TProv: Towards a Trusted Provenance-Aware Service Based on Trusted Computing

Wu Luo[1], Anbang Ruan[2], Qingni Shen[1], and Zhonghai Wu[1](✉)

[1] Peking University, Beijing, China
{lwyeluo,wuzh}@pku.edu.cn, qingnishen@ss.pku.edu.cn
[2] Octa Innovations, Beijing, China
ar@8lab.cn

Abstract. With the rapid development of cloud computing, system and data security become concerns due to user losing control of his machines and internal attacks. Provenance is an essential approach to establish data and system trustworthiness for cloud computing services, as it summarizes the history of objects and the actions performed on them. However, the current existing provenance-aware solutions either depend on applications in the user-space or fail to convey a genuine provenance information to a cloud user to do a further analysis. Thus they are vulnerable to a malicious privileged administrator or adversary attacking in an untrusted network. In order to solve these problems, we design TProv to establish a trusted provenance-aware service with the help of Trusted Computing. In addition, we introduce Merkle Hash Tree to reduce the length of Chain of Trust and enable parallel validation for the trustworthiness of provenance information, thus TProv decreases the overhead of the huge size of provenance information and the cost of operating trusted hardware, e.g. Trusted Platform Module. The experimental results reflect TProv's effectiveness and efficiency.

1 Introduction

Cloud Computing has attracted a lot of attention by facilitating customers access to computing services without owning any computing resources. However, system security and data security bring people's attention for the problems as user losing control of his machines in cloud [7] and internal attacks [23]. A number of security enhancement mechanisms have been proposed, e.g. access control, least privilege and intrusion detection. Nevertheless, there is still a key issue not being addressed, i.e. whether the proclaimed service components, including those security-enhancement components, have been genuinely enforced, and whether other unnecessary or even adverse components have not been loaded.

Genuinely recording and reporting the provenance of objects, e.g. files and processes, is an essential approach to establish data and system trustworthiness

© Springer International Publishing AG, part of Springer Nature 2018
H. Jin et al. (Eds.): ICWS 2018, LNCS 10966, pp. 67–83, 2018.
https://doi.org/10.1007/978-3-319-94289-6_5

for cloud computing services [4]. The provenance of an object can be characterized by the processes which have modified this object and recursively the provenance of all objects which were inputs for those processes [27]. Over the past two decades, provenance is adopted into different areas, e.g. network, storage and security enhancement [15,16,18,29]. However, adopting the current existing provenance-aware solutions to the cloud faces several challenges.

Firstly, due to the existence of internal attacks and even privileged adversaries in the cloud, assumption that applications in the user-space are trusted can be easily breached. Nevertheless, most of the existing provenance-aware systems [6,8,9] rely on some applications and services in the user-space. Even kernel-based provenance mechanisms [15,19] fail to work in malicious environments. Bates et al. claim to solve this problem and present a generic provenance-aware architecture called Linux Provenance Modules (LPM) [3]. Unfortunately, LPM also introduces the provenance recorder and storage back-ends in the user-space into its TCB, and anchors its hope to protect LPM components through SELinux. A strict SELinux policy is powerful to prevent these components from privileged adversaries. However, besides the cost to design and manage this strict SELinux policy, how to entrust a remote user that the cloud computing platform has genuinely enforced the proclaimed SELinux policy is also a problem.

Secondly, the user loses control of the services deployed in the cloud [7], and all information are transferred to the user through an untrusted network. Because of the existence of the network adversary [1] who may sniff or control the network and further temper or forge the packages, the integrity and trustworthiness of provenance information should be ensured. However, the vast majority of existing provenance-aware systems fail to support this capability. Lyle et al. [10] prepare to integrate Trusted Computing [14] into provenance-aware system, as the former provides remote attestation to enable a remote verifier to obtain genuine records of a target platform. Unfortunately, the provenance information the traditional Trusted Computing could collect is too coarse-grained to reflect the whole system provenance [3]. In addition, the efficiency of traditional Trusted Computing is limited due to the latency of operating trusted hardware [24].

In order to solve the aforementioned problems, our paper provides a trusted provenance-aware service named TProv based on Trusted Computing. As all applications and services in the user-space are viewed as untrusted, TProv is enhanced to prevent from a privileged adversary. In addition, TProv enables to genuinely transfer provenance information to a remote user through an untrusted network. We achieve these goals by designing an appropriate Chain of Trust (CoT). Furthermore, we leverage Merkle Hash Tree [13] to reduce the length of CoT and enable a parallel validation towards CoT rather than the traditional strict sequential validation. Key contributions in this paper are:

1. presenting a trusted provenance-aware service, which enables a cloud user to collect trusted provenance of a cloud machine in the case of a malicious local privileged adversary or adversary attacking in an untrusted network.
2. ensuring the efficiency of TProv by leveraging Merkle Hash Tree.
3. implementing a prototype of TProv.

The rest of this paper is organized as follows. Section 2 reviews Trusted Computing. Section 3 illustrates a possible scenario and overviews TProv. Sections 4 and 5 present TProv in detail. The evaluation results are given in Sect. 6. Section 7 reviews the related works. Section 8 presents some necessary discussion and our future works, and we conclude this paper in Sect. 9.

2 Trusted Computing

Trusted Computing [14] provides a hardware-based solution to validate the integrity of a remote platform. The core component for this technology is an embedded chip called Trusted Platform Module (TPM) [2]. TPM is viewed as the *Root of Trust (RoT)* and is trusted by default. The *Chain of Trust (CoT)* is built from the *RoT*. The last component in *CoT* measures the upcoming component and determines its trustworthiness if we enforce trusted boot [11]. The entire boot process is extended into *CoT*, including BIOS, GRUB, and finally the operating system kernel. After the kernel is loaded, this chain is extended to the applications layer via Integrity Measurement Architecture (IMA) [22].

