IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 12, 2020, accepted August 29, 2020, date of publication September 1, 2020, date of current version September 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3021033

RAitc: Securely Auditing the Remotely Executed Applications

LEI ZHOU[1], ENTAO LUO[2], AND GUOJUN WANG[3], (Member, IEEE)
[1]School of Computer Science and Engineering, Central South University, Changsha 410083, China
[2]School of Electronics and Information Engineering, Hunan University of Science and Engineering, Yongzhou 425199, China
[3]School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou 510006, China

Corresponding author: Guojun Wang (csgjwang@gzhu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61632009, in part by the Guangdong Provincial Natural Science Foundation under Grant 2017A030308006, and in part by the High-Level Talents Program of Higher Education in Guangdong Province under Grant 2016ZJ01.

ABSTRACT One of the most important security challenges in remote computing (e.g., cloud computing) is protecting users' applications running on the service platform from malicious attacks. Because remote users have little control over the platform, a malicious platform manager or platform-sharing guest acting as an adversary can easily create an untrustworthy execution environment. Prior studies have leveraged trusted third party (TTP)-based and trusted execution environment (TEE)-based approaches to mitigate such security issues, but these approaches still provide little transparency from the user's perspective. To address this challenge, we present a remote auditing approach based on an identified trust chain (RAitc) to analyze the correctness of remotely loaded applications. The chain is constructed with two goals: the first is to identify the remote platform to ensure that the user has a designated service system; the second is to build a trust chain from the user to the designated platform via verifiable computing-based module measurements and kernel-based application auditing. RAitc achieves a higher guarantee of safety in securely monitoring and verifying the integrity of remote applications executed by users. In addition, RAitc is both easier and more flexible for the extension of the trust base. Our implementation of RAitc protects users' remote execution environments while requiring an acceptable overhead on the target system in application auditing. We rigorously and comprehensively evaluated the effectiveness and performance of RAitc. The results show that RAitc performs effectively and has acceptable resource consumption.

INDEX TERMS Auditing, trust chain, identification, verifiable computing.

I. INTRODUCTION

Network computing nodes, such as the cloud, are intended for transfer of computational tasks from a computationally weaker client (i.e., a mobile device) to a remote powerful worker (i.e., a remote workstation or cloud server that provides services). This approach effectively improves the utilization ratio of computational resources and the quality of client services. However, the clients, who have less control over remote computing processing, may be suspicious of the results from a remote worker. A malicious platform-sharing attacker or a compromised worker can manipulate the computing result by attacking the remote computing platform [1], [2]. These compromising situations are reflected in the following aspects: the user's designated benign system might be replaced by a system with reduced performance or by a malicious system, and the user's designated application might be altered by malware.

To avoid such malicious remote services, users can either passively trust the platform, supported by security-enhanced workers [3]–[5], or actively seek to validate the remote service environment [6], [7]. To enhance cloud computing security, providers such as IBM and Huawei have proposed a series of security rules based on existing protection protectionmechanism—e.g., a network intrusion detection system (NIDS) [8] and distributed antivirus software [9]. However, such security mechanisms are deployed at the kernel level in the remote target system, which may already be compromised by untrusted platform providers. Because the users have little ability to audit the correctness of the

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

security mechanisms in the target system, providers have higher privileges regarding direct target system manipulation.

Thus, users seek assistance from trusted third parties (TTPs) and trusted execution environment (TEE) technologies so that they can actively assess the trustworthiness of remote services. For example, Zhang and Lee [10] proposed a new way to ensure module safety in an untrusted entity by introducing a TTP for data integrity verification and service validation. The basic TEE approaches generally use hardware or hypervisor protection to construct isolated environments to protect sensitive computing. Popular hardware features are introduced into commercial machines as transparent and trusted computing bases due to the secure and isolated execution environment—e.g., Intel Management Mode (IME) [11], Intel System Management Mode (SMM) [12], and TrustZone [13]. However, the hardware-assisted approaches [14] present the problem of addressing the semantic gap between hardware and software, which requires cooperative technology [15] to construct a secure channel between the TEE and the target system to help transfer and analyze the software data in the TEE without leaking sensitive data. Hypervisor-based approaches provide a virtual TEE with high privileges and isolation from the target virtual machine (VM) system. However, the virtual trust bases are extremely large, which causes more complex vulnerabilities [16].

In most remote outsourced computing situations, users expect a correct computing result from a designated platform with QoS assurance [17]. Therefore, researchers have developed verifiable computing (VC) approaches to verify the correctness of remote service results that do not attest to the security of the entire remote execution environment [18], [19]. This approach reduces the excessive overhead required for TEE maintenance. However, the existing VC approaches function effectively only for simple computations (i.e., single functions) and exhibit reduced performance with respect to executing general user applications. Additionally, VC (especially the noninteractive zero knowledge-based VC mechanism [20]) cannot solve the remote platform identification issue because the verifier generally possesses little information on the target device [21].

To address the above challenges, we trust part of the hardware existing in the target devices (Intel SMM on the x86 platform in our paper) to identify a designated system through identity cross-checking [22]. Note that Intel SMM, as the isolated execution environment, can elucidate the entire target application but requires a system switching which will halt the target host. Additionally, SMM-based target checking cannot effectively and synchronously implement dynamic checking. Fortunately, VC approaches are only one solution to dynamic checking of runtime function in remote devices even without TEE support. Thus, we propose a prototype system, RAitc, that uses an SMM-based module to identify the remote computational environment and implements a trust auditing chain between the user and identified system to ultimately check the loaded application. Our contributions are as follows:

- We propose a trusted remote application auditing system to verify the correctness of users' remote services that does not require trusting the remote OS.
- We leverage the Intel SMM in the target machine to assist with target device identification, which ensures that services are executed in the user-designated system.
- We leverage the VC mechanism to create a remote trust chain to verify the integrity of the auditing module in the target kernel, which protects the auditing function from malicious changes.
- We propose the RAitc prototype, along with practical experiments to examine the system's performance and effectiveness. The results show that RAitc effectively improves the ability of users to verify remote services.

The paper is organized as follows. We introduce the related works in section II. We discuss the covered technologies in section III, and illustrate the threat model and assumptions in section IV. Then we describe the overview of RAitc by introducing each component and the functions in section V. In addition, we implement the details in section VI to describe the key technologies and workflow of RAitc operation. We analyze the security of RAitc in section VII and evaluate the prototype in section VIII. Finally, we wrap up with the conclusion in section IX.

II. RELATED WORK

In this section, we describe the related work of RAitc. The topic of this work is related to several research areas and we illustrate as follows.

Trusted Execution Environment. Response to the security concern on computer computation, hardware manufactures, software providers, and researchers seek to design a trusted execution environment for the users. It commonly includes hardware-supported and virtualization-based TEE. The hardware-supported TEEs [23]–[25] directly build a physical isolated platform for secure-sensitive code execution. Such TEEs use existing hardware properties like Intel SGX, SMM, IME, and ARM TrustZone [26] to avoid extra hardware. Virtualization-based TEEs are suitable for cloud and other network computing, where the user's system is created based on the hypervisor and container [27], [28]. The former is more secure due to the physical isolation between the trusted and untrusted parties, but the vulnerabilities existing in its firmware and user interfaces will weaken the isolation. The latter is more flexible to develop in different platforms without considering the hardware difference. However, the large trust base on virtualization-based TEEs causes more bugs and overhead. Therefore, some researches proposed enhanced methods like vTZ [29] by joining the advantages of above two mechanisms.

Remote-assisted Malware Detection. Kernel-based malware detection is commonly used to protect enterprises and users' application [30], [31]. User applications generally call

kernel-based functions when executing, thus, kernel function based monitoring (e.g., system call interposition) can verify the runtime application. During a software execution, the corresponding events like thread creating, memory reading/writing, can be intercepted and checked. Unfortunately, kernel-level or higher-level attacks [32], [33] can bypass and compromise the above-mentioned detection, which means kernel-based detection itself should be protected as well. However, only local TEE-based protection may not completely trustworthy because a remote user has little control over the service platform.

Remote-assisted protection is necessary for enhancing the security of kernel-based detection. Previous researches generally use the memory snapshot [34] or other side-channel methods [35] to fetch memory data and analyze it in a remote trust server. It is only static malware analysis that cannot monitor the target's behavior [36]. Also, the semantic gap existing in binary code can reduce the accuracy of detection [37], [38]. Thus, an effective way to protect the target's detection is to remotely monitor the kernel-level detection module but not only binary analyzing. The fully homomorphic encryption [39] is the closest approach to achieve the goal but far from practical. Some researches like *Pinocchio* [40] can verify the correctness of simply remote function, but hard to handle complexity software. However, we can leverage the advantages of both local TEE and remote attestation to create a flexible and trusted kernel-based malware detection.

