MIPE: a practical memory integrity protection method in a trusted execution environment

Rui Chang, Liehui Jiang, Wenzhi Chen, Yang Xiang, Yuxia Cheng & Abdulhameed Alelaiwi

Cluster Computing
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-017-0833-4

Springer

Cluster Comput
DOI 10.1007/s10586-017-0833-4

MIPE: a practical memory integrity protection method in a trusted execution environment

Rui Chang[1] · Liehui Jiang[1] · Wenzhi Chen[2] · Yang Xiang[3] · Yuxia Cheng[4] · Abdulhameed Alelaiwi[5]

Received: 25 January 2017 / Accepted: 14 March 2017

Abstract With the rapid development of Internet of Things technology and the promotion of embedded devices' computation performance, smart devices are probably open to security threats and attacks while connecting with rich and novel Internet. Attracting lots of attention in embedded system security community recently, Trusted Execution Environment (TEE), allows for the execution of arbitrary code within environments completely isolated from the rest of a system. However, existing memory protection methods in a TEE are inadequate. In general, the software-based formal methods are not practical and the hardware-based implementation approaches lack of theoretical proof. To address the memory isolation and protection problems in TEE, in this paper, we propose a practical memory integrity protection method on an ARM-based platform, called MIPE, to defend against security threats including kernel data attacks and direct memory access attacks. MIPE utilizes TrustZone technique to create a isolated execution environment, which can protect the sensitive code and data against attacks. To present the integrity protection strategies, we provide the design of MIPE using B method, which is a practical formal method. We also implement MIPE on the Xilinx Zynq ZC702 evaluation board. The evaluation results show that the automatic proof rate of machines using B method is about 78.32%, and the proposed method is effective and feasible in terms of both load time and overhead.

Keywords TrustZone · B method · Threat tree model · Trusted execution environment · Memory integrity protection

✉ Rui Chang
crix1021@163.com

Liehui Jiang
jiangliehui@163.com

Wenzhi Chen
chenwz@zju.edu.cn

Yang Xiang
yang.xiang@deakin.edu.au

Yuxia Cheng
yuxia.cheng@deakin.edu.au

Abdulhameed Alelaiwi
aalelaiwi@KSU.EDU.SA

1 State Key Laboratory of Mathematic Engineering and Advanced Computing, Zhengzhou, China

2 Department of Computer, Zhejiang University, Hangzhou, China

3 Centre for Cyber Security Research, Deakin University, Burwood, VIC, Australia

4 School of Information Technology, Deakin University, Burwood, VIC, Australia

5 King Saud University, Riyadh 11543, Saudi Arabia

1 Introduction

With the rapid development of Internet of Things (IoT) technology and the promotion of embedded devices' computation performance [1], smart devices are probably open to security threats and attacks while connecting with rich and novel Internet [2]. Attracting lots of attention in embedded security community recently, Trusted Execution Environment (TEE) [3], allows for the execution of arbitrary code within environments completely isolated from the rest of a system.

TEE is the secure isolation execution environment supported by different technologies. Trusted Platform Module (TPM) focused on security key and encrypted objective [4]. However, it cannot defense against run-time attacks, which might lead to severe vulnerabilities in a system [5]. Kernel vulnerabilities detection scheme based on secure coprocessor

Published online: 25 March 2017

has already been proposed [6], but it only supplies isolation execution environment with a lack of controlling capability on system resources, such as memory and other exterior equipments. Intel Software Guard Extensions (SGX) has been the newest security technology of Intel since 2013 [7], but it is an open question how to utilize it on embedded platforms [8]. Virtual Machine Introspection (VMI) [9] runs a suspicious OS and the monitor programs on two VMs, respectively. However, due to the large size of hypervisor, it may contain a number of vulnerabilities that may be explored by malware to threaten the hypervisor.

Most recently, new capabilities of modern trusted hardware technologies allow for the execution of arbitrary code within environments completely isolated from the rest of the system [10–12]. The objective of trusted hardware technologies is to provide strong guarantees which ensure that adversaries cannot tamper with the execution of sensitive data. Due to the large code size and complexity of Operating System (OS) kernel, malicious codes can exploit known and unknown kernel vulnerabilities to threaten the OS and steal sensitive data from memory. However, existing memory protection methods in a TEE are inadequate. In general, the software-based formal methods are not practical [13,14], and the hardware-based implementation approaches lack of theoretical proof.

Motivated by the above research status, it is practically necessary to develop a memory integrity protection method that can establish the trustworthiness of embedded devices. In this paper, we first give the formal description of threat tree model, and construct a threat model based on "AND/OR" tree (Sect. 3). On the basis of threat tree model, then we provide the design of MIPE using B method, including initial specifications and refinement (Sect. 4). Furthermore, we implement MIPE on the Xilinx Zynq ZC702 evaluation board (Sect. 5). Our MIPE utilizes TrustZone technique to create a isolated execution environment, which can protect the sensitive code and data against attacks. We present the security analysis that MIPE can defend against security threats including kernel data attacks and direct memory access attacks (Sect. 6). The evaluation results show that the automatic proof rate of machines using B method is about 78.32%, and the proposed method is effective and feasible in terms of both load time and overhead (Sect. 7).

The main contributions of this paper are as follows:

(1) We propose a practical memory integrity protection method on an ARM-based platform, called MIPE, to defend against security threats including kernel data attacks and direct memory access attacks.
(2) To present the integrity protection strategies, we provide the design of MIPE using B method, which is a practical formal method.
(3) We implement MIPE on the Xilinx Zynq ZC702 evaluation board. The evaluation results show that the automatic proof rate of machines using B method is accepted and MIPE has small overhead.

2 Overview of trusted execution environment

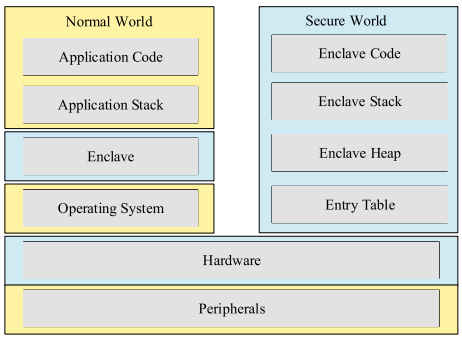
Trusted Execution Environment is a hotspot in current embedded system security research. TEE provides an isolation security execution environment, where the codes and data are of confidentiality and integrity. The secure characteritics include isolation execution, execution files integrity, run-time codes integrity, control flows integrity, etc.

