EAI Endorsed Transactions
on Security and Safety

Research Article **EAI.EU**

Confidential State Verification for the Delegated Cloud Jobs with Confidential Audit Log

Anyi Liu[1,*], Selena Haidar[1], Yuan Cheng[2], Yingjiu Li[3]

[1]Department of Computer Science and Engineering, Oakland University, 2200 N. Squirrel Road Rochester, MI 48309, USA.
[2]Department of Computer Science, California State University, Sacramento, 6000 J Street, Sacramento, CA 95819, USA.
[3]Singapore Management University, 81 Victoria Street, 188065, Singapore

Abstract

Cloud computing provides versatile solutions that allow users to delegate their jobs. An apparent limitation of most cloud audit services is that it is difficult for cloud users to tell the state of a delegated job. Although cloud users can provide a user-specified model, the model is susceptible to various exploits. In this paper, we present VERDICT, a system that amplifies the capability of the existing cloud audit services and allows users to check the state of a cloud job through the confidential model and audit log. VERDICT extracts the model from an application and keeps it in the encrypted form. The encrypted model is partitioned and updated with homomorphic encryption cipher in multiple sandboxes. The state of a delegated job can be checked by cloud users at any time. We implement VERDICT and deploy it in the VMs and containers on popular cloud platforms. Our evaluation shows that VERDICT is capable of reporting the state of a cloud job at run time, keeping the model confidential, and detecting malicious operations.

Received on 24 January 2019; accepted on 11 April 2019; published on 29 April 2019

Keywords: homomorphic encryption; finite state automaton; virtual machine; container.

doi:10.4108/eai.13-7-2018.162290

1. Introduction

Cloud computing is considered a ubiquitous and elastic computing paradigm, in which versatile and cost-efficient solutions allow users to delegate their jobs and store data. To ensure the accountability for their users, major IaaS cloud service providers (CSPs), nowadays, offer security and privacy-compliant audit services [1], [2], [3]. For instance, Google Cloud provides Cloud Audit Logging, which offers three types of audit logs, namely *admin activity*, *system event*, and *data access* [4]. Amazon AWS introduces CloudTrail [5], which records AWS API calls for cloud users. Microsoft Azure provides the activity log [6] and Azure Monitor [7], which allow users to analyze and learn performance issues, identify errors and failures and respond to alerts automatically.

Although the audit log generated by the current CSPs has been proven to be effective in monitoring the performance of applications, pinpointing to the errors, and facilitating forensic investigation [8], verifying the execution status of a delegated job still poses a number of challenges that cannot be met with the existing solutions. First, because most audit systems keep a daunting quality of audit log, it is difficult for them to provide a timely and appropriate reaction to online incidents. The recent settlement reported by the U.S. Department of Health and Human Services (HHS) clearly shows that the audit log of the applications that maintain electronically protected health information (ePHI) is significantly overlooked [9]. Therefore, a significant amount of human efforts are needed to comprehend and analyze the audit log, in order to understand the incidents and perhaps pinpoint the security breaches. It is always desirable to have an automated tool, orchestrated by the auditing system, to report the execution status of a cloud job in a timely manner. Second, the audit log, kept by the cloud infrastructure, might have been encrypted, which makes online status verification more challenging. Given that the audit log might contain the users' private information, most CSPs encrypt the audit log immediately after its generation. However, if a user runs a cloud-based analytic tool, such as Azure Monitor, it allows a malicious or honest-but-curious CSP to read the audit log in plaintext at runtime [10]. Third, existing online exploitation mitigation approaches provide limited state information of a remote application under

[*] Corresponding author. Email: anyiliu@oakland.edu

attestation. The techniques against control-flow hijackings, such as control-flow integrity (CFI) [11], fine-grained code randomization [12], and code-pointer integrity (CPI) [13], provide limited state information of the remote application to the verifier. Specifically, they can only tell *whether a control-flow attack occurred*, but cannot tell *what was the state of an application when the attack occurs*. Other techniques can precisely attest to the execution path [14–16]. However, these techniques mainly assume the availability of dedicated trusted hardware, such as TrustZone [17] and Intel Process Trace [18], which might not be available for certain CSPs.

We aim at tackling the aforementioned challenges by verifying the status of a delegated cloud job at runtime through a confidential model checker and confidential audit trial. In particular, our scheme ensures control-flow integrity (CFI) against code-reuse attacks [19]. In this context, the term *integrity* refers to all operations that obey the model of a program, such as the finite state machine (FSM). We achieve the confidentiality of the model under checking by first partitioning the model and then encrypting it via homomorphic encryption (HE) cipher [20]. To ensure the efficiency of running HE cipher, the ciphertext of each partition is kept in distributed but collaborative sandboxes (e.g., virtual machines (VMs) and containers). The ciphertext in each sandbox is updated while the audit log is generated online. To check the online status of a cloud job, a user can request the ciphertext of partitions from all sandboxes and thus calculate the state of cloud jobs by herself. Compared with the existing techniques that rely on trusted hardware [15, 17, 21, 22], our approach does not rely on any hardware support and thus can be deployed on the top of various platforms.

In summary, we make the following four contributions in this paper:

- We abstract a model to be checked from an application and keep it in the form of ciphertext. The model describes the expected behavior of a delegated job. It is partitioned and updated in multiple sandboxes (e.g., virtual machines or containers). Since the partitions of a model are encrypted with homomorphic encryption cipher, it keeps the model confidential and thus prevents the model from being subverted. In addition, the sandboxes never store secret keys and thus prevent the potential key leakage caused by side-channel attacks [23, 24].
- To further enhance the model's confidentiality, we decouple the rules that update the model from the model itself. In such a way, an adversary cannot reason about the state of the model even when the secret of the model is disclosed to the adversary. To minimize the run-time overhead of the HE cipher [25, 26], each online model update operation only incurs a bounded number of HE operations, which is *proportional to* the number of partitions[1]. We also present efficient algorithms that update the confidential model and report the state of the delegated job.
- To systematically counter control-data attacks that explicitly change the program's behavior, we classify the attacks that compromise the CFI by defining the abnormalities, including malicious deletion and repetition attacks. In addition, we present algorithms to detect such attacks.
- We implement a prototype system, namely VERDICT (**Ver**ifier for **D**elegated **I**ntegrated **C**loud **T**asks), and present a detailed evaluation in both VMs and containers from Amazon AWS [27], Microsoft Azure [28], Google Cloud [29], and IBM Cloud [30]. Our experiments demonstrate that VERDICT can identify the state of a delegated cloud job with a bounded online overhead for model updating and a reasonable offline overhead for model initialization and verification.

In contrast to our prior work [31], which proposed a preliminary framework that describes the functionalities of key components with limited technical details, we extend it with comprehensive coverage, more technical details, and security analysis in this paper. The remainder of the paper is organized as follows. In Section 2, we present the motivation and an overview of VERDICT. We then describe the threat model in Section 3. The detailed methodology, including how to construct the model and the algorithms about how to update the model and verify the state of the model, are elaborated in Section 4. We analyze the security properties of VERDICT in Section 5. In Section 6, we present the detection of two abnormalities that violate the expected behavior of a delegated job. Then, we describe the current implementation and evaluate the performance of VERDICT in Section 7. After that, we discuss the limitation of VERDICT and suggest possible solutions in Section 8. We review the related work in Section 9. Finally, we conclude the paper in Section 10.

2. Motivation and Overview

In this section, we first introduce a real-world example that motivates our approach (Section 2.1). We then define our research problem and provide an overview of the proposed system (Section 2.2).