Mechanism of Cleanroom Secure Service. The *cleanroom* model [41] is an ideal model, where the goal is to maintain a trusted execution environment for remote users. It defined such a network service model: the service store, denoted as *SS*, provides designated applications for the user's execution environment (*UEE*), including the remote server platform (*SP*) and local terminal (*UT*). It designed a cleanroom protocol to guarantee trusted collaborative computing between *SP* and *UT* if the protocol satisfies:

- *SS* is a trusted party and provides trust applications for both *SP* and *UT*, and
- computational resources dynamically deployed in *SP* can not be tampered by the dishonest *SP* manager, and
- computational resources dynamically deployed in *UT* can not be tampered by the malicious user, and
- users achieve trusted service through executing the legal application on the designated system.

Here both *SP* and *UT* are untrusted. If an application loaded from $NON - SS$ to *UEE*, it betrays the cleanroom protocol. To support such a protocol, the solution is to create a secure container along the untrusted platform with stronger isolation [26], [42]. Besides, the communication between each party goes through the secure channel [15], [43], [44]. However, this model is ideal and hard to implement in real computation due to existing hardware and software vulnerabilities [39], [45], [46]. Fortunately, we can build a practical model to perform approximate effect by trusting part of hardware components. That means we can use the TEE to identify the applications' execution environment [26], [27], [47], and the application is further checked by the remote attestation mechanism.

III. TECHNOLOGIES

In this section, we cover the background of trust-chain-based verification and the technologies involved in our design, including Intel SMM and remote VC.

A. TRUST CHAIN-BASED VERIFICATION

The trust chain was introduced by the Trusted Computing Group (TCG) [48] and first proposed to safeguard OS bootstrapping. The main operations of the trust chain gradually verify the correctness of the loaded module at each system level. It leverages the local hardware module [46] or hypervisor [49] as the root of trust (RoT). That is, a trust base exists at each step of trusted service attestation, and each node in the chain can be verified as trustworthy with the support of prior trust node.

The local trust chain works at high speed and has only small overhead due to hardening technologies but still exhibits many limitations. One factor is security: for example, the trust platform module (TPM) hardware-assisted trust chain, which generally functions as a BIOS guard during OS load checking, depending on a static workflow to verify and load the next OS module. The executed program is also coded and stored in firmware; thus, it can be blocked by malicious BIOS tampering or bootkits [50]. Furthermore, if the local hardware or hypervisor RoTs are under attack, then the base of chain might be compromised [51], leading to verification errors. Another factor is flexibility: to ensure trustworthiness, users have fewer interfaces for accessing the trust base. Thus, for complex application executions, the conventional trust chain cannot meet the strong dynamic verification requirements. Some researchers have extended the local trust chain-based verification to remote services [52]. Remote verification can then effectively mitigate the verification error caused by the compromised local chains. Furthermore, the user, acting as the verifier, can audit the remote computing platform.

B. SYSTEM MANAGEMENT MODE

SMM [53] is a special-purpose operating mode on Intel x86 platforms for which execution is similar to the available real and protected modes. SMM was designed to handle system-wide functions such as power management, system hardware control, or proprietary OEM-designed code. SMM is a hardware-guaranteed isolated environment that runs in protected mode including runtime user conveying systems. Compared with TPM-based trust attestation mechanisms, users can implement more complex and dynamic verifying mechanisms in SMM to check the target OS. Because SMM and the host share the same CPU cores, a checker can more easily obtain the host runtime information needed to solve the semantic gap between the memory and application state, which is a difficult issue for other hardware TEE-based approaches. Another advantage of SMM-based

security approaches is that they can retain the consistency of the host runtime state despite short pauses. In our paper, we leverage SMM to fetch the CPU identification in a trusted manner to enable further trust-chain construction.

C. REMOTE VERIFIABLE COMPUTING

In contrast to constructing a TEE for users, VC-based remote attestations focus only on guaranteeing the correctness of the outsourced computation. However, VC is more flexible with respect to verifying additional content in the service requirements of different users. Typical VC approaches, such as *Zaatar* [54], *Pantry* [55] and *zkSNARKs* [56], all construct circuit primitives for complex computations. These primitives allow an external entity (the verifier) to obtain assurance that an arbitrary segment of code, called by the target system, can be executed without tampering by any malware that may be present on an external computing device. In addition, primitive computing operations have achieved mathematical proof of execution completeness and correctness.

Most VC-based attestation approaches face difficulties when processing complex applications while also providing acceptable performance overhead. However, some studies (e.g., *Pinocchio* [40]) have proposed a practical VC mode that can effectively verify small signal functions, including mathematical formulas and hash algorithms. However, designing the corresponding logical analog circuit equations for further attestation on complex software is difficult.

In this paper, we leverage local trusted hardware (Intel SMM) and remote VC to construct a user-designated practical trust chain that can effectively audit users' remote applications in a trustworthy manner.

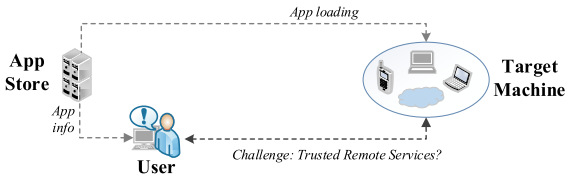

FIGURE 1. How can we trust a remote computing?.

IV. THREAT MODEL AND ASSUMPTIONS

A user who seeks to acquire correct computation services from a remote platform has little control over the remote devices or even over the OS. Figure 1 shows a model that users are unable to identify whether the right application is running on the correct platform and where the application—or even the platform—might be compromised by malicious managers or platform-sharing malevolent attackers [1], [2]. Therefore, in remote computing services, there are two main challenges: 1) how to ensure that the computing platform is the one that was designated and 2) how to ensure the correctness of application execution.

To simplify the model, we assume that applications loaded from the application store are trusted. We also assume that the communication channel used to load the application is safe by using extra secure technology, such as TLS [44]. Beyond that, we assume that the hardware and software providers do not implement collusion attacks because we trust parts of the hardware-supported components (using SMM) on the remote device.

In our threat model, an adversary is capable of attacking the target system, including the user's application and OS kernel; even worse, the adversary can relay the service execution to a malicious platform. Thus, the adversary can insert a malicious process and compromise the user's normal application process. We assume that the attacker cannot destroy the hardware system but that it can gain full control over the platform's software by exploiting kernel-level vulnerabilities. Thus, the user service results might be compromised, which requires appropriate actions.

V. SYSTEM ARCHITECTURE

The goal of RAitc is to effectively audit remote applications in a trustworthy manner to provide *clean* services for remote users. Figure 2 shows an overview of the auditing mechanism. To securely audit remote services, we construct an identified trust chain from the user to the target system. This construction process involves two key technologies: target identification and verifiable remote attestation. The former identifies whether the target machine is the user's designated platform, while the latter works to build an attestation chain from the user to the auditing module in the designated target system. Based on the identified trust chain, RAitc can trusty audit the runtime application. We illustrate each component and its specific functions below.

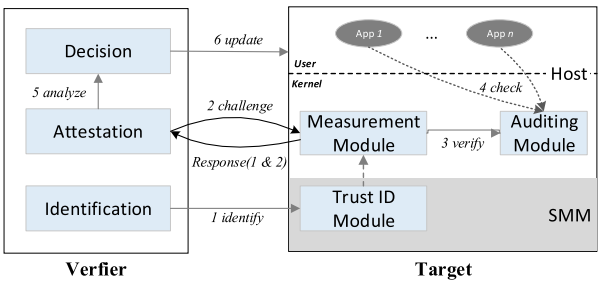

FIGURE 2. The overview of RAitc system's architecture.

A. SYSTEM COMPONENTS

In our approach, we assume that the user (verifier) is fully trusted, and RAitc also regards the Intel SMM in the target device as trustworthy. By using these tow trust bases, it cooperatively constructs the identified trust chain for user application auditing.

Verifier. We assume that each user owns a trustworthy and secure device for checking a remote target system (e.g., a mobile phone or personal laptop), denoted as *verifier*. We also assume that physically secure mechanisms and legal systems exist to protect the verifier from outside attacks. The verifier consists of three parts: an identification unit,

an attestation unit, and a decision unit. The identification unit is designed to contact both the SMM process and the user's runtime system in the target machine to obtain and cross-compare the identities from two isolated methods. The attestation unit is designed to execute the verification protocol. Combined with VC approaches, the verifier challenges the correctness of the measurement module execution and the integrity of the auditing module. The verification result is then returned to the user for further analyzing. The decision unit implements the signing process for the clean service protocol between the remote service provider and the users, and it stipulates the rules regarding what software are allowed to run in the users' computational systems: neither the service provider nor the users can change this software during execution, which maintains a clean runtime state for the remote users.