TEE supplies more secure execution environment than general-purposed OS, and more functions than Secure Element (SE, e.g., smart card and SIM card). The devices supporting TEE contain two execution environments (trusted and untrusted) that are physically separated. On mobile devices, TEE and mobile OS exist in parallel. They supply secure functions for abundant mobile environments.

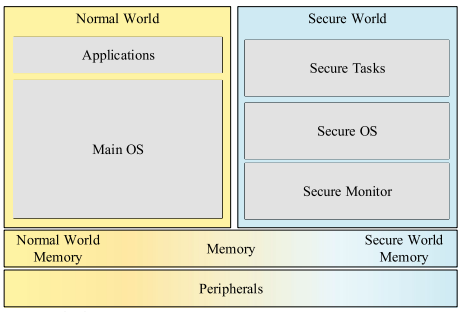
The research on key technologies of operation system support for TEE focuses on TPM [15], Intel SGX [16], and ARM TrustZone [17]. TPM focused on security key, and anything untrusted didn't know the key. Thus, anything encrypted by the key was considered secure. However, it cannot supply better support for flexible memory protection mechanism. SGX and TrustZone respectively adopted different support for memory protection mechanism in TEE.

Intel SGX adds 18 instructions to extend Intel Instruction Set Architecture (ISA) for software security [18]. It helps to define secure regions of code and data that maintain confidentiality even when an attacker has physical control of the platform and can conduct direct attacks on OS, VMM and memory. As shown in Fig. 1a, the trusted area is formed by the memory enclave, the rest belongs to the untrusted area. Peripherals are only accessed from the untrusted area since SGX does not extend to the system bus. The memory is separated into two domains and the secure domain is fixed.

ARM TrustZone is a hardware-based security extension technology, which is a hardware isolation mechanism to improve software security. It enables a single physical processor to execute codes in one of two possible operating worlds: Normal World (NW) and Secure World (SW). Accordingly, the memory is separated into two domains to run the dedicated OS and software. As shown in Fig. 1b, the secure memory is not fixed but strictly controlled in the isolation mechanism. The processor only runs in one world at a time, and running in the other world requires context switch. To date, TrustZone has been popularized and applied by many mainstream mobile manufacturers to achieve secure applications [19].



(a) The architecture of SGX



(b) The architecture of TrustZone

Fig. 1 The dividing of secure world and normal world

3 Threat model

Security threats are potential events that probably lead to unexpected results (e.g., Information Leakage and Denial of Service) [20]. Threat model represents the possible threats for a special system, describing the ways that the attackers probably launch potential threats or attacks.

Researchers have proposed several abstract representations for threat modelling, which are divided into three categories. Based on tree structure, they are fault tree, threaten tree, and attack tree. Based on the network structure, they are Petri nets [23], Generalized stochastic Petri nets [24], and aspect-oriented Petri nets [25]. Based on diagram structure, they are attack diagram [26] and UML sequence diagram [27].

3.1 Threat tree model

From an attacker's point of view, the attacked system consists of several targets of threats. Each target has vulnerability, and each successful attack to the vulnerability will probably do harm to the system. Threat model describes the decision-making process where the components in the attacked system go through. Threat model reflects attack methods which the attackers select, so the process of constructing threat model is the analysis process of possible potential threats.

We construct the threat model by "AND/OR" tree. Memory resource is taken as target of threats, and it is the root node of the tree. Root node is decomposed by several sub-objectives. Whether the type of father node is "AND" or "OR" depends on the logical relations of sub-objectives. Each of sub-objectives is decomposed according to the above iteration process until it is a leaf node. Such a way would finish the threat model construction.

Table 1 Notations and descriptions in TTM

Notations	Descriptions
Tr	The threat tree
N	A set of nodes
R	A set of relations
Pa	A set of attack paths
N^{root}	The root node of the threat tree
N^{leaf}	A set of leaf nodes
N^{and}	A set of "AND" nodes

Definition 1 The formal description of TTM is a seven-element set shown as follows: $\langle$ Tr, N, R, Pa, N^{root}, N^{leaf}, N^{and} $\rangle$. The notations and description in TTM are shown in Table 1.

n, n_{temp}, N_i, N_j are nodes of set N. R_{ij} is the element of set R, and R_{ij}=(N_i, N_j). n.Children is a set of children nodes. n.nodeType is the type of a node, whose value is either "AND" or "OR". VN is a set of visited nodes. QN is a queue of nodes. Pa_0 is a set of all the attack paths accessing to the root node of the threat tree.

Assumptions We assume an ARM-based architecture that implements the TrustZone extensions. We also assumes that it runs as a part of the secure world, while the target OS runs in the normal world. We also assumes that the whole system is loaded securely, including both the secure and the normal worlds. This process is straightforward using trusted boot. Intuitively, trusted boot only guarantees the integrity of the kernel during the boot-up process. It cannot guarantee the integrity of the kernel after the system runs and starts to interact with potential attackers.

There are three key algorithms for threat model construction as follows.

3.1.1 Threat tree construction algorithm

We construct the threat model by "AND/OR" tree. Root node is decomposed by several sub-objectives, which depend on the constitution of attacking means achieving the goal. Whether the type of father node is "AND" or "OR" depends on the logical relations of sub-objectives. A non-leaf node is an "AND" node if and only if it is implemented after each of its child nodes is implemented. The logical relations between any child nodes of "AND" node are "AND". It is worthwhile

to note that the default implementation order of its child nodes is from left to right. A non-leaf node is an "OR" node if and only if it is implemented when any of its child nodes is implemented. The logical relations between any child nodes of "OR" node are "OR".

Above procedures are iterated to implement the stepwise refinement of attacking means until the attack is indivisible (i.e., it is a leaf node). While the iteration process is over, the top-down construction is finished.

Algorithm 1: ThreatTreeConstruction(N, R)

Input: a set of nodes in the threat tree, N; a set of relations in the threat tree, R
Output: a threat tree, Tr

```
1  Tr = null;
2  if N^root = null then
3      Return;
4  Tr.root = N^root;
5  QN.enqueue(N^root);          /* Enqueue operation */
6  while QN ≠ null do
7      n = QN.dequeue();        /* Dequeue operation */
8      if n ∉ N^leaf then
9          if n ∈ N^and then
10             n.nodeType = "AND";
11         else
12             n.nodeType = "OR";
13         for each relation R_ij ∈ Tr.R do
14             if n = R_ij.N_i then
15                 n.Children.add(N_j);
16                 QN.enqueue(N_j);
17 Return Tr;
```

3.1.2 Accessibility checking algorithm

We analyze accessibility of nodes and check whether the circles exist in threat tree model. Based on breadth-first traversal way, each of the nodes is accessed in layers. The accessed nodes are added in an accessed set, and the accessed set is checked whether it is equated with constructed threat tree. If there exists a single node not in the accessed set, there is an unaccessible node in the model, and vice versa.