IMA ensures that all software components are measured into Platform Configuration Registers (PCRs) inside the TPM, including the executables, kernel modules, dynamic link library and files opened by *root*. It is implemented by the *PCR_Extend* instruction of TPM, which replaces the PCR value with a hash of the result of concatenating its original value and the most recently measurement's hash value. As a PCR can only be extended after a system restart, i.e. preforming *PCR_Extend*, all the values measured into it cannot be reversed. Each TPM has a number of 24 PCRs (from 0 to 23). The measurements of hardware, BIOS and bootloader stages are recorded into PCR0-7, while PCR10 records the events of IMA. Since the extend operation makes computationally impossible to recover the list of stored values backwards from the current content of a PCR, a Measurement Log (ML) is maintained to record the detailed information for the software components, representing the integrity status of the platform (Fig. 1).

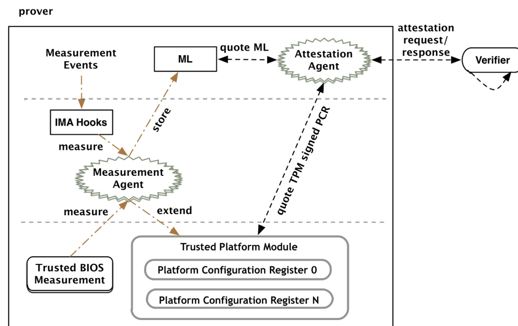


Fig. 1. Integrity Measurement Architecture

IMA has defined *Remote Attestation* to enable *verifier* to validate the integrity of *prover*. When receiving an attestation request, the Attestation Agent in *prover* collects integrity evidence, including PCR values, the signature signed by TPM and the ML. The signature signed by TPM contains the current PCR values. The private key (i.e. Attestation Identity Key, AIK) to generate signature can only be used inside a specific TPM. Hence, a valid signature represents for the identity of *prover* and can be used to determine the trustworthiness of transferred PCR values. The genuineness of ML is further determined by simulating *PCR_Extend* and matching the simulated result with trusted value of PCR10. If the validation result is positive, *verifier* searches his expected values and compares them with the trusted ML, and hence he can determine whether the *prover* is running as his expectation. The expectations of *verifier* are collected from the original source: the software and hardware manufacturers.

3 Scenario and Architecture Overview

3.1 Scenario

Figure 2 shows the threat model. We define *prover* as the provenance-aware platform (e.g. a machine in the cloud) and *verifier* as a cloud user to retrieve the provenance information of *prover* to do some analysis, e.g. identify malicious behavior. Besides the user-space applications and services in the *prover*, the communication channel between *verifier* and *prover* is also untrusted. We allow the adversary to be a local privileged adversary [1] or a remote adversary [1]. Specifically, a local privileged adversary is capable of obtaining *root* privileges and controlling all privileged or unprivileged software on *prover*, e.g. falsifying the provenance response sent to *verifier*. A remote adversary can remotely infect malwares, sniff all packets in the network and interfere with the communication to launch attacks, e.g. Man-in-the-Middle attacks. As many solutions [5,25] have been proposed to ensure kernel's integrity, we assume that the kernel is trusted.

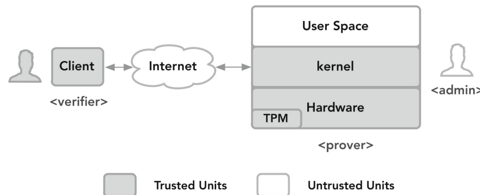


Fig. 2. Threat model of TProv

Based on this threat model, we consider a scenario which uses provenance to detect malicious process in *prover*. Given that an adversary successfully attacks *prover* via a user-space rootkit attack [12], e.g. mafix[1]. This rootkit replaces the

[1] http://forum.eviloctal.com/attachment.php?aid=1341.

good system applications, e.g. *ls, find, ps, netstat and ifconfig*, with trojaned system files to provide backdoor or hide the attackers presence [12]. Some tools (e.g. chkrootkit [17]) construct a basic digest library towards core system applications to prevent this attack. However, a local privileged adversary can modify these tools to hide himself. *Auditing* is a promising mechanism to record behaviors, yet transferring the audit log to a remote *verifier* is also challenging.

3.2 Architecture Overview

The overview of TProv is shown in Fig. 3. We call Measurement Events (MEs) as the events that should be recorded as provenance information, such as running binary programs, reading/writing/replacing a file, *fork*ing a new process, *insmod*ing a kernel module and IPC. LPM gives us a great idea to capture these events by writing hooks following LSM [3]. These hooks in TProv are called TProv hooks. When a ME occurs in *prover*, the corresponding TProv hook is triggered. The Measurement Agent in kernel measures this ME and writes the measurement result into PROV Log. In addition, kernel maintains a merkle hash tree (MHT) with a given tree height. The hash of the measurement generated by Measurement Agent is viewed as a leaf node of MHT. Once the MHT is full, i.e. all leaf nodes store measurement results, the root node's value of MHT is appended into RecordChain and extended into PCR via *PCR_Extend*. Meanwhile, data in RecordChain is exported into Chain Log.

