JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 35, 323-339 (2019)
DOI: 10.6688/JISE.201903_35(2).0005

# Real-time Auditing of the Runtime Environment for Cloud Computing Platforms

GWAN-HWAN HWANG, KUN-YIH HUANG, BO-SIANG LIAO,
YI-LING YUAN AND HUNG-FU CHEN
*Department of Computer Science and Information Engineering*
*National Taiwan Normal University*
*Taipei, 106 Taiwan*
*E-mail: ghhwang@csie.ntnu.edu.tw*

In this paper we show how to perform efficient auditing of the runtime environment for virtual machines in public cloud systems or standalone computer systems. The goal is to perform real-time integrity checking of executable codes and library files that will be dynamically linked before an application is launched. Auditing these binary files according to their hash values stored in a local machine is deficient because malware or viruses infecting those files can simultaneously alter their corresponding hash values. We propose an effective scheme to perform real-time auditing of such binary files. First, a status code that represents the current status of all executable codes and library files in the runtime environment and comprises only 32 bytes is downloaded from a trusted remote computer. Second, a full binary hash tree is used to perform efficient auditing of files that will be executed and linked by an application according to the downloaded status code. Finally, this application can then be launched safely. We used a real operating system to evaluate the performance of the proposed scheme, and the obtained experimental results demonstrated its feasibility.

***Keywords:*** cloud auditing, runtime-environment auditing, malware, virus, proof-of-violation

## 1. INTRODUCTION

The computing platform on which programs run includes a certain hardware architecture, an operating system, and runtime libraries. The auditing of the runtime environment should include integrity checking of software components in the computing platform. Runtime libraries typically comprise a huge number of files; for example, the libraries of Mac OS X El Capitan 10.11.5 when popular application software is installed comprise 149,487 directories and 717,976 files. The execution of applications always involves dynamically linking to many library files.

Cloud services are becoming a popular distributed technology because they allow users to rent well-specified computing, network, and storage resources without needing to spend massive amounts on integrating, maintaining, or managing the IT infrastructure. There are three main delivery models for cloud services: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service. IaaS is the most basic cloud-service model, in which the service provider offers virtual machines to users. Cloud users deploy their applications by installing operating-system images and their application

Received September 15, 2017; revised November 24, 2017; accepted January 15, 2018.
Communicated by Chang-Shing Lee.

software on the offered virtual machines. PaaS provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching application software.

In IaaS and PaaS, the computing platforms of users are executed in some kind of virtual machines [1, 2]. The images of these virtual machines are maintained and controlled by the service providers, and users have no knowledge about this is implemented. Images of virtual machines may be damaged or destroyed by internal errors or malicious security attacks. Malware or viruses could infect or be installed in the virtual machines without being detected by users because users do not have control of the images of their virtual machines. Some service providers support the live migration of virtual machines, and users cannot determine or are unlikely to be notified that their virtual machines have been migrated to other physical machines. Users of standalone computers can employ certain methods[1] to defend against the infection of malware or viruses because they have full control of their computers. Another situation is that when the images of virtual machines are destroyed, the service provider may restore these virtual machines using a backup of an early version of the images, and then deny that the user's latest version of the image has been lost. This situation can also allow a so-called roll-back attack [3] or replay attack [4]. If users cannot detect this kind of roll-back attack immediately, the presence of an outdated execution environment of their virtual machine can cause many problems. Antivirus tools do not work in this situation because the problem is not due to the virtual machine being infected with viruses. In addition, no antivirus software is able to uncover all computer viruses, especially new ones.

In this paper we present how to perform efficient auditing of the runtime environment of virtual machines in public cloud systems or standalone computers. The goal is to perform real-time *integrity checking* of executable codes of an application and dynamically linked library files[2] in a computing platform when the application is launched. An intuitive solution called integrity checking is used to keep all the hash values of executable codes and library files in the local computing platform so that the integrity of these files can be checked whenever the application is launched [5]. However, this scheme is deficient because if the computing platform is infected by a virus or the image of the virtual machine is controlled by a hacker, malware can be installed in some of the executable codes and the library files and the corresponding hash values of these infected files can be tampered with simultaneously since they are stored in the same computer. In addition, it is obvious that this scheme cannot resist a roll-back attack because recovering the virtual machine using a previous image will restore outdated executable codes and library files with their corresponding hash values. Besides, it would be quite inefficient to store the hash values of all the executable codes and library files in another remote computer because libraries comprise huge numbers of files.

We present a novel scheme to perform efficient real-time auditing of the runtime environment. Although the hash values of library files are stored in the local machine, a

[1] For example, if the computing platform is executing a Web server, the user can simply forbid any unknown software tool from executing, and can use a firewall to restrict the communication channels used by the computer.

[2] For example, the execution of the FaceTime application in Mac OS X El Capitan 10.11.5 requires 64 dynamically linked library files in /System/Library/ and /usr/lib/ to be loaded.

status code stored in a trusted remote computer is used to audit if executable codes and library files have been tampered with or modified. The hash values are stored in the local machine as a full binary hash tree (FBHTree). The status code is the *root hash* of the FBHTree, and it comprises only 32 bytes. Whenever application software is going to be launched, the latest status code is first downloaded. The hash values of the executable codes and library files that are going to be executed and linked are then inspected according to the status code. Finally, these inspected hash values are used to check if the library files have been modified. We employed a real operating system to evaluate the performance of the proposed scheme, and the experimental results demonstrate its feasibility.