3.1.3 Attack paths searching algorithm

Attack paths searching algorithm is basic to threat-perceived based memory protection technology. Different attacking means constitute several attack paths, and each attack path represents one of potential attacking means in order to damage the security. The following algorithm searches all the attack paths accessing to the target.

Algorithm 2: AccessibilityChecking(Tr)

Input: a threat tree, Tr
Output: Boolean accessible

```
1  Accessible=true;
2  VN=null; QN=null; n_temp=null;
3  if N^root = null then
4      Return;
5  else
6      VN.add(Tr.root)
7  for each relation R_ij ∈ Tr.R do
8      if Tr.root = R_ij.N_i then
9          QN.enqueue(N_j);
10 while QN ≠ null do
11     n_temp = QN.dequeue();
12     if n_temp ∉ VN then
13         VN.add(n_temp);
14         for each relation R_ij ∈ Tr.R do
15             if n_temp = R_ij.N_i then
16                 QN.enqueue(N_j);
17 for each node n ∈ Tr.N do
18     if n ∉ VN then
19         Accessible=false;
20 Return Accessible;
```

Algorithm 3: AttackPathsSearching(Tr)

Input: a threat tree, Tr
Output: a set of all the attack paths accessing to the root node of the threat tree, Pa_0

```
1  Pa = null; Pa_0 = null;
2  for j from Tr.N.size()-1 to 0 do
3      N_j = Tr.N.get(j);        /* get nodes reversely */
4      if N_j ∈ n_leaf then
5          Pa_j = N_j;
6      else
7          if N_j ∈ n_and then
8              get all attack paths of N_j s child nodes and the number
               of them;
9              sum of the number is n;
10             for l from 0 to n-1 do
11                 Paj.add(Paj[l]);
12         else
13             get the number of all attack paths of N_j s child nodes;
14             sum of the number is n;
15             for i from l to k do
16                 for t from l to N_i do
17                     Pa_j.add(Pa_j[t]);
18 Return Pa_0;
```

4 Design of MIPE using B method

The B method is a formal method enabling the development of secure programs. It uses concepts of first order

logic, set theory and integer arithmetics to specify abstract state machines that represent software behaviour. The security model can be verified using proof obligations to ensure its consistency. B method provides a refinement mechanism.

4.1 Initial specifications

The abstract descriptions of the specifications were refined to the **IMPLEMENTATION** descriptions by converting the nondeterministic sections to sequential processing. Initial specifications are shown in Fig. 2. Variables and descriptions are shown in Table 2.

4.2 Refinement

The **MACHINE** $Secure\_Memory(Smem)$ provides several operations respectively. They are shown as follows:

- *MemPut*, free memory blocks.
- *MemGet*, allocate memory blocks.
- *MemCreate*, create memory blocks.
- *MemQuery*, query the state of memory partition.
- *MemDelete*, delete the memory partition.
- .....

The **MACHINE** $Normal\_Memory(Nmem)$ also provides the operations above. B method covers software development process from an abstract specification to an implementation through successive refinement steps. Taking the precise mathematics semantics as the foundation, B method supports rigorous development process. Take *MemGet* as an example. The operations of *MemGet* is shown in Fig. 3, and the refinement of $Secure\_Memory(Smem)$ is shown in Fig. 4.

As shown above, the concrete model phase consists in completing the abstract model to get to a completely implementable B project. The only input of this phase is the abstract model and the goal is to implement it completely through refinement and importation breakdown. When the concrete model is fully proved, we are sure that the concrete model complies with the abstract model. The specifications of memory integrity protection in the proposed architecture are described in B.

5 Implementation of MIPE

In order to support TEE, a device needs to define a security perimeter separated by hardware from the main OS and applications, where only trusted code executes. We show TrustZone-enabled TEE architecture in Fig. 5. SW is represented on the right side of the figure (blue), where trusted components execute in TEE. All components outside the trusted area form the untrusted area called NW, where OS and applications execute in REE. The untrusted area is represented on the left side of the figure (yellow). Peripherals connected to the system bus belong to either of the two areas, or both of them. TrustZone relies on the so-called NS bit, an extension of the AMBA3 AXI system bus to separate the execution between SW and NW. A secure monitor mode and the trusted enclave domain in address space of OS kernel, which ensure the secure memory that cannot be tampered by the untrusted OS kernel, control the switch and migration between the two worlds. The role of the monitor mode software in a design is to provide a robust gatekeeper which manages the switches between the Secure and Non-secure processor states.

5.1 Context switching

TrustZone-based memory protection mechanism ensures that state of the world that the processor is leaving is safely saved, and the state of the world the processor is switching to is correctly restored. Normal world entry to monitor mode is tightly controlled. It is only possible via the following exceptions: an interrupt, an external abort, or an explicit call via an SMC instruction. The Secure world entry to the monitor mode is a little more flexible, and can be achieved by directly writing to CPSR, in addition to the exception mechanisms available to the Normal world.

The primary role of the monitor is to context switch resources that are needed in both worlds. Any secure state saved by the monitor should be saved into a region of Secure memory, so that the Normal world cannot tamper with it. Exactly what needs to be saved and restored for each switch depends on the memory management mechanism, and the software model used for inter-world communications. Figure 6 shows an example of switching signals by TrustZone protection controller (TZPC). The TZPC is configured as always Secure, the Timers and Real-Time Clock (RTC) as always Non-secure, and the Keyboard and Mouse Interface (KMI) has a programmable security state under software control. The TrustZone Memory Adapter (TZMA) enables a design to secure a region within an on-SoC static memory such as a ROM or a SRAM. Secure world software can program the TZPC at run-time to change the signal input to the AXI-to-APB bridge to switch the KMI from Secure to Non-secure or visa versa. As this figure also shows that the addition of the TZPC allows other signals on the SoC to be controlled dynamically.