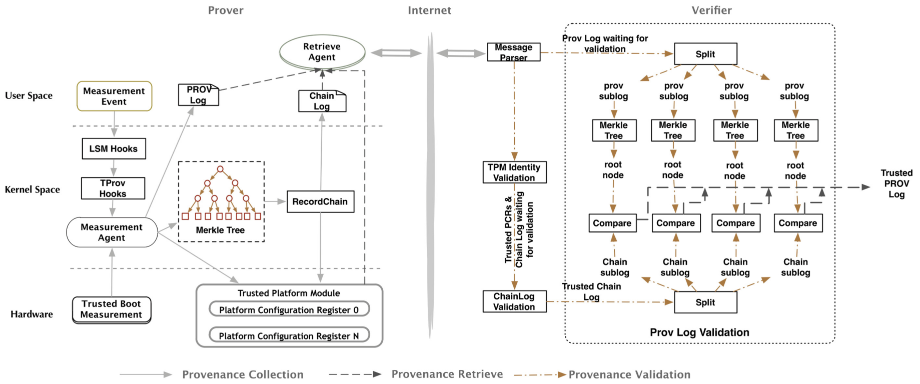


Fig. 3. Overview of TProv service

The whole procedure of handling ME is called as Provenance Collection (see Sect. 4), as the real lines show in Fig. 3. Other lines in Fig. 3 refer to the Provenance Attestation procedure (see Sect. 5), which starts from when a *verifier* sends requests to collect provenance information, and ends with when the *verifier* validates the trustworthiness of provenance received from *prover*.

Note that all components in the user-space of *prover* may be tampered by a local adversary. Meanwhile, even a non-tampered response generated by *prover* may be attacked by a remote adversary. Nevertheless, the Provenance Collection procedure establishes an one-by-one protection, i.e. TPM protects PCR which further protects the ChainLog, and finally the ChainLog protects the PROV Log. Hence any attack based on our threat model can be recognized by *verifier*.

Besides, introducing MHT gives adversary a chance to attack via modifying the memory which stores MHT. However, the runtime integrity of kernel can be protected by [5,25], and hence we do not consider this attack in this paper.

4 Provenance Collection

4.1 Definition of System Provenance

Before introducing TProv, we attempt to define the system provenance. Provenance of the whole system makes up of a directed graph, i.e. *G(V, E)*. The objects we focus on in *prover* consist of the processes and the files operated, i.e. *read/write/rename*, by these processes. All the processes and files compose of the nodes V in G. The edges E in G appear in the following cases:

1. When process B derives from process A, there is an edge from A to B.
2. When file F is operated by process P, there is an edge from P to F. The notation of this edge contains the type of operation, e.g. read/write/rename.
3. The IPC from process P to process Q introduces an edge from P to Q.

```c
int main() {
  int fd;
  fd = fork();
  if(fd == 0) {
    printf("I am child\n");
  } else if(fd > 0) {
    printf("I am parent\n");
  }
  return 0;
}
```

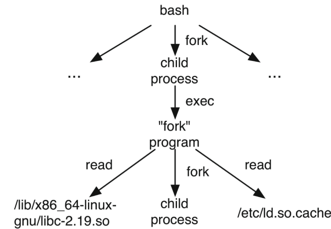


Fig. 4. Possible provenance (the right figure) for the fork program (the left figure)

Figure 4 gives a possible provenance graph for a "fork" program, which invokes syscall *fork* to generate a new child process in a *bash* environment. It indicates that the *bash* process *fork*s a new child process to run "fork" program, which further reads some necessary files, e.g. */etc/ld.so.cache*, and *fork*s a new child process. TProv should enable the *verifier* to restore a trusted provenance graph to do a further analysis, e.g. whether malicious behaviors exist in *prover*.

4.2 Workflow of Provenance Collection

ME Conversion. The inputs of Provenance Collection are the MEs (defined in Sect. 3.2) generated in the user-space of *prover*. MEs fall into kernel mode through syscalls. After this ME passes the DAC and MAC performed by LSM, TProv hooks are invoked and the Measurement Agent receives it. TProv hooks are designed according to LPM [3] to record the whole system's provenance, as the completeness of LSM Hooks has been demonstrated [28]. Measurement Agent measures the coming ME to generate some key-value pairs.

$$MA_{ME} = Measurement\ Agent\,(ME) = \{key : value\}_n \tag{1}$$

MA_{ME} should contain all necessary information that the *verifier* can utilize to do a further analysis. The relationship among MEs is essential to reconstructing the provenance graph. We enforce the Measurement Agent to allocate a unique and auto-increment `PROV_ID` for each process and file nodes when they are created. Besides `PROV_ID`, `PARENT_ID` is also recorded to indicate the node's parent. The `PARENT_ID` is the `PROV_ID` of the *current* process in kernel, afterwards, the relationship among MEs can be established by parsing `PROV_ID` recursively. Key `ACTION` tells us the type of operation the ME does and it is determined by the TProv hook's name. For example, value *PROV_FILE_WRITE* reflects to write to a file. Some special keys are included into MA_{ME}, e.g. key `ARGV` represents an executable program's parameters. The integrity of these key-pairs should also be ensured, so the Measurement Agent concatenates all key-value pairs and views its hash value as the value of key `NODE_HASH`.

