Received March 18, 2017, accepted April 30, 2017, date of publication May 10, 2017, date of current version October 25, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2703124

Building a Secure Scheme for a Trusted Hardware Sharing Environment

DI LU, JIANFENG MA, CONG SUN, QIXUAN WU, ZHAOCHANG SUN, AND NING XI
School of Computer Science and Technology, Xidian University, Xi'an 710126, China

Corresponding author: Di Lu (dlu@xidian.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61602357, Grant 61502368, Grant U1405255, in part by the National High Technology Research and Development Program (863 Program) under Grant 2015AA016007, in part by the China Postdoctoral Science Foundation Funded Project under Grant 2016M592762, in part by the Shaanxi Science and Technology Coordination and Innovation Project under Grant 2016TZC-G-6-3, in part by the Natural Science Basis Research Plan in Shaanxi Province of China under Grant 2016JM6034, and in part by the open project of the Key Laboratory of Grain Information Processing and Control (Henan University of Technology), Ministry of Education, under Grant KFJJ-2016-202.

ABSTRACT Trusted hardware sharing (THS) system can provide multiple trusted execution environments (TEE) via sharing the trusted hardware (e.g., sharing trusted platform module via virtualization) for stand-alone and isolation scenarios. However, the trusted function requests (TFRs) sent to the trusted hardware are emitted by multiple TEEs, which have to be processed by THS. Since different applications in different TEEs have different security requirements, the data in TFRs need to be protected from being leaked or modified in an unauthorized manner. To address this issue, we present a secure scheme for THS systems based on an information flow model that protects the sensitive data in TFRs. Each TFR is assigned a security level according to their owner, and processed in isolated environments with different security levels. We implement the prototype and conduct the experiments in both shared memory and isolated environments. The results indicate that the introduction of security mechanisms can lead to more time consumption on processing TFRs with the increase in the dimension of security levels. However, this degradation in performance is still acceptable and can be mitigated in the real world, because intensive TFR requests are not present as they are in the experimental environment.

INDEX TERMS Trusted hardware sharing, trusted computing, information flow, security level, lattice.

I. INTRODUCTION

For a long time, software-based security hardening (SSH) solutions, such as antivirus products and software firewalls, have been adopted to provide a protection mechanism for computing systems. However, there is no guarantee that SSH measures themselves can avoid being victims of cyber-attacks or being intentionally misused by malware. Because of this, SSH solutions alone do not always provide trust. To compensate for the lack of SSH, hardware supported security measures are proposed to provide trust enhancement for computing systems, such as personal computer, servers and embedded systems. Currently, the most popular hardware supported security hardening is the Trusted Platform Module (TPM) [1], which relies on a dedicated microprocessor to establish trust between communication partners. It comprises secure storage for cryptographic keys and cryptographic co-processors to provide reliable integrity measurement and remote attestation services.

With the help of TPM, computing system can establish a trusted execution environment (TEE) for OS and applications from system startup. With the upgrading of hardware performance of computing systems, the virtualization technique is widely adopted to provide multiple execution environments with various configurations on a single hardware. For embedded system, hardware can support multiple virtualized and isolated execution environment, such as MILS in avionic system [2]–[6]. However, an issue arises on how a TPM hardware serves multiple virtualized execution environments (VEE). To address this problem, current research works (refers to section II) concentrate on building a virtualized TPM for each VEE via the virtualization technique to achieve the goal of sharing one TPM hardware among multiple VEEs.

Figure 1 shows a typical example of the architecture of TPM sharing in a virtualization environment. The key component is the "virtualized TPM service (vTPMSvc)" module, which performs as an agent of the TPM hardware and is implemented in hypervisor. A vTPM is implemented as a virtual hardware device in each VM, which is managed by the vTPMSvc module. When an invocation to trusted function (also called a trusted function request, TFR, such as

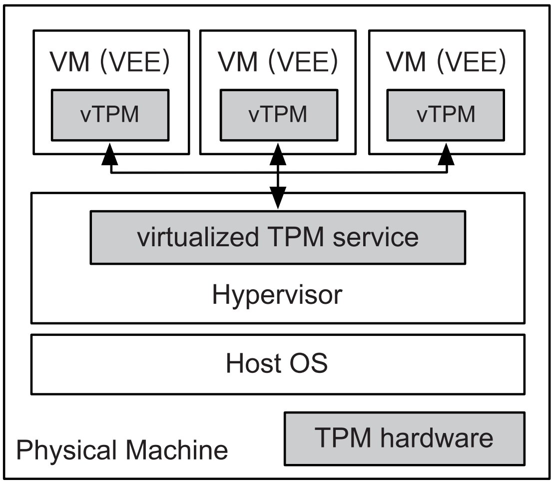


FIGURE 1. A typical scenario of TPM sharing in virtualization environment.

requesting cryptographic keys, integrity measurement, remote attestation services) occurs in a VM, the vTPM needs to deliver the invocation to the vTPMSvc module, which continuously hands over the request to the TPM hardware. When the TPM hardware finishes processing the request, the results are sent back to the VM. When multiple TFRs arrive at vTPMSvc, it needs to schedule the requests for either performance or security requirements.

With the virtualization technique, TPM functions are actually extended into each VM (or VEE) running on a single physical device. Thus, the trusted chain can be extended from the physical device to each nested virtualized environment, which guarantees the VEE can establish a TEE for itself. In cloud computing, multiple VMs are running as virtual servers providing services, such as web service and database service, on one physical machine with a TPM sharing mechanism. These services provided by VMs have different security requirements according to the service type, e.g., cryptographic services for different users or applications on particular security level. Hence, the TFRs invoked by VM also have security requirements, e.g., VM needs to request cryptographic keys from TPM via vTPM, and this procedure should be protected from being leaked. In this procedure, the TFR data should be prevented from unauthorized disclosure and modification. Moreover, the processing of continuous service data should not cause leakage or modification of sensitive data. To address both of the problems, we propose a security scheme based on *information flow control* to protect the trusted hardware sharing systems (THS).

Our threat model is one where the VEE may contain software with unintentional bugs, malicious codes or Trojan horses that cause information leakage. E.g., malicious codes may be injected into TFR by malware hidding in VEEs, which can be invoked while TFRs are being processed and can access other TFRs illegally. We focus on how TFR data generated by the VEE can be processed in a secure manner so that they are protected against unauthorized disclosure and modifications. Meanwhile, users can deliberately obtain information that they are not authorized to access.

In our paper, we first establish information flow constraints for the processing of TFR data. We formalize the information flow constraints in the THS environment and demonstrate how TFRs are processed with these information flow constraints. We also give the scheduling policy for the vTPMSvc module, which can improve the performance of request processing. We implement a prototype to demonstrate the feasibility of our approach and show how the performance is impacted by the security scheme as well as the improvement in vTPMSvc performance with our scheduling policy.

The rest of the paper is organized as follows. In Section II, we mention a few related works. In Section III, we conduct a brief discussion on TFR processing and introduce the security level to this procedure. In Section IV, we present our security model based on information flow control, and discuss TFR processing in detail. Moreover, our scheduling policy for the vTPMSvc is also discussed in this section. To prove the feasibility of our scheme, we propose the system implementation and evaluation in Section V. Finally, we conclude the paper in Section VI.

II. RELATED WORK

As mentioned before, different VEEs have different security requirements, e.g., the VEE providing critical services has higher security requirements than that providing normal services; hence, devices/VEEs security requirements need to be taken seriously when delivering a trust function request (TFR) in distributed environment. However, most recent work focus on how to share the single hardware TPM among multiple devices or VEEs, which lacks consideration of the security requirements of devices/VEEs.

For a virtualized environment, the hardware TPM is usually shared among VEEs via creating multiple virtualized TPMs (vTPMs), which extend TPM functionalities into VEEs. In [7], TPM is introduced to the virtualization platform as the trusted root, which can provide an authentication mechanism for both VMs and access users. The hardware TPM and the trust service are deployed on a Trust Validation Server (TVS). Trust service is virtualized into multiple instance (vTPM service) for servers containing virtual machines. Virtual machines are assigned virtualized TPMs (vTPMs) by sharing the single hardware TPM via vTPM services. This approach realizes the extension of the hardware TPM function and enhances the security of the virtualization platform. However, Sule *et al.* [8] designs and deploys a trusted cloud computing for power system applications. The hardware TPM is used as the trust root to establish the chain-of-trust for the infrastructure of the cloud platform. When all the software components of the cloud infrastructure are successfully measured, a software based TPM (vTPM) within the VM builds a new chain-of-trust for the components of the VM. Wang *et al.* [9] proposes the Trusted Cloud Platform, which also extends the security mechanism from the hardware TPM to the VM and VM monitor (VMM). With this architecture, the host OS, VMM and the VM images can be measured before startup, which achieves the establishment of the chain-of-trust from the hardware to the applications running in VMs.