An SMC function identifier once issued must never be reused. Additional SMC calls must take a new unused SMC identifier. Calls to removed SMC identifiers must return the Unknown SMC Function Identifier value. Incompatible argu-

MACHINE
$Secure\_Memory(Smem)$
CONSTRAINTS
$Smem <:$**NAT**
SETS
$Smid$
$STATE\{SW, NW, Monitor\}$
VARIABLES
S_used, $S\_allocated$, S_addr,
S_size, S_blocks, S_free, $state$
INVARIANT
$state = SW, Monitor \wedge$
$S\_used \in Smem \wedge$
$S\_allocated \in Smem \wedge$
$S\_used \in S\_allocated \wedge$
$S\_free <= S\_blocks$
OPERATIONS
$MemPut(id, addr1, addr2)$
$MemGet(id, addr1, addr2)$
$MemCreate(id, addr1, addr2)$
$MemQuery(id, addr1, addr2)$
$MemDelete(id, addr1, addr2)$
.....
END
END

(a) Secure memory

MACHINE
$Normal\_Memory(Nmem)$
CONSTRAINTS
$Nmem <:$**NAT**
SETS
$Nmid$
$STATE\{SW, NW, Monitor\}$
VARIABLES
N_used, $N\_allocated$, N_addr,
N_size, N_blocks, N_free, $state$
INVARIANT
$state = NW \wedge$
$N\_used \in Nmem \wedge$
$N\_allocated \in Nmem \wedge$
$N\_used \in N\_allocated \wedge$
$N\_free <= N\_blocks$
OPERATIONS
$MemPut(id, addr1, addr2)$
$MemGet(id, addr1, addr2)$
$MemCreate(id, addr1, addr2)$
$MemQuery(id, addr1, addr2)$
$MemDelete(id, addr1, addr2)$
.....
END
END

(b) Normal memory

MACHINE
$SYSTEM$
CONSTRAINTS
$mem <:$**NAT**
SETS
$STATE\{SW, NW, Monitor\}$
VARIABLES
$state$, $access$, mem, mem_used,
$mem\_allocated$, mem_addr, mem_size,
mem_blocks, mem_free
INVARIANT
$state \in STATE \wedge$
$access \in$ **BOOL** $\wedge$
$mem\_used \in mem \wedge$
$mem\_allocated \in mem \wedge$
$mem\_used \in mem\_allocated$
$mem\_free <= mem\_blocks$
OPERATIONS
.....
END
END

(c) System memory

MACHINE
S_data
SETS
$STATE1\{SW, NW, Monitor\}$
$STATE2\{SW, NW\}$
VARIABLES
$autho$, $state$, $attri$, $origi$
INVARIANT
$autho \in$ **BOOL** $\wedge$
$state \in STATE1 \wedge$
$attri \in STATE2 \wedge$
$origi \in STATE2 \wedge$
$(attri = NW \wedge state = NW) \Rightarrow autho =$ true
$(attri = SW \wedge state = NW) \Rightarrow autho =$ false
$(attri = SW \wedge state = \{SW, Monitor\}) \rightarrow autho =$ true
OPERATIONS
.....
END
END

(d) Secure data

Fig. 2 Initial specifications

ment changes cannot be made to an existing SMC call, a new call is required. Table 3 shows the recommended allocation of SMC identifier value ranges for different entities and purposes. The owner of a range is the entity who is responsible for that function in a specific SoC.

5.2 Virtual memory divide and exception handling

In general, the statuses of memory protection mechanism are initialization, allocation, checking, exception handling, and release. These statuses are described in Fig. 7.

Table 2 Variables and descriptions

Variables	Descriptions	Variables	Descriptions
S_used	Used memory blocks in SW	N_used	Used memory blocks in NW
S_allocated	Allocated memory blocks in SW	N_allocated	Allocated memory blocks in NW
S_addr	Starting address of memory partition in SW	N_addr	Starting address of memory partition in NW
S_size	The size of memory blocks in SW	N_size	The size of memory blocks in NW
S_blocks	The number of memory blocks in SW	N_blocks	The number of memory blocks in NW
S_free	Free memory blocks in SW	N_free	Free memory blocks in NW
State	The state of system	Autho	Access authority
Attri	The secure attribute of data	Origi	The initial state of data

OPERATIONS
$MemGet(id, addr1, addr2) =$
  PRE
    $id : mid \wedge addr1 < addr2 \wedge$
    $S\_free(id) < S\_blocks(id) \wedge$
    $addr1...addr2 <: S\_used \wedge$
    $addr2 - addr1 + 1 = S\_blocks(id)$
  IF
    $(attri = NW \wedge state = NW) \vee$
    $(attri = SW \wedge state = \{SW, Monitor\})$
  THEN
    $mem\_free(id) := mem\_free(id) - 1 \parallel$
    $mem\_used := mem\_used + (addr1...addr2)$
    .....
  END
END

Fig. 3 The operations of *MemGet*

IMPLEMENTATION
  $Secure\_Memory(Smem)$
REFINES
  $Secure\_Memory$
VALUES
  S_used, $S\_allocated$, S_addr,
  S_size, S_blocks, S_free, $state$
INVARIANT
  $state = NW \wedge$
  $N\_used \in Nmem \wedge$
  $N\_allocated \in Nmem \wedge$
  $N\_used \in N\_allocated \wedge$
  $N\_free <= N\_blocks$
OPERATIONS
  $MemGet(id, addr1, addr2) =$
    IF
      $attri = SW \wedge$
      $state = \{SW, Monitor\}) \wedge$
      $autho = \text{true}$
    THEN
      $S\_free(id) := S\_free(id) - 1 \parallel$
      $S\_used := S\_used + (addr1...addr2)$
      .....
  END
END

Fig. 4 The refinement of $Secure\_Memory(Smem)$

Only user mode is non-privileged mode for application process running, while other privileged modes are used to handle external interrupts, system calls and CPU exceptions, respectively. The entry instruction of the privileged mode is located at the system exception vector table, whose address is determined by the V bit of system control register and vector base address register. With TrustZone supporting, the two worlds have different page table base registers and vector vase registers. This brings independent virtual memory mapping and exception handling, and adds a new CPU privileged mode as well. Monitor mode is in charge of switching one world to the other. Owing to independent exception vector table, monitor mode uses the address space mapping of secure world, which is designated by MVBAR (Monitor Vector Base Address Register Specify). The dividing of the two worlds is showed in Fig. 8.