2.1. A Motivating Example

To further elaborate on the concepts developed earlier, we present a highly simplified version of a forensic program (*simple_traversal.c*) that is delegated to a CSP to acquire evidence from virtual disks through a virtual machine introspection library. In this example, the forensic

[1] The number of partitions is a configurable parameter, which depends on the level of the desired security.

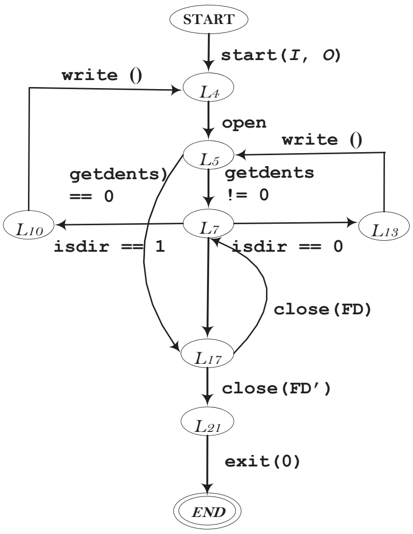
investigator expects the CPS to ensure the control-flow integrity of the delegated job. For example, the control flow should not be hijacked or deviate from the expected execution path.

Figure 1a shows the code snippet of *simple_traversal.c*, which traverses the file system of a virtual disk in the cloud. For simplicity, we ignore the detailed instructions, such as initializing the device drivers, archiving the files, maintaining the directory structures, or checking the integrity of raw data. All API calls made by the introspection library are underlined. The control-flow of *simple_traversal.c* can be modeled as a finite state machine (FSM), which is illustrated in Figure 1b. States of the FSM are labeled with line number L_n. The transitions are labeled with the corresponding API calls, which can be obtained from the audit log. As suggested by prior research, this program is vulnerable to control-data attacks [32], in which the program deviates from its expected behavior after launching. Figure 2 shows two control-flow hijackings examples, as reflected from audit log. In the first example (Figure 2a), the expected system call `write (1, entry->d_name)` (at line 10) was bypassed by the exploit. As a result, the entries in lines 3 and 4 were not recorded in audit log. In the second example (Figure 2b), one additional system call `write (1, entry->d_name)` was injected at runtime, and thus two extra audit entries were generated in lines 3 and 4. These two examples show that control-data attacks force the behavior of an application to deviate from its expected behavior, which is described by its corresponding FSM model.

It is worthwhile noting that the audit log might be encrypted and thus unintelligible. A cloud user should not be able to tell whether the application was executed as expected until she obtains the cryptographic key to decipher the encrypted audit log. This also poses a potential risk that a malicious and privileged user, who owns the cryptographic key, injects arbitrary content into the audit log and sends it to the cloud user.

```
1 void listdir(char *name)
2 {
3     DIR *dir; struct dirent *entry; int count;
4     dir = open(name, RD);
5     while (getdents(dir, entry, count) > 0) {
6       foreach (dir_entry in entry){
7         if (isdir(entry->d_type)) {
8            char path[1024];
9            snprintf(path, 1024, "%s/%s", name, entry->d_name);
10            write(1, entry->d_name);
11           listdir(path);
12        } else {
13           write(1, entry->d_name);
14        }
15      }
16    }
17    close(dir);
18 }
19 int main (int argc, char *argv[]){
20    listdir(argv[1]);
21    exit(0);
22 }
```

(a) The code snippet of *simple_traversal.c*



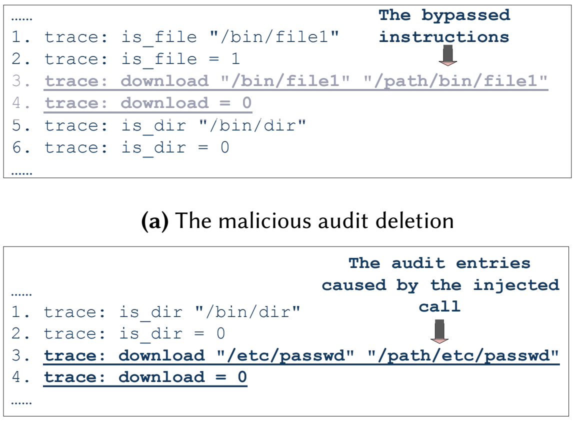
(b) The FSM of *simple_traversal.c*

Figure 1. A Motivating Example

2.2. System Overview

VERDICT is designed to fulfill two major objectives: 1) ensuring the *confidentiality* of the formal model that specifies the expected behavior of a delegated job and the audit log; and 2) guaranteeing that the state of a delegated job is verifiable, without disclosing the verification rules. As illustrated in Figure 3, VERDICT contains three key components: 1) *Event Generator* generates the audit events and encrypts them right after their generation; 2) *State Updater* cryptographically updates the model in ciphertext upon the newly generated audit log; and 3) *State Verifier* verifies the state of the delegated job based on the updated model and the audit events, both of which are confidential.

VERDICT verifies the state of a delegated job s in seven steps. First, the key generator generates the HE key pair and passes the public key, HK^+, to the encryption engine (step ❶). Once the event log e is generated, it will immediately be encrypted by HK^+ (step ❷). The timestamp of the audit event will be encrypted with an Order-Preserving Encryption (OPE) cipher [33]. The output of both ciphers will be kept as encrypted audit logs for future verification purposes (step ❸). A model of the delegated job, constructed from an application as an FSM, will be represented as a loosely-coupled data structure, namely the *dual-vectors*. The partial structure of dual-vectors is kept in sandboxes. The detailed process of how to define the dual-vectors will be elaborated in Section 4.1. Two vectors of the dual-vectors, namely U and V, are kept and updated in the two state-updaters SU_1 and SU_2 independently (steps ❹ and ❺). The detailed process of how to update U and V upon observation will be presented in Section 4.2. When a cloud user initiates a query to the state of a job, the recent copy of U and V, namely U' and V', are sent to the verification engine (step ❻). Then, the state verifier determines the current state of



```
......
trace: is_file "/bin/file1"
trace: is_file = 1
trace: download "/bin/file1" "/path/bin/file1"
trace: download = 0
trace: is_dir "/bin/dir"
trace: is_dir = 0
......
```

The bypassed instructions

(a) The malicious audit deletion

```
......
trace: is_dir "/bin/dir"
trace: is_dir = 0
trace: download "/etc/passwd" "/path/etc/passwd"
trace: download = 0
......
```

The audit entries caused by the injected call

(b) The malicious audit injection

Figure 2. Malicious Audit Manipulations

the delegated job. Optionally, the encrypted log will be fed into the state verifier for offline verification (step ❼).

3. The Threat Model

First, we assume that the state verifier is running inside a sandbox, whose trustworthiness is guaranteed by its underlying TCB. It generates and maintains the secret keys for HE and OPE. We also assume that both pseudorandom number generator (PRNG) and HE cipher used by the state verifier are *unbreachable*. The reason for this requirement is at least two-fold: First, it makes sure that the pseudorandom number used as a one-time pad for the HE cipher is *cryptographically secure* [34]. Second, it ensures the secrecy of the pseudorandom numbers, which might not be fully trusted. We describe the mechanism of the state updaters in Section 4.2. Given that the secret inside a sandbox is susceptible to the side-channel attacks [23], [35], [24], [36], [37], we assume that the state verifier is trustworthy when the secret key is generated. We also assume that cloud providers run *honest-but-curious* audit service. To ensure the integrity of the audit trail, the audit log must be encrypted right after its generation.