An example of the final MA_{ME} is shown below, which means that process whose `PROV_ID` is 5038 executes a binary program (value of key `ACTION`) named *cp*. The value of key `HASH` refers to the hash value of binary file `/bin/cp`.

```
NAME:/bin/cp HASH: 911babc64858747484fa00bbe4c2b50922150ede
ACTION:PROV_BPRM_CHECK_SECURITY ARGV:cp /var/prov_records /root/record.new
PROV_ID:5039 PARENT_ID:5038 NODE_HASH: 3cf54a63c131ac2a66b896c5f14b1e913059bddb
```

Then how can we reliably save these provenance entries? According to IMA [22], Measurement Agent appends the measurement result into kernel data structures and extends the PCR of TPM directly. Each measurement result of ME becomes a component of CoT. It means that kernel has to maintain CoT in the form of data structures. The longer the CoT, the larger the memory of kernel to be consumed and the longer the time for validation. Considering the huge size of MEs, this approach can not satisfy *high efficiency*. In order to reduce the memory consumption of kernel, the Measurement Agent writes MA_{ME} in the form of a *string* into PROV Log in the user-space directly, without recording any information of ME in the kernel. However, since the PROV Log locates in the user-space and is thus untrusted, it is essential to ensure the PROV Log's integrity. As we mentioned before, the value of key `NODE_HASH` (hereafter we call it *node_hash*) indicates the digest value of MA_{ME}, so recording a trusted *node_hash* ensures the integrity of PROV Log. Measurement Agent does not take charge of protecting *node_hash*, yet it just passes *node_hash* into the Merkle Tree.

Merkle Tree. TProv maintains a complete binary merkle hash tree (MHT) with a specified height. We donate MHT_h as a MHT whose height equals h. The root node of MHT_h lies in layer 0, while the leaf nodes lie in layer *h-1*. All nodes are initialized as zero. We call the initialization state as `AllZero`, otherwise the state is `non-AllZero`. We use $Node_i^l$ to represent the i-th node in layer l of MHT_h, where $l \in [0, h-1]$, $i \in [0, 2^l - 1]$ and $l, i \in \mathbb{Z}$. All ancestors of $Node_i^l$ are:

$$AncestorSet(Node_i^l) = \{Node_j^k \mid k \in [0, l-1] \cap j = \lfloor \frac{i}{2^{l-k}} \rfloor\} \tag{2}$$

When receiving *node_hash* from Measurement Agent, kernel locates the first `non-AllZero` leaf node (called WorkNode) of MHT_h and sets its value as *node_hash*.

$$WorkNode := \min_i \{Node_i^{h-1} | Node_i^{h-1} \neq \texttt{AllZero}\} \tag{3}$$

Updating leaf node results in updating MHT. For the sake of explanation, let's start with a necessary definition:

1. $\varphi_L[r(r_1 \xrightarrow{s} r_2); statement]$ means that the object operated is L (may be a Set), and the operating procedure is to traverse r from value r_1 to value r_2 with the step s, each time to complete *statement*.

Based on the above definition, the updating procedure of MHT_h can be expressed as Eq. 4, where *HASH()* indicates to calculate a digest value.

$$\varphi_{Node_j^k \in AncestorSet(WorkNode)}[k(h\text{-}2 \xrightarrow{-1} 0); Node_j^k := HASH(Node_{2j}^{k+1} || Node_{2j+1}^{k+1})] \tag{4}$$

More specifically, it operates the WorkNode's all ancestor nodes and updates them from the higher layer to the root node of MHT_h. For each ancestor node (e.g. $Node_j^k$), kernel concatenates its two children's values and calculates their digest value which will be set as the value of $Node_j^k$.

Once finishing the updating procedure, the Provenance Collection is accomplished if the WorkNode is not the MHT's last leaf node. Otherwise kernel passes the root node's value ($Node_0^0$) to RecordChain, and all leaf nodes of this MHT will be cleared and set as `AllZero`. Equation 5 shows the clear procedure, such that the WorkNode for the next ME is the first leaf node.

$$\varphi_{Node_j^{h-1}}[j(0 \xrightarrow{1} 2^{h-1} - 1); Node_j^{h-1} := \texttt{AllZero}] \tag{5}$$

RecordChain. Besides MHT_h, kernel maintains a double-linked list. When receiving the root node's value ($Node_0^0$) of MHT_h, RecordChain appends $Node_0^0$ into this double-linked list and extends this value into PCR11 via *PCR_Extend*. Kernel exports this list into Chain Log in the user-space. Note that PROV Log is appended by Measurement Agent directly in order to reduce the cost of storing data structures. However, Chain Log is exported into user-space via *securityfs*, which is used to print the data structures in kernel when a process opens this

file. As the *write* operation of this *securityfs* is disabled, modifying Chain Log in the user-space is in fact invalid. In addition, as the entries in Chain Log are $\frac{1}{2^{h-1}}$ times the number of entries in PROV Log, a reasonable h brings benefit to balance between the cost and security. In a word, Chain Log records root nodes of every full MHT and hence can be used to validate the integrity of PROV Log.

Besides protecting PROV Log, Merkle Tree and RecordChain also contribute to achieving *high efficiency*, as them reduce the length of CoT and enable parallel computing when validating PROV Log. Compared with IMA where all measurement events of IMA belong to CoT, the length of CoT in TProv is decreased to $\frac{1}{2^{h-1}}$ times. The longer the CoT, the more cost of operating TPM and validating CoT. In addition, we can leverage parallel validation to validate CoT. The details are shown in Sect. 5.

5 Provenance Attestation

5.1 Message Transferring

A *verifier* requests to collect provenance by sending *RetrieveReq* to *prover*.

$$RetrieveReq = \{nonce, PCR\ index\} \tag{6}$$

where *nonce* is generated at random, and PCR *index* indicates which PCRs are to be retrieved. *nonce* is used to prevent from the attacker using the past to forge the current message. The PCR *index* refers to PCR11 which records the CoT of TProv, i.e. root nodes' values of every full MHTs.

When receiving the *RetrieveReq*, the Retrieve Agent in *prover* collects the *RetrieveRes*, including PROV Log, Chain Log and information stored in TPM, i.e. PCR11, the *AIK Certificate* and the signature for PCR11 and nonce.