5.3 Integrity protection strategies

The TrustZone-enabled isolation protection mechanism is applied to protect memory mappings. In the case of an unreliable operating system kernel, the codes cannot be executed

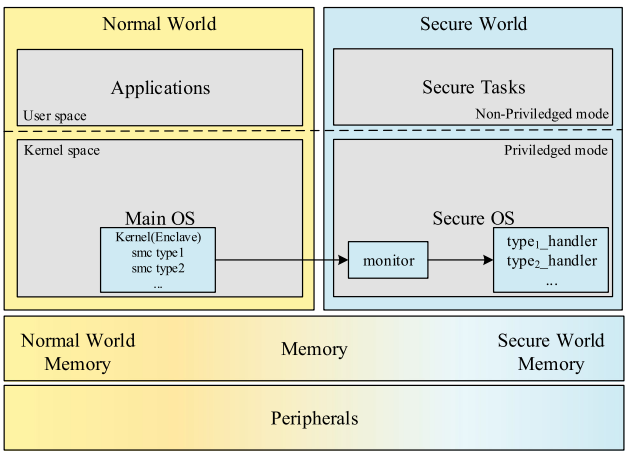


Fig. 5 The architecture of TrustZone-enabled device

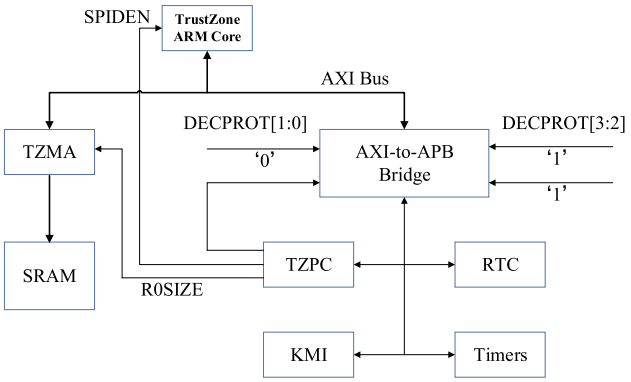


Fig. 6 An example of switching signals by TrustZone protection controller

in the target device, unless they are authorized by the integrity verification program. According to different types of attacks, we construct multilevel protection strategies.

First, based on integrity protection strategy of executable files and link library files, malicious processes are prevented from running. Monitor points are set up at the entry and return of system calls (e.g., exception handler). Creating process, loading executable files, and reading shared libraries are intercepted. Such a way could lead to that the initial status of the created process would be integrity.

Table 3 SMC identifier ranges

SMC function identifier	Reserved use and sub-range ownership
0x00000000-0x0100FFFF	Reserved for existing APIs
0x02000000-0x1FFFFFFF	General trusted OS
0x20000000-0x7FFFFFFF	Reserved
0x8000000-0x8000FEFF	ARM service calls
0x81000000-0x8100FEFF	CPU service calls
0x82000000-0x8200FEFF	SiP service calls
0x83000000-0x8300FEFF	OEM service calls
0x84000000-0x8400001F	PSCI SMC32 bit calls
0x84000020-0x8400FEFF	Standard service calls
0x8400FF00	Standard service call count
0x8400FF01	Standard service call UID
0x8400FF02	Reserved
0x8400FF03	Revision details
0x8400FF04-0x8400FFFF	Reserved
0xB0000000-0xB100FFFF	Trusted application calls
0xB2000000-0xBF00FEFF	Trusted OS calls
0xBF00FF00	Trusted OS calls count
0xBF00FF01	Trusted OS calls UID
0xBF00FF02	Reserved
0xBF00FF03	Revision details
0xBF00FF04-0xFF00FFFF	Reserved

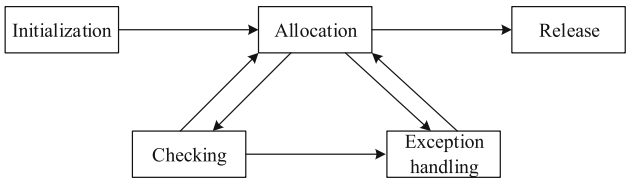


Fig. 7 The status of memory protection extraction

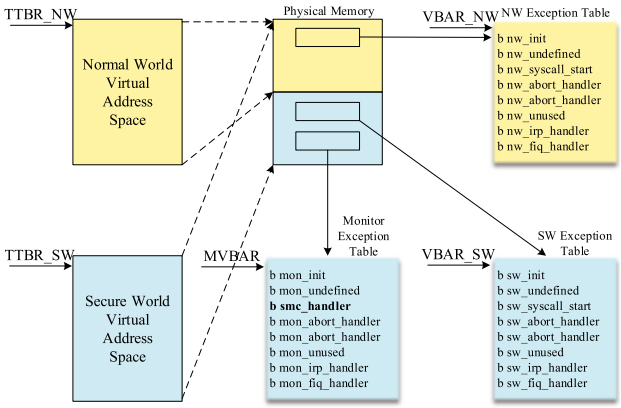


Fig. 8 Dividing of virtual memory and exception handler

Second, based on integrity protection strategy of run-time codes in memory, injection attacks for run-time codes are prevented. With WXN (Write eXecute Never) protection mechanism, the writable and executable permissions are mutually exclusive, so it's feasible to distinguish the executable codes from stack data in memory. Such a way could prevent above-mentioned injection attacks with hardware support. In other words, SW just keeps WXN-enabled mechanism in the security strategy. Additionally, the periodical measurement on the process code segments could enhance the strategy as well.

Third, based on control flow integrity protection strategy, control flow hijack attacks (e.g., code reuse) are prevented. We explore TrustZone-based isolation technology to protect run-time flow integrity, which is on the basis of the shadow stack method. The following security strategies are basic requirements. Each function call must match a return, and the destination address of function call must be included in the white list. Furthermore, the destination address of returned instruction must be consistent with return address of shadow stack. For the executable files, whose load base addresses are fixed, the white list of function's jump addresses can be accessed from the central control server directly. For the relocatable dynamic link library files, the actual addresses is got by adding the loading base addresses and the relative addresses of white list.

5.4 Integrity verification method

Goals of the integrity verification are monitoring sensitive instructions and protecting memory. The former is to ensure

the integrity of system when its status is changed. The latter is to guarantee that the integrity verification is always under control of corresponding program. When a page fault occurs, NW creates a mapping between virtual address and physical address, and sends SMC instruction to trap into SW. SW checks previous status of physical page to judge whether the address is mapped to the virtual address. Such a way would prevent some special attacks, e.g., Man-in-the-Middle.

To prevent the attacks from kernel, the integrity verification program can be built in the monitor, and it should implement the real-time intervention of system calls. The intervened instructions include system call, hardware interrupt (INT 0x10), and SMC instruction, which is used to switch status of system between NW and SW.

The following procedures should be implemented. The return address of system call is replaced by the monitor module, and an illegal address is created by the system call to protect interruption. Then, instructions trapped into monitor module are executed. The monitor module checks codes and returns the address of quondam system call after integrity is ensured. All registers' status (including CR3 page table register, IP, SP, etc.), the stack status of kernel in clients, and the system call number in EAX, need to be checked.

Meanwhile, in order to ensure the integrity of executable files, the following two situations must be assured. The one is that the system executes with integrity and none of user processes is modified. The other is that any modification of the user processes is first perceived by the integrity verification program.