The capabilities of an adversary can be modeled as follows. First, the adversary can hijack the behavior of an application by launching a control-data attack [32]. The behavior of the hijacked application can be recorded by the audit trail, whose content is secure. Second, the trustworthiness of each sandbox that runs a state updater might not be guaranteed. Once a sandbox is compromised, the adversary is capable of obtaining the ciphertext of the encrypted model, as well as running brute-force attacks to break the ciphertext. Third, the adversary, with high administrative privileges, can launch an insider attack to introspect memory space relevant to HE operations and try to infer the rules of model updating. Even though the HE ciphertext is unbreachable, an advanced adversary can manipulate memory content by replacing HE ciphertext with an arbitrary value. Therefore, we assume that the memory address of the ciphertext might not be continuous, which can be guaranteed by some existing techniques, including address space layout randomization [38] or Oblivious RAM [39, 40]. Finally, a collusion attack [41] might be possible if an adversary can control several sandboxes in which the state-updaters are deployed. We assume that the adversary can compromise partial sandboxes, but not all of them. The countermeasure of how to defend collusion attacks will be discussed in Section 8.

4. Detailed Design

In this section, we present the detailed design of VERDICT in the order of model initialization, updating, and verification.

4.1. Constructing Secure Model

To construct a model that describes the expected behavior of an application and keeps its content confidential, we need to overcome at least two challenges. First, we need to formalize the model of a delegated job to represent a generic application and keep tracking its state based on the observation. Second, the model should be encrypted in such a way that it is cryptographically difficult for an adversary to learn any rule or reason about the state of an application. To overcome the first challenge, we formalize the model as a finite state machine (FSM) [42], which has been extensively applied as an efficient way to check the behavior model of a system [43], model verification [44], and intrusion detection [45]. The notations used in the remainder of the paper are listed in Table 1.

Table 1. Summary of Notations

Notation	Definition
j	The delegated job.
$\mathcal{M}_j$	The FSM that models j.
$DV(\mathcal{M}_j)$	The dual-vector of $\mathcal{M}_j$.
$sizeof(\alpha)$	The size of vector α $\quad(\alpha = u$ or $v)$.
$\mathbb{R}_e$	The set of e-Relation.
$\mathbb{R}_i$	The set of i-Relation.
HK^+	The public key of HE cryptographic cipher.
$+++HK^+$	The public key of HE cryptographic cipher.
HK^-	The private key of HE cryptographic cipher.
$En^k(m)$	The encryption of message m with key k.
$De^k(m)$	The decryption of message m with key k.
$C.plain$	The plaintext of ciphertext C.

Definition 1 (The Security Model $\mathcal{M}_j$). The model used to describe the behavior of a delegated job s can be defined as a deterministic FSM $\mathcal{M}_j$, which is a quintuple $\mathcal{M}_j$ = $(\Sigma, S, s_0, \Delta, S_F)$:

- Σ is an *event alphabet* with a finite number of symbols. In our scheme, each symbol e represents an observable audit event from a delegated job.

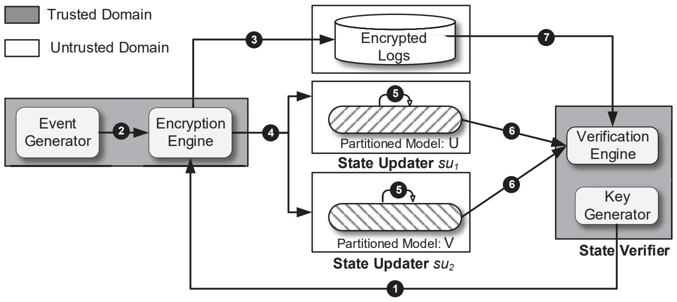


Figure 3. An Overview of VERDICT

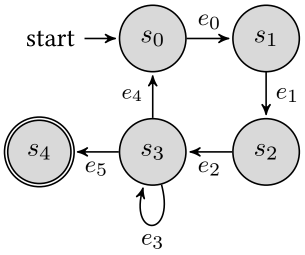


Figure 4. The sample finite state machine $\mathcal{M}_j$

	e_0	e_1	e_2	e_3	e_4	e_5
s_0	s_1	-	-	-	-	-
s_1	-	s_2	-	-	-	-
s_2	-	-	s_3	-	-	-
s_3	-	-	-	s_3	-	-
s_3	-	-	-	-	s_0	-
s_3	-	-	-	-	-	s_4

Figure 5. The transition function $\mathcal{M}_j.\Delta$

- S is a finite set of states. Here, a state s_i could be a location in a program [45], a unique value of the Program Counter (PC) register, or a unique snapshot of the program's call stack [46].
- s_0 is the initial state, where $s_0 \in S$.
- Δ is a state transition function: $\Delta : S \times \Sigma \rightarrow S$.
- S_F is the set of final states. In accordance with the general definition of FSM, we include an additional state $s_e \in S_F$, which indicates an erroneous state that leads the FSM to halt with errors.

Example 1. Figure 4 illustrates a sample $\mathcal{M}_j$ comprises the alphabet $\Sigma = \{e_i\}$ $(0 \le i \le 5)$, the set of states $S = \{s_i\}$ $(0 \le i \le 4)$, the initial state s_0, and the final state set $F = \{s_4\}$. Figure 5 tabulates the transition function $\mathcal{M}_j.\Delta$.

To overcome the second challenge, our scheme makes a key observation: if we can keep the secrecy of 1) the set of states S and 2) the transition function Δ, we will be able to prevent the adversary from reasoning about the current state of a system. However, to meet this requirement is not easy because the model of the delegated job might run upon an untrusted node. If an adversary can somehow correlate the memory locations relevant to each HE operation, she could potentially infer the states and transition function of a model. This challenge motivates us to decouple the states from the transition function, and run partitions of the model independently. To do that, we define a unique data structure for $\mathcal{M}_j$, namely *Dual-Vector* $DV(\mathcal{M}_j)$, which keeps the state information of $\mathcal{M}_j$. The definition of $DV(\mathcal{M}_j)$ is listed below:

Definition 2 (The Dual-Vector DV($\mathcal{M}_j$)). A DV($\mathcal{M}_j$) comprises two vectors of finite size, namely $U[sizeof(U)]$ and $V[sizeof(V)]$. In the vectors, the initial value of each element is first randomly chosen and encrypted by HK^+. Then, the vectors U and V are saved in different sandboxes.