This paper is organized as follows: Section 2 introduces the system architecture, two intuitive solutions, and the proposed scheme, Section 3 presents the implementation details and experimental results, related work is described in Section 4, and conclusions are drawn in Section 5.

## 2. SYSTEM ARCHITECTURE

In this section we first present the general system architecture of the proposed scheme. To support real-time auditing, a *hash values archive* (HVA) is installed in the local machine to store all hash values of executable codes and library files (Fig. 1). An *agent* is executed to perform auditing whenever application software is going to be launched. We assume that the loading, linking, and execution of the application software are performed and controlled by the agent, and that the agent cannot be hacked[3]. The status code is a very small[4] and so allows the efficient and secure verification of the HVA contents.

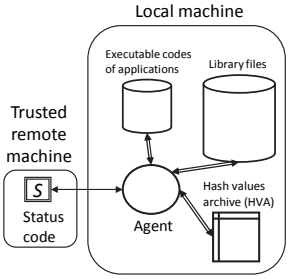


Fig. 1. The system architecture.

Launching application software in the local machine involves the following consecutive steps: (1) the agent downloads a *status code S* from a trusted remote machine; (2) the agent calculates a status code $S'$ from the HVA; (3) the agent compares $S$ and $S'$[5] to verify if the hash values in the HVA are valid; (4) the corresponding executable codes of

[3] Some schemes can prevent the agent from being hacked (*e.g.*, [6]). The executable code of the agent can be stored in another machine. When the user starts operating the local machine, he/she manually downloads the executable code of the agent but then can execute the agent only after entering a password. So even if a hacker controls the image of the virtual machine of the local machine, he/she cannot restart the agent to load, link, and execute the application software. The details are beyond the scope of this paper.

[4] It comprises 32 bytes if Secure Hash Algorithm (SHA)-256 is used as the hash function.

[5] $S$ and $S'$ should be equal.

the application software and library files that need to be dynamically linked are examined according to the hash values in the HVA; and (5) if the verification passes, the agent loads, links, and executes the application software. The key to maximizing the efficiency of the system is verifying the HVA according to the downloaded status code, since the HVA stores a huge number of hash values. Two intuitive solutions for real-time auditing of the HVA are presented in Sections 2.1 and 2.2, and the proposed FBHTree scheme is presented in Section 2.3.

### 2.1 Intuitive Solution One: Storing the Hash Values in a Key – Value Pair

The first intuitive solution is to store the pathname and hash value of the corresponding executable code or library file as a *key-value pair* ($P_i$, $H_i$) in the HVA, where $P_i$ and $H_i$ are the pathname and hash value, respectively, of the $i$th file. If there is a total of $N$ executable codes and library files, the status code is the hash value of the concatenation of the hash values of all key-value pairs in the HVA; that is, $S$ = hash(hash($P_1$|$H_1$)| hash($P_2$|$H_2$)| … |hash($P_N$|$H_N$))[6]. This intuitive solution is very inefficient because all of the hash values of the pathname and file hash pairs have to be concatenated in order to verify if pairs in the HVA are correct. As we have already mentioned, the local machine contains a huge number of executable codes and library files. It is time-consuming to calculate $S'$ according to the HVA, and so we present an improved scheme in section 2.2.

### 2.2 Intuitive Solution Two: Storing the Hash Values in an *m*-ary Hash Tree

The second intuitive solution is to represent the HVA in an *m-ary hash tree* (also called a Merkle tree). An *m*-ary tree is a rooted tree in which each node has no more than *m* children. A hash tree is a tree of hashes in which the leaves are hash values of files and the top of the tree is occupied by the root hash (also called the top hash or master hash) [7]. An example of a file directory is given in Fig. 2 and Fig. 3 illustrate its corresponding hash tree. Each file in a directory is associated with its hash value. We denote $h$(f) as the calculated hash value of file or directory f. The hash value of a directory is the hashing of the concatenation of all the hash values of files and directories in it, such as $h$(d3) = hash($h$(f2) | $h$(d6) | $h$(f3)); note that $h$(f2) = hash(f2). Invocating the hash function may take a while because f2 could be a large file.

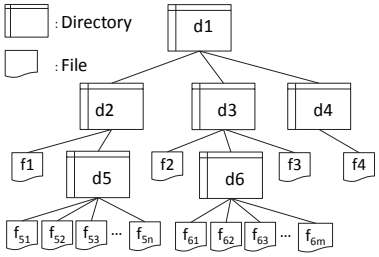


Fig. 2. An example of a file directory.

[6] A cryptographic hash function such as SHA-1, Whirlpool, Tiger, or SHA-256 is used for the hashing, and is denoted as hash (filename). This is actually an invocation of the hash function, and "|"represents concatenation.

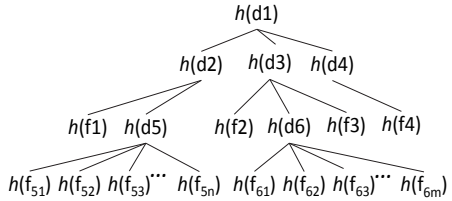


Fig. 3. The hash tree corresponding to the file directory in Fig. 2.

The root hash is the hash value of the root directory; that is, $h(\text{d1}) = \text{hash}(h(\text{d2}) \mid h(\text{d3}) \mid h(\text{d4}))$. For an $m$-ary tree, "$m$" is actually the maximum number of files and directories in a single directory.