5.2 Trustworthiness of Chain Log

When *RetrieveRes* reaches *verifier* through an untrusted network, the Message Parser in *verifier* attempts to resolve each part of *RetrieveRes*. If the resolution fails, the *RetrieveRes* has been tampered. Otherwise, the Message Parser passes the PROV Log that has not yet been verified to the PROV Log Validation module, and passes the others of *RetrieveRes* to the TPM Identity Validation.

TPM Identity Validation module does the following steps. Firstly, it validates the identity of TPM via *AIK Certificate* in *RetrieveRes*. Secondly, it retrieves the AIK public key from *AIK Certificate* and resolves the PCR11 and nonce from the signature with the AIK public key. Finally, it compares the resolved results with PCR11 in *RetrieveRes* and *nonce* in *RetrieveReq* respectively. Note that any failure of these steps means that the *RetrieveRes* has been tampered. If all steps are successful, trusted PCR values are obtained, and the *RetrieveRes* is up to date as the *nonce* matches the one the *verifier* sent.

The trusted PCR11 and the Chain Log waiting for validation are passed into the ChainLog Validation module as inputs. During the Provenance Collection,

entries in Chain Log are extended into PCR11 one-by-one. Through simulating the `PCR_Extend` operation towards Chain Log, a simulated value (*s-PCR*) can be calculated (Eq. 7). *s-PCR* is initialized as `AllZero`.

$$\varphi_{entry_i \in ChainLog}[i(0 \xrightarrow{1} |ChainLog| - 1); s\text{-}PCR := HASH(s\text{-}PCR \parallel entry_i)] \tag{7}$$

Finally, the *verifier* compares *s-PCR* with the trusted PCR11. If the match fails, *RetrieveRes* has been tampered.

5.3 Trustworthiness of PROV Log

When the Chain Log has been validated to be trusted, the PROV Log Validation module prepares to work with the inputs, including the trusted Chain Log and the PROV Log waiting for validation.

SPLIT. Split contributes to making a division towards the input log and outputting a Set of entries. For Chain Log, each element of the outputted Set, i.e. *SC*, refers to an entry in Chain Log, while for PROV Log, each element of the outputted Set, i.e. *SP*, refers to adjacent 2^{h-1} entries.

$$\begin{aligned} SC &= \{entry_0, entry_1, ..., entry_{|Chain\ Log|-1}\} \\ SP &= \{\{entry_{s*i+0}, entry_{s*i+1}, ..., entry_{s*i+2^{h-1}-1}\} \mid \\ &\quad s = 2^{h-1} \cap i \in \{0, 1, ..., \lfloor \frac{|PROV\ Log|}{2^{h-1}} \rfloor - 1\} \end{aligned} \tag{8}$$

Since every 2^{h-1} entries of PROV Log make up of a MHT in kernel, and all root nodes are recorded into Chain Log, the size of *SC* and *SP* should be equal.

Validation Rule 1 (size comparison): *The outputted Sets for splitting Chain Log and PROV Log should satisfy Eq. 9.*

$$|SC| = |SP| \; or \; |Chain\ Log| = \lfloor \frac{|PROV\ Log|}{2^{h-1}} \rfloor \tag{9}$$

MHT. For each element of *SP*, i.e. SP^j, which is a Set with 2^{h-1} entries of PROV Log, we have to reconstruct a MHT with height h. Let SP^j_i as the i-th element of SP^j, and thus SP^j_i also refers to an entry in PROV Log. Define MHT^j_h as the MHT reconstructed by SP^j and $Node^{j,l}_k$ as the k-th node in layer l of MHT^j_h. Note that the *node_hash* for each entry in PROV Log is viewed as a leaf node to construct MHT. Define *node_hash* for SP^j_i as $SP^j_i.node_hash$. Therefore, all $SP^j_i.node_hash$ for SP^j make up the leaf nodes of MHT^j_h.

$$\varphi_{Node^{j,h-1}_i}[i(0 \xrightarrow{1} 2^{h-1} - 1); Node^{j,h-1}_i := SP^j_i.node_hash] \tag{10}$$

Additionally, considering that *node_hash* is calculated by all other key-value pairs for a entry in PROV Log, we can validate the integrity of each entry in PROV Log during the procedure of updating the leaf nodes of MHT.

Validation Rule 2 (entry's integrity comparison): *If all steps in Eq. 11 return True, the integrity of entries in SP^j is satisfied.*

$$\varphi_{SP_i^j}[i(0 \xrightarrow{1} |SP^j|-1); Compare(SP_i^j.node_hash, HashSet(SP_i^j \setminus node_hash))] \quad (11)$$

where $SP_i^j \setminus node_hash$ represents the key-value pairs of SP_i^j except key `NODE_HASH`, and $HashSet\,(S)$ indicates to concatenate all elements of Set S and calculate its digest value. $Compare\,(A, B)$ returns `True` if the value of A equals value B.

We can recalculate the root node's value through updating nodes from layer $h-2$ to the lowest layer, i.e. layer 0. Updating a node means to concatenate its two children nodes' values and set the returned value's digest as this node's new value. Equation 12 shows the updating procedure towards nodes in layer k.

$$\varphi_{Node_i^{j,k}}[i(0 \xrightarrow{1} 2^k-1); Node_i^{j,k} := HASH(Node_{2*i}^{j,k+1}||Node_{2*i+1}^{j,k+1})] \quad (12)$$

Validation Rule 3 (PROV Log's trustworthiness determination): After all MHTs are updated, a trusted PROV Log should satisfy Eq. 13.

$$\varphi_{MHT_h^j}[j(0 \xrightarrow{1} |SP|-1); Compare(Node_0^{j,0}, SC^j)] \quad (13)$$

where $Node_0^{j,0}$ is the root node of the j-th MHT and SC^j refers to the j-th element of SC, i.e. the j-th entry of Chain Log.