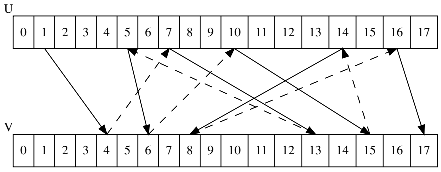
Along with DV($\mathcal{M}_j$), we also define two relations: *explicit relation* and *implicit relation*, namely *e-Relation* and *i-Relation*, which are used as an alternative to the existing transition function $\mathcal{M}_j.\Delta$. More specifically, e-Relations are used to update an FSM based on the observed events, while i-Relations are used to verify the current state of $\mathcal{M}_j$. The definitions of e-Relations and i-Relations are listed below:

Definition 3 (e-Relation $\mathbb{R}_E$ and i-Relation $\mathbb{R}_I$). Given an FSM $\mathcal{M}_j$ that expresses a delegated job s, for each transition $\delta_i \in \mathcal{M}_j.\Delta$, there is a mapping relation $\mathbb{R} : \delta_i \rightarrow [\![index_u, index_v]\!]$ and $\mathbb{R} \in \mathbb{R}_E \cup \mathbb{R}_I$, in which $index_\alpha$ $(\alpha = u$ or $v)$ refers to the set of indices in the vectors U and V, respectively. There are two kinds of relations:

- The e-Relation $\mathbb{R}_E$ is a binary relation, denoted as $\mathbb{R}_E^i : [\![index_u^i,\ index_v^i]\!]$, which corresponds to one transition δ_i: $s_i \times e_i \rightarrow s_{i+1}$. The state s_{i+1} is the next state given the current state s_i and the event e_i, respectively.
- The i-Relation $\mathbb{R}_I$ is a binary relation, denoted as $\mathbb{R}_I^i : [\![index_v^i,\ index_u^{i+1}]\!]$, which corresponds to two transitions δ_i: $s_i \times e_i \rightarrow s_{i+1}$ and δ_{i+1}: $s_{i+1} \times e_j \rightarrow s_{i+2}$ in sequence. The corresponding e-Relation of the

Transition	$\mathbb{R}_E$
δ_0	$[\![\ 1,\ \ 4\ ]\!]$
δ_1	$[\![\ 7,\ 13\ ]\!]$
δ_2	$[\![\ 5,\ \ 6\ ]\!]$
δ_3	$[\![\ 10,\ 15\ ]\!]$
δ_4	$[\![\ 14,\ \ 8\ ]\!]$
δ_5	$[\![\ 16,\ 17\ ]\!]$

(a) The e-Relations correspond to transitions δ_i.



(b) The dual-vector DV($\mathcal{M}_j$).

Figure 6. The time spent on updating the model

two transitions are $\mathbb{R}_E^i : [\![index_u^i, index_v^i]\!]$ and $\mathbb{R}_E^{i+1} : [\![index_u^{i+1}, index_v^{i+1}]\!]$.

We further elaborate on the relations of $\mathbb{R}_E$ and $\mathbb{R}_I$, with the sample FSM shown in Example 1.

Example 2. Figure 6a illustrates the transitions δ_i ($0 \leq i \leq 5$) and their corresponding $\mathbb{R}_E$. Each row in the table represents an $\mathbb{R}_E^i$. Figure 6b illustrates the $\mathbb{R}_E$ and $\mathbb{R}_I$ derived from DV($\mathcal{M}_j$). Specifically, the solid arrow that points from an element of U to an element of V is corresponding to an e-Relation, while the dotted arrow that points from an element of V to an element of U is corresponding to an i-Relation. For example, the transition $\delta_1 : s_0 \times e_1 \rightarrow s_1$ (row 2) is mapped to the e-Relation $\mathbb{R}_E^1 : [\![1, 4]\!]$, which is corresponding to the solid arrow points from $U[1]$ to $V[4]$. Similarly, the transition $\delta_2 : s_2 \times e_2 \rightarrow s_3$ (row 3) is mapped to the e-Relation $\mathbb{R}_E^2 : [\![7, 13]\!]$, which corresponds to the solid arrow pointing from $U[7]$ to $V[13]$. Given these two adjacent e-Relations, the i-Relation is $\mathbb{R}_I^1 : [\![4, 7]\!]$ corresponds to the dotted arrow points from $V[4]$ to $U[7]$.

The design of decoupling the formal model as dual-vector and manipulating its content with e-Relation and i-Relation has several advantages. First, since the state update operations and state verification operations are dictated by $\mathbb{R}_E$ and $\mathbb{R}_I$, respectively, it prevents an adversary from inferring the transition function from the state updaters. When $\mathcal{M}_j$ is updated, the state updaters update the elements in U and V independently by following the relations in $\mathbb{R}_E$, which is public to an adversary. However, when the state of $\mathcal{M}_j$ is verified, the state verifier follows the relations in $\mathbb{R}_I$, which is unknown to the adversary. Therefore, only the state verifier can identify the state of the model.

Second, as an online component, each state updater is only responsible for performing HE re-computation against the element that matches the index of the corresponding e-Relation. Only one HE addition or subtraction operation is needed in each vector for an observed event. This property ensures that HE operations in each sandbox are bounded at run time, though the HE operations are considered as computationally costly.

```
Algorithm  Update_Partitions
Input: The pair $P_i = \{index_\alpha, E(r)\}$ ($\alpha = u$ or $v$), the vectors
    $U$ and $V$, which contain the HE ciphertext, and the public key
    of HE crypto $HK^+$.
Output: The updated vectors $U'$ in $SU_1$ and the $V'$ in $SU_2$.
1: switch updater do
2:     ▷ Perform HE subtraction on Vector $U$.
3:     case $SU_1$
4:         HE_Sub ($U[index_u]$, $En^{HK^+}(r)$, $HK^+$);
5:     ▷ Perform HE addition on Vector $V$.
6:     case $SU_2$
7:         HE_Add ($V[index_v]$, $En^{HK^+}(r)$, $HK^+$);
End of Algorithm
```

4.2. Updating Secure Model

Once the model of the delegated job is partitioned and deployed in different sandboxes, they will be updated based on the encrypted audit log. Recall that in Section 2.2, the model of the delegated job is partitioned into two vectors (U and V) and saved in the state updaters (SU_1 and SU_2), respectively. The key challenge of updating the elements in the state updater is that both U and V only contain the HE ciphertext. If updated, they can only be updated with the HE ciphertext as well.

To update elements in U and V with HE ciphertext, we take the following steps: for each event e_i, which satisfies the transition δ_i: $s_i \times e_i \rightarrow s_{i+1}$, the event generator first produces two pairs, namely $P_i = [index_\alpha,\ En^{HK^+}(r)]$ ($\alpha = u$ or v). Then, the pairs are sends to SU_1 and SU_2, respectively. In each pair, the $index_\alpha$ ($\alpha = u$ or v) are the indices of an e-Relation $R_E^i : [\![index_u^i,\ index_v^i]\!]$. $En^{HK^+}(r)$ is the HE ciphertext of pseudorandom number r that has been encrypted by HE public key HK^+. Algorithm *Update_Partitions* presents the procedure of updating vectors U and V upon receiving a pair. This algorithm uses two generic HE operations, namely *HE_Add* for HE addition and *HE_Sub* for HE subtraction. The algorithm is executed each time the state updater receives a P_i. For the state updater that contains U, the element at $index_u$, $U[index_u]$, will be *homomorphically* subtracted by $En^{HK^+}(r)$ (line 4). For the state updater that contains V, the element at $index_v$, $V[index_v]$, will be *homomorphically* added by $En^{HK^+}(r)$ (line 7). Since Algorithm *Update_Partitions* ensures that a transition δ_i in $\mathcal{M}_j.\Delta$ only impose one HE operation (either HE addition or HE subtraction) to a vector, the time complexity of algorithm *Update_Partitions* is $O(1)$. It is worth noting that to send $index_\alpha$ ($\alpha = u$ or v) as plaintext is vulnerable to security breaches. The solution to solve this problem will be discussed in section 8.