Target. The target system resides on a powerful machine that provides extra computational resources to other users; the target system might be a workstation or personal computer with a high-performance configuration. We trust its assembled hardware components, which are developed by an upscale manufacturer with closed technology. We use Intel SMM in our system, which is isolated from the host system. In addition, we assume that no collusion attacks are generated by the service provider or by hardware vendors. In our design, we trust SMM but regard the entire operating system as an attack risk. We develop a Trust ID module that is inserted into the SMM handler; this module provides a trusted target identity for the users. A kernel-based auditing module is employed to detect system interruptions and verify the integrity of the loaded application. A measurement module is developed based on verifiable computation. The security is ensured by constructing a trust chain from the verifier to the target; in contrast, previous researchers constructed a chain only from the local hardware chip (regarded as the TPM) to the OS. We design a software-based attestation mechanism to verify the functions in the target system. RAitc implements a small kernel-based auditing module and a VC-based measurement module to create a trusted chain to verify the auditing module.

Assistance. Some entities provide assistance to support remote services. One example involves applications loaded from a trusted application store. In addition, abstract information is first sent to the verifier.

B. EXECUTION WORKFLOW

Secure system initialization is an essential requirement of RAitc. Before system execution, we first deploy the necessary trusted modules to the verifier and target machine under the assumption that no attacks occur during the initiation stage. Below, we describe the workflow with respect to secure auditing operations for remote applications. The details of each step are shown in Figure 2.

- First, a user and a service provider both sign a clean application protocol. The requested application is loaded from a trusted application store, which can be checked by both the verifier and the target system. Note that we first consider the consistency of the application from the application store but not whether the application is malicious.
- Second, the verifier identifies the remote system by cross-checking identification attributes of the target system (step 1). By cross-comparing the identity obtained from the SMM and the extracted host identity (steps 2 & 3), the user can ensure that the designated platform is loaded for remote service rather than a fake platform.
- Third, the verifier executes the VC session to check the integrity of the auditing module using trusted remote memory access and hash computing. The security is checked using mathematical proofs of verifiable computation. Using these (steps 2 & 3) operations, users can deploy a secure auditing module in the target system.
- Fourth, the loaded application is checked (step 4) by the auditing module, and the result is returned to the verifier for further analysis (step 5). Based on the verification result and any new user requirements, the users can dynamically update (step 6) the application execution protocol.

In RAitc, the trusted parties include the verifier device and the SMM in the target device. We assume that network communications between the verifier and target are secure by using the TLS protocol. The identification and trust chain constructed of trusted parties can then provide secure auditing services for remote user application execution.

VI. IMPLEMENTATION

In this section, we describe the implementation of RAitc. We focus on the design of each component and the details of the key technologies employed in our system.

A. CLEAN REMOTE SERVICE PROTOCOL

We define ''Clean'' as a pure application running on a specific system. We first request a remote platform CSP_r for remote computation, and verify that device CSP_c is created as requested. We whitelist the designated applications $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$, where $\mathcal{H} = \{H_1, H_2, \ldots, H_n\}$ is a corresponding identifier for each application, and n is the maximum number of applications. $\mathcal{A}$ is identified by the users. The verifier calculates the application's identity $H_i = hash(\alpha_i^1, \alpha_i^1, \alpha_i^2, \ldots, \alpha_i^k)$, where $\alpha_i^j, j \in [0, k]$ represent the information features from A_i, and k is the maximum number of application features.

To maintain correct protocol execution, we implement the identification operation to ensure that the protocol is deployed on the right platform; then, we build the trust chain to verify whether the executed application satisfies the signed application protocol. The details are provided in the following section.

B. HARDWARE-ASSISTED IDENTIFICATION

The quality of the computation platform is a key factor in ensuring the QoS of computing services. However, a malicious provider can change the user-designated platform when remote services are applied and executed via a cuckoo or MIMT attack [57]. One example involves replacing a high-cost platform with a lower-cost hardware and software configuration.

Suppose that a remote computer offers two execution environments: an SMM-based trusted mode (ST) and an untrusted mode (UT). ST is generally constructed with less hardware and software, constituting a small trusted computing base (TCB) that we trust in current model. UT is the system used to execute the user's remote task, while UT_d is the predesignated remote computation system. UT_f is a forged remote computation system. Therefore, before application execution by UT, users should first ensure that the remote system is the one configured by the users. SMM is a trusted model based on isolated memory that cannot be accessed by the host system. We assumed that the process of running in SMM is trusted. Furthermore, we preconfigure the network interface register to ensure that SMM can be automatically triggered through a specific network package [58].

Identity Extraction. We insert the Trust ID module into SMM to identify the host OS. The program is designed to read the hardware identity in ST, including the CPU ID and NIC ID. The hardware identity running in ST has two features: (1) the identities are difficult for an adversary to change, and (2) the extraction program can remain trusted because it cannot be compromised by an adversary from UT. This identity information is sent to the verifier. The verifier generates a random number to SMRAM and then copies it to host memory, after which number is used to calculate the host identification with the host-extracting hardware identity.

Algorithm 1 illustrates the identity cross-checking workflow. The unique hardware ID is configured by the hardware vendors and compared to the ID securely fetched from the remote platform. In summary, the verifier can identify (1) which device was provided for the user and (2) whether the services provided by the designated OS will be executed on the corresponding hardware platform. Evaluating the time required for SMM-based identity extraction and host-based identity extraction is the key to determining whether a relay attack exists in the identification operation.

Time Verification. If an adversary fakes the extraction process but provides a matching hardware ID, then the random number is also redirected to the adversary's computer to calculate the final identification: this requires yet another time interval T_a, which causes a delay sufficiently different for the verifier to detect. However, the host-extracted timestamp T_2 can also be changed by an adversary. This change can be verified by using the total time required for the entire identification stage.

Algorithm 1 Hardware-Assisted Identification

```
% trigger SMM on Target
generate random number r
send a trigger command (cmd, r)
% switch to SMM on Target
if cmd ← true then
    get id(smm)
    relay r to host via memory
    calculate T1 = smm_time()
    return id(smm) and T1 to verifier
    resume
%return to host on Target
if r not NULL then
    get host_core_id
    calculate id(host) = hash(host_core_id, r)
    calculate T2 = host_time()
    return id(host) and T2 to verifier
if equal(id(smm), id(host), r) is true then
    identification successful
else
    session failed
```

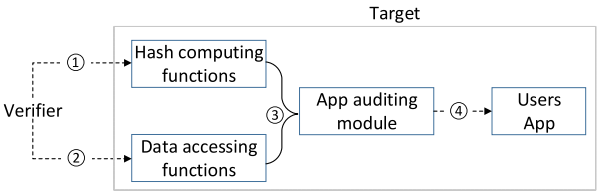

FIGURE 3. The workflow of the trust chain.

C. TRUST CHAIN FOR APPLICATION ATTESTATION

After identifying the user's designated system, the verifier then deploys the trust chain for the application auditing module (AM). Figure 3 shows each step of the chain construction. The measurement module in the target machine includes two types of functions $\mathcal{F} = \{F_{sc}, F_{ac}\}$, where F_{sc} is a hash computing function that remotely checks the integrity of functions in the auditing module (step 1). Here we use $sha1$ and this can be switch to any other hash functions for different security requirements. F_{ac} represents the data accessing functions, which are used to read function text in the auditing module (step 2). AM consists of the following four functions: $func\_hook$, $addr\_locate$, app_audit, and $result\_report$, we define $C_M^i, i \in [1, n]$ to represent the memory segment M of the ith function's text code. $\mathcal{F}$ and $\mathcal{M}$ are publicly loaded to target UT. Then, we verify C_M^i through proof-based verifiable computation (step 3). If the final attestation of the measure module function is correct, then it must satisfy the hash value check $H(M_i) = H(C_M^i)$. Based on the trust application auditing module, we finally audit the loaded application in step 4.

1) CONSTRAINS FOR MEASUREMENT MODULE VERIFICATION

We build a chain to gradually verify the correctness of functions' execution in the measurement module. We first verify the hash computing functions based on the *Zaatar*. Then we verify the data fetching functions based on the *Pantry*. We implement a trust remote attestation mechanism by constructing special constrains on verifiable computing.