For sharing the hardware TPM among devices, Feng *et al.* [10] propose Trusted Execution Environment Module (TEEM), a portable Trusted Computing module that can provide trusted computing functionalities for various computing platforms such as desktop machines and mobile devices. TEEM is designed as a secure TPM service running in the secure world of TrustZone, and a prototype is implemented on a general ARM SoC development board. To pave the way for utilizing TPM in cross-device scenarios, Chen *et al.* [11] proposes cTPM, which extends the original TPM design by adding an additional root key to the TPM and making that root key available for sharing with the cloud. This approach actually achieves TPM sharing by extending the scope of the root key utilization. Raj *et al.* [12] proposes firmware-TPM (fTPM), an end-to-end implementation of a TPM using ARM TrustZone, for ARM-based mobile devices. fTPM is actually a software defined TPM that relies on the security features of the ARM processor. Hence, fTPM can also be considered as the hardware security mechanism that is shared by applications via a virtualized TPM. To address the lack of trusted hardware for mobile devices, Proskurin *et al.* [13] proposes a secure element based TPM (seTPM) for trust establishment in mobile devices. seTPM is actually a software deployed in a GlobalPlatform-defined secure element, which can be shared by multiple applications in the mobile device via a seTPM driver embedded in the host OS kernel. Constantin *et al.* [14] presents a trusted architecture for a partitioned multicore processor based on TPM. A trusted component is designed in the OS kernel (trusted kernel), which can virtualize the hardware TPM into multiple vTPMs for different partitions. In this architecture, the trusted kernel achieves TPM sharing and acts as a vTPM manager.

III. TRUSTED FUNCTION REQUEST PROCESSING

Figure 2 shows the security requirement for each VEE, and the processing of TFR in the THS environment. Services, applications or guest OS can invoke TFRs, such as requesting cryptographic key and VEE validation, via vTPM module in VEE. As the figure shows, A TFR denoted as $TFR^{SL_i}_{VEE1.svc_m}$ (i and m represent the label of the security level and service respectively), issued by $VEE1.svc_m$ (a service application in VEE1) is transferred to vTPM, which continuously delivers the request to the vTPMSvc module. After vTPMSvc finishes processing $TFR^{SL_i}_{VEE1.svc_m}$ (including TFR security level verification and the scheduling procedure), the TFR is sent to the TPM hardware (path ①). Finally, the $TFR^{SL_i}_{VEE1.svc_m}$ is completed in the TPM hardware, and the resulting data, $RSLT^{SL_i}_{VEE1.svc_m}$, are delivered back to the $VEE1.svc_m$ in the reverse path (path ②).

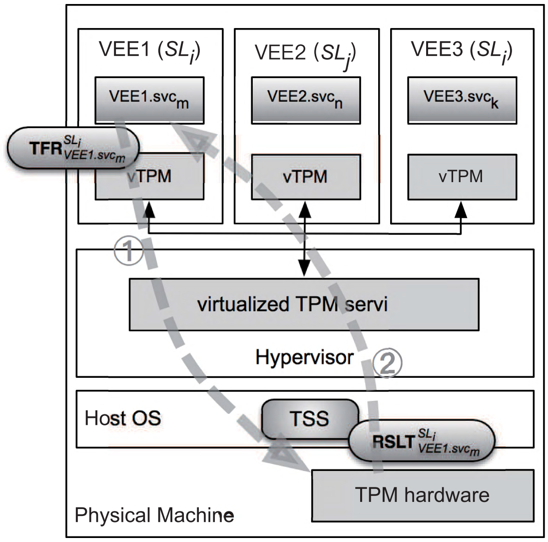


FIGURE 2. Security requirements in VEEs and the processing of TFR in the THS environment.

Let us consider the security requirements, as shown in Figure 2, we assume that VEE1 and VEE3 are in security level SL_i and VEE2 is in security level SL_j (The security level of VEE refers to that of TFRs invoked by the VEE). Without loss of generality, we assume $SL_i \preccurlyeq SL_j$, which means level SL_i is dominated by level SL_j (the *domination* relation of security is discussed in section IV-A). Considering a trusted function request, $TFR^{SL_i}_{VEE1.svc_m}$, generated in VEE1, is to be transferred to the vTPM module of VEE1. $TFR^{SL_i}_{VEE1.svc_m}$ indicates that the TFR is from service application svc_m, and is in security level SL_i, which is inherited from the VEE1. vTPM then sends the request to the vTPMSvc module of the hypervisor. vTPMSvc provides isolation for the TFR from different VEE in different security levels. Thus, TFR processing is constrained in a particular processor in vTPMSvc, named the TFR processing unit (TPU), which can process TFR in a required security level (a TPU has multiple TFR processing queues to satisfy the requirement of processing TFRs in various security levels). Finally, $TFR^{SL_i}_{VEE1.svc_m}$ is completed in the hardware TPM.

When the hardware TPM finishes processing $TFR^{SL_i}_{VEE1.svc_m}$, the result $RSLT^{SL_i}_{VEE1.svc_m}$ is generated. Then, the result data $RSLT^{SL_i}_{VEE1.svc_m}$, including the status information (successful or failed) and result data, are sent back to the vTPMSvc module, which finally transfers the result to $VEE1.svc_m$. During this procedure, $RSLT^{SL_i}_{VEE1.svc_m}$ is also handled in security constraints, including secure queues in the vTPMSvc module.

According to the discussion above, the most significant mechanism in vTPMSvc is the isolation among TFR processing queues (TPQ) in various security levels. All the TPQs are managed by a *TPQ Monitor (TPQM)*, which provides isolation for TPQs. If one TFR contains malicious code, it cannot access TFRs in the other TPQs. Moreover, each TFR in TPQ is encrypted, thus, TFR data will not leak to other

TFRs in the same TPQ (the same security level). Besides, the whole TPU, including TPQs and TPQM, is also isolated from other modules, which prevents the TPU from being attacked by malicious code. The implementation of TPU will be further discussed further in section V.

IV. SECURITY MODEL

In this section, we demonstrate the design details of our security scheme. First, the information flow control approach is introduced, and then, according to this, our constraints for TFR processing are proposed. Secondly, we present the procedure of TFR processing with our security constraints. Finally, to demonstrate how to improve the performance of vTPMSvc in processing requests generated by multiple VEEs, our request scheduling policy is proposed.

A. INFORMATION FLOW CONTROL

In the following, we present an information flow model for the THS system to protect against improper leakage and disclosure. An information flow model adapted from the *lattice* [15], [16] structure is used to establish our model.

A THS system has multiple VEEs providing services, which can be partitioned into *conflict of interest (COI)* classes according to the type of services they provide. The VEEs providing the same type of service are in direct competition with each other. Thus, preventing the sensitive information in TFRs with different security requirements from being leaked during processing is significant. The definition of the *CoI* is given in the following.

Definition 1: The VEEs providing the same type of services are partitioned into a number of conflict of interest classes denoted by $\Phi_1, \Phi_2, \ldots, \Phi_n$. Each CoI contains VEEs providing the same type of service, which can be denoted as $CoI_i = \{VEE_1, VEE_2, \ldots, VEE_m\}$, where $m \geq 1$ and $1 \leq i \leq n$.

According to Definition 1, a CoI set, e.g., Φ_k, represents a type of service, which consists of VEEs providing this service. For example, there are three CoI sets $\Phi_1 = \{VEE_1, VEE_2, VEE_3\}$, $\Phi_2 = \{VEE_a, VEE_b, VEE_c\}$ and $\Phi_3 = \{VEE_1, VEE_b, VEE_2\}$, and we have $\Phi_1 \cap \Phi_2 = \varnothing$, which means no VEE provides both service Φ_1 and Φ_2. Moreover, $\Phi_1 \cap \Phi_3 = \{VEE_1, VEE_2\} \neq \varnothing$ which means both VEE_1 and VEE_2 provide the same service Φ_1 and Φ_3.

In addition, some VEEs may provide different services from one another, which are also not in direct competition with each other. These VEEs are considered as the ones providing complementing services in the THS system. We define the notion of complementing interest (CI) class and discuss its significance.

Definition 2: The VEEs providing complementing services are represented as an n-element vector $\Omega = [VEE_1, VEE_2, \ldots, VEE_n]$, where $VEE_k \in \Phi_k \cup \{\perp\}$ and $1 \leq k \leq n$. The vector Ω is a CI class. $VEE_k = \perp$ signifies that the CI class does not contain services from any VEE in Φ_k. $VEE_k \in \Phi_k$ indicates that the CI class contains services from the corresponding VEE in COI class Φ_k.