Algorithm *Verification_State*

```
Input: The updated vectors $U'$ and $V'$, the original copy of
    partitions $U_{init}$ and $V_{init}$, and the pseudorandom number $r$.
Output: The ID of the current state of the model $\mathcal{M}_j$ if success,
    or -1 if fail.
1:  ▷ Cancel out the non-final states.
2:  for each $\delta_i \in \Delta$
3:      $(index_v, index_u) \leftarrow$ Lookup_Imp $(i)$ ;
4:      $temp \leftarrow$ HE_Add($V[index_v], V[index_u]$);
5:      if $De^{HK^-}(temp) == V[index_v].plain + U[index_u].plain$ then
6:          diff $= De^{HK^-}(V[index_v]) - V[index_v].plain$;
7:          $temp\_V[index_v] \leftarrow$ HE_Sub($V[index_v], En^{HK^+}$(diff));
8:          $temp\_U[index_u] \leftarrow$ HE_Add($U[index_u], En^{HK^+}$(diff));
9:  ▷ Now, start to traverse the transition function and determine the
    final state
10: for each $\delta_i \in \Delta$
11:     $(index_u, index_v) \leftarrow$ Lookup_exp $(i)$ ;
12:     if $(De^{HK^-}(temp\_U[index_u]) == U_{init}[index_u].plain)$ and $(De^{HK^-}(temp\_V[index_v]) == V_{init}[index_v].plain + r)$ then
13:         return $i$;
14:     if $De^{HK^-}(temp) == V[index_v].plain + U[index_u].plain$ then
15:         return -1;
End of Algorithm
```

4.3. Verifying Secure Model

Once the cloud user requests to check the state of an FSM that corresponds to the monitored program of interest, she will request both U' and V' from SU_1 and SU_2 and perform the *Verification_State* algorithm. U' and V' denote the vectors that have been updated in the state updater. As illustrated in the algorithm, it takes the current values of U' and V' as input and outputs the ID of the state as defined by the FSM if succeeded, or -1 if failed.

The algorithm uses standard HE encryption $En^k(m)$ and HE decryption $De^k(m)$. In addition, two functions *Lookup_Exp* and *Lookup_Imp* are used to retrieve the indices from the explicit and implicit relations, respectively. Nevertheless, the initial copies of U and V, namely U_{init} and V_{init}, are also needed. The algorithm contains two parts. Lines 1 - 8 cancel out the intermediate states, which are the states that have been involved in the state transition during run-time but are not included in the set of final states. Lines 10 - 15 traverse the transition function and determine the final state.

5. Security Analysis

In this section, we analyze the security properties that our scheme entails, including the trade-off of the vector size and the resilience against cryptanalysis.

First, three parameters are important to preserve the secrecy of the model. The first two parameters are the sizes of vector U and V, namely $sizeof(U)$ and $sizeof(V)$, which are governed by the complexity of the FSM, or more specifically, the number of states and transitions in the FSM. The values of $sizeof(U)$ and $sizeof(V)$ might be different, as long as they can accommodate all the e-Relations. The third parameter is $En^{HK^+}(r)$, which is the HE ciphertext of the pseudorandom number r. This parameter should be first generated by the state verifier and then passed to SU_1 and SU_2. Each time when the *Update_Partitions* algorithm is called, the $En^{HK^+}(r)$ will be performed against the ciphertext of U and V. As we have assumed, the HE ciphertext is unbreachable, thus the adversary may not be able to decipher this parameter. Second, as it was stated in Section 3, our scheme cannot prevent an adversary from stealing the ciphertext of U and V and launching a brute-force attack to try to crack HE ciphertext. However, even if the adversary successfully deciphers the plain-text content of U and V, if she cannot reason about the complete set of $\mathbb{R}_I$ from $\mathbb{R}_E$, she will still not be able to tell the state of the model. Further, to successfully infer i-Relation also requires that the adversary controls all the state updaters and observes every HE update operation. Obviously, this requirement contradicts our assumption that the adversary can only compromise partial sandboxes that run the state updates, but not all of them. Third, it is possible that an adversary deliberately sabotages the scheme by updating the secure model arbitrarily, without executing Algorithm *Update_Partitions*. If that is the case, such an attack can easily be detected because Algorithm *Verification_State* will return -1. Finally, since the state updaters only have HE ciphertext and HE public key, the adversary will not obtain sufficient information about the behavior model even if she launches a side-channel attack.

6. Detecting Abnormal Activities

As an important design goal, VERDICT can not only tell the state of a delegated job but also detect abnormal activities that violate the control-flow integrity by deviating from the expected behavior model. We categorize such abnormal activities into two categories: malicious *deletion* and malicious *repetition*. Specifically, malicious deletion deliberately omits certain critical steps during the execution of the delegated job; while malicious repetition intentionally repeats certain steps during the execution of the delegated job.

Figure 7 illustrates an example of detecting malicious deletion. In the figure, the symbols $\oplus$ and $\ominus$ indicate the operations of HE addition and subtraction, respectively. The HE summation of $V[index_v^{k-1}]$ and $U[index_u^k]$ ($k = i$ and $i + 1$) in the first column indicates a verification step that follows the i-Relation. The second column indicates the summation of the plaintext values of these two cells, which is C_i. If one event is deliberately missing, it will cause one additional HE addition and one additional HE subtraction in two consecutive verification steps, as shown in the third

Calculation	Init. Values	Extra	Decrypted value
	in plaintext	HE Ops.	in plaintext
$V[index_v^{i-1}] \oplus U[index_u^i]$	C_i	$\oplus En^{HK^+}(r)$	$C_i + r$
$V[index_v^{i}] \oplus U[index_u^{i+1}]$	C_{i+1}	$\ominus En^{HK^+}(r)$	$C_{i+1} - r$

Figure 7. Malicious deletion attack

Calculation	Init. Values	Extra	Decrypted values
	in plaintext	HE Ops.	in plaintext
$V[index_v^{i-1}] \oplus U[index_u^i]$	C_i	$\ominus En^{HK^+}(r)$	$C_i - r$
		$\ominus En^{HK^+}(r)$ for $n-1$ times	$C_1 - r \times (n-1)$
$V[index_v^{i}] \oplus U[index_u^{i+1}]$	C_{i+1}	$\oplus En^{HK^+}(r)$	$C_{i+1} + r$
		$\oplus En^{HK^+}(r)$ for $n-1$ times	$C_{i+1} + r \times (n-1)$

Figure 8. Malicious repetition attack

column. As a result, the recovered plaintext will produce $C_i + r$, where r is the pre-defined pseudorandom number specified and known by the state verifier.

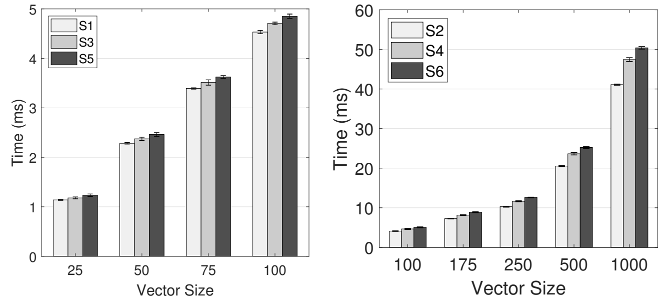
Figure 8 illustrates an example of detecting malicious repetition. Similar to the previous example, we follow i-Relations and calculate the HE summation of $V[index_v^{k-1}]$ and $U[index_u^k]$ ($k = i$ and $i+1$). If an event is deliberately repeated once, it will cause one additional HE subtraction and one additional HE addition in two consecutive verification steps. Thus, the recovered plaintext will be $C_i - r$ and $C_i + r$ for the two steps. Likewise, if an event repeats more than once, say n times, the verification results will be $C_i - r \times (n-1)$ and $C_i + r \times (n-1)$.