Design *Constrains 1*. Suppose to construct QAP (Quadratic Arithmetic Program) for hash computing. Function F_{sc} has n_1 input elements of field $\mathbb{F}$, where QAP Q over $\mathbb{F}$ contains three sets of $m+1$ polynomials $\mathcal{V} = \{v_k(x)\}$, $W = \{w_k(x)\}$, $\mathcal{Y} = \{y_k(x)\}$, for $k \in 0 \ldots m$, and a target polynomial t_x. Meanwhile, $\mathcal{F}$ has n_2. $\mathcal{F}$ totally has $N = n_1 + n_2$ elements.

If and only if there exist coefficients $(p_{N+1}, \ldots, p_m)$, ensuring the $(p_1, \ldots, p_N) \in \mathbb{F}^N$ is valid for entire I/O elements configuration. In other words, there must exist some polynomial $h(x)$ such that $h(x) \cdot t(x) = p(x)$. While, in our designed system, the hash $\mathcal{F}$ is developed based on the *Zaatar* project.

Design *Constrains 2*. Suppose that F_{sc} is the data accessing function which read data from random access memory (RAM). We need to fetch the contents of C_M^i in memory for further attesting *AM*. By leveraging the *GetBlock* mechanism provided by *Pantry*, we verify whether the output of RAM block accessing equals to the input. Then, the constrains should satisfy such condition: if and only if the return value is the hash of the applied name.

We fetch the memory data of the *AM* code as the input parameters, we call the *Load* interface. According to the primitives: $block = GetBlock(name)$ shown in algorithm 2, where if correct execution, $H(block) = name$, H denote a collision-resistant hash functions (CRHF).

Algorithm 2 Verifiable Data Block Primitive

1: GetBlock(*name n*):
2: $S : name \rightarrow block \cup \perp$
3: $block \leftarrow$ read block with name n in block store S
4: assert $n == H(block)$
5: return *block*

Then, we design $block = memory\_block(C_M^i)$, and the H is also implemented with simple hash algorithm to improve the effectiveness comparing using the *SHA-2* algorithm. By leveraging the *Merkel-tree* searching, the verifier supplies a digest as part of the input to the target computation. One way to bootstrap this operation is to create a small amount of state locally, then compute the digest directly. After that, target sends the output to the verifier, which uses the verification machinery to track the changes.

We now describe the constraints that enforce the model.The code $b = GetBlock(n)$ compiles to constraints $\mathcal{C}_{H_{-1}}$, where the input variable X represents the name, the output variable Y represents the block contents; and $\mathcal{C}_{H_{-1}}$ $(X = n, Y = b)$ is satisfiable if and only if $b \in \mathcal{H}^{-1}(n)$ (i.e., $(\mathcal{H}(b) = n)$. The finally corresponding constraints are:

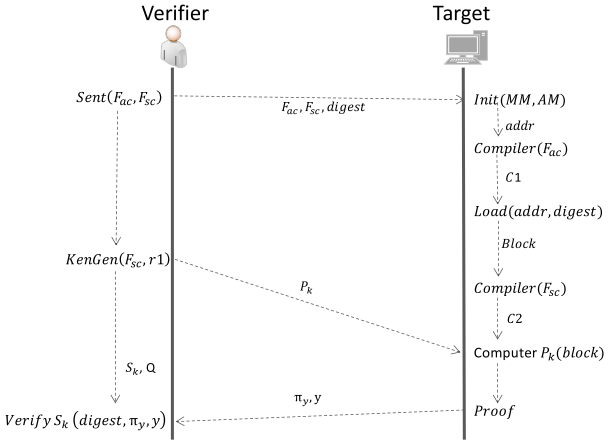

FIGURE 4. The attestation for *AM* based on trust chain.

$\mathcal{C} = \{Y - B - X_1 = 0\} \cup \mathcal{C}_{H_{-1}}(X = X_2, Y = B)$. Where the notation $X = X_2$ and $Y = B$ means that, in $\mathcal{C}_{H^{-1}}$ above, the appearances of X are relabeled X_2 and the appearances of Y are relabeled B. Notice that variable B is unbound in $\mathcal{C}(X_1 = x_1, X_2 = x_2, Y = y)$. To assign $B = b$ in a way that satisfies the constraints, $\mathcal{P}$ must identify a concrete b. Presumably from storage, such that $H(b) = x2$. From *Pantry*'s random memory accessing operations, the verifiable blocks provide the required names-are-hashes referencing scheme, and the *GetBlock* invocations compile to constraints that force target to exhibit a witness-path. Thus, using $\mathcal{C}_{load}$ to denote the constraints, $\mathcal{C}_{Load}(X = (a, d), Y = v)$ can be satisfied only if the digest d is consistent with address *addr* holding value v, which is the guarantee that Load is supposed to be providing.

With the above constraints, the functions in measurement module can be compiled to arithmetic circuit computation, and verified by existing verifiable computing approaches. More proof details can be referred in *Pepper* project.

2) VERIFICATION FOR APPLICATION AUDITING MODULE

To protect the *AM* from being compromised, we check the integrity of the functions in *AM*. We implement the workflow of the remote attestation, defined as $RA_{TB} = (KenGen, Compute, Verify)$. Figure 4 shows the process of the RAitc's proof-based verifiable computing, and the follow describes the details of each operation.

- *Sent* $(\mathcal{F}) \rightarrow (Target)$: $\mathcal{F}$ is developed by the verifier and load to target system. Besides, the $\mathcal{M}$ is initialized in target system. Those two modules are constructed to be the base of attestation. Before attestation, $\mathcal{F}$ and $\mathcal{M}$ both compile to kernel functions and C_M^i deploy in an fixed memory address *addr* which forward to the verifier in advance.
- $Compiler(F_{ac}, F_{sc})$: Target compiles the F_{ac} and F_{sc} to circuit computations, which can be executed by existing verifiable computing approaches.
- $Load(addr, digest)$: The target executes the load operation based on the *Constrains 2*, and access the special

memory block which is the function's text segment in AM. In our design, this block will not send to the verifier immediately.

- *KenGen*(F_{sc}, $r1$): Verifier generates a random security parameter $r1$, and the ECC-based randomized key generation algorithm [55] generates a public key P_k that encodes the parameters. It also computes a matching secret key S_k, which is kept security by the verifier.
- *Compute* $P_k(block)$: Using the public key P_k from the verifier, the target computes an encoded version of the function's output $y = F(x)$ with the block from *Load* operations.
- *Verify* $S_k(digest, \pi_y, y)$: Using the secret key S_k and the the *digest* of name of verified functions, the verification algorithm converts the target's encoded output π_y into the output of the function, e.g., $y = F(x)$ or outputs π indicating that y does not represent the valid output of F on *digest*.
- *Proof*: This randomly checks the verifiable computation. The verified result can be accepted or rejected by the verifier. Here, if $y \neq F_{sc}(block)$ and the verifier reject the result in the possibility $Pr > 1 - \epsilon$, where ϵ is very small, we regard the attestation is in a trusted state.

The security proof of this attestation system is determined by the ratio of erroneous results accepted by the verifier. The adversary may succeed in producing an output that convinces the verification algorithm to accept the wrong output value. We design a random selection method to verify F. It selects multiple x from software lists as the input values, where some of the selections never run in the target. The result returned by the auditing module check should be correct even if the verifier does not know whether the target system is safe, e.g., it might contain other malicious added code. This scheme has been proved to be protected through a static file integrity verification scheme and memory analysis.

3) AUDITING FOR DESIGNATED APPLICATION

With the above verification steps, the measurement module is developed as a trust base for the target system. In addition, the measurement module enables to check the integrity of host kernel function, which protect the auditing module from attack. Therefore, we have created a trust chain from the verifier to local AM, which guarantees the remote loaded application out of integrity damaged.

We depend on auditing module to hook application, locate the corresponding memory space, check the integrity of text code, and provide the digital certificate for applications. In the implementation, we directly create a system call intercepted module and place it into the OS kernel. We build the auditing module through modular fashion mechanism [59], which creates little association with other system kernel module except specific self-defined functions. After the auditing module is initialized, the target system attempts to load some software that belong or beyond the clean service protocol. That software installation would triggers corresponding system calls, then the monitor running in the kernel hook the system call information from system table like System Services Descriptor Table (SCT in Linux kernel). The message from system call instructions are analyzed to address what software is running in OS, we get the store path of software executive file.