From both Definition 1 and 2 we can obtain that if $VEE_k \in \Phi_k$, Φ_k may not be unique. For example, referring to the example above, VEE_b provides services Φ_2 and Φ_3, assuming a CI class Ω_p containing VEE_b, thus, $VEE_b \in \Phi_2 \vee VEE_b \in \Phi_3$, which means one VEE may provide multiple services. Moreover, assuming $\Omega_p = [VEE_1, VEE_2]$, thus, VEE_1 and VEE_2 have to provide complementing services. However, both VEE_1 and VEE_2 are in Φ_1, which means VEE_1 and VEE_2 are in the same COI class, which contradicts our assumption. Hence, Definition 2 forbids multiple VEEs that are part of the same COI class from being assigned to the same CI class.

In the following, we define the security model for the THS system. As mentioned in Section III, each TFR is associated with a security level that captures its sensitivity. Security level associated with a TFR indicates which entities (user, application, TFR processing queue, etc.) can access or modify it. Since VEEs have different security levels, VEEs in the same COI class may have different security levels. E.g., assuming VEE_i and VEE_j are in different security levels, however, both of them provide the same service Φ_p; thus, we have $\Phi_p = \{VEE_i, VEE_j\}$ with two elements in different security levels. Moreover, VEEs in the same CI class may have the same security levels. E.g., we assume that VEE_i and VEE_j have the same security level and provide complementing services, which can be denoted by a vector $\Omega_p = [VEE_i, VEE_j]$. VEE_i provides different services from VEE_j, but they are in the same security level. Next, we show how security levels are represented.

Definition 3: A security level is denoted as an n-element vector $[s_1, s_2, \ldots, s_n]$, where $s_j \in \Phi_j \cup \{\perp\} \cup \{\top\}$ and $1 \leq j \leq n$. $s_j \in \Phi_j$ indicates that the TFRs are generated by corresponding VEE in Φ_j; $s_j = \perp$ signifies that the TFRs are generated by the VEE not in Φ_j; $s_j = \top$ signifies that the TFRs are generated by more than one VEEs in Φ_j.

According to Definition 3, the security level vector signifies which VEE generates the TFR, and which COI class the VEE belongs to. Assuming we have three COI classes denoted as $\Phi_1 = \{VEE_1, VEE_2, VEE_3\}$, $\Phi_2 = \{VEE_a, VEE_b, VEE_c\}$ and $\Phi_3 = \{VEE_1, VEE_b, VEE_2\}$. The TFR generated by VEE_a in Φ_2 has a security level of $SL_{\Phi_2.VEE_a} = [\perp, VEE_a, \perp]$. Similarly, the TFR generated by VEE_3 in Φ_1 has a security level $SL_{\Phi_1.VEE_3} = [VEE_3, \perp, \perp]$.

Considering a cloud system established based on SOA architecture, one service may consist of multiple services running in multiple VEEs. Thus, one TFR can contain information regarding multiple VEEs. For example, the service Φ_1 in VEE_1 requests service Φ_2 in VEE_b, in order to verify the identity of VEE_1, VEE_b has to request the TPM to execute the verification. Thus, the verification request can be packed as a TFR denoted as $TFR^{VLD}_{[VEE_b.\Phi_2, VEE_1.\Phi_1]}$, where the superscript *VLD* indicates the TFR type, and $[VEE_b.\Phi_2, VEE_1.\Phi_1]$ indicates the *source* ($VEE_b.\Phi_2$) and *target* ($VEE_1.\Phi_1$) entities. $VEE_1.\Phi_1$ means VEE_1 belongs to COI class Φ_1. Although $TFR^{VLD}_{[VEE_b.\Phi_2, VEE_1.\Phi_1]}$ is generated by VEE_b, it also contains information about VEE_1

(Because VEE_1 will send some information to VEE_b when it requests service Φ_2). In this case, the security level of $TFR^{VLD}_{[VEE_b.\Phi_2,VEE_1.\Phi_1]}$ is $[VEE_1, VEE_b, \bot]$. Moreover, when service Φ_1 in VEE_1 requests services Φ_2 in VEE_b and service Φ_3 in VEE_2, both of $VEE_b.\Phi_2$ and $VEE_2.\Phi_3$ have to verify $VEE_1.\Phi_1$ via the TPM. Thus, the verification TFR has the form of $TFR^{VLD}_{[[VEE_b.\Phi_2,VEE_2.\Phi_3],VEE_1.\Phi_1]}$, where the *source* part of the subscript is a list consisting of two VEEs. Since the TFR contains information of VEE_1, VEE_2 and VEE_b as well as the information of the corresponding services Φ_1, Φ_2 and Φ_3, the security level of the TFR has the form of $[VEE_1, VEE_2, VEE_b]$.

More generally, assuming $VEE_1.\Phi_1$ and $VEE_2.\Phi_2$ request the same trusted function of the TPM simultaneously (E.g., requesting the TPM to generate a key), their TFRs can be combined as $TFR^{Fx}_{[[VEE_1.\Phi_1,VEE_2.\Phi_2],\cdot]}$. Fx denotes one type of function of the TPM (VLD is actually one type of Fx), and the *source* part of the subscript is a list containing both of $VEE_1.\Phi_1$ and $VEE_2.\Phi_2$, whereas the *target* part is determined by Fx (e.g., when Fx is VLD, the *target* part is the VEEs being verified). Thus, the security level of the TFR is $[VEE_1, VEE_2, \bot]$. Moreover, the RSLT data have the same security level as the corresponding TFR.

Definition 4: Assuming **SL** *is the set of security levels, which is denoted as* $\mathbf{SL} = \{SL_1, SL_2, \ldots, SL_n\}$. *We say security level* SL_1 *is dominated by* SL_2*, denoted as* $SL_1 \preccurlyeq SL_2$*, if the following equation holds:* $\forall i_k = 1, 2, \ldots, n$, $(SL_1[i_k] = SL_2[i_k]) \vee (SL_1[i_k] = \bot) \vee (SL_2[i_k] = \top)$. *Considering any two levels* $SL_i, SL_j \in \mathbf{SL}$*, if neither* $SL_i \preccurlyeq SL_j$*, nor* $SL_j \preccurlyeq SL_i$*, they are incomparable.*

As described in definition 4, $SL_x[i_k]$ refers to the i_kth element (such as VEE, $\bot$ or $\top$) in the level SL_x, which is denoted as a vector. $(SL_1[i_k] = SL_2[i_k])$ implies that the corresponding elements in the two arrays are equal in security levels (two identical vectors). $SL_1[i_k] = \bot$ refers to a level $[\bot, \bot, \ldots, \bot]$, which is *public* to SL_2. However, $SL_2[i_k] = \top$ signifies that SL_2 is *trusted*, which refers to the top security level. Thus, we can obtain that the level $[\top, \top, \ldots, \top]$ (known as the *"trusted"* level) dominates all the other levels, whereas the level $[\bot, \bot, \ldots, \bot]$ (the *"public"* level) is dominated by all levels, and each security level is dominated by itself. For example, level $[VEE_2, \bot, VEE_b]$ is dominated by $[VEE_2, VEE_c, \top]$ which is then dominated by $[VEE_2, \top, \top]$. $[VEE_2, \top, \top]$, which is dominated by $[\top, \top, \top]$. Thus, we have:

$$[VEE_2, \bot, VEE_b] \preccurlyeq [VEE_2, VEE_c, \top] \preccurlyeq [VEE_2, \top, \top] \preccurlyeq [\top, \top, \top].$$

However, for example, $[VEE_2, \bot, VEE_b]$ and $[VEE_3, \bot, VEE_a]$, $[\bot, \bot, \top]$ and $[VEE_3, \top, VEE_1]$ are incomparable.

As mentioned before, RSLT data are the result of the corresponding TFR and generated by the hardware TPM. Hence, RSLT inherits the security level from the corresponding TFR. When the RSLT data arrives at the vTPMSvc, it will be put into the resulting queue (part of the RSLT processing unit) with the same security level and waits to be sent to the corresponding VEE. Since all TFRs are finally processed by the hardware TPM, the hardware TPM is the *trusted* entity with level $[\top, \top, \top]$, which means the TPM hardware is always in the top level.