7. Implementation & Evaluation

We implemented VERDICT using C++ (around 500 lines of code). We used `HElib` [47], a fully homomorphic encryption (FHE) library that supports HE operations. We used `libguestfs` [48] to generate audit logs. We evaluated VERDICT on four popular cloud infrastructures, including 1) Amazon AWS [27], 2) Google Cloud [29], 3) Microsoft Azure [28], and 4) IBM Cloud (DC cluster) [30]. To demonstrate that VERDICT can leverage different kinds of sandboxes, we deployed the online component of VERDICT in both VMs and containers on each infrastructure, whose configurations are listed in Table 2. For simplicity, we use S_i $(1 \leq i \leq 6)$ to denote the VMs of different size. To accommodate the updater with different sizes of the encrypted vectors, we choose different computing nodes from a variety of cloud infrastructures. The ranges of the vector sizes are also listed. Apparently, the larger the vector size, the higher-performance VMs we have acquired from cloud infrastructures.

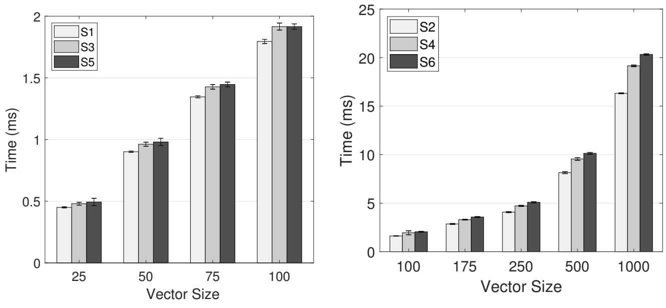
7.1. Computational Overhead

The first set of experiments is to analyze the time spent on initializing, updating, and verifying the model with vectors of different sizes. We run this experiment in each VM 100 times and calculate the mean and standard deviation of the execution time. From the experiments, we made the following observations across different infrastructures. First, as illustrated in Figure 9 and Figure 10, the time spent on initializing and verifying the model is nearly linear to the sizes of vectors. All three infrastructures show nearly similar timing statistics for model initialization and verification. Second, since the online component of VERDICT, the state



(a) vector sizes: 25 - 100 **(b)** vector sizes: sizes: 100 - 1000

Figure 9. The time spent on initializing the model



(a) vector sizes: 25 - 100 **(b)** vector sizes: sizes: 100 - 1000

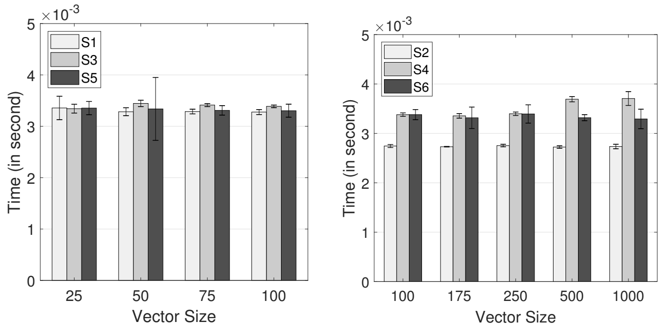
Figure 10. The time spent on verifying the model

Table 2. System Configurations of VMs on Different Cloud Infrastructures

	Amazon AWS		Google Cloud		Microsoft Azure	
Settings	`t2.small (S1)`	`t2.xlarge (S2)`	`n1-standard-1 (S3)`	`n1-standard-4 (S4)`	textttStandard DS4 v3 (S5)	`DS2 v4 (S6)`
CPU	1 vCPU	4 vCPUs	1 vCPU	4 vCPUs	1 vCPU	4 vCPUs
Memory	2 GB	16 GB	2 GB	15 GB	2 GB	16 GB
Vector Sizes	25 – 100	100 – 1000	25 – 100	100 – 1000	25 – 100	100 – 1000

Table 3. System Configurations of VMs on Different Cloud Infrastructures

	A1	A2	A3	A4	A5
Settings	t2.small	t2.medium	t2.large	t2.xlarge	t2.2xlarge
CPU	1vCPU	2vCPU	2vCPU	4vCPU	8vCPU
Memory	2GB	4GB	8GB	16GB	32GB



(a) vector sizes: 25 - 100 **(b)** vector sizes: sizes: 100 - 1000

Figure 11. The time spent on updating the model

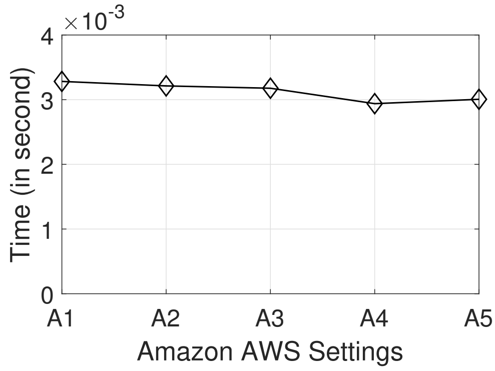


Figure 12. The update time for different configurations of VMs

updater, only performs one HE addition or subtraction operation on each vector at run time, it incurs constant computational costs when performing HE operations. As illustrated in Figure 11, for a particular infrastructure, the average time spent on each VM is nearly constant. This information will guide us to determine the performance upper bound of each auditable event. For those time-critical applications, whose timing requirement is more lenient than the performance upper bound, we can adapt VERDICT without worrying about impairing accountability. Third, it shows the different performances for different infrastructures: for the same configuration, the VMs in Amazon AWS outperform the VMs in Google Cloud and Microsoft Azure, regarding all timing metrics.

7.2. The Update Time

The second set of experiments study how different configurations of VMs affect the time spent in updating vectors. For this set of experiments, we use five VMs using different configurations as tabulated in Table 3. Figure 12 illustrates that a higher configuration of a VM does not significantly improve the performance of updating vectors. In other word, the state updaters, which are the online component of VERDICT, achieve similar performance, regardless of the configuration of VMs.

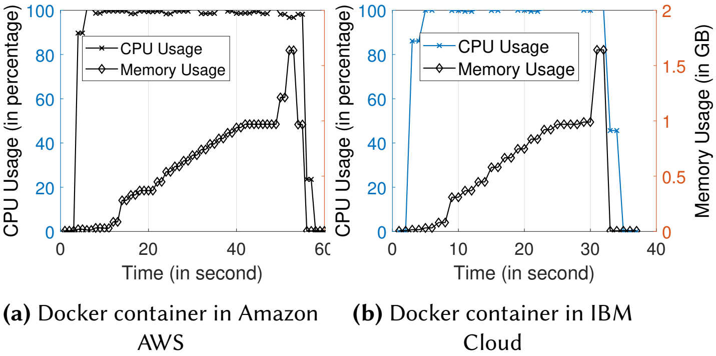
7.3. The Applicability of Using the Container

The last set of experiments is to measure the resource consumption of executing VERDICT inside a container, during its execution. We install Docker containers [49] in both Amazon AWS and IBM Cloud (DC cluster). Figure 13a and Figure 13b illustrate the CPU and memory usage by running VERDICT in a Docker container in Amazon AWS and IBM Cloud, respectively. The containers in both infrastructures show a similar performance pattern.

8. Discussion

In this section, we first discuss some limitations of the proposed scheme. Then, we suggest the possible solutions, Finally, we identify some directions for our future study.

