ (\forall j.\ j \in GIDS \rightarrow idc \notin s.core(j))\wedge\\ &\quad\mathbf{(v)}\ (\forall j.\ j \in GIDS \wedge j \neq i \wedge j \neq OSID \rightarrow\\ &\quad\delta(oknow(s, j) \cup s.data_{regs}(idc)\\ &\quad\cup\ clt(s.data_{mem}, s.mem(i))) \cap s.pdata(j) = \phi)\\ &\mathbf{Act}_{\mathsf{StoE}^2}\ i\ idc\ s\ s' \overset{def}{=}\\ &\quad newc = (s.core(i) := s.core(i) \cup \{idc\})\wedge\\ &\quad newk = (s.know(i) := s.know(i) \cup s.data_{regs}(idc)\\ &\quad\cup\ clt(s.data_{mem}, s.mem(i)))\wedge\\ &\quad s \sim_{core,know=newc,newk} s'.\end{aligned} \tag{18}$$

The guard **(ii)** and **(iii)** refines the guard **(i)** and **(ii)** in StoE^1, respectively. The guard **(iv)** states that the core *idc* has not been used by any guest. The guard

(v) states that for any guest except guest i and the OS, private data of the guest cannot be analyzed or synthesized from the union of the knowledge of other guests and the set of data in memory and registers that can be accessed by guest i. In the action, the state variable *core* is changed and the set of data in memory and registers that can be accessed by guest i is added to the knowledge of guest i. The function *clt* is used for collecting data in memory. Formally, $clt(f, m) = \{d | x \in m, d \in f(x)\}$.

4) Changing the number of used cores. Specifically, a guest may start using more or less cores when the guest is executing. To preserve confidentiality in this event, we add the guard which is similar to the guard **(v)** in StoE^2.

5) Other operations performed by the shielding system. The shielding system may perform other operations for managing the memory or registers. For example, the shielding system may write data to the memory or registers, clear the data in the memory or registers, reallocate the memory for a guest, or generate data, *e.g.*, keys used for encrypting data of guests. We show the event that writes data to memory in the following.

$$\begin{aligned}
&\mathbf{Grd}_{\mathsf{SWrite}^2_{mem}}\ d\ ld\ s \overset{def}{=} \\
&\quad \textbf{(i)}\ \forall\ j_1 . j_1 \in GIDS \wedge \\
&\quad s.core(j_1) \neq \phi \wedge ld \cap s.mem(j_1) \neq \phi \rightarrow \\
&\quad (\forall j_2 .\ j_2 \in GIDS \wedge j_2 \neq j_1 \wedge j_2 \neq OSID \rightarrow \\
&\quad \delta(oknow(s, j_2) \cup d) \cap s.pdata(j_2) = \phi) \\
&\mathbf{Act}_{\mathsf{SWrite}^2_{mem}}\ d\ ld\ s\ s' \overset{def}{=} \\
&\quad newd = update_{mem}(s.data_{mem}, ld, d) \wedge \\
&\quad newk = update_{knwl}(s.know, s.core, s.mem, ld, d) \wedge \\
&\quad s \sim_{data_{mem}, know = newd, newk} s'.
\end{aligned} \tag{19}$$

The guard **(i)** states that if the intersection between ld and memory that can be accessed by the guest j_1 is not empty, for any guest except guest j_1 and the OS, private data of the guest cannot be analyzed or synthesized from the union of the set of data to be written, *i.e.*, d, and the knowledge of other guests. In the action, the shielding system writes d to ld and the knowledge of guests that can access a part of memory locations in ld is added with d. Formally, $(update_{knwl}(k,c,m,ld,d))(i) = \{k(i),\ \text{if}\ (m(i) \cap ld = \phi) \vee (c(i) = \phi);\ k(i) \cup d,\ \text{if}\ (m(i) \cap ld \neq \phi) \wedge (c(i) \neq \phi)\}$. We omit the definition of other operations in this paper.

5 Case Studies

In this section, we model two shielding systems, *i.e.*, TrustVisor [21] and OSP [11], based upon the refinements of our general model.

5.1 Case 1: TrustVisor

TrustVisor [21] is an open-source hypervisor used for shielding SCAs, which are called Pieces of Application Logic (PALs) by TrustVisor. TrustVisor provides code integrity as well as data integrity and confidentiality for SCAs. Besides, TrustVisor leverages the features of modern processors to reduce the performance overhead caused by protecting SCAs from the OS and its applications. We now explain TrustVisor's functions related to our goals as follows.

1) *registration*: The *registration* interface allows the OS to register SCAs. When the OS calls *registration*, TrustVisor prepares an environment for launching a new SCA, *i.e.*, the memory to be accessed by the SCA. The memory cannot be accessed by the OS and the data on the memory should be prepared. In practice, before using *registration*, the OS sets a region of the memory and the data on the memory, and pass the region as the parameters in *registration*. After *registration* is called, TrustVisor checks the parameters, and sets the corresponding memory unaccessible by the OS. Besides, TrustVisor prepares the context of the SCA, *i.e.*, the data to be loaded to the registers, according to the parameters in *registration*.

2) *invocation*: Following *registration*, the OS may invoke an SCA by calling *invocation*. When the OS calls *invocation*, the OS is switched into suspending mode, and the selected SCA is started to execute. TrustVisor firstly saves the context of the OS, and then loads the context of the SCA. Besides, the OS produces input and passes the input to the SCA. Specifically, the OS firstly puts the input in memory region specified in the parameters of *invocation*. Then, after calling *invocation*, TrustVisor copies the data in the memory region to the memory that can be accessed by the SCA. Finally, the SCA is allowed to use the core of the CPU. Note that the design of TrustVisor only supports using a single CPU core, and TrustVisor can manage multiple SCAs.