Note that, based on the dominance relation among security levels, entities (VEE, TPU, etc.) can only process the data (such as TFR/RSLT data) with dominated security levels. Assuming a service set Φ containing n types of services, VEE_x provides only one type of service Φ_k $(1 \leq k \leq n)$, we have $VEE_x \in \Phi_k$. Thus, VEE_x has the security level with the form of $SL_{\{\Phi_k.VEE_x\}} = [\bot, \bot, \ldots, VEE_x, \ldots, \bot]$, where $\|SL_{\Phi_k.VEE_x}\| = n$ and VEE_x is the kth element of the vector, and the subscript $\{\Phi_k.VEE_x\}$ indicates that VEE_x provides service Φ_k. The level $SL_{\Phi_k.VEE_x}$ implies that when VEE_x provides service Φ_k, it can only access the TFR data generated by itself and receive the corresponding RSLT data. Moreover, according to the definition 1, a VEE can provide two or more services; thus, a VEE may appears in multiple CoI classes. Assuming VEE_x provides two services Φ_p and Φ_q, its security level has the form of

$$SL_{\{\Phi_p.VEE_x,\Phi_q.VEE_x\}} = [\bot, \ldots, \bot, VEE_x, \bot, \ldots, \bot, VEE_x, \bot, \ldots, \bot],$$

where the two VEE_x are the pth and the qth elements in the vector. Thus, when VEE_x provides services Φ_q and Φ_q, it can access its own TFR data and the corresponding RSLT data. Particularly, if VEE_x begins to provide a new service Φ_γ (without loss of generality, we assume $\Phi_\gamma \in \Phi$), its security level must be updated to:

$$SL_{\{\Phi_p.VEE_x,\Phi_q.VEE_x,\Phi_\gamma.VEE_x\}} = [\bot, \ldots, \bot, VEE_x, \bot, \ldots, \bot, VEE_x, \bot, \ldots, \bot, VEE^*_x, \bot, \ldots, \bot]$$

where the new added element VEE^*_x is the γth element. Conversely, if a VEE stops to provide an existing service, the corresponding security level vector has to be updated. Note that, security level vector also indicates what types of service the VEE provides, e.g., $SL_{\{\Phi_p.VEE_x,\Phi_q.VEE_x,\Phi_\gamma.VEE_x\}}$ signifies that VEE with this level provides services Φ_p,Φ_q and Φ_γ. On the other hand, CoI classes Φ_p,Φ_q and Φ_γ contain VEE_x. Hence, after updating the security level, the corresponding CoI classes also have to be updated.

Definition 5: Assuming SL_i *and* SL_j $(SL_i \neq SL_j \vee \|SL_i\| = \|SL_j\|)$ *are two security levels, a security combination of* SL_i *and* SL_j *is defined as* $SL_i \bigoplus SL_j$. *The binary operator* $\bigoplus$ *denotes the combination of two security levels. Thus, we have* $SL_i \preccurlyeq SL_i \bigoplus SL_j$ *and* $SL_j \preccurlyeq SL_i \bigoplus SL_j$.

Note that, according to definition 5, the constraint $\|SL_i\| = \|SL_j\|$ is indispensable, for the security level with smaller length has fewer types of service so that it can not be combined with the level having a larger length. For example, assuming $\Phi_1 = \{VEE_1, VEE_2, VEE_3\}$, $\Phi_2 = \{VEE_a, VEE_b, VEE_c\}$ and $\Phi_3 = \{VEE_1, VEE_b, VEE_2\}$, two security level vectors $SL_i = [VEE2, VEEc, \bot]$ and $SL_j = [\top, VEE_a, VEE_1]$, thus, $SL_i \bigoplus SL_j = [\top, [VEE_a, VEE_c], VEE_1]$.

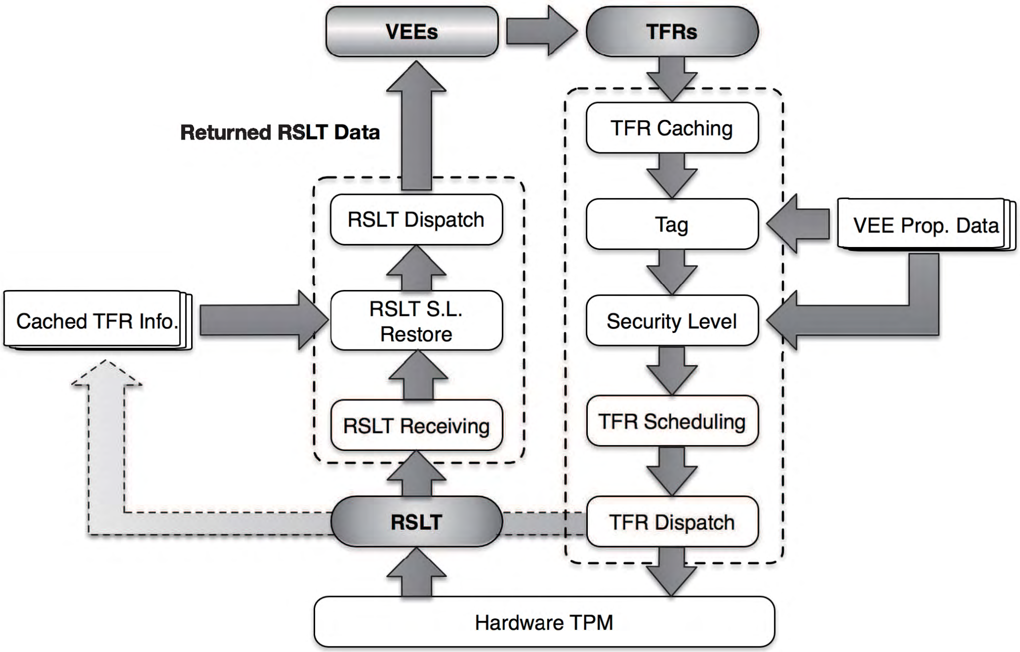


FIGURE 3. Cycle of TFR/RSLT data processing.

We have $SL_i \preccurlyeq [\top, [VEE_a, VEE_c], VEE_1]$ and $SL_j \preccurlyeq [\top, [VEE_a, VEE_c], VEE_1]$.

From the example above, we can obtain that $X \bigoplus Y = X(i) \bigoplus Y(i)$, where $X, Y \in \{\Phi_x\} \cup \{\perp\} \cup \{\top\}$, and i indicates the i^{th} element of security level X and Y. Particularly, $\perp \bigoplus X = X$, where $X \in \{\Phi_x\} \cup \{\perp\}$. Conversely, $\top \bigoplus X = \top$, where $X \in \{\Phi_x\} \cup \{\top\}$. Moreover, $X \bigoplus Y$ is the *least upper bound* of X and Y. Operator "$\bigoplus$" can be used in combination with two similar security levels. E.g., services $VEE_1.\Phi_1$ and $VEE_b.\Phi_2$ request service $VEE_2.\Phi_3$, and their TFRs have security levels $[VEE_1, \perp, VEE_2]$ and $[\perp, VEE_b, VEE_2]$, respectively. Since service $VEE_2.\Phi_3$ is the common requesting target, both security levels can be combined into $[VEE_1, VEE_b, VEE_2]$. Hence, through security level combination, TFRs can be combined for higher processing performance.

B. TFR/RSLT DATA PROCESSING

In this section, we discuss the details of TFR/RSLT data processing with the security mechanism. As shown in figure 3, the processing of TFR/RSLT data can be divided into two parts, TFR (right part in figure 3) and RSLT processing (left part in figure 3). The dotted box in the figure indicates that these procedures are conducted in the vTPMSvc module, which are protected by the hypervisor.

First, the TFRs are generated by VEEs and sent to the *"Tag"* procedure, which checks each TFR and tags it with the properties of the corresponding VEE and service, such as VEE/service ID/name, service type, priority, IP address and VEE/service owner information. Next, each TFR is assigned a security level by the *"Security Level"* procedure based on the database storing the security level information of the VEEs. After that, TFRs are scheduled by the *"TFR Scheduling"* procedure, (the details regarding TFR scheduling are discussed in section IV-C). During scheduling, the scheduler checks the security level of each TFR, and decides in which TPQ the TFR should be enqueued. If the TPQ with appropriate security level does not exist, the TPU will create a new TPQ. Next, the scheduler fetches a TFR from one TPQ and sends it to the *"TFR Dispatch"* procedure. Finally, procedure *"TFR Dispatch"* caches the properties of each TFR (such as security level, VEE information), and sends the TFR to the hardware TPM sequentially (hardware TPM only accepts sequential access).

When the hardware TPM finishes processing a TFR, the corresponding RSLT data are generated. Since the RSLT data are sequentially generated by the hardware TPM, the RSLT needs to be cached temporally before it can be further processed (in the procedure *"RSLT Receiving"*). The cached RSLT data are then fetched in procedure *"RSLT Security Level Restore (RSLT S.L. Restore)"* and will be assigned properties according to the corresponding TFR information cached in the *"TFR Dispatch"* procedure. Finally, in the *"RSLT Dispatch"* procedure, RSLT data are fetched from RSLT cache and sent to the corresponding VEEs.

C. TFR SCHEDULING

According to the discussion of TFR/RSLT processing details in section IV-B, we present a further discussion on the internals of TFR scheduling. We begin by introducing the internals of the TFR scheduling procedure, including the key components. Then, scheduling algorithms are proposed and discussed.