The root hash is used as the status code. The advantage of using an $m$-ary hash tree is only part of the hash tree needs to be retrieved to calculate the root hash. A partial hash tree is part of the complete hash tree that omits the nodes under some of the directories [8]. For example, the partial hash tree shown in Fig. 4 (a) omits the nodes under directories d2 and d3. If the valid root hash is known, we only need to perform the following hash calculations to verify if the hash value of f4 is valid in the HVA: (1) retrieve $h(\text{d2})$, $h(\text{d3})$, and $h(\text{f4})$, and then (2) compute $S' = \text{hash}(h(\text{d2}), h(\text{d3}), \text{hash}(h(\text{f4}))$. If downloaded status code $S$ equals $S'$, the hash value of file f4 [*i.e.*, $h(\text{f4})$] must be valid. Compared with the intuitive solution presented in Section 2.1, this method only requires a subset of the hash values in the HVA to be retrieved in order to validate if the hash value of a file is valid according to a status code. However, since directories may contain many files, it could be necessary to retrieve many hash values of files and concatenate them when performing the audit. For example, verifying whether the hash value of file $\text{f}_{52}$ is valid requires the retrieval of $h(\text{f1})$, $h(\text{f}_{51})$, $h(\text{f}_{52})$, …, $h(\text{f}_{5n})$, $h(\text{d3})$, and $h(\text{f4})$, and then performing hash (hash($h(\text{f1})$, hash($h(\text{f}_{51}) \mid h(\text{f}_{52}) \mid \ldots \mid h(\text{f}_{5n})$)), $h(\text{d3})$, $h(\text{d4})$) to obtain the root hash. Retrieving these hash values has a high overhead associated with traversing many nodes in an $m$-ary hash tree, which motivates us to develop a more efficient scheme.

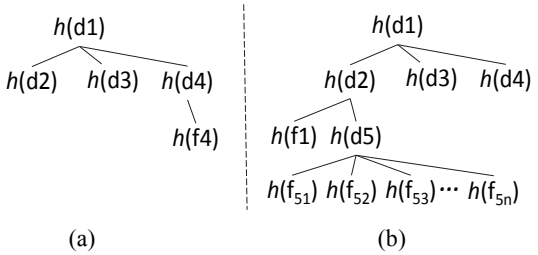


Fig. 4. (a)-(b) Two partial hash trees of the hash tree shown in Fig. 2.

### 2.2 Proposed Scheme: Storing the Hash Values in an FBHTree

We propose storing the hash values in an FBHTree that is not only a full binary tree but also a hash tree. The hash value of a file is stored in one of the leaf nodes in the

FBHTree. It is obvious that the structure of an FBHTree is impossible to match that of a file system in which a single directory always has multiple files and directories. We define an index function Γ to compute the location of the hash value of a file in the FBHTree according to the pathname of the file; for example, if Γ(/d1/d2/f1) = 3, then the hash value of file f1 is stored in the leaf node of the FBHTree with an ID of 3. The core idea of the proposed scheme is that a slice of an FBHTree is sufficient for deriving its root hash. In the following, we first define the FBHTree and its slice, and then show how to derive the root hash from a slice.

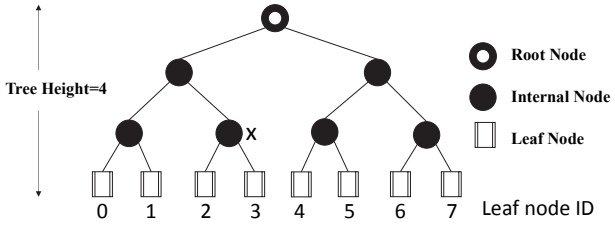


Fig. 5. An FBHTree that is four nodes high.

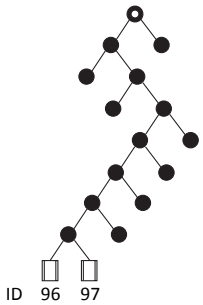


Fig. 6. A slice of an FBHTree that is nine nodes high.

The hash value of a file is stored in one of the leaf nodes in the FBHTree according to index function Γ. Fig. 5 shows an FBHTree that is four nodes high and has eight leaf nodes. Since it must be a full binary tree, it has $2^{N-1}$ leaf nodes and is $N$ nodes high. Index function Γ returns the ID of a leaf node, and we propose the following function:

$$\Gamma(\text{pathname}) = \text{SHA-256}^{7}\ (\text{pathname}) \bmod 2^{N-1}.$$

That is, Γ returns 0 to $2^{N-1}-1$ if the FBHTree is $N$ nodes high. In Section 3 we show that this index function can distribute hash values into leaf nodes of an FBHTree quite evenly. It is obvious that hash values of different files may be stored in the same leaf node due to collision of the Γ function. In such cases multiple pairs of hash values of pathnames and binary values of files are stored in the same leaf node. For convenience, we denote these as *PB pairs*[8]. For example, if Γ(/home/doc/t1/f1.txt) = Γ(/home/public/

[7] The SHA-256 algorithm generates an almost-unique, fixed-size 256-bit (32-byte) hash.
[8] A PB pair is 64 bytes in size because the hash value derived from SHA-256 is 32 bytes.

img/f2.jpg) = 5, then the two PB pairs [hash(/home/doc/t1/f1.txt), hash(binaries of f1.txt)] and [hash(/home/public/img/f2.jpg), hash(binaries of f2.jpg)] are stored in the same leaf node with an ID of 5.

The hash value of a leaf node is computed first by concatenating PB pairs stored in it and then applying a hash function. The hash value of an internal node and a root node is the hash of the concatenation of hash values of its two child nodes. Referring to Fig. 5, the hash value of internal node X is the hash of the concatenation of hash values of child nodes with *leaf node IDs* of 2 and 3.