3) *termination*: When an SCA has completed executing and returns to the OS, *termination* is called through a return point set by TrustVisor in the stack used by the SCA. When an SCA calls *termination*, the OS and the executing SCA is switched into executing mode and suspending mode respectively. During the process, TrustVisor saves the context of the executing SCA, and copies the context of the OS. Besides, the SCA produces output and passes the output to the OS. Specifically, the SCA firstly puts the output in a memory region specified by TrustVisor. Then, after calling *termination*, TrustVisor copies the data in the memory region to the memory region specified in the parameters of *invocation*.

4) *unregistration*: TrustVisor zeros all execution state associated with the SCA specified by parameters of

unregistration. When the OS unregisters an SCA by calling *unregistration*, TrustVisor deallocates all memory allocated for the SCA, clears data in the deallocated memory, and allocates the memory to the OS.

5) hv_{seal}: When an SCA calls hv_{seal}, TrustVisor encrypts data in the specified blocks of memory using the symmetric key, *e.g.*, k, owned by the TrustVisor, writes the encrypted data to the memory allocated for the SCA, and continues executing the SCA. Here, TrustVisor binds the identifier of the SCA i to the encrypted data d. In other words, the ciphertext outputted by hv_{seal} are formed as $Enc(k, Cons(d, i))$.

6) hv_{unseal}: Corresponding to hv_{seal}, when an SCA calls hv_{unseal} for decrypting $Enc(k, Cons(d, i))$, TrustVisor ensures that d is originally sealed by i, *i.e.*, i is the identifier of the SCA, before writing the decrypted data to the memory that can be accessed by the SCA. Then, TrustVisor continues executing the SCA.

States (S^3_{tv}): We show the record for states in S^3_{tv} in the following.

$$S^3_{tv} \overset{def}{=} \begin{aligned} & S^2 + func : option\ tfun \\ & + ctxt : tguest \mapsto (tdata\ set) \\ & + symkey : option\ tkey \end{aligned} \quad (20)$$

The state variable *func* denotes which function provided by TrustVisor, *e.g.*, *registration*, is currently being executed. If $func = NONE$, then no function is running. Note that TrustVisor only uses a single core of the CPU, therefore there is at most one executing function at each state. We divide the process of calling functions into several steps, and each step can be refined from our general model. Therefore, the state variable *func* records not only the executing function's name, but also the function's current step and parameters. For example, $Ivc(s : step_{ivc})(p : prmt_{ivc})$ is a subtype of *tfun*. If $func = Some(Ivc(IVC_{sv}, prm_{ivc}(i)))$, it means that the function *invocation* is being executed, the current step in *invocation* is saving the context of the OS, and the selected SCA is guest i. Here, $step_{ivc}$ is the type of steps of *invocation*, and each of the step is defined as a constant. $prmt_{ivc}$ is the type of parameters of *invocation*, and formally, $prmt_{ivc} \overset{def}{=} prm_{ivc}(t : tguest)$. In S^3_{tv}, we also add *ctxt* and *symkey* representing contexts and symmetric key held by TrustVisor, respectively.

Events: As mentioned, the process of calling each function is divided into several steps, and each step is modeled as an event refined from *S2* or a new event in which the state variables in former refinement level do not change. We show the example of modeling functions *invocation* and hv_{seal} as follows.

1) *invocation*: The process is divided into 5 steps. In the first step, the OS is switched into suspending mode. Hence, the event is refined from EtoS^2.

$$\begin{aligned} & \mathbf{Grd}_{\mathsf{EtoS}^3_{ivc}}\ i\ s \overset{def}{=} \\ & \quad \mathbf{(i)}\ (s.func = NONE) \wedge \mathbf{(ii)}\ (s.core(OSID) \neq \phi) \wedge \\ & \quad \mathbf{(iii)}\ (i \in GIDS \wedge i \neq OSID) \\ & \mathbf{Act}_{\mathsf{EtoS}^3_{ivc}}\ i\ s\ s' \overset{def}{=} \\ & \quad newc = (s.core(OSID) := \phi) \wedge \\ & \quad s \sim_{func, core = Some(Ivc(IVC_{sv}, prm_{ivc}(i))), newc} s'. \end{aligned} \quad (21)$$

The guard **(i)** states that there is no function that is executing. The guard **(ii)** states that the OS is occupying the CPU, which means the OS is executing. In guard **(iii)**, guest i is selected to execute. Therefore in the action, the OS no longer uses the CPU so that the mode of the OS is switched into suspending mode. On the other hand, the step turns into IVC_{sv}.

In the second step IVC_{sv}, TrustVisor saves the context of the OS. Since context is saved to the new state *ctxt* in S^3_{tv} and no other state is changed, we model the step as a new event.

$$\begin{aligned} & \mathbf{Grd}_{\mathsf{Sv}^3_{ivc}}\ i\ s \overset{def}{=} \\ & \quad \mathbf{(i)}\ s.func = Some(Ivc(IVC_{sv}, prm_{ivc}(i))) \\ & \mathbf{Act}_{\mathsf{Sv}^3_{ivc}}\ i\ s\ s' \overset{def}{=} \\ & \quad newc = (s.ctxt(OSID) := s.data_{regs}(UCORE)) \wedge \\ & \quad s \sim_{func, ctxt = Some(Ivc(IVC_{copy}, prm_{ivc}(i))), newc} s'. \end{aligned} \quad (22)$$

The guard ensures that the event occurs only when the previous step has been executed, in which the *func* is changed into $Some(Ivc(IVC_{sv}, prm_{ivc}(i)))$. In the action, the data on the registers are assigned to $ctxt(OSID)$, and the step turns into IVC_{copy}. Here, *UCORE* is a constant representing the identifier of the single core used by TrustVisor.