NX (Non-execute) bit of page is enabled by the integrity verification program, viz. it sends signals to OS whether the instructions of the page can be executed. Once executable codes in user-code pages are protected by NX, it will trap into the monitor module immediately. If a malicious OS modifies the initial address of user segment when the user page table changes, it must trap into the monitor module as well. The reason is that the system call is triggered in such a process. By comparing the addresses of user segment, the integrity is verified successfully. After checking the integrity of user process, the full page is set as readable and executable by integrity verification program. If there is an attack attempting to modify this page, it will still trap into the monitor module to verify the integrity. Such verification cycles ensure the integrity.

6 Security analysis

Throughout Sect. 5, we discussed the security guarantee provided by Enclave and Monitor in SW. In this section, we summarize these guarantees by discussing how they defeat security threats. We categorize security threats by the attack surface they target.

6.1 Kernel attack surface

The first threat to the kernel is to run a modified binary during system load-time. As mentioned in Sect. 3, TrustZone-enabled devices assumes the presence of trusted boot to defeat that threat. It is the only guarantee that the loaded kernel binary is instrumented to remove control instructions. Once that binary is loaded, the memory protection will guarantee that it cannot be modified. The security guarantees provided by both control instruction emulation and memory protection are non-bypassable because there is nowhere in the normal world that can execute the emulated instructions or modify the translation tables. Modern kernels usually support extending their code using loadable modules. These modules are not verified by trusted boot because they are loaded after the system starts. Our scheme supports these kernel code extensions as long as they are known to the system and they are subject to instrumentation to remove control instructions from their binaries. We rely on an orthogonal system to verify the binaries of loadable kernel modules before they are loaded into memory. This system restricts the modules to be loaded a predefined set of modules.

6.2 Kernel data attacks

As mentioned in Sect. 5.3, TrustZone prevents direct access of kernel data from user space. User space will have the PXN access restriction so it can not escalate its privilege to access the privileged kernel data. Moreover, our scheme will prevent kernel data from being double mapped to user space. For this particular security guarantee, our scheme uses the kernel to get information about the memory layout. This is acceptable because this type of attacks can only originate from the user space. The philosophy behind using TrustZone to provide this protection is to avoid vulnerabilities that usually exist in the huge kernel code base. Such vulnerabilities have been previously exploited to give unauthorized access to kernel data [21].

6.3 Direct memory access attacks

Hardware peripherals are sometimes allowed to bypass the MMU and do a Direct Memory Access (DMA) to the physical memory. Attackers may use an exploit to trick the kernel into allowing these devices to directly access its memory. DMA attacks are not a threat to our scheme. The secure world is inherently secure against DMA using the TrustZone protection mechanism. DMA attacks that aim at modifying the kernel binary or the translation tables can be stopped using instruction emulation. Our scheme needs to further instrument the kernel so that the kernel cannot manage the DMA controller. The exact implementation will differ according to the used hardware platform.

Table 4 Automatic proof rate of machines

Component	Proof obligations	Proved automatically	Proved interactively or manually	Automatic proof rate (%)
AddressData	4	4	0	100
Task	62	58	4	93.55
S_Memory	42	36	6	85.71
Object	8	8	0	100
N_Memory	36	31	5	86.11
SYSTEM	72	46	26	63.89
S_data	57	40	17	70.18
Monitor	18	9	9	50
Event_Control	10	10	0	100
Total	309	242	67	78.32

7 Experimental evaluation

7.1 Evaluation of the automatic proof

We implement above scheme by Atelier B 4.2.1 [22]. Atelier B is a consolidated tool that is used in many projects both in the academia and in the industry. The Atelier B has numerous tools such as a powerful editor with the ability to warn the user in the case of mistyping or even potential typing errors, automatic proof obligation generator, automatic prover, and an interactive prover. The automatic prover of the Atelier B is very effective. As shown in Table 4, the automatic proof rate of proof obligations is about 78.32% (242 out of 309).

One of the advantages of the proposed hardware-assisted architecture is verified and demonstrated through the design and development of the memory management system in a trusted execution environment. There is much more to come.

- The memory management system can be kept small, minimizing the critical program.
- A small kernel makes it easy for the correctness of its design to be formally verified.
- The hardware-assisted architecture provides an isolated environment for multiple components of an embedded system.

7.2 Evaluation of secure world

We rely on Xilinx Zynq ZC702 evaluation board which is one of the few platforms fully supporting TrustZone. In this experiment, we observe the difference in performance when a workload is executed in secure and kernel space respectively. This is specially relevant for the pervasive OS support operations (e.g., memory operations). The TrustZone operating system is Sierraware's GPL version of Open Virtualization. We use Linux Kernel (version 4.0.1) as the operating system running in the NW, together with a light command-based version of Ubuntu.

Table 5 The output of microbenchmarks

Syscall	SW (s)	μs/c	NW (s)	μs/c	Number of calls
Read	2.6366	32	2.5512	30	87,969
Write	1.5806	16	1.2898	14	96,008
Close	0.4128	10	0.2264	4	45,137
Wait4	0.0131	13,100	0.0209	20,900	1
Open	0	0	0.0001	2	44

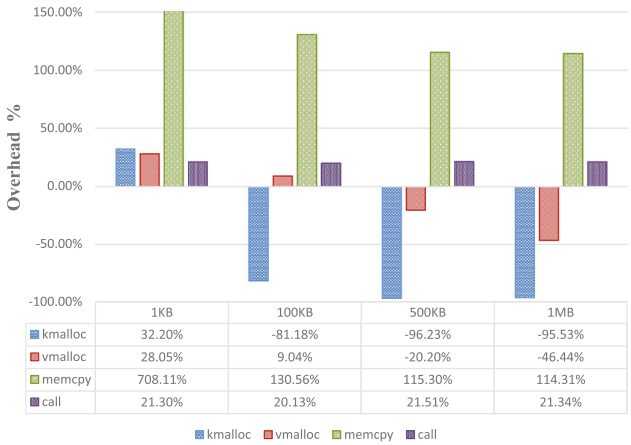
As shown in Table 5, the output of program is the result when syscall mediation through the secure area enabled or not with 10,000 executed instructions per system call. Column outputs are syscall name, seconds, microseconds per call, and number of calls, respectively.

The metric we use is the overhead introduced by the secure space, defined as:

$$Overhead = \frac{\boldsymbol{T}_{secure} - \boldsymbol{T}_{kernel}}{\boldsymbol{T}_{kernel}} \quad (1)$$

Figure 9 shows the overhead of executing memory operations in secure space. It presents the overhead of allocating memory in secure space (compared to kernel space) using *kmalloc* and *vmalloc*, the overhead of copying memory with *memcpy*, and the overhead of executing a *call* to secure space.