A slice of a leaf node with an ID of $i$ is a binary tree denoted as slice($i$), which contains leaf node $i$, the root node, all the internal nodes between leaf node $i$ and the root node, and child nodes of all of these internal nodes. Fig. 6 shows slice(96) of the FBH-Tree that is nine nodes high. We can derive the root hash of an FBHTree if we have one of its slices. This is because we have the hash values of all the internal nodes from a leaf node to the root node and all the children of these internal nodes in a slice of an FBHTree. Therefore, if we have the correct root hash, $\gamma$, of an FBHTree, we can first verify if the root hash of slice($i$) is $\gamma$ If slice($i$) passes this verification, then PB pairs stored in leaf node $i$ must be correct.

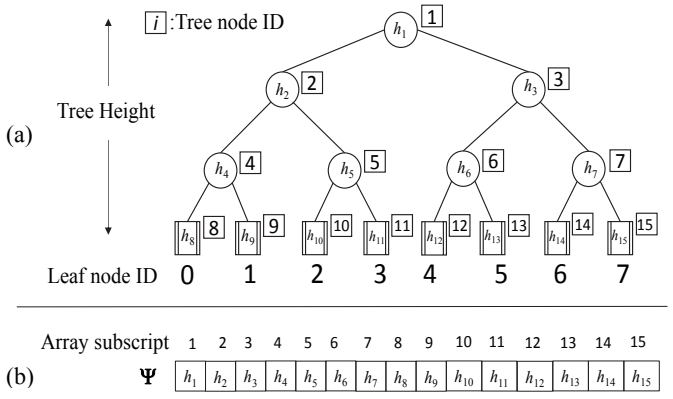


Fig. 7. (a) An FBHTree that is four nodes high; (b) A one-dimensional array Ψ.

We propose storing an FBHTree in a one-dimensional array, with PB pairs of a leaf node stored in a resizable array. For an FBHTree with a height of $N$ nodes, we employ a one-dimensional array with $2^N-1$ elements to store hash values of internal and leaf nodes and $2^N-1$ resizable arrays to store PB pairs of leaf nodes. Fig. 7 shows an FBHTree that is four nodes high and the corresponding one-dimensional array, Ψ, that has $2^4-1 = 15$ elements. Note that each node has a *tree node ID*; these IDs are allocated sequentially according to the search order in a breadth-first search. In addition, each leaf node also has a leaf node ID, $I$, that can simply be translated into the tree node ID, $X$, by $X = I + 2^{N-1}$. The tree node ID is actually the array subscript of the one-dimensional array and corresponds to the return value of index function Γ. In additional, each leaf node is associated with a resizable array to store some PB pairs. Algorithm 1 shows the validity of a hash value of a file is verified according to a valid status code.

**Algorithm 1:** Verify if a hash value of a file is valid. Assume that the height of the FBHTree is $N$ nodes.

**Input:** *FPname*: The pathname of the file to be verified.
Ψ: A one-dimensional array that stores an FBHTree; PB($i$) denotes the PB pairs stored in the leaf node with an ID of $i$.
$S$: The latest downloaded status code.

(1) $I = \Gamma(FPname)$ // Obtain the leaf node ID
(2) $X = I + 2^{N-1}$ // Translate the leaf node ID, $I$, into a tree node ID, $X$
(3) $Y = \Psi[X]$
(4) WHILE ($X \neq 1$) DO // From the bottom of the tree up to the root node
    IF $X$ is an even number THEN
        $Y$ = hash($Y$ | Ψ[$X$+1])
    ELSE
        $Y$ = hash(Ψ[$X$–1]) | $Y$)
    END IF
    $X = \lfloor X / 2 \rfloor$[9] // Unconditional rounding
  END WHILE
(5) IF ($Y \neq S$) THEN Report "Some hash values are not valid."
  ELSE
    Compute the hash value of PB($I$), $\alpha$.
    IF ($\alpha \neq \Psi[X]$) THEN Report "Some hash values are not valid."
    ELSE
      Obtain the PB pair of *FPname* in PB($I$). Assume the hash value of *FPname* is $h$.
      Report that $h$ is valid.
    END IF
  END IF

We employ the FBHTree and the array shown in Fig. 7 to illustrate Algorithm 1. We assume that the Γ function has $I = 3$ in step 1, and then $X = 11$ in step 2 and $Y = \Psi[11]$ in step 3. The first iteration of the WHILE loop produces $Y$ = hash(Ψ[10]|$Y$), and the second and third iterations produce $Y$ = hash(Ψ[4]|$Y$) and $Y$ = hash($Y$|Ψ[3]), respectively. Finally, after step 4, $Y$ = hash(hash(Ψ[4] | hash(Ψ[10]|Ψ[11])) | Ψ[3]). In step 5 we have to check if the hash values of PB(11) that stores the PB pair of *FPname* equal Ψ[11].

## 3. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We conducted a series of experiments to evaluate the performance of the proposed scheme, which was implemented using the Java programming language. The digest function was "java.security.MessageDigest" with the SHA-256 algorithm. The local machine was a Macbook Air 2014 with a 1.4-GHz Intel Core i5 processor and Mac OS X El Capitan 10.11.5 as the operating system. The library files were stored in 149,487 directories with 717,976 files. The target application software installed included Face-Time, LINE, Pages, Mail, Maps, Calendar, Preview, and Photos. This computer setup is representative of a contemporary computer system with popular application software installed.