Algorithm 3 Workflow of Auditing *firefox*

```
% detect the f(firefox) in Target.
calculate size(f)
set mem_addr is the memory base f code.
initialize remote_attestation_access()
set input is the Data(mem_addr, size)
if input not NULL then
    calculate input with verifiable computing
    verify the output1.
% additional detect dynamic changing.
switch to SMM
if mem_addr not NULL then
    calculate Data(mem_addr, size)
    calculate T4 = smm_time()
    return output2 and T4 to the verifier
    resume
if equal(output1, output2, T4) is true then
    firefox is trusted
```

The algorithm 3 describe an example (*firefox* execution file) about auditing load application. We assume an adversary can modify the memory to replace the legal application's normal process, e.g., using Address Translation Redirection Attack. To protect against that, we leverage SMM to check the integrity of memory operation functions in the meantime. Also, we check the PMU register to counter the assemble instructions from the SMM side. The final cross-checking ensures the applications running in safety status.

In summary, we build a trust chain for the target starting from the verifier, and we use the remote attestation mechanism to check whether the auditing module is broken or is performing incorrect computing operations. The verifier can verifies AM timed or randomly. The entire workflow of RAitc can be summarized by the following two stages:

Preparation: create an association between the user and the application. A user is assigned a protocol with a manager to limit the software that can execute. The methods might function via network access or human visits. The necessary preparation work, which includes user authentication, control of access privileges, and logging, should be completed before auditing. The system will then acquire a user identity $U = (username, id, S_{.level}, \cdots)$, where $S_{.level}$ indicates the user's security level requirement.

Auditing: compute the application identity. Each procedure (including applications and OSes) in an application pool needs a unique identity v for integrity attestation. Here, we directly hash the execution code of the application for further auditing. Generally, function code is unchangeable in

memory space. Before confirming v as an identity, the service needs to be compared to an original hash value v_o from the application store. When $v = v_o$, we store S and v in the software database in an association table.

VII. SECURITY ANALYSIS

In this section, we evaluate the security of the system. For a trust chain from the verifier to the target machine, the security of each node and the communication between every pair of nodes should be proved. We independently analyze the identification and the verifiable computation used in our system. We then evaluate the security of the entire chain and consider attack possibilities.

A. SECURITY FOR IDENTIFICATION

Suppose that SMM is the trust base in an x86 target machine, then the data fetched via SMM is regarded as trustworthy. Since the SMM and host system run with the same CPU, RAitc can then achieve same CPU features for identifying the current device. Otherwise, the identities are different. From the Algorithm 1, we defined follow parameters: the network channel between the verifier and the target's SMM, denoted as L_1; the network channel from the verifier to the target's host, denoted as L_2. The identification extraction in SMM and host are O_1 and O_2 respectively. Thus, the trust identification is defined $S_{identify} = (s(L_1), s(O_1), s(L_2), s(O_2))$, where s represents secure proof.

To prove the security guarantee s, we analyze each stage of identification. First, the verifier is a trusted platform owned by the users, we trust the verifier's program. Besides, L_1 and L_2 using TLS protocol against the data tampering attack from the network. We mentioned that network message attacks are out of our scope. Thus, we regard the communication channels between the verifier and target are trusted, that means $(s(L_1), s(L_2))$ is true. Second, O_1 in SMM is an isolated and secure operation. We add an independent network driver into the SMI handler, which builds a separate communication channel between the verifier and target's SMM. Thus, $(s(L_1), s(O_1), s(L_2))$ is acceptable. To ensure trust O_2, target system runs simple HMAC algorithm with the random number r. After computing $outputs = hash(r, identity)$ in O_2, its *outputs* are checked by the verifier to verify whether the identity is from the target.

However, the *hash*() function may be forwarded to a forged system through malicious fetching the r and target's identity. To detect such an attack, we monitor the time interval of identification. We firstly measure the necessary time cost of identification $\mathcal{T} = sum(T_{L_1+L_2}, T_{O_1}, T_{s2h}, T_{O_2})$, where T_{s2h} is the system switching time between SMM and host. While $T_{L_1+L_2}$ is the network handshake and communication time which depends on network conditions, here we reasonably assume it is a constant value. If a relay attack is in target host, the O_2 and the *hash* function might be executed in forged system, which introduce the delay time cost $T_{relay} = sum(T_{fetch} + T_{return})$, where T_{fetch} is the time to fetch the random number r and forward it to the forged system, T_{return} is the time to send the result of *hash* function to the target. We define a ratio threshold λ, where $(\frac{T_{relay}}{\mathcal{T}} > \lambda)$ is significant in the test. Therefore, T_{relay} is a detectable value by using the introspecting approaches [60], and the identity extraction was proven to be trusted. By comparing the identities extracted from the SMM and host, the result trusty identifies the designated target machine.

B. SECURITY FOR TRUST CHAIN

The verifier is another key trust base for RAitc. The verification of the trust chain from the verifier to the measurement module is developing with existing verifiable computation mechanisms. In our design, F_{sc} is a *hash* module, and F_{ac} is a memory accessing module based on the *Pantry*'s component. That means, the proof of completeness similar to the *Pantry*. The following steps show the corresponding security analysis.

First, the verifiable computing should satisfy: the function F_{sc} calculates the data set $\mathcal{D}$, to achieve the hash value set $\mathcal{G}$, it should exist a valid transcript $\gamma = \{x, y\}$, where the data is the input $x \in \mathcal{D}$, the hash value is the output variable which set to $y \in \mathcal{H}$. Previous verifiable computing, like Geppetto [61], proven that γ is unique. That means, given an input data block, F_{sc} returns a unique hash value, the execution of this function is deterministic. That's similar to function F_{ac}. Given an input digest, memory accessing of F_{ac} returns a corresponding memory data block, the hash of such data block equals to the digest.

Second, the verifier compiles functions with constraints of $\mathcal{C}$. Given the input digest and multi-round of proofing, the verifier obtains a list y' calculated by the target. For all $y' \neq y$, the possibility of attestation result $Pr\{(verifier, target)(\mathcal{C}, x, y') = 1\} \leq \epsilon$. That because the used hash algorithm H in our based *Zaatar* project is a collision-resistance, that is means, an adversarial $\mathcal{A}$ cannot produce a collision in H with the succeed probability $\geq \epsilon$. Therefore, we can promise that constraints $\mathcal{C}$ can lead to the correct result. Also, there is a sequence of memory accessing operations existing in target, the attestation succeeds probability still $\leq\epsilon$, the proof can also refer the *Pantry*'s proof [55]. In addition, to construct a collision-resistance hash function H, the verifiable computing algorithm needs to satisfy the proof-of knowledge (PoK) property. With the above analysis, we know that the measurement module can provide the correct attestation function.

In summary, with the device identification and security proof of the attestation chain, we finally create a trust chain from the user to the target AM. Then, we audit the loaded application by checking whether the application is legal on the designated system.

C. SECURITY DISCUSSION

Memory or cache attacks on the target system caused by higher-privilege malware can dynamically change code at runtime, which will block the normal workflow of RAitc.

This type of attack can be detected by RAitc because the target sends either an error or no result. Note that our system does not focus on safeguarding the target system but instead on elucidating whether a loaded application is compromised. If the target system is broken, then we rely on assisted security policies to recover or update the system. Moreover, we can join other SMM-based introspection approaches [12] to co-detect those attacks. In addition, we assume that the network condition is sufficiently stable to support an acceptable communication interval. Other risks, such as transient attacks and DoS attacks, also exist and threaten current security mechanisms. We can mitigate them by implementing random round attestation to make interval prediction by attackers more difficult. However, such risks cannot be comprehensively removed; thus, they remain issues for future consideration.

VIII. EVALUATION

In this section, we evaluate the performance of our system. The security system underwent rigorous testing and evaluation that included both validating the effectiveness of its protection and measuring its impact on system performance.

A. EXPERIMENTAL SETUP

All experiments were performed in a laboratory environment with two computers equipped with Intel x86 200 series chipsets. Both the verifier and target machine had Intel i7 cores with 8 GB of memory and ran Ubuntu 16.04, the VC software based on *Zaatar* and the *Pepper* project. Additionally, we developed SMM-based and host-kernel-based secure modules.

B. EFFECTIVENESS

RAitc was evaluated in two phases. First, we set up an execution environment involving the executable modules waiting to be deployed, which included the Trust ID module, measurement module, and auditing module. To identify the target platform, we preinserted the Trust ID module into the SMM handler. To verify the correctness of the loaded applications, we deployed the chain built by the attestation module in both the user's device and the target device. We used Coreboot and SeaBIOS for the target BIOS to boot the users' remote computation systems.