Except **Validation Rule 1**, all validation steps can be determined with multi-threading, as they only concern about a specific element of SP and SC during the calculation. Hence the cost of validation can be reduced.

6 Evaluation

The *prover* in our prototype is Ubuntu 14.04 with our modified kernel to support TProv, along with 4 GB memory and 4 processor cores. A TPM1.2 chip is equipped in *prover*. The *verifier* is Ubuntu 14.04 with the default kernel and 2 GB memory.

6.1 Effectiveness: Process Behavior Track

We firstly demonstrate TProv's effectiveness. We utilize the scenario in Sect. 3.1 to show that TProv contributes to tracking process behaviors. Figure 5 describes a part of provenance graph created by analyzing the PROV Log towards `mafix` process, i.e. `mafix/root/#4645`. Nodes in this graph are made up of processes and files to be written. Node `/bin/bash#4441` represents for process `/bin/bash` whose `PROV_ID` equals 4441. The file movement is identified by a character "→", for instance, node in the bottom left corner with label `mafix/bin/hide` → `/usr/lib/libsh/hide#31185` means to replace `/usr/lib/libsh/hide` with file `mafix/bin/hide`.

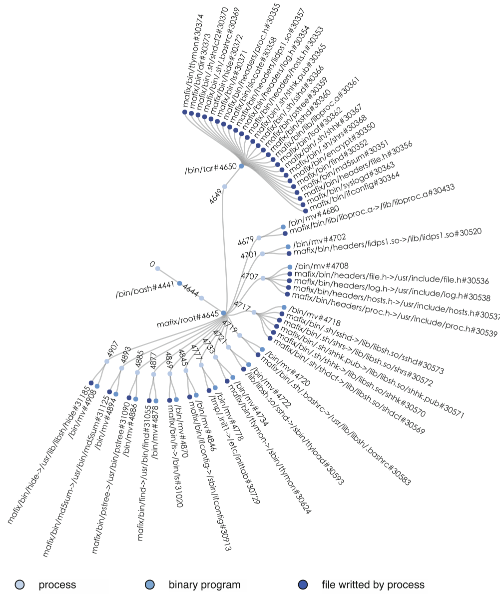


Fig. 5. Provenance information towards *mafix* process.

As Fig. 5 shows, *mafix* process, i.e. node `mafix/root#4645`, firstly runs a binary program `/bin/tar#4650` to extract a lot of files and save them into directory `mafix/bin`. After that, mafix runs *mv* several times to replace the good system binaries with files extracted by itself. For example, process with `PROV_ID` 4877, which is created by *mafix*, runs `/bin/mv#4878` program, and replaces `/usr/bin/find` with `mafix/bin/find` in node with `PROV_ID` 31055. Hence the information that a lot of system binaries have been replaced can be acquired.

6.2 Efficiency: Performance of Provenance Collection

Besides the effectiveness of TProv, we utilize `LMbench3`[2], a well-known tool for performance analysis, to evaluate our modified kernel which implements Provenance Collection procedure. The evaluation is divided into three cases: (1) `Origin`: the default kernel 3.13.11. (2) `TProv`: kernel which implements the Provenance Collection procedure based on kernel 3.13.11. (3) `TProv-w/o-mht`: compared with `TProv`, kernel in this case does not maintain MHT, yet just extends all measurement results into PCR and writes them into PROV Log directly.

Table 1 shows the evaluation results, where in each case we run `LMbench3` for 10 times and calculate its average overhead. The MHT's height is set as 10. For

[2] http://www.bitmover.com/lmbench/.

Table 1. Performance of provenance collection

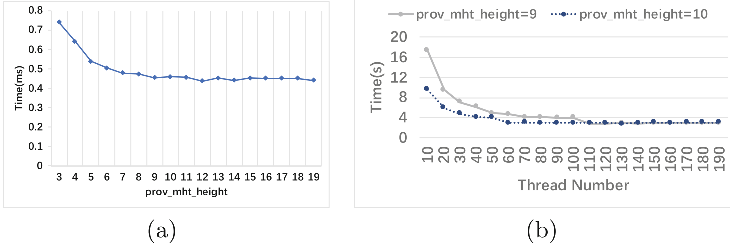
Type	Origin	TProv	TProv-w/o-mht
Processor, Processes - times in microseconds - smaller is better			
null call	0.07	0.07 (0.00%)	0.07 (0.00%)
null I/O	0.16	0.1609 (0.57%)	0.16 (0.00%)
fork process	138	150.3636 (8.96%)	306.7273 (122.27%)
exec process	600.3636	689 (14.76%)	1251 (108.37%)
File & VM system latencies in microseconds - smaller is better			
0 K file create	7.7379	8.1674 (5.55%)	8.1667 (5.54%)
0 K file delete	6.9002	6.9333 (0.48%)	6.9037 (0.05%)
10 K file create	16.1727	16.7273 (3.43%)	16.5938 (2.60%)
10 K file delete	9.8112	9.9845 (1.77%)	9.8428 (0.32%)
**Local* communication bandwidths in MB/s - bigger is better*			
Pipe	2436.1818	2465.6364 (1.21%)	2522 (3.52%)
AF Unix	6989.4545	6957.6364 (−0.46%)	6673.1818 (−4.52%)
Bcopy (libc)	6558.8364	6547.2273 (−0.18%)	6547.25 (−0.18%)
Bcopy (hand)	4193.4182	4191.8727 (−0.04%)	4173.75 (−0.47%)
Mem read	8785.6364	8824.6364 (0.44%)	8817.7 (0.36%)
Mem write	6003.2727	5952 (−0.85%)	5965 (−0.64%)

case TProv and TProv-w/o-mht, Table 1 shows them with a percent overhead calculation against case Origin. For most measurements, the evaluation results of case TProv and TProv-w/o-mht cost a little more than case Origin. However, there are four measurements whose additional overheads are noteworthy: *fork process, exec process, 0 K file create and 10 K file create*. It is reasonable as more than two hooks are triggered in these cases. Additionally, compared with case `TProv`, `case TProv-w/o-mht` usually costs more. The reason is that kernel in this case does not maintain a MHT, instead each valid ME causes a TPM operation.