First, homomorphic encryption offers an appealing feature of performing computations over ciphertext, while being complained about its low efficiency in handling



(a) Docker container in Amazon AWS

(b) Docker container in IBM Cloud

Figure 13. The CPU and memory usage for Docker containers in AWS and IBM Cloud

high-performance applications [50]. Although we made similar observations through the experiments, we conclude that the slow-down incurred by homomorphic encryption does not limit its applicability for most applications, which do not require an online response. The online component of VERDICT, the state updater, performs only one HE operation (either addition or subtraction) upon an observed audit record, which is nearly constant. The performance bottlenecks, including vector initialization and state verification, can be performed offline. Thus, our approach is still applicable to online auditing systems, if the performance of state updaters satisfies the timing constraint of those applications. In our future study, we will test real-world applications and explore the possible solution to mitigate the performance gap.

Second, recall that the event generator sends $P_i = [index_\alpha, En^{HK^+}(r)]$ $(\alpha = u$ or $v)$ to the vectors. The plaintext of $index_\alpha$ poses security breaches. One possible solution is that the event generator first creates two mirror vectors, namely Mirror_U[*sizeof(U)*] and Mirror_V[*sizeof(V)*], which are of equal size to U and V, respectively. Then, *Mirror_U* and *Mirror_V* are initialized as follows:

$$Mirror\_U[i] = \begin{cases} En^{HK^+}(r), & \text{if } i = index_u; \\ En^{HK^+}(0), & \text{if } i \neq index_u; \end{cases} \tag{1}$$

and

$$Mirror\_V[i] = \begin{cases} En^{HK^+}(r), & \text{if } i = index_v; \\ En^{HK^+}(0), & \text{if } i \neq index_v; \end{cases} \tag{2}$$

After that, the mirror vectors *Mirror_U* and *Mirror_U* are sent to SU_1 and SU_2, respectively. Finally, each element in *Mirror_U* will perform a HE subtraction with the element on the same index in U. Similarly, each element in *Mirror_V* will perform a HE addition with the element on the same index in V. This process preserves the secrecy of indices while still allowing the specific elements in U and V to be updated. The only drawback is that it will increase the time complexity from $O(1)$ to $O(n)$. More cost-efficient schemes will be developed to reduce the computational complexity and still keep the information confidential.

Finally, in this paper, we primarily focus on retaining the secrecy of the model, without constructing a TCB. Our current design requires a model to be generated as the result of a static analysis from a program. The model that we have studied is deterministic, instead of probabilistic. As a future direction, we will apply some learning algorithms, such as Hidden Markov Model (HMM), to build a non-deterministic and probabilistic model. In addition, we will consider the side-channel attacks, which could possibly infer the execution path of the delegated job by analyzing the resources used by VERDICT.

9. Related Work

Homomorphic Encryption As a cryptosystem that preserves privacy for secure multi-party computation, Homomorphic Encryption (HE) allows an untrusted party to perform computating upon encrypted data. The two primary types of HE cryptosystems are partial homomorphic encryption (PHE) and fully homomorphic encryption (FHE). PHE supports the HE operations/primitives of additive homomorphic encryption and multiplicative homomorphic encryption separately, while FHE supports both additive homomorphic encryption and multiplicative homomorphic encryption over ciphertext simultaneously. Gentry [50] proposed the first FHE scheme based on lattice-based cryptography with numerous follow-up work [25, 26, 51]. Although researchers show that HE is computationally costly and only supports limited cryptographic operations [52], they have nonetheless been widely adopted in various applications [53], [54], and [55]. Li et al. [54] introduced a distributed incremental data aggregation scheme, in which data collected from smart meters can be aggregated along the route from the source meter to the collector unit. HE is used to protect user's private data en route. As a result, the involved meters cannot tell any intermediate or final results. Hong et al. [55] presented a collaborative search log sanitization scheme, which allows multiple parties to collaboratively generate search logs with boosted utility while satisfying differential privacy. To this end, a protocol, namely *CELS*, was proposed to meet the privacy-preserving objectives. Compared with existing HE applications, our approach serves for a specific purpose of verifying the state of a delegated job through the encrypted audit log it generates. It encrypts the behavior model extracted from an application and updates the state of that model through the HE cipher. In such a way, the encrypted model can be deployed on a platform whose TCB might not have been constructed.

Recent research demonstrates that HE might be vulnerable to various attacks. Bogos et al. [34] performed cryptanalysis against the HE scheme proposed by Zhou and Wornell [56], which encrypts data in the form of integer vectors. The analysis results show that the specific HE cryptosystem is vulnerable to at least three attacks: the broadcast attack, the chosen-ciphertext key recovery attack, and the chosen-plaintext decryption attack. Our paper does not restrict any specific HE cryptosystem and explores the application of a general HE scheme that is aimed at enabling arbitrary computing tasks. The research of a specific vulnerable HE cryptosystem is out of the scope of this paper.

Behavior Modeling for Anomaly Detection A number of anomaly detection approaches, based on behavior modeling, have been proposed to detect host-based intrusion [45, 46]. Sekar et al. [43] presented a framework that detects anomalous program behaviors using FSM. As a follow-up, Bhathar et al. [45] proposed a technique that can generate the behavior model of the program based on learning the temporal properties of system call arguments. Both approaches model the expected behavior of a program as an FSM or an augmented FSM, which can be used to detect the abnormalities that deviate from the expected behaviors. A common assumption of both approaches is that the machine that runs the intrusion detection system is trustworthy; therefore, there is no need to preserve the secrecy of the behavior model. Unfortunately, this assumption can be invalided due to the recent exploits that compromised the OS and the hypervisor. Compared with these approaches, VERDICT makes realistic assumptions on the trustworthiness of the computing nodes by preserving the confidentiality of the behavior model. Meanwhile, the specific features of the HE cryptosystem allow the encrypted behavior model to be used for verifying the state of the delegated job. Therefore, it is more appropriate for a computational environment, in which the guarantee of its trustworthiness might not be available.

Remote Attestation In general, the problem presented in this paper is closely related to the remote attestation problem [57], in which a remote party, namely the *verifier*, verifies the trustworthiness or the integrity of the software running upon an untrusted device, namely the *prover* [14]. Most remote attestation approaches are static, which only prove the integrity of software initially loaded by a prover [58–60] and are incapable of detecting run-time attacks [61]. Recent technology enables a verifier to detect run-time attacks based on control-flow correctness such that it overcomes the limitations of static attestation [14, 62]. The enforcing techniques, such as control-flow integrity (CFI) [11], enable a verifier to detect the attacks that explicitly violate the model that describes a software normal behavior [61] or implicitly cause a valid but unintended program execution behavior [32, 38]. The primary difference in objective between the remote attestation and our approach is that the remote attestation is to let a verifier determine the degree of trust in the software from the platform of a prover, while our approach determines the state of the software, which is delegated by the user. The technique that verifies the correctness of control-flow still trusts the verification result generated by the prover, which might be compromised.

10. Conclusions

To ensure the accountability and model checking of the delegated jobs running on the cloud, we presented VERDICT that verifies the state of a delegated job through a confidential behavior model and audit log. We leveraged the technique of Homomorphic Encryption, partitioned the model, and stored the partitions in a distributed manner to achieve confidentiality and privacy. This scheme can be deployed across various computing environments without relying on any specific trusted hardware. We presented the algorithms to update and verify the state of a delegated job as well as to detect abnormalities that violate the control-flow integrity. Lastly, our experimental evaluation demonstrated that the VERDICT only introduces a bounded performance overhead for the HE operations invoked in model updating and verification.