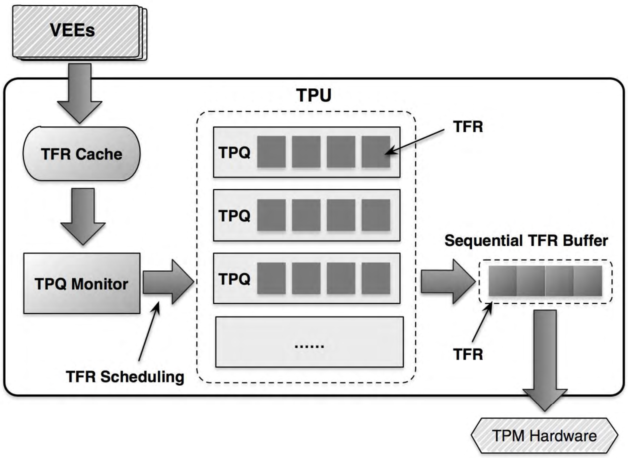


FIGURE 4. TFR scheduling internals.

Figure 4 shows the key phases involved in TFR scheduling and the details of related components, including *TFR Cache*, *TPU/TPQ* and *Sequential TFR Buffer*. As mentioned before, multiple VEEs can send TFRs to vTPMSvc simultaneously, and a cache (TFR Cache) is set in TPU to balance the difference between the rate of TFR processing and submitting. Each TFR in the cache is then fetched in order by TFR Monitor, which schedules TFRs to appropriate TPQs according to their security levels. Finally, TFRs are fetched from each TPQ by a *"Round-Robin"* algorithm and cached in the Sequential TFR Buffer, which ensures that the hardware TPM can be accessed in a sequential manner. In this section, we focus on TFR scheduling, which is performed by the TFR Monitor.

TABLE 1. Symbol table for algorithm 1 and 2.

Symbol	Description
CCH_{TFR}	TFR Cache
TFR	Current TFR
SL_{TFR}	Security level of current TFR
$\mathbf{Q}_{TPQ}$	TPQ list
$\hat{Q}$	TPQ with minimal dominating security level
SL_Q	Security level of current TPQ
Q^*	New created TPU

Table 1 shows the symbols used in algorithm 1. Note that, $\mathbf{Q}_{TPQ}$ denotes a list including all TPQs in TPU. The current processing TPQ is represented by Q. TFR denotes the latest TFR fetched from the TFR Cache. Algorithm 1 fetches TFR from the TFR Cache, and ensures that the TFR can be delivered to the appropriate TPQ. In the outer loop, the algorithm checks the TFR Cache (CCH_{TFR}) and picks the TFR at the head of CCH_{TFR}. Note that, function $PickMinDomSLQueue(sl, \mathbf{Q}_{TPQ})$ is used to pick a queue (denoted by $\hat{Q}$) with minimal security level from $\mathbf{Q}_{TPQ}$, which can dominate sl. E.g., $sl = [VEE_1, \bot, \bot]$, there is a security level list $\mathbf{SL} = [VEE_1, VEE_a, \bot]$, $[VEE_1, \top, \bot]$, $[\top, \top, VEE_3]$. We have $\mathbf{SL}_1 \preccurlyeq \mathbf{SL}_2 \preccurlyeq \mathbf{SL}_3$ ($\mathbf{SL}_i$ denotes the i_{th} element of $\mathbf{SL}$). Since $sl \preccurlyeq \mathbf{SL}_1$, $\mathbf{SL}_1$ is

Algorithm 1 TFR Scheduling Algorithm

Require: CCH_{TFR},TFR Cache;
Ensure: Does the TFR be successfully delivered to the TPU? (Boolean value);
repeat
  $TFR \Leftarrow Dequeue(CCH_{TFR})$;
  $\mathbf{Q}_{TPQ} \Leftarrow$ Get TPQ list;
  $\hat{Q} \Leftarrow PickMinDomSLQueue(\mathbf{Q}_{TPQ})$;
  if $\hat{Q}$ is found **then**
    $Enqueue(\hat{Q}, TFR)$;
  else
    Create a new queue Q^* with level SL_{TFR};
    $Enqueue(Q^*, TFR)$;
  end if
until CCH_{TFR} is empty

the minimal dominating security level to sl. Algorithm 2 gives the details of function *PickMinDomSLQueue*. If no proper queue ($\hat{Q}$) is found for *TFR*, a new queue, Q^*, is created with the same security level as *TFR*.

Algorithm 2 Pick Queue With Minimal Dominating Security Level (*PickMinDomSLQueue*)

Require: $\mathbf{Q}_{TPQ}$, SL_{TFR};
Ensure: $\hat{Q}$ index in $\mathbf{Q}_{TPQ}$;
$i \Leftarrow 1$;
$tmp\_sl \Leftarrow null$;
$tmp\_idx \Leftarrow -1$;
while $i \leq \|\mathbf{Q}_{TPQ}\|$ **do**
  $Q_i \Leftarrow$ get the $i_{th}Q$ in $\mathbf{Q}_{TPQ}$;
  $SL_i \Leftarrow$ get security level of Q_i;
  if $SL_{TFR} = SL_i$ **or** $SL_{TFR} \preccurlyeq SL_i$ **then**
    if $tmp\_sl = null$ **or** $SL_i \preccurlyeq tmp\_sl$ **then**
      $tmp\_sl \Leftarrow SL_i$;
      $tmp\_idx \Leftarrow i$;
    end if
  end if
  $CheckLongTimeIdle(Q_i)$;
  $i \Leftarrow i + 1$;
end while
return tmp_idx;

The outer loop of algorithm 1 will terminate when the TFR Cache is empty. Hence, the execution time depends on the size of the TFR Cache. Furthermore, function *PickMinDomSLQueue* has a time complexity of $\Theta(n)$, which is determined by $\|\mathbf{Q}_{TPQ}\|$. Function *CheckLongTimeIdle* just checks whether the queue is empty for long time and flags each empty queue; thus, its time complexity is $\Theta(1)$. Hence, the time complexity of the algorithm 1 is $\Theta(n^2)$.

V. PROTOTYPE IMPLEMENTATION AND EVALUATION

A. PROTOTYPE IMPLEMENTATION

We have implemented a prototype of secured vTPMSvc, which contains more than 4500 lines of C++ codes and

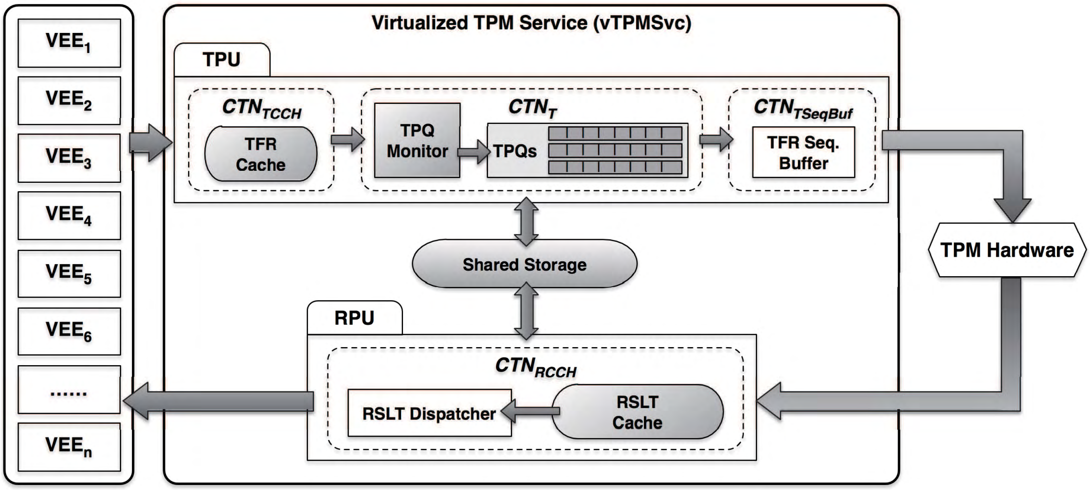


FIGURE 5. A prototype of secured vTPMSvc implementation.

consists of TFR Cache, TPQ, TFR Seq. Buffer and RSLT Cache (RSLT Dispatcher belongs to RSLT Cache). We use TPM emulator as the shared hardware TPM. The architecture of our prototype is shown in figure 5. The cycle of TFR/RSLT processing contains two primary procedures, *TFR processing* and *RSLT processing*. VEEs generate TFRs and send them to TPU, which processes these TFRs according to their security levels and delivers them to the hardware TPM. The RSLT data are generated by the hardware TPM when it finishes the TFRs, and the RSLT data will be dispathed to VEEs. Note that, a Shared Storage is set to temporarily store information related to the TFRs being processed, which will be restored to corresponding RSLT by RPU.