The remote attestation algorithm was developed based on *Zaatar* and extended to form the measurement module. We developed an LKM-based hooking and analyzing module as the auditing module. Both the measurement and auditing modules were deployed in the target system. When deploying a trusted system, the goal was to monitor whether the loaded application was legal with respect to the current runtime stage or user requirements.

First, the remote attestation approach might be relayed to another malicious machine. Therefore, target identification is necessary for trust chain construction. To evaluate the target identification operation, we fetched the CPU identities (e.g., *EA060900FFFB8B0F* for CPU0 in our testbed) in both SMM and the target host system. To ensure a correct target platform for users, the CPU identities should be the same for one platform. Our experimental results show the effectiveness of CPU identity fetching and comparison. The machine and the OS system can be identified by SMM-based identification approaches. Even a malicious attack must forge a trusted auditing module on the runtime system; therefore, it cannot simulate a user-designated remote platform.

Some attacks may bypass the auditing module through compromising the auditing functions themselves. For example, kernel-level attacks (e.g., amark [62]) can change the auditing function instructions in kernel memory. However, by using the security-proven VC mechanism, we can verify the integrity of auditing functions. Thus, we set the application auditing module to a fixed kernel memory space at system loading stage. The functions in each module shown in Table 1, were audited after being developed. The identity of the application and its attached attributes, such as name and size, were stored by the verifier for ongoing attestation purposes. Through the proven VC mechanism, we verified the integrity of each function in the auditing module via memory extraction and hash computing. The experimental results in Table 1 show that the integrity of all auditing functions can be verified by RAitc.

TABLE 1. Functions exist in auditing module.

Key files	Hash value	Size($Byte$)	Attested
audit_manager.c	595e***91c1	1,310,660	✓
data_mem_access.c	6f15***6454	1,311,844	✓
func_hook.c	16dd***9cd9	1,137	✓
addr_locate.c	23a5***ad46	903	✓
app_audit.c	e5e6***eafb	2,605	✓
result_report.c	283e***a6db	353	✓

Finally, by deep-going verifying the correctness of the identified trust chain, the next step is to verify the effectiveness of RAitc. We choose 4 system commands and 7 applications for our evaluation experiments. The system commands are allowed to run in OS because those procedures are common functions that is default setting with clean service protocol. 4 out of the applications are authorized in our proposed application protocol, and the rest are unauthorized. Furthermore, a self-defined the program represents the unknown (malware) software. The expected result is that the legal application runs normally while the unauthorized software and malware are stopped by our auditing system. The experiment result shows that all applications are audited by RAitc, the illegal and malicious software are blocked. In some situations, the benign application might be attacked and change to be a malicious one. To simulate such scenarios, we change some code in benign functions in the test. For example, the application *firefox* normally runs in an authorized state but it is stopped in a compromised state, as shown in Table 2. Additionally, the user requirement is dynamically changed and the manager needs to update its protocol lists in more complex situations.

TABLE 2. The auditing result based on RAitc.

Software Type	Name	Execution Allowed
System command	ls, cd, cp, mv	✓
Authorized	Nvicat, firefox, Chrome, Eclipse	✓
Unauthorized	VLC player, RhythmBox, KMPlayer	×
Malicious	Compromised firefox, ProcessHacker.ko	×

In summary, our system constructs a useful trust chain for remotely attesting to the security of the user's platform, which can effectively monitor the loaded application status and detect malicious behaviors in the user's remote computing environment.

C. EFFICIENCY

We analyzed the performance at each stage and calculated the total overhead to evaluate the system performance.

1) IDENTIFICATION

The identification operation works in both SMM and protected mode. Because SMM suspends the host OS, it temporarily halts the user's runtime system. We evaluated the time overhead of the identification process, and the results are shown in Table 3. From steps 1 to 5, the operations work in SMM, which requires $57\mu s$. This is a very short and acceptable time to validate the user's service. Note that we do not consider ''network package missing'' errors on the SMM side since we regard the network communication as stable.

TABLE 3. Identification overhead in each stage.

Steps	Operations	Time
1	cmd_recv	0
2	net_smm	$7\mu s$
3	sid_get	$11\mu s$
4	net_call	$23\mu s$
5	rand_send & resume	$16\mu s$
6	hid_get	$9\mu s$
7	cal_fuzzy & net_call	$3ms$

By verifying the target system's identification, we build a trusted chain from the verifier to the designated target. The attestation on the trust chain requires time and system resources (CPU and memory) for checking computations. Before the application runs on the user's remote platform, we must spend time and memory on attesting to the initial software. The overhead costs on two operations: auditing module checking and application auditing. The two operations can be executed synchronously, and the overhead of auditing module checking shows in Table 4.

2) MEASURING

The platform resource overhead for the auditing module checking is high due to the *p_compute* process running in the target. During the auditing module checking, we record the system overhead through sampling and analyzing the CPU and memory consumption. Figures 5 and 6 show the CPU and memory usage ratio, respectively, once those outsource verification executions have occurred. It takes nearly $10s$ to verify whether the hash function outputs match, and two of four cores are fully used for the remote checking computation. The remaining two cores can be used to execute normal processes in the target system. Thus, our trust chain construction imposes considerable overhead, but its advantages are that RAitc will not halt other executing system operations. The memory occupation ratio is also high due to the number of *QAP* computations. Similarly, Figures 7 and 8 show the overhead required to verify the normal process's

TABLE 4. Time overhead of trust chain attestation.

Steps	hash_computing avg time(s)	mem_accessing avg time(s)	Total avg time(s)
key_gen	1.01	9.01	10.02
p_delay	5.20	6.96	12.16
p_compute	8.07	28.7	36.77
v_verify	1.75	1.62	3.37

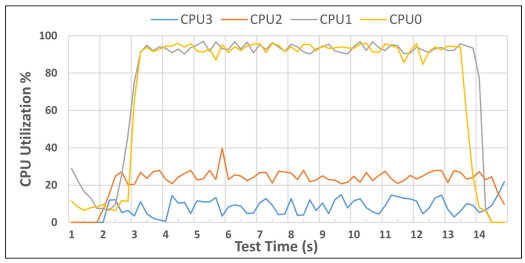

FIGURE 5. CPU overhead on attestation of hash computing function.

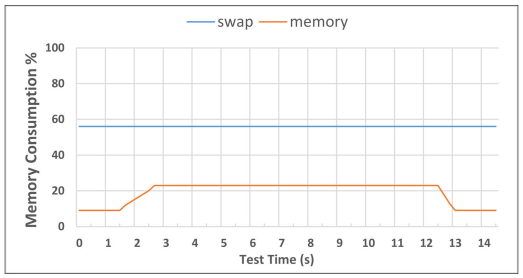

FIGURE 6. Memory overhead on attestation of hash computing function.

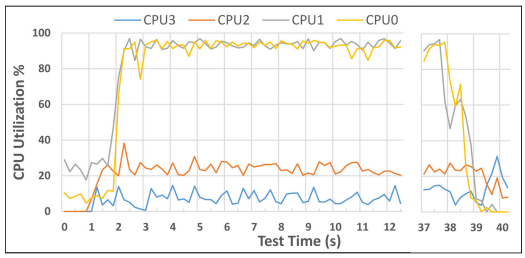

FIGURE 7. CPU Overhead on attestation of memory accessing function.

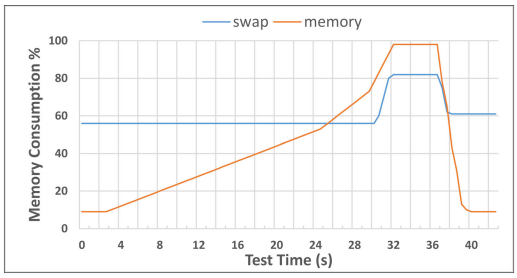

FIGURE 8. Memory Overhead on attestation of memory accessing function.

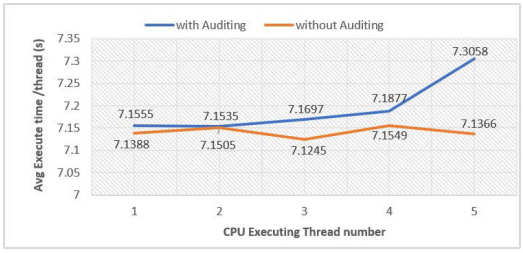

FIGURE 9. CPU benchmark test during application auditing.

memory data access. Note that this test result is for accessing less 10MB memory and the overhead depends on the size of involved memory blocks. Moreover, the overhead required for checking data accessing is larger than that required for hash computing due to the additional memory block fetches and digest computing operations. On our testbed, the auditing module checks cost nearly half the CPU and memory resources, but this is acceptable because it takes only seconds and does not block normal system operations.