In the third step IVC_{copy}, TrustVisor manages the memory for preparing the input of *invocation* before SCA i is executed. Specifically, the data d are generated by the OS as input, which are stored in *ls*, and then copied to the location *ld*, which belongs to SCA i. The step is refined from the event SWrite^2_{mem} in *S2*.

$$\begin{aligned} & \mathbf{Grd}_{\mathsf{Copy}^3_{ivc}}\ i\ ls\ d\ ld\ s \overset{def}{=} \\ & \quad \mathbf{(i)}\ s.func = Some(Ivc(IVC_{copy}, prm_{ivc}(i))) \wedge \\ & \quad \mathbf{(ii)}\ ld \subseteq s.mem(i) \wedge \mathbf{(iii)}\ ls \subseteq s.mem(OSID) \wedge \\ & \quad \mathbf{(iv)}\ d = clt(s.data_{mem}, ls) \\ & \mathbf{Act}_{\mathsf{Copy}^3_{ivc}}\ i\ ls\ d\ ld\ s\ s' \overset{def}{=} \\ & \quad newd = update_{mem}(s.data_{mem}, ld, d) \wedge \\ & \quad s \sim_{func, data_{mem} = Some(Ivc(IVC_{load}, prm_{ivc}(i))), newd} s'. \end{aligned} \quad (23)$$

Note that the guard **(i)** implies there is no guest that is executing. Therefore when Copy^3_{ivc} occurs, the guard **(i)** in SWrite_{mem} is ensured, and the knowledge of any guest is not changed for SWrite^2_{mem}, *i.e.*, $update_{knwl}(s.know,\ s.core,\ s.mem,\ ld,\ d) = s.know$. The step then turns into IVC_{load}.

In the fourth step, TrustVisor loads the context of the SCA i. It is also refined from the event in *S2*, in which TrustVisor writes data to registers.

$$\mathbf{Grd}_{\mathsf{Load}^3_{ivc}}\ i\ s \overset{def}{=}$$
$$\textbf{(i)}\ s.func = Some(Ivc(IVC_{load}, prm_{ivc}(i)))$$
$$\mathbf{Act}_{\mathsf{Load}^3_{ivc}}\ i\ s\ s' \overset{def}{=}$$
$$newd = (s.data_{regs}(UCORE) := s.ctxt(i)) \wedge$$
$$s \sim_{func, data_{regs} = Some(Ivc(IVC_{StoE}, prm_{ivc}(i))), newd}\ s'. \tag{24}$$

The guard simply checks if it is at the fourth step, *i.e.*, IVC_{load}. In the action, the context, which is stored by TrustVisor and modeled as a global state *ctxt*, is assigned to the registers, *i.e.*, $data_{regs}$. Then the step turns into IVC_{StoE}.

In the final step, the mode of SCA i is switched into executing mode, therefore the event is refined from StoE^2.

$$\mathbf{Grd}_{\mathsf{StoE}^3_{ivc}}\ i\ s \overset{def}{=}$$
$$\textbf{(i)}\ s.func = Some(Ivc(IVC_{StoE}, prm_{ivc}(i)))$$
$$\mathbf{Act}_{\mathsf{StoE}^3_{ivc}}\ i\ s\ s' \overset{def}{=}$$
$$newc = (s.core(i) := s.core(i) \cup \{UCORE\}) \wedge$$
$$newk = (s.know(i) := s.know(i) \cup s.data_{regs}(UCORE)$$
$$\cup\ clt(s.data_{mem}, s.mem(i))) \wedge$$
$$s \sim_{func, core, know = NONE, newc, newk}\ s'. \tag{25}$$

Here, the guard is refined that it only checks whether the step is IVC_{StoE}. The guards defined in StoE^2 are true according to the definitions of previous steps. Finally, the state variable *func* turns into *NONE*.

2) hv_{seal}: We divide the process into 3 steps. In the first step, the mode of SCA i is switched from executing mode to suspending mode. Therefore the step is also refined from EtoS^2.

$$\mathbf{Grd}_{\mathsf{EtoS}^3_{seal}}\ i\ d\ s \overset{def}{=}$$
$$\textbf{(i)}\ (s.func = NONE) \wedge \textbf{(ii)}\ (s.core(i) \neq \phi) \wedge$$
$$\textbf{(iii)}\ (i \in GIDS \wedge i \neq OSID) \tag{26}$$
$$\mathbf{Act}_{\mathsf{EtoS}^3_{seal}}\ i\ d\ s\ s' \overset{def}{=} newc = (s.core(i) := \phi) \wedge$$
$$s \sim_{func, core = Some(Seal(SEAL_{enc}, prm_{seal}(i))), newc}\ s'.$$

The guards state that SCA i is executing. Besides the actions refined from EtoS^2, we add action that the step turns into $SEAL_{enc}$.