6.3 Efficiency: Influence of MHT's Height

The MHT's height is the only parameter for TProv. In order to enable the user of TProv to adjust the MHT's height to fit for his own requirement, we add a new parameter with the key *prov_mht_height* into the configuration or command line of Grub, e.g. */boot/grub/grub.cfg* for Ubuntu.

Figure 6(a) shows the average time usage for adding 600000 entries. When a lower *prov_mht_height* is set, the average time for collecting entries is higher. The result is acceptable due to reducing the frequency of performing TPM operations. However, due to the increased cost of updating MHT when setting a larger *prov_mht_height*, the average time for collecting entries tends to be stable.



(a) (b)

Fig. 6. (a) Average time usage for adding 600 thousands entries for different *prov_mht_height*; (b) Time usage for validation with different *prov_mht_height*.

6.4 Efficiency: Provenance Validation with Multi-threading

We claim that TProv enables parallel validation to decrease the time usage. We evaluate it with two variables: the MHT's height and the thread numbers used to validate PROV Log. For each experiment, we record the time usage to validate the integrity of PROV Log after finishing `split`. Before running this experiment, the number of PROV Log's and Chain Log's entries are 27821 and 55 respectively when the *prov_mht_height* equals 10, and they reach to 27143 and 106 respectively when the *prov_mht_height* equals 9.

Figure 6(b) shows our results. It indicates two points. Firstly, when increasing the thread number, the time for `validate` is reduced as expected until the thread number reaching to the number of Chain Log's entries, for instance, when *prov_mht_height* is set as 10, time usage tends to be stable when thread number reaches 55. Secondly, a higher *prov_mht_height* requires less thread number to stabilize the time usage, i.e. 106 threads when *prov_mht_height* equals 9, but 55 threads when it is set as 10. Compared with IMA which is limited to do a strict sequential validation, TProv reaches *High efficiency*.

7 Related Work

Trusted Computing. Based on IMA, binary remote attestation enables a *verifier* to determine the current state of *prover*. Due to the limited computation capability of a TPM chip, the latency of a remote attestation is unacceptable. Stumpf et al. [26] propose the Timestamped Hashchain-based Attestation to compensate this deficiency. Policy-Reduced IMA(PRIMA) and Privilege-Based Remote Attestation [20] achieve ML reduction and hence decrease the delay of Remote Attestation. Additionally, Sadeghi and Stüble propose a delegation based attestation in [21]. They highlight the drawbacks of binary remote attestation and demonstrate how property based attestation to be realized.

Provenance-aware Mechanisms. The majority of proposed provenance-aware mechanisms depend on applications in the user-space. SPADE [6] is a daemon in the user-space to record the provenance from interfaces provided by

operating system, e.g. the audit mechanisms. Hasan et al. present SPROV [8] and SPROV2 [9] to utilize a signature-chaining mechanism to provide a platform-independent, efficient, highly-configurable and modular library. However, an adversary in our threat model can easily forge/disable this library to tamper the provenance information. In contrast, PASS [15] is a well-known provenance-aware system to collect provenance in the VFS layer. Hi-Fi [19] is implemented as a LSM module to collect the whole system provenance, although it blocks the installation of other security modules based on LSM. LPM [3] solves some issues of Hi-Fi, but the provenance recorder and storage back-ends in the user-space are assumed as a part of TCB. Lyle et al. [10] attempt to integrate Trusted Computing to provide a trusted provenance system. Nevertheless, the provenance information it could collect are incompleteness, i.e. just focus on code execution and ignores other activities like IPC and network. Activities whose digests are not changed are also being ignored. Additionally, it requires frequent TPM operations and is limited to the strict sequential validation.

8 Discussion and Future Work

Firstly, MHT maintained in the kernel of *prover* may not be full when the Provenance Attestation occurs, i.e. leaf nodes in the state of `AllZero` exist. In this situation, the last $|PROV\ Log|\%2^{h-1}$ entries in PROV Log are ignored in the validation procedure. It can be accepted as the PROV Log is much huger than these *remaining entries* with a reasonable height of MHT. Increasing the Provenance Attestation frequency is also helpful.

Secondly, we adopt the idea of LPM to construct TProv Hooks following by LSM. LSM is demonstrated to achieve completeness [3] such that this solution is effective to collect the whole system provenance. However, the hooks for network message transferring are not implemented. We will figure it out in the future.

Finally, we do not show algorithms to filter the entries of a trusted PROV Log for the purpose of constructing a provenance graph with information only concerned by a certain *verifier*. We will do it in the future.

9 Conclusion

In this work, we introduced TProv to establish trusted provenance-aware service for cloud computing in the case of adversary being capable to be a privileged administrator or attacking in an untrusted network. We firstly presented the Provenance Collection procedure to show that TProv collects provenance which only depends on hardware and kernel, such that the size of TCB is reduced. Provenance Attestation procedure based on Trusted Computing ensures that any attacks from untrusted user-space of *prover* and untrusted network can be detected from the side of *verifier*. In addition, considering the overhead of the huge size of provenance information and the cost of operating TPM, we introduced an approach to leverage Merkle Hash Tree to guarantee efficiency. The experiment result reflects the effectiveness and efficiency of our solution.