[9] $\lfloor r \rfloor$ means the floor value of $r$; that is, the largest integer less than or equal to $r$.

Before determining the total auditing time, we first investigated if the proposed index function Γ is good enough to distribute a huge number of library files into the leaf nodes of an FBHTree evenly. For calculating the root hash of the slice of a leaf node that stores some PB pairs, we need to compute the hash values of these PB pairs separately, then concatenate them, derive the hash of the concatenated values, and compute the hash values of all internal nodes along the path to the root node. If many PB pairs are stored in a leaf node, it may take too long to compute the root hash from the slice of a leaf node.

We have previously presented some experimental results obtained in a preliminary investigation [9]. The FBHTree is applied to check for violations in single file operations in cloud storage. We set the height of the FBHTree from 9 to 17 nodes: the numbers of leaf nodes were 256, 1024, 4096, 16384, and 65535 for tree heights of 9, 11, 13, 15, and 17 nodes, respectively. We stored different numbers of PB pairs in an FBHTree. The file stock ratio is calculated as the number of PB pairs divided by the number of leafs. We monitored for the presence of collisions for stock ratios of 33%, 66%, and 100%. In all situations the average and maximum numbers of stored PB pairs in a single nonempty leaf node were less than 2 and 8, respectively. The worst case was for a tree height of 17 nodes and a file stock ratio of 100%, for which the maximum number of PB pairs in nonempty leaf nodes was 8. Further details are available elsewhere [9].

However, these experimental results are not sufficient to demonstrate that an FBHTree is suitable for auditing the runtime environment. Our target machine contained 717,976 library files, and we stored these files in FBHTrees with different heights. The total number of stored PB pairs in an FBHTree was also 717,976. This main data structure is a one-dimensional array, with some resizable arrays used to store PB pairs that collide. The number of resizable arrays equals the number of leaf nodes. Referring to Table 1, an FBHTree with a height 10 nodes consists of a one-dimensional array that stores $2^{10}-1$ hash values (each one is 32 bytes) and 512 resizable arrays.

**Table 1. The collision of index function Γ.**

| Height | No. of leafs | $\alpha$ | $\beta$ |
|---|---|---|---|
| 4 | 8 | 89747.25 | 89908 |
| 6 | 32 | 22436.16 | 22646 |
| 8 | 128 | 5609.27 | 5794 |
| 10 | 512 | 1402.32 | 1502 |
| 12 | 2048 | 350.58 | 421 |
| 14 | 8192 | 87.64 | 127 |
| 16 | 32768 | 21.91 | 43 |
| 18 | 131072 | 5.50 | 18 |
| 20 | 524288 | 1.84 | 10 |

Height: Height of an FBHTree
No. of leafs: Total number of leaf nodes in the FBHTree
$\alpha$: Average number of stored PB pairs in a nonempty leaf node
$\beta$: Maximum number of stored PB pairs in a nonempty leaf node

We implemented the following four different schemes for the system architecture shown in Fig. 1. In addition to the two intuitive solutions and the proposed FBHTree scheme, we also implemented a simplified scheme of the intuitive solution, in which the pathname and hash value of a library file is stored in a key-value pair but without veri-

fying that it is correct based on the status code. We denote this last scheme as the *pure integrity check* scheme. The key-value pair scheme is presented in Section 2.1. Both the pure integrity check scheme and key-value pair schemes employ java.util.HashMap to store the hash value pairs of hashes of the pathname and binaries of a file. The *m*-ary hash tree scheme presented in Section 2.2 is constructed using java.util.ArrayList. The FBHTree is stored in a one-dimensional array and the PB pairs of leaf nodes are stored in java.util.LinkedHashMap.

We compared different aspects of the four schemes. We first compared the build-up time and the memory required to store the HVA in the four schemes. Referring to Table 2, the build-up time includes the time required to calculate the hash values for all of the 717,976 files and to store them in the HVA. The hash values for all of the files were calculated in about 509.2s. The key-value pair scheme is very similar to the pure integrity check scheme; an additional time of 516.1–509.2 = 6.9s was required to store 717,976 pairs in a java.util.HashMap object and compute the status code. Thus, the memory requirement of the two schemes was identical. The *m*-ary hash tree scheme needed to store the names of 149,487 directories and 717,976 files in elements of the Java java.util.ArrayList object, and so it needed more memory. When the height of the FBHTree increases, the memory required increases because the number of internal and leaf nodes increases.

**Table 2. The build-up time and required memory for the HVA.**

| Scheme | | Build-up time (s) | Memory required (MB) |
|---|---|---|---|
| Pure integrity check | | 516.1 | 57.5 |
| Key-value pair | | 522.6 | 57.5 |
| *m*-ary hash tree | | 566.5 | 87.2 |
| FBHTree | Height=4 | 518.0 | 57.5 |
| | Height=6 | 518.3 | 57.5 |
| | Height=8 | 518.8 | 57.6 |
| | Height=10 | 518.6 | 57.7 |
| | Height=12 | 518.3 | 58.1 |
| | Height=14 | 518.8 | 59.9 |
| | Height=16 | 518.9 | 67.1 |
| | Height=18 | 520.8 | 133.5 |
| | Height=20 | 520.1 | 206.9 |

In the second part of our experiment we measured the running times required to verify the hash value of a file according to a status code and to calculate the new status code when a hash value of a file is updated. Referring to Table 3, the number of files in the HVA was 717,976. The pure integrity check scheme does not need to verify the hash value of the file according to the status code. In contrast, the key-value pair scheme needed to concatenate 717,976 hash values in order to verify the hash value of a file. The average running times required for verifying and deriving a new status code were 17.2 and 18.11 ms, respectively. The *m*-ary hash tree scheme only needs to calculate a small portion of hash values of a node, so it takes less time to verify the hash value according to the status code compared with the key-value pair scheme. For an FBHTree with a height of at least 12 nodes, the average running times for the two operations were always

less than 1 and 2 ms, respectively. Although the FBHTree scheme for trees with heights greater than 10 nodes overwhelms the other three schemes, Table 3 only demonstrates that the FBHTree scheme is good at auditing a single file. It remains necessary to further investigate the running time when auditing all the related executable codes and library files when an application is launched.