In the second step, TrustVisor encrypts data in memory that can be accessed by SCA i, and save the encrypted data. The step is refined from the event SWrite^2_{mem} in *S2*.

$$\mathbf{Grd}_{\mathsf{Seal}^3_{enc}}\ i\ ls\ d\ ld\ k\ s \overset{def}{=}$$
$$\textbf{(i)}\ s.func = Some(Seal(SEAL_{enc}, prm_{seal}(i))) \wedge$$
$$\textbf{(ii)}\ ld \subseteq s.mem(i) \wedge \textbf{(iii)}\ s.symkey = Some(k) \wedge$$
$$\textbf{(iv)}\ ls \subseteq s.mem(i) \wedge \textbf{(v)}\ d \in clt(s.data_{mem}, ls)$$
$$\mathbf{Act}_{\mathsf{Seal}^3_{enc}}\ i\ ls\ d\ ld\ k\ s\ s' \overset{def}{=}$$
$$newd = update_{mem}(s.data_{mem}, ld,$$
$$\{Enc(k, Cons(d, Id(i)))\}) \wedge$$
$$s \sim_{func, data_{mem} = Some(Seal(SEAL_{StoE}, prm_{seal}(i))), newd}\ s'. \tag{27}$$

The guard **(i)** checks whether the step is $SEAL_{enc}$. In the guards, we also make constraints on the parameters of the event. Specifically, the data d in memory *ls* are encrypted, and the encrypted data are saved to *ld*, which is located at memory that can be accessed by SCA i. In guard **(iii)**, TrustVisor must have generated the encryption key before the event occurs.

In the final step, the mode of an SCA is switched back to executing mode. Hence, the step is refined from the event StoE^2. Since the refined event is similar to StoE^3_{ivc}, we simply provide the definition as follows and omit the explanations.

$$\mathbf{Grd}_{\mathsf{StoE}^3_{seal}}\ i\ s \overset{def}{=}$$
$$\textbf{(i)}\ s.func = Some(Seal(SEAL_{StoE}, prm_{seal}(i)))$$
$$\mathbf{Act}_{\mathsf{StoE}^3_{seal}}\ i\ s\ s' \overset{def}{=}$$
$$newc = (s.core(i) := s.core(i) \cup \{UCORE\}) \wedge$$
$$newk = (s.know(i) := s.know(i) \cup s.data_{regs}(UCORE)$$
$$\cup\ clt(s.data_{mem}, s.mem(i))) \wedge$$
$$s \sim_{func, core, know = NONE, newc, newk}\ s'. \tag{28}$$

Results: Since we have proved properties of memory isolation and data confidentiality in previous refinement levels, to guarantee the properties as well, we only need to individually prove that the guards and actions correctly refined in each event, *i.e.*, the theorem Eq. (4),(5) are satisfied in *S3*. Therefore, it is unnecessary to prove that each event satisfies the properties. Moreover, for many events in different functions are refined from the same functions, *e.g.*, StoE^2, our method also reduces the complexity of modeling the shielding systems.

We notice a potential security threat in using the function *terminate*. When TrustVisor copies the output prepared by an SCA to memory allocated for the OS, private data of the SCA may be in the output, if the function is not carefully used by the designers of SCAs. Therefore in the next steps, data confidentiality is violated because the private data in the output is added to the knowledge of the OS. Formally, in the step of copying the output of an SCA, TrustVisor copies the set of data stored in ls to ld.

$$\begin{aligned}
&\mathbf{Grd}_{\mathsf{Copy}^3_{trm}}\ i\ ld\ d\ ls\ s \overset{def}{=}\\
&\quad \textbf{(i)}\ s.func = Some(Trm(TRM_{copy}, prm_{trm}(i)))\wedge\\
&\quad \textbf{(ii)}\ ld \subseteq s.mem(OSID) \wedge \textbf{(iii)}\ ls \subseteq s.mem(i)\wedge\\
&\quad \textbf{(iv)}\ d = clt(s.data_{mem}, ls)\\
&\mathbf{Act}_{\mathsf{Copy}^3_{trm}}\ i\ ld\ d\ ls\ s\ s' \overset{def}{=}\\
&\quad newd = update_{mem}(s.data_{mem}, ld, d)\wedge\\
&\quad s \sim_{func, data_{mem} = Some(Trm(TRM_{lod}, prm_{trm}(i))), newd}\ s'.
\end{aligned} \tag{29}$$

The step is refined from SWrite^2_{mem}. Here private data of SCA i may be in d.

Then, in the step of switching the mode of the OS to executing mode, the set of data stored in memory allocated for the OS is added to the knowledge of the OS.

$$\begin{aligned}
&\mathbf{Grd}_{\mathsf{StoE}^3_{trm}}\ i\ s \overset{def}{=}\\
&\quad \textbf{(i)}\ s.func = Some(Trm(TRM_{StoE}, prm_{trm}(i)))\\
&\mathbf{Act}_{\mathsf{StoE}^3_{trm}}\ i\ s\ s' \overset{def}{=}\\
&\quad newc = (s.core(OSID) := s.core(OSID) \cup \{UCORE\})\wedge\\
&\quad newk = (s.know(OSID) := s.know(OSID)\cup\\
&\quad s.data_{regs}(UCORE) \cup clt(s.data_{mem}, s.mem(OSID)))\wedge\\
&\quad s \sim_{func, core, know = NONE, newc, newk}\ s'.
\end{aligned} \tag{30}$$

We briefly illustrate the reason why data confidentiality is violated in these steps. Since it can be proved that StoE^3_{trm} occurs only if Copy^3_{trm} have occurred, it implies that d in Copy^3_{trm} is a subset of $clt(s.data_{mem}, s.mem(OSID))$, *i.e.*, the set of data stored in memory allocated for the OS. Therefore, private data in d may be added to the knowledge of the OS.