In Open Virtualization, when the secure area initializes at boot time, a memory pool is created using a technique similar to the buddy system. Since the secure area has a limited amount of memory, it is relatively cheap to pre-allocate and manage this pool. As a result, secure tasks can obtain dynamic memory at constant cost. The design principle followed in Open Virtualization is that secure tasks will not need to allocate big amounts of memory, and allocations will not occur as often as in a general purpose OS. In kernel space however, Linux uses slab allocation, which is designed to



	1KB	100KB	500KB	1MB
kmalloc	32.20%	-81.18%	-96.23%	-95.53%
vmalloc	28.05%	9.04%	-20.20%	-46.44%
memcpy	708.11%	130.56%	115.30%	114.31%
call	21.30%	20.13%	21.51%	21.34%

Fig. 9 Overhead of executing memory operations in secure space

allocate large amounts of memory while minimizing external fragmentation. This explains that secure area is faster at allocating memory as the requested size increases.

When using *vmalloc*, the kernel allocates virtually contiguous memory that may or may not be physically contiguous; with *kmalloc*, the kernel allocates a region of physically contiguous (also virtually contiguous) memory and returns the pointer to the allocated memory. While *kmalloc* is normally faster that *vmalloc* it depends very much on the requested allocation size and how fragmented the memory is. In our tests *kmalloc* clearly deteriorates faster than *vmalloc*; we execute each memory operation a million times in order to avoid misleading results due to scheduling, locking, etc. When looking at individual allocations, *kmalloc* does perform better for small allocation sizes.

8 Related work and future work

8.1 Threat model

McDermott proposed attack modeling by Petri net, which described the status of the system and security-related entities by places of Petri net [23]. Dalton et al. proposed an attack modeling method by generalized stochastic Petri net [24]. Xu et al. proposed an aspect-oriented Petri nets, which was a strict formalization method for threat-driven modeling and verification of security software [25]. Phillips et al. analyzed vulnerabilities in computer network by attack diagrams. Sheyne et al. explored a model checking tool for generating and analyzing attack graphs [26]. Wang et al. proposed a threat representation method by UML sequence diagram, and utilized the model-driven security testing method to discover run-time threats [27].

8.2 SGX-based

Professor Ahmad-Reza Sadeghi from Technische Universit Darmstadt (CASED) of Germany pursued his studies on Trusted Execution Environments of embedded system security. They gave the theoretical analysis for embedded system security with Intel SGX support [28]. Georgia Institute of Technology achieved a project openSGX simulating SGX by QEMU [29], which is the first attempt to use SGX in embedded field.

8.3 TrustZone-based

TrustZone-based technologies applied to mobile terminals, such as Apple SecureEnclave, Samsung KNOX and so forth [30]. Professor Azab and Professor Ning Peng from North Carolina State University developed applications for KNOX and explored their new technologies [31]. Ge X and Vijayakumar H et al. proposed a protection scheme for kernel integrity on mobile embedded devices based on TrustZone without implementation [32]. On .Net platform, a security scheme based on TrustZone TEE was jointly developed by Microsoft Research and Lisbon University [33]. Researchers from CASED proposed a new code provisioning paradigm, which ran on secure hardwares with an isolated execution environment [34].

8.4 Future work

As demonstrated in the previous works, our MIPE is an effective solution to provide the integrity protection of execution files, run-time codes, and control flows with affordable performance overhead. Based on our initial hypothesis, the future work will include the following three parts. Firstly, we would like to improve the current algorithms supporting increasing nodes of a threat tree. Secondly, it would be interesting to make improvements upon the verified process and implementations using B method. Finally, we are going to provide the memory integrity verification by formalization in the future.

9 Conclusion

In this paper, We try out a novel practical memory protection method for the security of embedded devices. We propose and implement a practical memory integrity protection method on an ARM-based platform, called MIPE, to defend against security threats including kernel data attacks and direct memory access attacks. To present the integrity protection strategies, we provide the design of MIPE using B method, which is a practical formal method. With growing multifarious threats in embedded devices, this work provides

effective integrity protection for execution files, run-time codes, and control flows. We present the security analysis and show through evaluation that the proposed scheme is effective and feasible in terms of both load time and overhead. The evaluation results also show that the automatic proof rate of machines using B method is about 78.32%. It is expected that our scheme would support potential applications on TrustZone-enabled devices in the future.

Acknowledgements Thanks to project supported by the National Natural Science Foundation of China (No. 61572516). The authors extend their appreciation to the International Scientific Partnership Program (ISPP) at King Saud University, Riyadh, Saudi Arabia for funding this work through the Project No. ISPP#0069.