The system contains two primary units, TPU and RPU, which take charge of processing TFR and RSLT, respectively. In order to ensure that the key components are not accessed or modified by unauthorized and malicious applications or users, we isolate these components with *container* (the *container* depends on the system implementation), so that the malicious applications in one component cannot affects other components. In our prototype, *Linux Container* (LXC) [17], [18] is adopted to provide a isolation environment for each component. Since LXC provides separate address space, it is more secure than shared memory environment. Shown as figure 5, CTN_{TCCH} denotes the containers for TFR cache, and CTN_T represents the containers for TPQ. TFR Sequential Buffer (TFR Seq. Buffer) is contained by $CTN_{TSeqBuf}$, whereas RPU has on container CTN_{RPU}. which includes RSLT Cache and RSLT Dispatcher. Thus, both of TPU and RPU are container managers.

As mentioned in Section IV, TPQ Monitor provides in-queue isolation to prevent TFR from being accessed or modified by malicious code hiding in other TFRs. In our implementation, each node of TPQ (the TPQ node is actually TFR) will be allocated a *shadow memory address*, which is managed and protected by TPQ Monitor and cannot be used to access other nodes' addresses without the TPQ Monitor. Thus, malicious code in a node cannot access other nodes' data. Moreover, in-queue isolation is also adopted in TFR/RSLT Cache and TFR Seq. Buffer, which contain queue monitors not explicitly shown in the figure. Since the three components (TFR/RSLT Cache, TFR Seq. Buffer) need to process all of the TFR or RSLT data, their security levels are the *trusted* level ($[\top, \top, \top]$).

B. SYSTEM EVALUATION

In this section, the system performance evaluation is proposed, including *in-container* and *stand-alone* environments. For in-container environments, each component is deployed in an individual container, which can communicate with other components via a socket. For stand-alone environments, the components are running on the single machine, whose intercommunication depends on interprocess communication.

In addition, our evaluation consists of three security constraints, including non-security (*NON_SEC*), low-security (*LOW_SEC*) and high-security (*HIGH_SEC*). For *NON_SEC*, all the components have a FIFO queue to process TFRs and RSLTs without a security level. *LOW_SEC* provides security-level-aware queue to process TFRs and RSLTs in each components. Particularly, multiple queues with different security levels for processing various TFR are adopted in TPU. For *HIGH_SEC*, in addition to the security guarantees in *LOW_SEC*, the *shadow address* is adopted in each queue; thus, the real memory addresses are protected while accessing to queues or queue elements.

We conducted our evaluation on a computer with a quad-core CPU (Intel Q8400), 8GB RAM (DDR3) and 500GB hard disk (7200rpm). Both of the host and the container operating systems are Ubuntu 16.04.1 x86_64. Our test data are 10000 TFRs with random security levels in 3, 5, 10, 15, 20, 25, 30 dimensions. We will evaluate and discuss the time consumption of TFR/RSLT processing in different

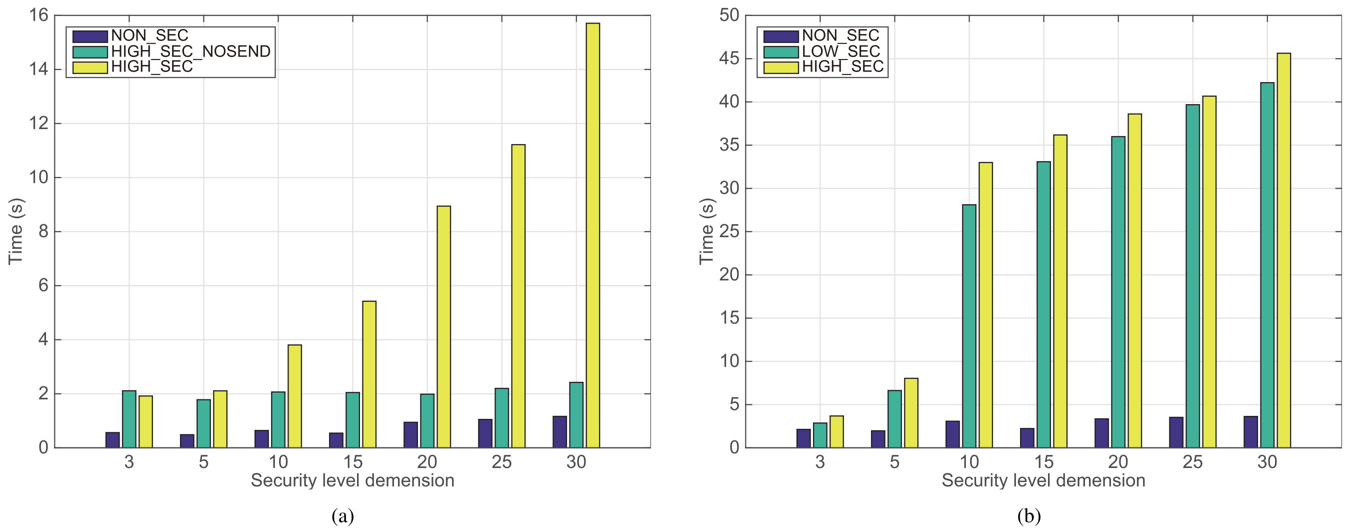


FIGURE 6. **(a) Time consumption of TFR processing in TFR Cache; (b) Time consumption of TFR processing in TPQ.**

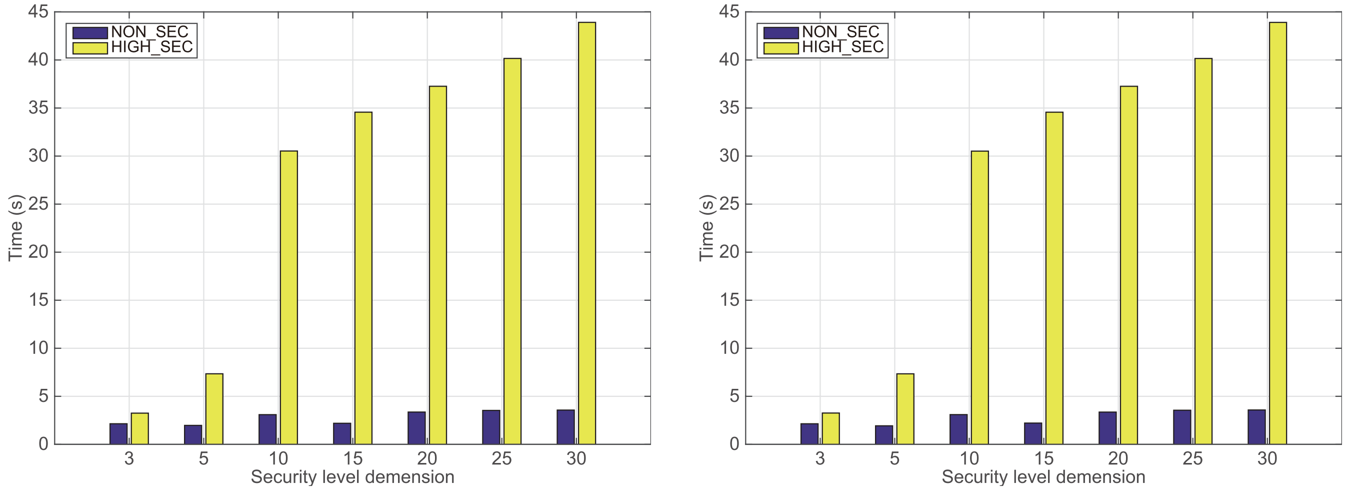
components as well as in the whole process. To focus on the performance of the primary four components (TFR Cache, TPQ, TFR Seq. Buffer and RSLT Cache), the perfomance in terms of TFR processing of TPM is ignored. In addition, the time consumption in our evaluation also contains semaphore operations, memory allocation/release, etc., which exists in each component and each security constraint. Hence, the extra time overheads will not affect our evaluation.

C. IN-CONTAINER ENVIRONMENT

Figure 6 (a) shows the time consumption of TFR processing in the TFR Cache. Note that the TFR/RSLT Cache and TFR Seq. Buffer only have one queue to cache TFRs/RSLTs, which have to process data with any security levels; hence, they are assigned to the top security level. Thus, for the three components, there are only two security constraints, *NON_SEC* and *HIGH_SEC* (*HIGH_SEC_NOSEND* indicates that the cached TFRs are fetched from the TFR Cache, but not to be sent to the TPQ). For the *NON_SEC* case, the time consumption experiences no significant change (about $0.2s \sim 1.0s$) with the increase of in the dimension of the security level. However, it takes more time (about $1s \sim 1.5s$) for *HIGH_SEC_NOSEND* to process TFR compared with *NON_SEC*, which is caused by the conversion from *shadow address* to *real address*. For *HIGH_SEC*, it has a similar time consumption to *HIGH_SEC_NOSEND* does in the first two dimensions (3 and 5). However, from dimension 10, the time consumption has a distinct increment, and reaches nearly 16 seconds at dimension 30. The increasing time overhead is primarily caused by the TPQ which has limited processing capacity (shown as figure 6 (b)). With the increase in the security level dimension, TPQ has to queue the TFRs according to their security levels, which is a time-consuming process, in particular, when the address conversion is involved. In addition, the higher dimension of security level also increases the amount of data (TFR data) being transmitted via the socket, which has finite-size buffer. TPQ has to finish handling the data in the current buffer before it can continue to receive data from the TFR Cache (in our experiment, the TFR is emitted at an extremely fast rate; hence, the socket buffer can be quickly filled). This also reduces the performance of the TFR Cache.