In design of TrustVisor, it is suggested that SCAs encrypt private data of themselves in their outputs by calling hv_{seal}. To validate the suggestion, we replace the step Copy^3_{trm} with $\mathsf{CopyStrn}^3_{trm}$ in the formal model.

$$\begin{aligned}
&\mathbf{Grd}_{\mathsf{CopyStrn}^3_{trm}}\ i\ ld\ d\ ls\ k\ s \overset{def}{=}\\
&\quad \textbf{(i)}\ s.func = Some(Trm(TRM_{copy}, prm_{trm}(i)))\wedge\\
&\quad \textbf{(ii)}\ ld \subseteq s.mem(OSID) \wedge \textbf{(iii)}\ ls \subseteq s.mem(i)\wedge\\
&\quad \textbf{(iv)}\ d = clt(s.data_{mem}, ls) \wedge \textbf{(v)}\ s.symkey = Some(k)\wedge\\
&\quad \textbf{(vi)}\ (\forall d_1. d_1 \in d \wedge (parts(d_1) \cap pdata(i) \neq \phi) \rightarrow\\
&\quad \exists d_2. d_1 = Enc(k, Cons(d_2, Id(i))))\\
&\mathbf{Act}_{\mathsf{CopyStrn}^3_{trm}}\ i\ ld\ d\ ls\ k\ s\ s' \overset{def}{=}\\
&\quad newd = update_{mem}(s.data_{mem}, ld, d)\wedge\\
&\quad s \sim_{func, data_{mem} = Some(Trm(TRM_{lod}, prm_{trm}(i))), newd}\ s'.
\end{aligned} \tag{31}$$

Compared with Copy^3_{trm}, guard **(v)** and **(vi)** are added. The guard **(v)** states that TrustVisor has generated the encryption key used in hv_{seal}. The guard **(vi)** models the suggestion that if d_1 contains SCA i's private data, then d_1 should be the output of hv_{seal} called by SCA i. Finally, data confidentiality is proved to be preserved when the guards are added.

5.2 Case 2: OSP

OSP is a shielding system that aims to overcome the weakness of TrustZone-based approaches and hypervisor-based approaches in ensuring safe execution of security sensitive codes (SCCs), *i.e.*, SCAs, on mobile devices. TrustZone-based approaches bloat the TCB of the system as they must increase the code base size of the most privileged software. The most privileged software is used for supporting the execution of SCCs in the secure world. Hypervisor-based approaches incur performance overhead on mobile devices that are already suffering from resource restrictions. Therefore, OSP uses a hybrid approach that utilizes both TrustZone and a hypervisor to not only avoid executing SCCs in the secure world, but also mitigate performance overhead by activating the hypervisor. Specifically, OSP consists of the OSP hypervisor, which protects and manages the SCCs, and the OSP core, which controls and configures OSP overall. The OSP core resides in the secure world, while the OSP hypervisor is implemented in the normal world for an additional TEE for executing SCCs. Thus the TCB bloating of the secure world is suppressed. The OS in the normal world cannot access resources, *e.g.*, memory, used in the TEE. OSP deactivates its hypervisor when there is no SCC that is executing, and therefore, performance overhead in activating the hypervisor is mitigated.

We now explain OSP's functions related to our goals as follows.

1) *SCC_register*: An SCA is registered by the OS by calling *SCC_register*. When the OS calls *SCC_register*, OSP prepares the execution environment of a new SCA. OSP allocates memory for the SCA, initializes contexts, *i.e.*, data to be written to registers, for the SCA and resumes the OS. The allocated memory is reserved for all SCAs, and isolated from memory allocated for the OS.

2) *SCC_invoke*: After *SCC_register* for an SCA, the SCA can be invoked by the OS. When the OS calls *SCC_invoke*, OSP performs two phases of execution. In the first phase, a CPU core originally used by the OS is stopped and assigned to the selected SCA by OSP. OSP saves the context of the OS for the core, loads the context of the SCA, and executes the SCA. Besides, the OS produces the input for the SCA to the memory region specified by parameters of *SCC_invoke*. OSP copies the input to memory allocated for the SCA. If the SCC finishes its work, OSP performs the second phase. Specifically, the SCA turns into suspending mode, and the CPU core originally used by the SCA is reassigned to the OS. OSP saves the context of the SCA, loads the context of the

OS for the core and makes the OS use the core. Besides, the SCA produces the output for the OS to the memory region specified by OSP. Then, OSP copies the output to the memory allocated for the OS. Note that the OS may use several CPU cores simultaneously, and therefore when the OS calls *SCC_invoke*, the OS may not be turned into suspending mode.

3) *SCC_unregister*: OSP completely clears every relevant state of an SCA specified by parameters of *SCC_unregister*. When the OS calls *unregistration*, OSP deallocates all memory originally allocated for the SCA, clears data in the deallocated memory, and resumes the OS.

States and Events: The record for states for OSP is similar to S^3_{tv}, except that *symkey* is unused, since OSP does not provide hv_{seal} or hv_{unseal}. We also omit the modeling of events, for it is similar to the modeling of events in TrustVisor as well.