**Table 3. The running time of operations in FBHTree.**

| Scheme | | $\gamma$ | $\delta$ |
|---|---|---|---|
| Pure integrity check | | Nil | Nil |
| Key-value pair | | 17.20 | 19.11 |
| *m*-ary hash tree | | 6.83 | 18.51 |
| FBHTree | Height=4 | 27.57 | 34.81 |
| | Height=6 | 7.10 | 9.13 |
| | Height=8 | 2.26 | 9.13 |
| | Height=10 | 0.47 | 2.18 |
| | Height=12 | 0.22 | 1.90 |
| | Height=14 | 0.09 | 1.92 |
| | Height=16 | 0.10 | 1.64 |
| | Height=18 | 0.07 | 1.71 |
| | Height=20 | 0.08 | 1.57 |

Height: Height of an FBHTree

$\gamma$: Average running time for verifying the hash value of a file according to status code (in ms)

$\delta$: Average running time for calculating the new status code when a hash value of a file is updated (in ms)

Table 4 lists the required running time for auditing executable codes and used library files[10] of dynamic linking for some applications. We first show the running time for calculating the hash values of linked library files. For example, the FaceTime application will link 64 files in /System/Library and /usr/lib. We computed the hash values of these 64 library files and compared them with the corresponding hash values in the HVA. The overhead of auditing the runtime environment is the additional time required to verify the hash values in the HVA according to the downloaded status code. Since the status code comprises only 32 bytes, it was not necessary to add the network transfer time for downloading the status code.

We present the running time and overhead of performing the auditing of several applications. The overhead is calculated by dividing the difference between the running time for auditing using library files according to the status code and the running time of the pure integrity check scheme by the running time of the pure integrity check scheme. For example, for the FaceTime application, the pure integrity check scheme required 2114.32 ms and running time for the *m*-ary hash tree scheme was 2631.32 ms, and hence the overhead was $(2631.32 - 2214.32)/2214.32 = 18.83\%$. The experimental results show that the key-value pair scheme is very inefficient, with an overhead ranging from 173.68% to 835.49%. The *m*-ary hash tree scheme improves the key-value pair scheme, but the maximum overhead of 138.69% is still high. For an FBHTree with a height greater than 10 nodes, the overhead was always under 1%, which demonstrates that the FBHTree scheme is both feasible and efficient. Although collision occurred for the FBH-

[10] The used dynamically linked library files can be obtained by executing the "otool" command in Mac OS.

Tree that was 12 nodes high is serious, the overhead is still acceptable because the pure integrity check needs to hash all binaries of the dynamically linked library files. The required computation time for the FBHTree with a high number of collisions in leaf nodes is still small compared with the computational time required to hash the library files.

**Table 4. Running time for auditing used library files for selected applications (in ms).**

| Scheme | | Facetime | Line | Pages | Mail | Maps | Calendars | Preview | Photos |
|---|---|---|---|---|---|---|---|---|---|
| No. of linked library files | | 64 | 26 | 36 | 46 | 37 | 44 | 30 | 68 |
| Pure integrity check | | 2214.32 | 1241.71 | 3283.03 | 3059.24 | 767.9 | 1069.32 | 703.11 | 2481.5 |
| Key-value pair | | 13251.24 | 5839.72 | 8985.14 | 8985.14 | 7089.80 | 9146.04 | 6577.50 | 13990.68 |
| (Overhead) | | 498.43% | 370.30% | 173.68% | 193.70% | 823.27% | 755.31% | 835.49% | 463.80% |
| *m*-ary hash tree | | 2631.32 | 1600.37 | 3986.86 | 3986.86 | 1671.14 | 1960.27 | 1623.95 | 5923.20 |
| (Overhead) | | 18.83% | 28.88% | 21.44% | 30.32% | 117.62% | 83.32% | 130.97% | 138.69% |
| sdfd | Height=4 | 3664.80 | 1853.77 | 4031.34 | 4048.33 | 1590.14 | 2081.20 | 1441.42 | 4031.62 |
| | (Overhead) | 65.50% | 49.299% | 22.79% | 32.33% | 107.08% | 94.63% | 105.01% | 62.47% |
| FBHtree | Height=6 | 2649.86 | 1413.86 | 3502.17 | 3356.66 | 1008.40 | 1368.88 | 919.38 | 2922.44 |
| | (Overhead) | 19.67% | 13.86% | 6.67% | 9.73% | 31.32% | 28.02% | 30.76% | 17.77% |
| | Height=8 | 2360.94 | 1298.14 | 3353.99 | 3159.45 | 847.26 | 1166.02 | 778.95 | 2626.08 |
| | (Overhead) | 6.62% | 4.54% | 2.16% | 3.28% | 10.33% | 9.04% | 10.79% | 5.83% |
| | Height=10 | 2247.48 | 1256.75 | 3296.95 | 3079.50 | 784.00 | 1089.58 | 730.28 | 2515.72 |
| | (Overhead) | 1.50% | 1.21% | 0.42% | 0.66% | 2.10% | 1.89% | 3.86% | 1.38% |
| | Height=12 | 2228.88 | 1245.80 | 3287.03 | 3069.25 | 772.44 | 1077.40 | 709.20 | 2490.30 |
| | (Overhead) | 0.66% | 0.33% | 0.12% | 0.33% | 0.59% | 0.76% | 0.87% | 0.35% |
| | Height=14 | 2222.06 | 1242.53 | 3285.48 | 3063.10 | 769.72 | 1073.20 | 709.20 | 2487.20 |
| | (Overhead) | 0.35% | 0.07% | 0.07% | 0.13% | 0.24% | 0.36% | 0.87% | 0.23% |
| | Height=16 | 2217.72 | 1243.25 | 3286.10 | 3062.69 | 770.40 | 1070.26 | 704.86 | 2488.44 |
| | (Overhead) | 0.15% | 0.12% | 0.09% | 0.11% | 0.33% | 0.09% | 0.25% | 0.28% |
| | Height=18 | 2220.82 | 1243.17 | 3285.17 | 3062.69 | 769.04 | 1072.36 | 704.55 | 2485.96 |
| | (Overhead) | 0.29% | 0.12% | 0.07% | 0.11% | 0.15% | 0.28% | 0.22% | 0.18% |
| | Height=20 | 2219.58 | 1242.48 | 3285.11 | 3061.05 | 769.72 | 1071.94 | 705.48 | 2485.96 |
| | (Overhead) | 0.24% | 0.06% | 0.06% | 0.06% | 0.24% | 0.25% | 0.34% | 0.18% |