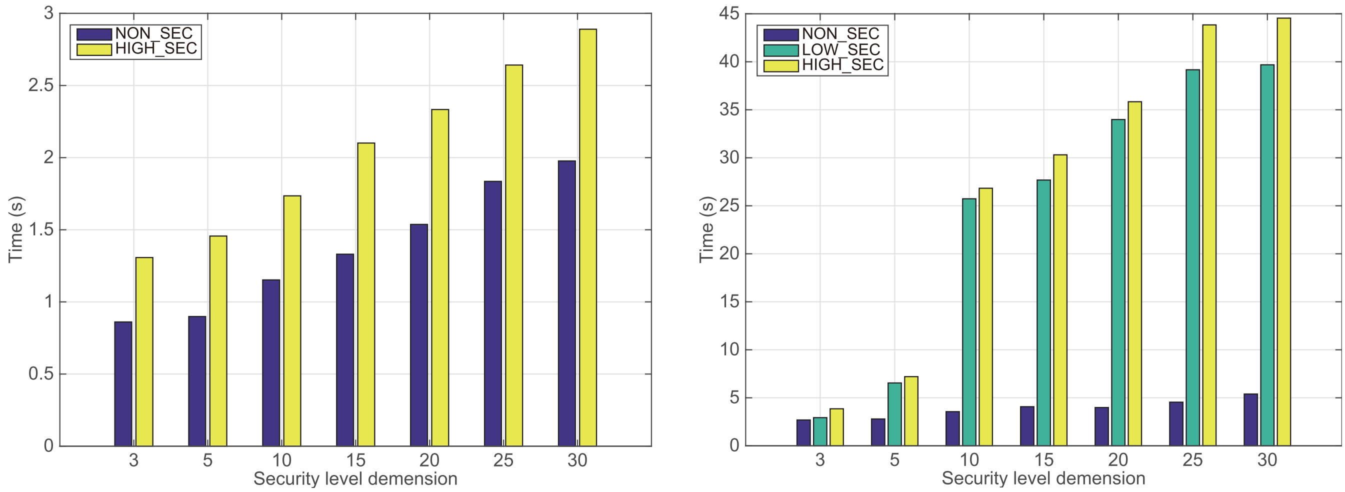
Shown as figure 6 (b), for *NON_SEC*, similar to figure 6 (a), the time consumption does not change significantly due to no address conversion, operations on multiple queues or security level comparison. The dimension does not distinctly affect the performance. However, *LOW_SEC* and *HIGH_SEC* spend more time on TFR processing than *NON_SEC* does. Particularly, from dimension 10, there is a noticeable increase in time consumption, which lasts until dimension 30 (about 46 seconds). According to the discussion about figure 6 (a), the time consumption is mainly due to the limited TPQ processing power, including address conversion between *shadow address* and *real address* and the increase in data amount in the socket buffer etc. Moreover, the difference in time consumption between *LOW_SEC* and *HIGH_SEC* is mainly due to address conversion.

As shown in figure 7 (a) and (b), since there is no security mechanism, the time consumption of *NON_SEC* does not change significantly in all security level dimensions. However, for *HIGH_SEC*, the increment of time consumption is also caused by the security operations, including address conversion, security level comparison etc. In addition, both *NON_SEC* and *HIGH_SEC* in figure 7 (a) and (b) show the same trends as that of TPQ (figure 6 (b)). This is also caused by the limited processing power of TPQ. TFR Seq. Buffer has to wait until TPQ finishes processing TFRs and sends them out. Thus, the performance of TFR Seq. Buffer is greatly affected by that of TPQ. Moreover, the time consumption of TFR processing of the TPM is also subject to the TFR Seq. Buffer. Since the TPM runs in batch mode, the output is also serial. This will transfer the performance of



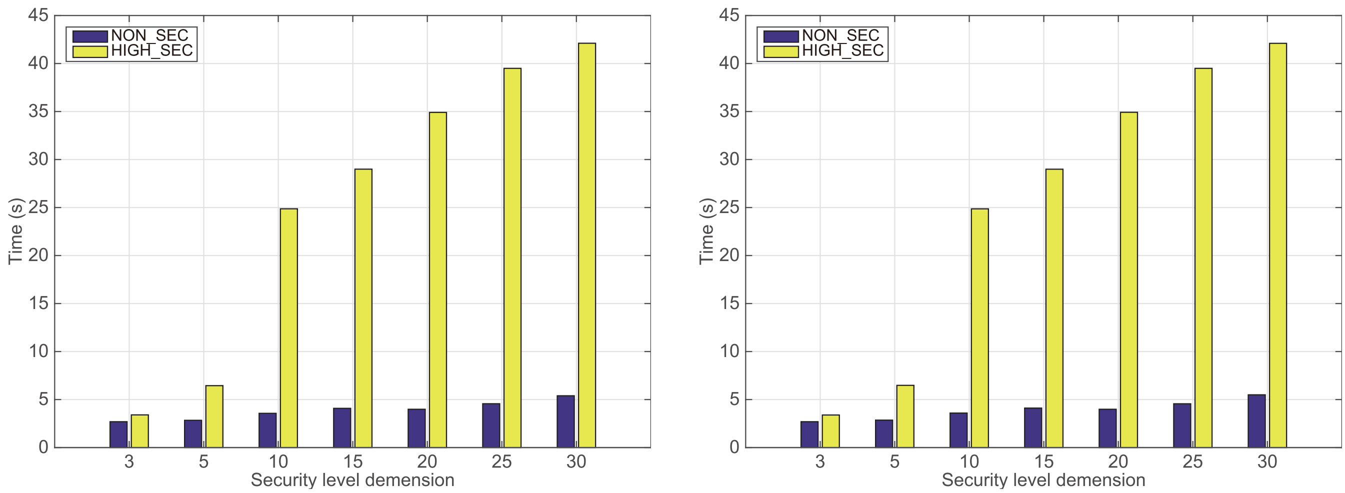
(a) (b)

FIGURE 7. **(a) Time consumption of TFR processing in TFR Seq. Buffer; (b) Time consumption of RSLT processing in RSLT Cache.**



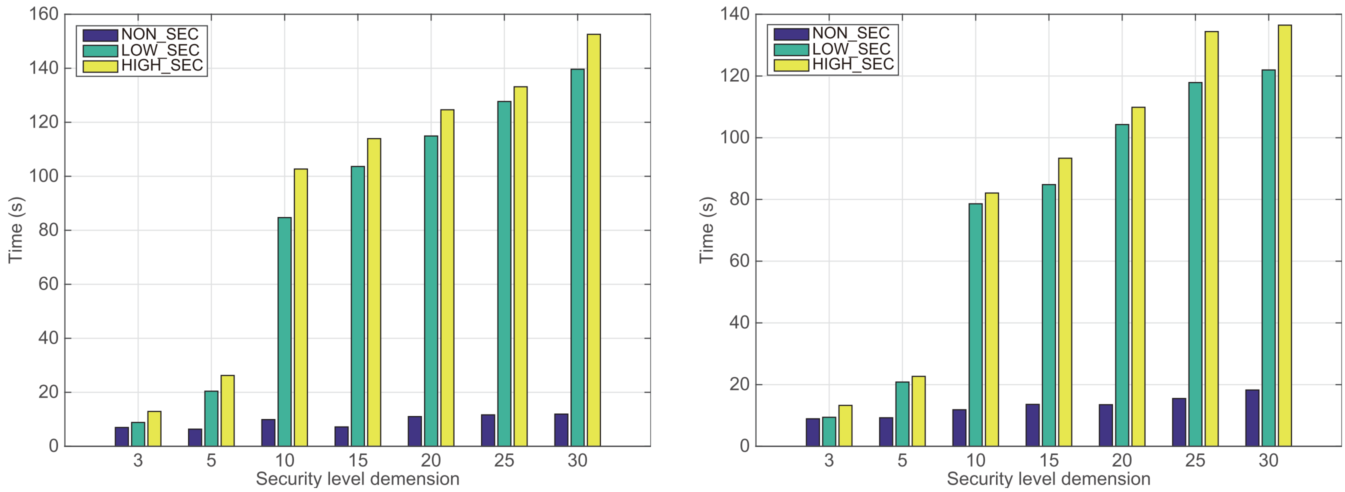
(a) (b)

FIGURE 8. **(a) Time consumption of TFR processing in TFR Cache; (b) Time consumption of TFR processing in TPQ.**



(a) (b)

FIGURE 9. **(a) Time consumption of TFR processing in TFR Seq. Buffer; (b) Time consumption of RSLT processing in RSLT Cache.**



(a) (b)

FIGURE 10. **(a) Total time consumption in in-container environment; (b) Total time consumption in stand-alone environment.**

TFR Seq. Buffer to the RSLT Cache, which makes the two figures (figure 7 (a) and (b)) show nearly the same trends.