Results: We also discover a potential security threat in using the function *SCC_invoke*. The threat is quite similar to the one we noticed in TrustVisor. Formally, first, if the OS is in executing mode when OSP copies the output of the SCA, d is added to the knowledge of the OS.

$$\begin{aligned}
&\textbf{Grd}_{\mathsf{Copy2}^3_{ivk}}\ i\ idc\ ld\ d\ ls\ s \overset{def}{=}\\
&\quad \textbf{(i)}\ s.func = Some(Ivk(IVK_{copy2}, prm_{ivk}(i, idc)))\wedge\\
&\quad \textbf{(ii)}\ ld \subseteq s.mem(OSID) \wedge \textbf{(iii)}\ ls \subseteq s.mem(i)\wedge\\
&\quad \textbf{(iv)}\ d = clt(s.data_{mem}, ls)\\
&\textbf{Act}_{\mathsf{Copy2}^3_{ivk}}\ i\ idc\ ld\ d\ ls\ s\ s' \overset{def}{=}\\
&\quad newd = update_{mem}(s.data_{mem}, ld, d)\wedge
\end{aligned} \tag{32}$$

If $s.core(OSID) \neq \phi$
then $newk{=}(s.know(OSID){:=}s.know(OSID)\cup\ d)$
else $newk{=}s.know)\ \wedge\ s\ \sim\ func, data_{mem},\ know\ =\ Some(Ivk(IVK_{lod2}, prm_{ivk}(i, idc))),\ newd, newks'$.

Second, if the OS is in suspending mode when $\mathsf{Copy2}^3_{ivc}$ occurs, d is written to ld for $\mathsf{Copy2}^3_{ivc}$.

$$\begin{aligned}
&\textbf{Grd}_{\mathsf{RC2}^3_{ivk}}\ i\ idc\ s \overset{def}{=}\\
&\quad \textbf{(i)}\ s.func = Some(Ivk(IVK_{RC}, prm_{ivk}(i, idc)))\\
&\textbf{Act}_{\mathsf{RC2}^3_{ivk}}\ i\ idc\ s\ s' \overset{def}{=}\\
&\quad newc = (s.core(OSID) := s.core(OSID) \cup \{idc\})\wedge\\
&\quad (\mathsf{if}\ s.core(OSID) \neq \phi\ \mathsf{then}\\
&\quad newk = (s.know(OSID) := s.know(OSID)\cup\\
&\quad s.data_{regs}(idc))\\
&\quad \mathsf{else}\ \ newk = (s.know(OSID) := s.know(OSID)\cup\\
&\quad s.data_{regs}(idc) \cup clt(s.data_{mem}, s.mem(OSID))))\wedge\\
&\quad s \sim func, core, know = NONE, newc, newks'
\end{aligned} \tag{33}$$

It can be proved that d in $\mathsf{Copy2}^3_{ivk}$ is a subset of $clt(s.data_{mem},\ s.mem(OSID))$ in $\mathsf{RC2}^3_{ivk}$. Therefore, private data in d may be added to the knowledge of the OS.

In design of OSP, it is suggested that SCAs encrypt private data of themselves in their outputs. We assume that SCAs encrypt private data in the output by using their private symmetric keys. Formally, we replace the step $\mathsf{Copy2}^3_{ivk}$ with $\mathsf{Copy2Strn}^3_{ivk}$ in the formal model.

$$\begin{aligned}
&\textbf{Grd}_{\mathsf{Copy2Strn}^3_{ivk}}\ i\ idc\ ld\ d\ ls\ s \overset{def}{=}\\
&\quad \textbf{(i)}\ s.func = Some(Ivk(IVK_{copy}, prm_{ivk}(i, idc)))\wedge\\
&\quad \textbf{(ii)}\ ld \subseteq s.mem(OSID) \wedge \textbf{(iii)}\ ls \subseteq s.mem(i)\wedge\\
&\quad \textbf{(iv)}\ d = clt(s.data_{mem}, ls)\wedge\\
&\quad \textbf{(v)}\ (\forall d_1.d_1 \in d \wedge (parts(d) \cup s.pdata(i) \neq \phi) \rightarrow\\
&\quad \exists d_2.\exists k.d_1 = Enc(k, d_2)\wedge\\
&\quad k = invKey(k) \wedge Key(k) \in s.pdata(i)).\\
&\textbf{Act}_{\mathsf{Copy2Strn}^3_{ivk}}\ i\ idc\ ld\ d\ ls\ s\ s' \overset{def}{=}\\
&\quad newd = update_{mem}(s.data_{mem}, ld, d)\wedge\\
&\quad (\mathsf{if}\ s.core(OSID) \neq \phi\ \mathsf{then}\\
&\quad newk = (s.know(OSID) := s.know(OSID) \cup d)\\
&\quad \mathsf{else}\ \ newk = s.know)\wedge\\
&\quad s \sim func, data_{mem},\\
&\qquad know = Some(Ivk(IVK_{lod}, prm_{ivk}(i, idc))),\\
&\qquad newd, newks'.
\end{aligned} \tag{34}$$

Here, $\mathsf{Copy2Strn}^3_{ivk}$ refines SWrite^2_{mem} in *S2*. Compared with $\mathsf{Copy2}^3_{ivk}$, guard **(v)** is added. Guard **(v)** states that if d_1 contains SCA i's private data, then d_1 should be a ciphertext encrypted by a symmetric key that is privately owned by SCA i. Finally, we prove the validity of $\mathsf{Copy2Strn}^3_{ivk}$.

6 Related Work

6.1 Shielding Systems

The current shielding systems can be divided into two categories: (1) Systems using modern instructions in Intel or AMD chips. (2) Systems using TrustZone Technology.