Acknowledgment. This work was supported by National Natural Science Foundation of China under Grant No. 61672062.

References

1. Abera, T., Asokan, N., Davi, L., Koushanfar, F., Paverd, A., Sadeghi, A.R., Tsudik, G.: Invited-things, trouble, trust: on building trust in IoT systems. In: Proceedings of the 53rd Annual Design Automation Conference, p. 121. ACM (2016)
2. Alliance, T.C.P.: Main specification. Version 1, pp. 1–284 (2000)
3. Bates, A.M., Tian, D., Butler, K.R., Moyer, T.: Trustworthy whole-system provenance for the Linux kernel. In: Usenix Security, pp. 319–334 (2015)
4. Bertino, E.: Data trustworthiness—approaches and research challenges. In: Garcia-Alfaro, J., Herrera-Joancomartí, J., Lupu, E., Posegga, J., Aldini, A., Martinelli, F., Suri, N. (eds.) DPM/QASA/SETOP -2014. LNCS, vol. 8872, pp. 17–25. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17016-9_2
5. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 292–307. IEEE (2014)
6. Gehani, A., Tariq, D.: SPADE: support for provenance auditing in distributed environments. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012. LNCS, vol. 7662, pp. 101–120. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35170-9_6
7. Gholami, A., Laure, E.: Security and privacy of sensitive data in cloud computing: a survey of recent developments. arXiv preprint arXiv:1601.01498 (2016)
8. Hasan, R., Sion, R., Winslett, M.: The case of the fake picasso: preventing history forgery with secure provenance. In: FAST, vol. 9, pp. 1–14 (2009)
9. Hasan, R., Sion, R., Winslett, M.: SPROV 2.0: a highly-configurable platform-independent library for secure provenance. In: ACM Conference on Computer and Communications Security (CCS) (2009)
10. Lyle, J., Martin, A.P., et al.: Trusted computing and provenance: better together. In: TaPP (2010)
11. Maliszewski, R., Sun, N., Wang, S., Wei, J., Qiaowei, R.: Trusted boot (tboot) (2015)
12. Manap, S.: Rootkit: attacker undercover tools. Pers. Commun. (2001)
13. Merkle, R.C.: Protocols for public key cryptosystems. In: 1980 IEEE Symposium on Security and Privacy, p. 122. IEEE (1980)
14. Mitchell, C.: Trusted Computing, vol. 6. IET, Stevenage (2005)
15. Muniswamy-Reddy, K.K., Holland, D.A., Braun, U., Seltzer, M.I.: Provenance-aware storage systems. In: USENIX Annual Technical Conference, General Track, pp. 43–56 (2006)
16. Muniswamy-Reddy, K.K., Macko, P., Seltzer, M.I.: Provenance for the cloud. In: FAST, pp. 15–28 (2010)
17. Murilo, N., Steding-Jessen, K.: Chkrootkit-locally checks for signs of a rootkit (2014)
18. Nguyen, D., Park, J., Sandhu, R.: Dependency path patterns as the foundation of access control in provenance-aware systems. In: TaPP (2012)
19. Pohly, D.J., McLaughlin, S., McDaniel, P., Butler, K.: Hi-Fi: collecting high-fidelity whole-system provenance. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 259–268. ACM (2012)

20. Rauter, T., Höller, A., Kajtazovic, N., Kreiner, C.: Privilege-based remote attestation: towards integrity assurance for lightweight clients. In: Proceedings of the 1st ACM Workshop on IoT Privacy, Trust, and Security, pp. 3–9. ACM (2015)
21. Sadeghi, A.R., Stüble, C.: Property-based attestation for computing platforms: caring about properties, not mechanisms. In: Proceedings of the 2004 Workshop on New Security Paradigms, pp. 67–77. ACM (2004)
22. Sailer, R., Zhang, X., Jaeger, T., Van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: USENIX Security Symposium, vol. 13, pp. 223–238 (2004)
23. Salem, M.B., Hershkop, S., Stolfo, S.J.: A survey of insider attack detection research. In: Stolfo, S.J., Bellovin, S.M., Keromytis, A.D., Hershkop, S., Smith, S.W., Sinclair, S. (eds.) Insider Attack and Cyber Security. AIS, vol. 39, pp. 69–90. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-77322-3_5
24. Son, J., Koo, S., Choi, J., Choi, S.J., Baek, S., Jeon, G., Park, J.H., Kim, H.: Quantitative analysis of measurement overhead for integrity verification. In: Proceedings of the Symposium on Applied Computing, pp. 1528–1533. ACM (2017)
25. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing kernel security invariants with data flow integrity. In: NDSS (2016)
26. Stumpf, F., Fuchs, A., Katzenbeisser, S., Eckert, C.: Improving the scalability of platform attestation. In: Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, pp. 1–10. ACM (2008)
27. Tariq, D., Baig, B., Gehani, A., Mahmood, S., Tahir, R., Aqil, A., Zaffar, F.: Identifying the provenance of correlated anomalies. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 224–229. ACM (2011)
28. Zhang, X., Edwards, A., Jaeger, T.: Using CQUAL for static analysis of authorization hook placement. In: USENIX Security Symposium, pp. 33–48 (2002)
29. Zhou, W., Fei, Q., Narayan, A., Haeberlen, A., Loo, B.T., Sherr, M.: Secure network provenance. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 295–310. ACM (2011)