D. STAND-ALONE ENVIRONMENT

Figure 8 (a), (b) and figure 9 (a), (b) show the time consumption of the system with different security constraints and security level dimensions in a stand-alone environment. In this scenario, all the components are processes in the same operating system. Compared with the in-container environment, we can find some important similarities:

1) For the four components (TFR/RSLT Cache, TPQ and TFR Seq. Buffer), the time consumption does not change drastically with the constraint *NON_SEC* at any security level dimensions. This indicates that without security, the system performance is not significantly affected by the dimension of security level.
2) Both TFR Seq. Buffer and RSLT Cache consume nearly the same time in processing TFR at any dimensions; this is also caused by the delay in security operations in TPQ (refers to the discussion about figure 6).

Moreover, in figure 10 (a), the time consumption of *HIGH_SEC* has small increases from dimension 3 ∼ 30, which is in sharp contrast to figure 6 (a); in stand-alone mode, the components are processes whose intercommunications do not depend on the virtual network interfaces of the containers, and the components can directly use local memory without being limited by the container's memory capacity. Thus, when the amount of data increases with the dimensions, the TPQ will not delay on receiving and processing a large number of TFRs, and thus TFR Cache must wait.

Figure 10 shows a comparison of the total time consumption for the in-container and stand-alone environment. Although in-container mode consumes more time than stand-alone mode, it provides stronger isolation for the components and is therefore a more secure mechanism compared with stand-alone mode (shared memory). In addition, the increased more time cost of in-container mode does not affect its application. For example, at dimension 30, for *HIGH_SEC*, in-container mode consumes only 16 seconds more than the stand-alone mode, which can still be accepted in the real applications for a more secure guarantee. Moreover, in reality, the speed of the TFR generation is not as fast as in our experiment, and the security level dimensions are also not as high; hence, TPQ has enough time to deal with TFRs. In such circumstances, the system has good usability based on the premise of ensuring security.

VI. CONCLUSION

In this paper, we establish a secure scheme for trusted hardware sharing systems (THS), which protects the sensitive data in trusted function requests (TFRs) from being leaked, disclosed and modified in an unauthorized manner by malicious software or users. In our scheme, we first build a security level model for the THS system based on an information flow model. Then, the TFRs are assigned different security levels according to their owners (such as applications, services) and processed in isolated environments with different security levels. This mechanism enhances the security of TFR processing under the THS system in stand-alone and in-container environments. We have developed a prototype, and our experiment results show that, in the case of a large number of high-rate concurrent TFRs, the isolation environment and the increasing security level dimension will lead to the degradation of system performance. However, in the real world, since the rate of TFR generation is lower than that of the experimental environment in most of the use case, the delay caused by the security mechanism will not seriously affect the usability of the system.

REFERENCES

[1] W. Arthur, D. Challener, and K. Goldman, *A Practical Guide to TPM 2.0*. New York, NY, USA: Springer, 2015.

[2] P. Parkinson and A. Baker, "High assurance systems development using the MILS architecture," Wind River Systems, Inc., Alameda, CA, USA, White Paper, 2010.

[3] *Two Architecture Approaches for MILS Systems in Mobility Domains (Automobile, Railway and Avionik)*, Zenodo, Genève, Switzerland, Jan. 2015.

[4] P. J. Parkinson, "Applying MILS to multicore avionics systems," in *Proc. Int. Workshop Mils, Archit. Assurance Secur. Syst. (HIPEAC)*, 2016, pp. 1–9.

[5] S. H. VanderLeest and D. White, "MPSoC hypervisor: The safe & secure future of avionics," in *Proc. IEEE/AIAA 34th Digit. Avionics Syst. Conf. (DASC)*, Sep. 2015, pp. 6B5-1–6B5-14.

[6] D. Muench, M. Paulitsch, and A. Herkersdorf, "IOMPU: Spatial separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non-transparent bridges," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun., IEEE 7th Int. Symp. Cyberspace Safety Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, Aug. 2015, pp. 1037–1044.

[7] Z. Yu, W. Zhang, and H. Dai, "A trusted architecture for virtual machines on cloud servers with trusted platform module and certificate authority," *J. Signal Process. Syst.*, vol. 86, pp. 327–336, Mar. 2016.

[8] M.-J. Sule, M. Li, G. A. Taylor, and S. Furber, "Deploying trusted cloud computing for data intensive power system applications," in *Proc. IEEE 50th Int. Univ. Power Eng. Conf. (UPEC)*, Sep. 2015, pp. 1–5.

[9] J. Wang *et al.*, "POSTER: An E2E trusted cloud infrastructure," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1517–1519.

[10] W. Feng, D. Feng, G. Wei, Y. Qin, Q. Zhang, and D. Chang, "TEEM: A user-oriented trusted mobile device for multi-platform security applications," in *Proc. Int. Conf. Trust Trustworthy Comput.*, 2013, pp. 133–141.

[11] C. Chen, H. Raj, S. Saroiu, and A. Wolman, "cTPM: A cloud TPM for cross-device trusted applications," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2014, pp. 187–201.

[12] H. Raj *et al.*, "fTPM: A software-only implementation of a TPM chip," in *Proc. 25th USENIX Secur. Symp. (USENIX Security)*. Austin, TX, USA, Aug. 2016, pp. 841–856.

[13] S. Proskurin, M. Weiß, and G. Sigl, "seTPM: Towards flexible trusted computing on mobile devices based on GlobalPlatform secure elements," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.*, 2016, pp. 57–74.

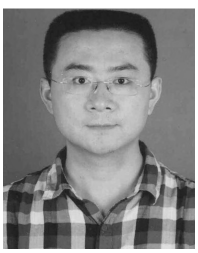
[14] A. Constantin, J. Cook, A. De, A. Constantin, J. Cook, and A. De, "Trusted hardware for partitioned multicore," in *Encyclopedia of Database Systems*. New York, NY, USA: 2009, pp. 3191–3192.

[15] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[16] R. S. Sandhu, "Lattice-based enforcement of chinese walls," *Comput. Secur.*, vol. 11, no. 8, pp. 753–763, 1992.

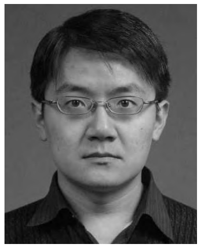
[17] M. Helsley, "LXC: Linux container tools," IBM DeveloperWorks, New York, NY, USA, Tech. Library, 2009.

[18] *Linux Container*, accessed on Jun. 9, 2017. [Online]. Available: https://linuxcontainers.org



DI LU received the B.S., M.S., and Ph.D. degrees from Xidian University, China, in 2006, 2009, and 2014, all in computer science and technology. He is currently a Lecturer with the School of Computer Science and Technology, Xidian University. His research interests include cloud computing, system, and network security.

JIANFENG MA received the B.S. degree from Shaanxi Normal University in 1982 and M.S. and Ph.D. degrees from Xidian University in 1992 and 1995, respectively, all in computer science. He is currently a Professor with the School of Computer Science and Technology, Xidian University. He has authored over 150 journal and conference papers. His research interests include information security, cryptography, and network security.



CONG SUN received the B.S. degree in computer science from Zhejiang University, in 2005, and the Ph.D. degree in computer science from Peking University, in 2011. He is currently an Associate Professor with the School of Computer Science, Xidian University. His research interests include information flow security and program analysis.

QIXUAN WU received the B.S. degree in computer science from Xidian University, in 2015, where he is currently pursuing the degree with the School of Computer Science. His research interests include trusted computing and dynamic fault tree analysis.

ZHAOCHANG SUN received the B.S. degree in computer science from Xidian University, in 2015, where he is currently pursuing the degree with the School of Computer Science. His research interests include trusted computing and runtime verification.

NING XI received the B.S., M.S., and Ph.D. degrees from Xidian University, China, in 2008, 2011, and 2014, all in computer science and technology, all in computer science and technology. He is currently a Lecturer with the School of Computer Science and Technology, Xidian University. His major research is in home network, service computing, and network security.