Intel and AMD chips: Flicker [22] leverages Trusted Platform Module (TPM) and Intel TXT [17] or AMD SVM [30] to execute security sensitive code in isolation with the OS. Since it uses new features of processors, specially designed hardware or modifications for the OS are not needed in protecting applications, and it only requires that as few as 250 lines of additional code are trusted. TrustVisor [21] provides code integrity as well as data integrity and confidentiality for selected portions of an application. The goal is to leverage the features of modern processors to overcome the tradeoff between achieving a high level of security and high performance. It is achieved by implementing a software-based "micro-TPM" which attests the existence of isolated execution to an external entity.

InkTag [15] is proposed to directly address the Iago attacks [8] in systems that solely protect memory of applications from untrusted OS. It simplifies the design of hypervisor by forcing the untrusted operating system to participate in its own verification. Haven [5] enables users to run applications on cloud hosting services without having to trust the service provider. It protects confidentiality and integrity of the user's applications from the platform on which it runs (i.e., the cloud service provider's OS, VM and firmware). MiniBox [20] is the first two-way sandbox for x86 native code, which not only protects a benign OS from a misbehaving application, but also protects an application from a malicious OS. MiniBox can be applied in Platform-as-a-Service cloud computing to provide two-way protection between a customer application and the cloud platform OS.

TrustZone: Though the secure world of ARM TrustZone [1] is used for executing security critical applications, the increased number of the applications makes the size of the most privileged software in the secure world complex and therefore, vulnerable. Hence, shielding systems, such as OSP [11] and PrivateZone [18], are proposed for handling the problem. OSP [11] relies on a hybrid approach that utilizes both TrustZone and a hypervisor to implement an additional execution environment for securely executing applications.

This scheme, called on-demand hypervisor activation, has been efficiently and securely implemented by leveraging the memory protection capability of TrustZone. PrivateZone [18] is a framework to enable individual developers to utilize TrustZone resources. Using PrivateZone, developers can run Security Critical Logics (SCL) in a Private Execution Environment (PrEE). The advantage of PrivateZone is its leveraging of TrstZone resources without undermining the security of existing services in the TEE.

6.2 Refinement of Security Systems

The design of TAP [28] is motivated by the phenomenon that recent proposals for trusted hardware platforms, such as Intel SGX [23] and the MIT Sanctum processor, offer compelling security features but lack formal guarantees. It is proved that SGX and Sanctum are refinements of TAP under certain parameterizations of the adversary, demonstrating that these systems implement secure enclaves for the stated adversary models. Specifically, TAP satisfies three security properties that entail secure remote execution: integrity, confidentiality and secure measurement. TAP is currently limited to concurrent execution on a single-threaded single-core processor.

Klein *et al.* [19] present post-hoc verification of the seL4 microkernel from an abstract specification down to its C implementation. The functional correctness is verified that the implementation of seL4 always strictly follows the high-level abstract specification of kernel behavior. Here, the refinement is used to prove the conformance between formalizations at different levels.

Zhao *et al.* [33] propose a security model for information flow security in certification of separation kernels and a refinement framework on ARINC 653 compliant Separation Kernels (ARINC SKs). According to code-to-spec review, they find six security flaws in the ARINC 653 standard and three flaws in ARINC SK implementations.

Refinement and verifications on security protocols have been studied as well. Sprenger *et al.* [27] propose to verify security protocols by stepwise refinement. Their refinement strategy guides the transformation of abstract security goals into protocols that are secure when operating over an insecure channel controlled by a Dolev-Yao-style intruder. They have implemented their method in Isabelle/HOL and used it to develop different entity authentication and key transport protocols. Huang *et al.* [16] make fine-grained refinement on TPM-based security protocols on the application level. The purpose is to guide the design of TPM-based protocol applications, which are generally security-critical and error-prone in implementation. The framework introduces a modified Dolve-Yao adversary model, where the normal entities outside TPM may also perform malicious operations.

6.3 Verifications without Refinement

There are also researches on formally verifying security systems without refinement [31, 32]. Barthe *et al.* [3, 4] formalize in the Coq proof assistant an idealized model of a hypervisor, and formally establish that the hypervisor ensures strong isolation properties between the different operating systems, and guarantees that requests from guest operating systems are eventually attended. Sinha *et al.* [25, 26] formally verifies confidentiality of applications running on Intel SGX. The main concerns are vulnerabilities of divulging secrets in the application caused by incorrect use of SGX instructions or memory safety errors.

7 Conclusions

We develop a formal framework for analyzing security properties ensured by shielding systems. We analyze the property of memory isolation and data confidentiality, and propose four refinement steps for guiding verification of shielding systems. Potential security threats in using the systems are found.

One of our future work is to refine the shielding systems into fine-grained pseudo-code level with full verification, in which the soundness of specific goals in each shielding system is also proved. We also plan to extend our framework with more realistic hypotheses to support side-channel attacks [2, 9, 24] towards shielding systems.

Acknowledgments

The research is supported by National Natural Science Foundation of China under Grant No.61572453, No.61202404, No.61520106007, No.61170233, No.61232018, No.61572454, and Anhui Provincial Natural Science Foundation, No.1508085SQF215. We gratefully acknowledge the anonymous reviewers for their valuable comments.

References

[1] T. Alves, "Trustzone: Integrated hardware and software security," *White Paper*, 2004. (`https://www.researchgate.net/publication/244521018_Trustzone_Integrated_Hardware_and_Software_Security`)

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O. Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pp. 689–703, Nov. 2016.