## 4. RELATED WORK

The Trusted Platform Module (TPM) is an international standard for a secure cryptoprocessor [10, 11]. A TPM-enabled computer can check the integrity of the machine during the boot process, enabling protection and detection mechanisms to function in hardware during the preboot period, in the secure boot process, or even in the application software. Sule *et al.* showed a prototype implementation of a trusted cloud computing deployment on a computer network where both trusted computing integrity measurement/verification are based on the TPM [12]. Berger *et al.* proposed a solution called scalable attestation which combines the benefits of secure boot and trusted boot to address integrity and monitoring issues of the cloud environment based on TPM [13]. However, performing integrity checking only during boot time is insufficient since malware and viruses can infect the runtime environment at any time.

Mishra surveyed methods for virus detection and their limitations [5], and discussed the application and advantage of integrity checking. Integrity checking is called “inocu-

lation" in the commercial Norton AntiVirus product marketed by Symantec Corporation. However, this integrity checking cannot defend against a roll-based attack by the service provider in the IaaS and PaaS service models because the malware and viruses can alter corresponding hash values stored in the local machine when they infect binary files. Related work includes advanced intrusion detection environment (AIDE) which is a file and directory integrity checker [14]. Yue *et al.* proposed an approach to protect virtual machine image integrity in the cloud [15]. In order to reduce the starting time, a periodic scanning program needs to verify the entire image. Wang *et al.* proposed a dynamic integrity validation framework which introduces a trusted third party (TTP) to collect the integrity information and detect remotely the integrity violations on virtual machines periodically. The attacker may choose the attack time between two consecutive detection cycles to escape the verification [16]. Kaczmarek and Wrobel proposed a system that enables file integrity verification implemented as an in-kernel Linux security modules [17]. Database of initial cryptographic hashes are stored in kernel space. The kernel can perform integrity verification when a system call is invoked. However, this scheme cannot apply in cloud computing environment as the image of kernel is controlled by the cloud service provider.

Intrusion detection systems (IDS) for software applications have drawn much attention in recent years [18-23]. An IDS is a mechanism used to monitor system and network situations, collect useful data such as suspicious activities and environmental context, and analyze such data to detect malicious intentions. The challenge of intrusion detection is to build effective predictive models with low error rates by utilizing and integrating various data resources. Some works focus the intrusion detection of virtual machines. Garfinkel and Rosenblum proposed an approach called virtual machine introspection [24], in which the activity of the host is analyzed by directly observing hardware states and inferring software states based on a priori knowledge of its structure by a virtual machine monitor. Ibrahim *et al.* presented the CloudSec monitoring appliance that provides active, transparent, and real-time security monitoring for virtual machines hosted in an IaaS cloud platform [25]. CloudSec utilizes virtual machine introspection techniques to provide fine-grained inspection of the physical memory of virtual machines, without requiring any security code to be installed in the virtual machines. Hizver and Chiueh developed a real-time kernel data structure monitoring system that leverages the OS analysis capabilities of volatility, an open source computer forensics framework, to simplify and automate analysis of virtual machine execution states [26]. Garfinkel *et al.* presented an architecture for trusted computing, called Terra, that uses a trusted virtual machine monitor (TVMM) to partition a tamper-resistant hardware platform into multiple isolated virtual machines [27]. The hardware and TVMM can act as a trusted party to allow closed-box virtual machines to cryptographically identify the software they run. However, since the virtual machine monitor is maintained and controlled by the service provider, these schemes do not allow the user to audit an untrusted platform in the cloud.