3) AUDITING

Application checking by the auditing module will delay the application execution. We tested the corresponding time overhead for the hooking, locating, auditing, and reporting stages. We directly examined the hooking *fork* and *execve* system calls and located the memory space for the target application. We evaluated the overhead based on the *sysbench* benchmark [63] through CPU and memory testing, it is reasonable to use CPU and memory overhead to evaluate the performance of our RAitc because no IO other latency added in our application auditing module. In the experiments, we tested the benchmark evaluation of CPU operation and memory access under different concurrent thread conditions. For example, for a kernel executable module case with a size of 1.9KB, RAitc generally takes 91*ms* to perform application execution event hooking and system call analysis, 1.83*ms* to perform memory addressing, and 1.61*ms* to check the integrity of the application code. The overhead of the locating and auditing operations depends on the size of the application: a large application requires more time for these two stages. We test *sysbench* with an increasing number of threads (from 1 to 5) to simulate normal service environment, and the Figure 9 shows few additional changes in CPU consumption while the auditing module executes, it generally adds less

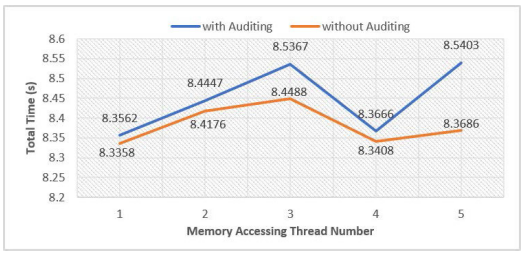

FIGURE 10. Memory benchmark test during application auditing.

than 2.37% latency, and Figure 10 shows similar memory overhead increases due to the auditing function execution and application memory accesses, it generally adds less than 2.05% latency. Compared with the total system resources, the auditing process occupied only a small ratio, and exerted little effect on normal user computation.

We did not show the network performance because the actual performance is affected by the network communication conditions. In addition, the sessions caused by identification, VC and application auditing occupy less than 3% of the total network flow based on package monitoring with the *Wireshark* tools, which is negligible in a practical execution environment.

4) COMPARISON

To compare various approaches, we built an execution environment to simulate ring 3 to ring -2 scenarios. The virtualization execution environment is built with QEMU and KVM, where QEMU also simulate the SMM execution (similar to KShot [64]). Then we respectively deploy the test approaches, where the domain 0 and VM both use Ubuntu 16.04 and kernel 4.15. Note that ring-3 based Nighthawk is installing on the same platform and does the same introspection work [11]. Thus, we install the Kernel- [8], Hypervisor- [65] and SMM-based defender [66], and use the *mpstate* to monitor each CPU utilization. Here we test the CPU consumption with and without installing defenders. We developed the shell script to orderly install and uninstall some applications from Table 2 (i.e., *ls*, *firefox*, *KMPlayer*, etc).

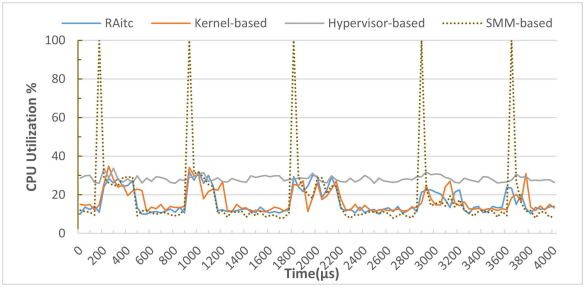

FIGURE 11. The comparison between different level defenders.

Figure 11 shows the consumption ratio in the system, where the result shows that our approach gets a kernel-based

defender approximated performance, which is better than hypervisor-based approaches. The SMM-based approach will cause a temporary pause on the current system (100% CPU utilization), because the system will switch to SMM during the checking. In addition, there is a semantic gap existing in the latter two approaches which needs extra time and computation to extract the application information. Nighthawk is a co-processor based approach that computation is executed on another core which will not affect the host system, that is the reason that Nighthawk only creates some shared memory accessing consumption.

The key point of RAitc is focusing on the security for a remote user who cannot directly access the execution platform. We compared those approaches in security aspects shown in Table 5. Simply, the lower level approaches have higher security protection due to the isolation and smaller TCB, but similarly, need more work to solve the semantic gap issue. Since we can remotely verify the correctness of functions execution, our approaches give a chance by using trustchain to audit the remote application with an acceptable consumption, with effectively increase the user's trustworthiness for remote execution.

TABLE 5. Comparing with other similar approaches.

Approach	Trust Base[1]	Sync-check	Semantic Gap
Kernel-based defender [8]	OS Kernel [r1]	✓	✗
KVMI [65]	Hypervisor [r(-1)]	✗	✓
SMMcheck [66]	SMM [r(-2)]	✓	✓
RAitc	SMM [r(-2)] *Verifier*	✓	✗
Nighthawk [11]	IME [r(-3)]	✗	✓

[1] smaller ring number means higher privilege level in computer system [11].

RAitc audits the applications in OS-level and mitigates the semantic gap problem, which exhibits greater flexibility for extension of the application auditing functionality. In addition, the time overhead of the auditing process is a short interval for application initialization, which is necessary and acceptable for users' security requirements. Note that, the time and resource overhead of trust chain building is not covered in this comparison, because even such operations synchronizing with auditing but not interfere with each other.

IX. CONCLUSION

In this paper, we presented RAitc, a trusted remote application auditing system, that provides identification of remote target platforms and attestation for loaded applications. We assumed that a security module could depend only on the target kernel, might be bypassed by a relay attack or by rootkits, or might be compromised by malware with higher privileges. To address this issue, we first trust the part of the hardware features on the target platform (i.e., SMM) that assists in fetching the CPU identity but assume it might be subject to a relay attack. Based on this definite platform, we then leverage VC approaches to create a chain of trust from the user to the remote target system. Therefore, we can correctly verify the integrity of the auditing module in the target system's kernel. The auditing module consists of general kernel hooks, memory access checks, and hash computing functions with protection from RAitc. Thus, RAitc combines hardware-assisted trusted computing and remote VC mechanisms to co-construct the trust chain for an application. We developed a prototype version of RAitc to implement the attestation functions and tested its effectiveness and performance in a real experimental environment. The results show that RAitc effectively protects the user's remote service environment with an acceptable level of overhead.

REFERENCES

[1] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2014, pp. 990–1003.

[2] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. M. Marvel, "Catch me if you can: A closer look at malicious co-residency on the cloud," *IEEE/ACM Trans. Netw.*, vol. 27, no. 2, pp. 560–576, Apr. 2019.

[3] S. H. Islam, "A provably secure ID-based mutual authentication and key agreement scheme for mobile multi-server environment without ESL attack," *Wireless Pers. Commun.*, vol. 79, no. 3, pp. 1975–1991, Dec. 2014.

[4] R. M. B. Blessy, "A survey on network security attacks and prevention mechanism," *J. Current Comput. Sci. Technol.*, vol. 5, no. 2, pp. 1–5, 2015.

[5] I. Ghafir, V. Prenosil, J. Svoboda, and M. Hammoudeh, "A survey on network security monitoring systems," in *Proc. IEEE 4th Int. Conf. Future Internet Things Cloud Workshops (FiCloudW)*, Aug. 2016, pp. 77–82.

[6] Y. Wang, T. Uehara, and R. Sasaki, "Fog computing: Issues and challenges in security and forensics," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, vol. 3, Jul. 2015, pp. 53–59.

[7] F. A. M. Ibrahim and E. E. Hemayed, "Trusted cloud computing architectures for infrastructure as a service: Survey and systematic literature review," *Comput. Secur.*, vol. 82, pp. 196–226, May 2019.

[8] M. A. Ambusaidi, X. He, and P. Nanda, "Unsupervised feature selection method for intrusion detection system," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2015, pp. 2497–2507.

[9] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proc. 16th ACM Conf. Comput. Commun. Secur. (CCS)*, Chicago, IL, USA, 2009, pp. 400–409.

[10] T. Zhang and R. B. Lee, "Monitoring and attestation of virtual machine security health in cloud computing," *IEEE Micro*, vol. 36, no. 5, pp. 28–37, Sep. 2016.

[11] L. Zhou, J. Xiao, K. Leach, W. Weimer, F. Zhang, and G. Wang, "Nighthawk: Transparent system introspection from ring-3," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2019, pp. 217–238.

[12] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A dependable introspection framework via system management mode," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–12.