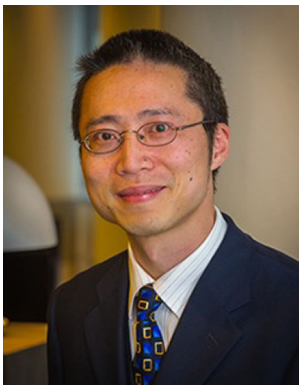
References

1. Wang, L., Hu, S., Betis, G., et al.: A computing perspective on smart City[J]. IEEE Trans. Comput. **65**(5), 1337–1338 (2016)
2. Cheng, C., Lee, J., Jiang, T., et al.: Security analysis and improvements on two homomorphic authentication schemes for network coding[J]. IEEE Trans. Inf. Forensics Secur. **11**(5), 993–1002 (2016)
3. González, J., Bonnet, P.: Towards an open framework leveraging a trusted execution environment[C]. In: The 5th International Symposium on Cyberspace Safety and Security , pp. 458-467 (2013)
4. Gustafson, A., Schunnesson, H., Galar, D., Mkemai, R.: TPM Framework for Underground Mobile Mining Equipment. A Case Study. Springer, New York (2011)
5. Nerella, V.K.S.: Exploring run-time reduction in programming codes via query optimization and caching[J]. Dissertations and Theses-Gradworks (2013)
6. Smith, S.W.: Secure Coprocessor[M]. Springer, New York (2011)
7. Mckeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. ACM (2013)
8. Kim, S., Shin, Y., Ha, J., Kim, T., Han, D.:. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. Acm Workshop on Hot Topics in Networks (2015)
9. Deng, Z., Zhang, X., Xu, D.: Spider: stealthy binary program instrumentation and debugging via hardware virtualization. Computer Security Applications Conference, pp. 289–298. ACM (2013)
10. Alves, T.: TrustZone : Integrated Hardware and Software Security. White Paper (2004)
11. Lesjak, C., Hein, D., Winter, J.: Hardware-security technologies for industrial IoT: TrustZone and security controller[C]// Industrial Electronics Society, IECON 2015—Conference of the IEEE. IEEE (2015)
12. Sun, H., Sun, K., Wang, Y., Jing, J., Jajodia, S.: TrustDump: Reliable Memory Acquisition on Smartphones. Lecture Notes in Computer Science, vol. 8712, pp. 202–218 (2014)
13. Ren, W., Zeng, L., Liu, R., Cheng, C.: F2AC: a lightweight, fine-grained, and flexible access control scheme for file storage in mobile cloud computing[J]. Mob. Inf. Syst. **2016**, 1–9 (2016)
14. Ren, W.: uLeepp: an ultra-lightweight energy-efficient and privacy-protected scheme for pervasive and mobile WBSN-cloud communications[J]. Ad Hoc Sens. Wirel. Netw. **27**(3), 173–195 (2015)
15. Kim, M., Kim, Y., Ju, H., et al.: Design and implementation of mobile trusted module for trusted mobile computing[J]. IEEE Trans. Consum. Electron. **56**(1), 134–140 (2010)
16. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.: Software guard extensions instructions and programming model. In: Proceedings of the 2013 HASP Workshop. Intel Corporation (2013)
17. Alves, T., Felton, D.: Trustzone: Integrated Hardware and Software Security. ARM white paper (2004)
18. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C: VC3: trustworthy data analytics in the cloud using SGX. IEEE Secur. (2015)
19. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven[J]. ACM Trans. Comput. Syst. **33**(3), 1–26 (2015)
20. Chi, C., Tao, J., Liu, Y., et al.: Security analysis of a homomorphic signature scheme for network coding[J]. Secur. Commun. Netw. **8**(18), 4053–4060 (2015)
21. Android Rooting Method: Motochopper. http://hexamob.com/how-to-root/motochopper-method (2015)
22. ClearSy. Atelier B 4.2.1 Free. http://www.atelierb.eu/en/2015/03/09/atelier-b-4-2-1-free-2/ (2016)
23. Dalton, G.C., Mills, R.F., Colombi, J.M., et al.: Analyzing Attack Trees Using Generalized Stochastic Petri Nets[J], pp. 116–123. IEEE (2006)
24. Xu, D., Nygard, K.E.: Threat-driven modeling and verification of secure software using aspect-oriented Petri nets[J]. IEEE Trans. Softw. Eng. **32**(4), 265–278 (2006)
25. Xu, D., Nygard, K.: A Threat-Driven Approach to Modeling and Verifying Secure Software[C]//20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), pp. 342–346. Long Beach, CA, USA (2005)
26. Wang, L., Wong, W., Xu, D.: A threat model driven approach for security testing. The 3rd International Workshop on Software Engineering for Secure Systems (2007)
27. Hwang, D.D.: Securing embedded systems. IEEE Secur. Priv. **4**, 40–49 (2006)
28. Trusted Execution Environments Intel SGX. http://sigops.org/sosp/sosp13/ (2014)
29. Intel SGX Emulation using QEMU. https://github.com/sslab-gatech/opensgx (2015)
30. Nuno, S., Himanshu, R., Stefan, S., Alec, W.: Using ARM trustzone to build a trusted language runtime for mobile applications. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 67–80 (2014)
31. Ahmed MA, Peng, N., Jitesh, S., Quan, C., Rohan, B., Guruprasad, G., Jia, M., Wenbo, S.: Hypervision across worlds real-time Kernel protection from the ARM trustZone secure world. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1028–1031 (2014)
32. Xinyang, G., Hayawardh, V., Trent, J.: Sprobes Enforcing Kernel Code Integrity on the TrustZone Architecture. Eprint Arxiv (2014)
33. Hoekstra, M., Lal, R., Pappachan, P., Rozas, C., Phegade, V.: Using innovative instructions to create trustworthy software solutions. In: Proceedings of the 2013 HASP Workshop (2013)
34. Dmitrienko, A., Heuser, S., Nguyen, T.: Market-Driven Code Provisioning to Mobile Secure Hardware. Springer, Berlin (2015)

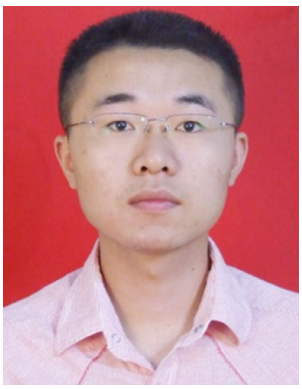
Rui Chang was born in 1981. She received the Master degree from Wuhan University of technology, Wuhan, China. She is currently a Ph.D. candidate and an Associate Professor working in State Key Laboratory of Mathematic Engineering and Advanced Computing, Zhengzhou, China. Her areas of research include computer architecture, system software, embedded systems, and security.

Liehui Jiang was born in 1967. He is currently a Professor and Ph.D. Supervisor with the State Key Laboratory of Mathematic Engineering and Advanced Computing, Zhengzhou, China. His main research interests include computer architecture, reverse engineering and security. He has published over 100 refereed papers. He is a senior member of China Computer Federation.

Wenzhi Chen was born in 1969. He received the Ph.D. degree from Zhejiang University, Hangzhou, China. He is currently a Professor and a Ph.D. Supervisor with the College of Computer Science and Technology, Zhejiang University. His areas of research include computer graphics, computer architecture, system software, embedded systems, and security. He has published over 80 refereed papers. He has served as an editorial board member of several international journals, including IEEE Transactions on Information Forensics and Security. He is a Senior Member of the IEEE and China Computer Federation.



Yang Xiang received his Ph.D. in Computer Science from Deakin University, Australia. He is currently a full professor and the Director of Centre for Cyber Security Research, Deakin University. His research interests include network and system security, data analytics, distributed systems, and networking. He has published more than 200 research papers in many international journals and conferences. He serves as the Associate Editor of IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Security and Communication Networks (Wiley), and the Editor of Journal of Network and Computer Applications. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a Senior Member of the IEEE.



Yuxia Cheng received the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2015. He is currently an associate research fellow at the School of Information Technology, Deakin University. His current research interests include multicore architecture, operating systems, virtualization and security.

Abdulhameed Alelaiwi is a faculty member of Software Engineering Department, at the College of Computer and Information Sciences, King Saud University. Riyadh, Saudi Arabia. He received his Ph.D. degree in Software Engineering from the College of Engineering, Florida Institute of Technology-Melbourne, USA. He has authored and co-authored many publications including refereed IEEE/ACM/Springer journals, conference papers, books, and book chapters. His research interest includes software testing analysis and design, cloud computing, and multimedia. He is a member of IEEE.