Acknowledgment

The work is supported by the National Science Foundation under Grant No. DGE-1723707 and Michigan Space Grant Consortium. The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

References

[1] Crinimal Justice Infromation Services (CJIS), [Online]. Available: https://www.fbi.gov/services/cjis.

[2] C. S. Alliance, "CSA Security, Trust & Assurance Registry (STAR)," [Online]. Available: https://cloudsecurityalliance.org/star/#_overview.

[3] Cyber Essentials Plus, [Online]. Available: https://www.cyberessentials.org/.

[4] Cloud Audit Logging, [Online]. Available: https://cloud.google.com/logging/docs/audit/.

[5] AWS CloudTrail, [Online]. Available: https://aws.amazon.com/cloudtrail/.

[6] Monitor Subscription Activity with the Azure Activity Log, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-monitor/platform/activity-logs-overview.

[7] Azure Monitor, [Online]. Available: https://azure.microsoft.com/en-us/services/monitor/.

[8] Amazon Web Services: Risk and Compliance - Amazon S3 - AWS, [Online]. Available: https://s3-us-west-2.amazonaws.com/naspovaluepoint/1484182671_DLT_File_08_-_Appendix_2_AWS_Risk_and_Compliance_Whitepaper.pdf.

[9] U.S. Department of Health & Human Services, "$5.5 million HIPAA settlement shines light on the importance of audit controls," [Online]. Available: https://www.hhs.gov/about/news/2017/02/16/hipaa-settlement-shines-light-on-the-importance-of-audit-controls.html.

[10] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 340–353.

[12] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated Software Diversity,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer Integrity,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 147–163.

[14] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-FLAT: Control-Flow Attestation for Embedded Systems Software,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[15] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, “Efficient Protection of Path-Sensitive Control Security,” in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 131–148.

[16] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, “Enforcing Unique Code Target Property for Control-Flow Integrity,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.

[17] ARM, *ARM Security Technology - Building a Secure System using TrustZone Technology*, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[18] Intel Processor Trace (IPT), [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[19] T. Bletsch, “Code-reuse Attacks: New Frontiers and Defenses,” Ph.D. dissertation, 2011.

[20] C. Gentry, “Fully Homomorphic Encryption Using Ideal Lattices,” in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178.

[21] Intel Corporation, “Intel Trusted Execution Technology: Software Development Guide,” Tech. Rep., [Online]. Available: Available:http://download.intel.com/technology/security/downloads/315168.pdf.

[22] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, “Intel Software Guard Extensions Support for Dynamic Memory Management Inside an Enclave,” in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, 2016, pp. 101–109.

[23] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM Side Channels and Their Use to Extract Private Keys,” in *Proceedings of 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 305–316.

[24] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing,” in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 557–574.

[25] S. Halevi and V. Shoup, “Faster Homomorphic Linear Transformations in HElib,” in *CRYPTO (1)*, vol. 10991. Springer, 2018, pp. 93–120.

[26] X. Liu, R. Deng, K.-K. Choo, Y. Yang, and H. Pang, “Privacy-preserving outsourced calculation toolkit in the cloud,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, March 2018.

[27] Amazon Web Services, [Online]. Available: https://aws.amazon.com/.

[28] Microsoft Azure Cloud Computing Platform and Services, [Online]. Available: https://azure.microsoft.com/en-us/.

[29] Google Cloud Platform, [Online]. Available: https://cloud.google.com/.

[30] IBM Cloud, [Online]. Available: https://www.ibm.com/cloud/.

[31] A. Liu and G. Qu, “H-verifier: Verifying confidential system state with delegated sandboxes,” in *Science of Cyber Security*, F. Liu, S. Xu, and M. Yung, Eds. Springer International Publishing, 2018, pp. 126–140.

[32] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data Attacks Are Realistic Threats,” in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005, pp. 12–12.

[33] A. Boldyreva, N. Chenette, Y. Lee, and A. O’Neill, “Order-Preserving Symmetric Encryption,” in *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, 2009, pp. 224–241.

[34] S. Bogos, J. Gaspoz, and S. Vaudenay, “Cryptanalysis of a Homomorphic Encryption Scheme,” Cryptology ePrint Archive, Report 2016/775, 2016, [Online]. Available: https://eprint.iacr.org/2016/775.

[35] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 719–732.

[36] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software Grand Exposure: SGX Cache Attacks Are Practical,” in *Proceedings of the 11th USENIX Workshop on Offensive Technologies*, 2017, pp. 11–11.

[37] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, “Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures,” in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, 2016, pp. 38–55.

[38] Y. Jang, S. Lee, and T. Kim, “Breaking kernel address space layout randomization with intel tsx,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 380–392.

[39] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, “Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 837–849.

[40] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, “Constants Count: Practical Improvements to Oblivious RAM,” in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 415–430.

[41] A. Bosu, F. Liu, D. D. Yao, and G. Wang, “Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 71–85.

[42] V. M. Glushkov, “The Abstract Theory of Automata,” *Russian Mathematical Surveys*, vol. 16, pp. 1–53, 1961.

[43] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, “A Fast Automaton-based Method for Detecting Anomalous Program Behaviors,” in *Proceedings 2001 IEEE Symposium on Security and Privacy*, 2001, pp. 144–155.

[44] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines-a survey,” *Proceedings of the IEEE*, vol. 84, pp. 1090–1123, August 1996.

[45] S. Bhatkar, A. Chaturvedi, and R. Sekar, “Dataflow Anomaly Detection,” in *Proceedings of the 2006 IEEE Symposium on*

Security and Privacy, 2006, pp. 48–62.

[46] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003, pp. 62–75.

[47] HElib: the library that implements homomorphic encryption (HE), [Online]. Available: https://github.com/shaih/HElib.

[48] Libguestfs, [Online]. Available: http://libguestfs.org/.

[49] Docker Container, [Online]. Available: https://www.docker.com/.

[50] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, 2009.

[51] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *Advances in Cryptology*, 2013, pp. 75–92.

[52] Z. Brakerski and V. Vaikuntanathan, "Efficient Fully Homomorphic Encryption from (Standard) LWE," in *Proceedings of the IEEE 52nd Annual Symposium on Foundations of Computer Science*, 2011, pp. 97–106.

[53] M. Hirt and K. Sako, "Efficient Receipt-Free Voting Based on Homomorphic Encryption," in *Proceedings of the 2000 International Conference on the Theory and Application of Cryptographic Techniques*, 2000, pp. 539–556.

[54] F. Li, B. Luo, and P. Liu, "Secure Information Aggregation for Smart Grids Using Homomorphic Encryption," in *Proceedings of the 1st IEEE International Conference on Smart Grid Communications*, 2010, pp. 327–332.

[55] Y. Hong, J. Vaidya, H. Lu, P. Karras, and S. Goel, "Collaborative Search Log Sanitization: Toward Differential Privacy and Boosted Utility," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 504–518, 2015.

[56] H. Zhou and G. Wornell, "Efficient Homomorphic Encryption on Integer Vectors and Its Applications," in *Proceedings of the 2014 Information Theory and Applications Workshop*, 2014, pp. 1–9.

[57] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O. Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of Remote Attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, June 2011.

[58] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012, pp. 5–8.

[59] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New Results for Timing-Based Attestation," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 239–253.

[60] A. Seshadri, M. Luk, and A. Perrig, "SAKE: Software Attestation for Key Establishment in Sensor Networks," in *Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems*, 2008, pp. 372–385.

[61] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 552–561.

[62] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware," in *Proceedings of the 54th Annual Design Automation Conference*, 2017, pp. 24:1–24:6.