[13] L. Zhou, F. Zhang, and G. Wang, "Using asynchronous collaborative attestation to build a trusted computing environment for mobile applications," in *Proc. IEEE SmartWorld, Ubiquitous Intell. Comput., Adv. Trusted Comput., Scalable Comput. Commun., Cloud Big Data Comput., Internet People Smart City Innov. (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug. 2017, pp. 1–6.

[14] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the semantic gap in trusted execution environments," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–51.

[15] J. Jang and B. B. Kang, "Securing a communication channel for the trusted execution environment," *Comput. Secur.*, vol. 83, pp. 79–92, Jun. 2019.

[16] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, Cambridge, MA, USA, Sep. 2008, pp. 1–20.

[17] M. Kiperberg, A. Resh, and N. J. Zaidenberg, "Remote attestation of software and execution-environment in modern machines," in *Proc. IEEE 2nd Int. Conf. Cyber Secur. Cloud Comput.*, Nov. 2015, pp. 335–341.
[18] J. Keuffer, R. Molva, and H. Chabanne, "Efficient proof composition for verifiable computation," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2018, pp. 152–171.
[19] Y. Wang, Y. Shen, and X. Jiang, "Practical verifiable computation–a mapreduce case study," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 6, pp. 1376–1391, Jun. 2018.
[20] W. Song, B. Wang, Q. Wang, C. Shi, W. Lou, and Z. Peng, "Publicly verifiable computation of polynomials over outsourced data with multiple sources," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 10, pp. 2334–2347, Oct. 2017.
[21] A. Kosba, C. Papamanthou, and E. Shi, "XJsnark: A framework for efficient verifiable computation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 944–961.
[22] Z. Zhang, H. Zhang, and S. Liu, "Coarse-fine convolutional neural network for person re-identification in camera sensor networks," *IEEE Access*, vol. 7, pp. 65186–65194, 2019.
[23] W. Sun, R. Zhang, W. Lou, and Y. Thomas Hou, "REARGUARD: Secure keyword search using trusted hardware," in *Proc. IEEE IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 801–809.
[24] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1521–1538.
[25] L. Batina, P. Jauernig, N. Mentens, A.-R. Sadeghi, and E. Stapf, "In hardware we trust: Gains and pains of hardware-assisted security," in *Proc. 56th ACM/IEEE Design Automat. Conf. (DAC)*, Jun. 2019, pp. 1–4.
[26] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "PrOS: Light-weight privatized se cure OSes in ARM trustzone," *IEEE Trans. Mobile Comput.*, vol. 19, no. 6, pp. 1434–1447, Jun. 2020.
[27] S. Park, C. H. Kim, J. Rhee, J.-J. Won, T. Han, and D. Xu, "CAFE: A virtualization-based approach to protecting sensitive cloud application logic confidentiality," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 4, pp. 883–897, Jul. 2020.
[28] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from hypervisor and host operating system exploits," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1357–1374.
[29] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM trustzone," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 541–556.
[30] A. Anto, R. S. Rao, and A. R. Pais, "Kernel modification APT attack detection in Android," in *Proc. Int. Symp. Secur. Comput. Commun.* Springer, 2017, pp. 236–249.
[31] X. Gu, H. Zhang, and S. Kim, "CodeKernel: A graph kernel based approach to the selection of API usage examples," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 590–601.
[32] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.
[33] P. Bhat and K. Dutta, "A survey on various threats and current state of security in Android platform," *ACM Comput. Surveys*, vol. 52, no. 1, pp. 1–35, Feb. 2019.
[34] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 11–20.
[35] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 169–174.
[36] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-Learning-Guided typestate analysis for static Use-After-Free detection," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 42–54.
[37] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," Black Hat, 2017.
[38] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *Proc. 26th Eur. Signal Process. Conf. (EUSIPCO)*, Sep. 2018, pp. 533–537.
[39] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.
[40] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 238–252.
[41] L. Zhou, X. Liu, Q. Liu, and G. Wang, "A cleanroom monitoring system for network computing service based on remote attestation," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2016, pp. 451–457.
[42] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure until proven updated: Analyzing AMD SEV's remote attestation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1087–1099.
[43] S. H. Islam and G. P. Biswas, "A more efficient and secure ID-based remote mutual authentication with key agreement scheme for mobile devices on elliptic curve cryptosystem," *J. Syst. Softw.*, vol. 84, no. 11, pp. 1892–1898, Nov. 2011.
[44] S. Chen, S. Jero, M. Jagielski, A. Boldyreva, and C. Nita-Rotaru, "Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2019, pp. 404–426.
[45] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 991–1008.
[46] S. Han, W. Shin, J.-H. Park, and H. Kim, "A bad dream: Subverting trusted platform module while you are sleeping," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 1229–1246.
[47] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *Proc. USENIX Annu. Tech. Conf. (USENIXATC)*, 2016, pp. 565–578.
[48] H. Brandl. (2004). *Trusted Computing: The TCG Trusted Platform Module Specification*. [Online]. Available: http://www. wintecindustries. com/orderdesk/TPM/Documents/TPM1. 2_-_Basics.pdf
[49] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, and D. Mattoon, "ftpm: A software-only implementation of a TPM chip," in *25th USENIX Secur. Symp. (USENIX Secur. 16)*, 2016, pp. 841–856.
[50] W. Showalter, "A universal windows Bootkit: An analysis of the MBR Bootkit 'HDRoot,'" in *Proc. 50th Hawaii Int. Conf. Syst. Sci.*, 2017.
[51] Z. Ning and F. Zhang, "Understanding the security of ARM debugging features," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 602–619.
[52] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," *Lect. Notes Comput. Sci.*, vol. 6223, no. 3, pp. 465–482, 2010.
[53] J. Yao. (2017). *SMM Protection in EDK II*. [Online]. Available: https://uefi.org/sites/default/files/resources/Jiewen%20Yao%20-%20SMM%20Protection%20in%20%20EDKII_Intel.pdf
[54] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 253–268.
[55] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proc. 24th ACM Symp. Operating Syst. Princ. (SOSP)*, 2013, pp. 341–357.
[56] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 926–943.
[57] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan, "The password reset MitM attack," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 251–267.
[58] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "HyperCheck: A hardware-AssistedIntegrity monitor," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 4, pp. 332–344, Jul. 2014.
[59] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A programmable programming language," *Commun. ACM*, vol. 61, no. 3, pp. 62–71, 2018.
[60] L. Zhou, Y. Shu, and G. Wang, "A software detection mechanism based on SMM in network computing," in *Proc. Int. Conf. Secur., Privacy Anonymity Comput., Commun. Storage*. Springer, 2016, pp. 134–143.
[61] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 253–270.
[62] GitHub. (2019). *amark*. [Online]. Available: https://github.com/amanone/amark
[63] GitHub. (2020). *sysbench*. [Online]. Available: https://github.com/akopytov/sysbench

[64] L. Zhou, F. Zhang, J. Liao, Z. Ning, J. Xiao, K. Leach, W. Weimer, and G. Wang, "KShot: Live kernel patching with SMM and SGX," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2020, pp. 1–13.
[65] J. Pfoh, C. A. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *Proc. Int. Workshop Secur.* (Lecture Notes in Computer Science), vol. 7038, T. Iwata and M. Nishigaki, Eds. Tokyo, Japan: Springer, Nov. 2011, pp. 96–112.
[66] F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A framework to secure peripherals at runtime," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2014, pp. 219–238.

LEI ZHOU is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Central South University, China. He has been a Visiting Student with Wayne State University, Detroit, MI, USA. His research interests include cloud security computing and hardware assisted system security.

ENTAO LUO is currently an Associate Professor with the School of Electronics and Information Engineering, Hunan University of Science and Engineering, Yongzhou, China. His research interests include security and privacy issues in cloud computing.

GUOJUN WANG (Member, IEEE) received the B.Sc. degree in geophysics, the M.Sc. degree in computer science, and the Ph.D. degree in computer science, with Central South University, China, in 1992, 1996, and 2002, respectively. He is currently a Pearl River Scholarship Distinguished Professor of Higher Education, Guangdong, a Doctoral Supervisor and the Vice Dean of the School of Computer Science and Cyber Engineering, Guangzhou University, China, and the Director of the Institute of Computer Networks, Guangzhou University. He has been listed in Chinese Most Cited Researchers (Computer Science) by Elsevier from 2014 to 2019. His research interests include artificial intelligence, big data, cloud computing, Internet of Things (IoT), block chain, trustworthy/dependable computing, network security, privacy preserving, recommendation systems, and smart cities. He is a Distinguished Member of CCF and a member of ACM and IEICE.