[3] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "Formally verifying isolation and availability in an idealized model of virtualization," in *17th International Symposium on Formal Methods (FM'11)*, pp. 231–245, June 2011.

[4] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "Cache-leakage resilient OS isolation in an idealized model of virtualization," in *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pp. 186–197, June 2012.

[5] A. Baumann, M. Peinado, and G. C. Hunt,Shielding applications from an untrusted cloud with haven in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pp. 267–283, Oct. 2014.

[6] G. Bella, "Formal correctness of security protocols," *Information Security and Cryptography*, 2007. (`https://www.springer.com/gb/book/9783540681342`)

[7] E. Boiten and J. Abrial, "Modeling in event-b system and software engineering," *Journal of Functional Programming*, vol. 22, no. 2, pp. 217, 2012.

[8] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, pp. 253–264, Mar. 2013.

[9] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*, pp. 7–18, Apr. 2017.

[10] Y. Cheng, X. Ding, and R. H. Deng, "Efficient virtualization-based application protection against untrusted operating system," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security (ASIACCS'15)*, pp. 345–356, Apr. 2015.

[11] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *USENIX Annual Technical Conference (USENIX ATC'16)*, pp. 565–578, June 2016.

[12] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[13] G. Kahn G. Huet and C. Paulin-Mohring, *The Coq Proof Assistant: A Tutorial: Version 7.2*, RT-0256, 2002.

[14] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with ARM trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pp. 488–501, June 2017.

[15] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, pp. 265–278, Mar. 2013.

[16] W. Huang, Y. Xiong, X. Wang, F. Miao, C. Wu, X. Gong, and Q. Lu, "Fine-grained refinement on tpm-based protocol applications," *IEEE Trans. Information Forensics and Security*, vol. 8, no. 6, pp. 1013–1026, 2013.

[17] C. Intel, "Lagrande technology preliminary architecture specification," *Intel Corporation*, 2006. (`http://kib.kiev.ua/x86docs/SDMs/315168-002.pdf`)

[18] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable & Secure Computing*, no. 99, pp. 1–1, 2016.

[19] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2, 2014.

[20] Y. Li, J. M. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *USENIX Annual Technical Conference (USENIX ATC'14)*, pp. 409–420, June 2014.

[21] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *31st IEEE Symposium on Security and Privacy (S&P'10)*, pp. 143–158, May 2010.

[22] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the EuroSys Conference (EuroSys'08)*, pp. 315–328, Apr. 2008.

[23] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP'16)*, pp. 10, June 2016.

[24] M. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: eradicating controlled-channel attacks against enclave programs," in *24th Annual Network and Distributed System Security Symposium (NDSS'17)*, 2017. (`https://www.cc.gatech.edu/~slee3036/papers/shih:tsgx.pdf`)

[25] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*, pp. 665–681, June 2016.

[26] R. Sinha, S. K. Rajamani, S. A. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *22nd ACM Conference on Computer and Communications Security (CCS'15)*, pp. 1169–1184, June 2015.

[27] C. Sprenger and D. A. Basin, "Developing security protocols by refinement," in *17th ACM Conference on Computer and Communications Security (CCS'10)*, pp. 361–374, Oct. 2010.

[28] P. Subramanyan, R. Sinha, I. A. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *24th ACM Conference on Computer and Communications Security (CCS 2017)*, pp. 2435–2450, 2017.

[29] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. Martins, A. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[30] A. Virtualization, "Secure virtual machine architecture reference manual," *AMD Publication*, vol. 33047, 2005.

[31] C. Wang, Y. Xiong, W. Cheng, W. Huang, H. Xia, and J. Huang, "A general formal framework of analyzing selective disclosure attribute-based credential systems," *International Journal of Network Security*, vol. 19, no. 5, pp. 794–803, 2017.

[32] M. Wu, J. Chen, and R. Wang, "An enhanced anonymous password-based authenticated key agreement scheme with formal proof," *International Journal of Network Security*, vol. 19, no. 5, pp. 785–793, 2017.

[33] Y. Zhao, D. Sanan, F. Zhang, and Y. Liu, "Refinement-based specification and security analysis of separation kernels," *IEEE Transactions on Dependable & Secure Computing*, no. 99, pp. 1–1, 2017.

Biography

Jiabin Zhu is a Ph.D. candidate in school of Computer Science and Technology, University of Science and Technology of China. His current research interests formal methods and information security.

Wenchao Huang received the B.S. and Ph.D degrees in computer science from University of Science and Technology of China in 2005 and 2011, respectively. He is currently an associate professor in School of Computer Science and Technology, University of Science and Technology of China. His current research interests include mobile computing, information security, trusted computing and formal methods.

Fuyou Miao received his Ph.D of computer science from University of Science and Technology of China in 2003. He is an associate professor in the School of Computer Science and Technology, University of Science and Technology of China. His research interests include applied cryptography, trusted computing and mobile computing.

Cheng Su is a Ph.D. candidate in school of Computer Science and Technology, University of Science and Technology of China. His current research interests formal methods and information security.

Baohua Zhao received His M.S. Degree from Beijing University of Technology in 2016. He is a director in Computing Technology and Applications Research Institute, Global Energy Interconnection Research Institute. His main research interests include trusted computing, information security and computer technology.

Yan Xiong received the B.S., M.S., and Ph.D degrees from University of Science and Technology of China in 1983, 1986 and 1990 respectively. He is a professor in School of Computer Science and Technology, University of Science and Technology of China. His main research interests include distributed processing, mobile computing, computer network and information security.