Wei *et al.* proposed a management system for images of virtual machines that controls access to images, tracks their provenance, and provides users and administrators with efficient image filters and scanners for detecting and repairing security violations [28]. However, in that system it is necessary to periodically perform virus scanning of the images of virtual machines for detecting and fixing vulnerabilities. Haeberlen *et al.* proposed using an accountable virtual machine monitor [29] that can detect a faulty vir-

tual machine by observing the log file and messages sent and received in the virtual machine. Santos and Lopes proposed an approach to build dependable virtual machines. It is based on trusted computing and model checking: trusted computing allows for low-level attestation of the software of a virtual appliance, and model checking provides for the automatic verification of the software's high-level configuration properties [30]. Win *et al.* proposed a virtualization security solution which aims to provide comprehensive protection of the virtualization environment. When a virtual machine is first created, the security monitor calculates the hash value of the memory contents and stores it. It periodically calculates the hash value of the guest virtual machine's process table and compares it with the stored value. The hash values are then passed to the control monitor, which compares it with previously obtained values. The results from these analyses will indicate the presence of a malware attack [31].

Viswanathan and Mishra proposed a software system that utilizes operating systems' kernel feature for file system monitoring to detect changes for authenticating modifications to dynamic and active website contents. SHA-1 hash for every Web element inside the Web server is stored in NV-RAM memory of the trusted platform module using TPM owner authorization value. Since NV-RAM is limited, it cannot support runtime environment for a real computer platform [32].

Our previous study of real-time proof-of-violation auditing for cloud storage systems employed the FBHTree to audit single file access with the results showing that the FBHTree can be used to simultaneously and continuously audit file operations and collect cryptographic proofs [9]. This scheme needs less than twice the time to finish a file operation in all the situations compared with normal file operations without real-time POV and outperforms previous work [8] with an improvement in performance by one to two orders of magnitude.

## 4. CONCLUSION AND FUTURE WORK

Performing only periodic auditing of the runtime environment is deficient because malware and viruses may be installed in or infected executable codes and library files at any time. In the cloud environment, users have no control of the images of their own virtual machines, which makes this problem even more serious. Real-time auditing of the runtime environment is a solution for obtaining high security. The need to perform auditing before every application is launched makes efficiency a crucial factor to consider.

In this paper we have proposed employing the FBHTree to perform real-time auditing. Hash values of executable codes and library files are distributed to the leaf nodes of an FBHTree by hash values and modular operations, and then efficient searching and auditing of executable codes and library files are performed. The performance of the proposed scheme has been evaluated on the Mac OS X platform, with the experimental results demonstrating that the overhead introduced by the auditing process is always less than 1% if the FBHTree is at least 12 nodes high.

## REFERENCES

1. J. Smith and R. Nair, "The architecture of virtual machines," *IEEE Computer*, Vol. 38,

2015, pp. 32-38,.
2. "VirtualBox," https://www.virtualbox.org/.
3. J. Feng, Y. Chen, D. Summerville, W. S. Ku, and Z. Su, "Enhancing cloud storage security against roll-back attacks with a new fair multi-party non-repudiation protocol," in *Proceedings of IEEE Consumer Communications and Networking Conference*, 2011, pp. 521-522.
4. A. Shraer, I. Keidar, C. Cachin, Y. Michalevsky, A. Cidon, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *Proceedings of ACM Workshop on Cloud Computing Security*, 2010, pp. 19-29.
5. U. Mishra, "Methods of virus detection and their limitations," *SSRN eJournal*, 2010, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1916708.
6. A. M. Dunn, O. S. Hofmann, B. Waters, and E. Witchel, "Cloaking malware with the trusted platform module," in *Proceedings of the 20th USENIX Conference on Security*, 2011, p. 26.
7. R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, 1987, pp. 369-378.
8. G.-H. Hwang, W.-S. Huang, and J.-Z. Peng, "Real-time proof of violation for cloud storage," in *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science*, 2014, pp. 394-399.
9. G.-H. Hwang and H.-F. Chen, "Efficient real-time auditing and proof of violation for cloud storage systems," in *Proceedings of the 9th IEEE International Conference on Cloud Computing*, 2016, pp. 132-139.
10. Trusted Computing Group, "TPM main specification," https://www.trustedcomputing group.org/tpm-main-specification/.
11. F. Yu, H. Zhang, B. Zhao, J. Wang, L. Zhang, F. Yan, and Z. Chen, "A formal analysis of Trusted Platform Module 2.0 hash-based message authentication code authorization under digital rights management scenario," *Security and Communication Networks*, Vol. 9, 2016, pp. 2802-2815.
12. M.-J. Sule, M. Li, G. A. Taylor, and S. Furber, "Deploying trusted cloud computing for data intensive power system applications," in *Proceedings of the 50th International Universities Power Engineering Conference*, 2015, pp. 1-5.
13. S. Berger, K. Goldman, D. Pendarakis, D. Safford, E. Valdez, and M. Zohar, "Scalable attestation: A step toward secure and trusted clouds," in *Proceedings of IEEE International Conference on Cloud Engineering*, 2015, pp. 185-194.
14. Advanced Intrusion Detection Environment (AIDE), http://aide.sourceforge.net/.
15. X. Yue, L. Xiao, W. Zhan, Z. Xu, L. Ruan, and R. Liu, "An optimized approach to protect virtual machine image integrity in cloud computing," in *Proceedings of the 7th International Conference on Cloud Computing and Big Data*, 2016, pp. 75-80.
16. C. Wang, C. Liu, B. Liu, and Y. Dong, "DIV: Dynamic integrity validation framework for detecting compromises on virtual machine based cloud services in real time," *China Communications*, Vol. 11, 2014, pp. 15-27.
17. J. Kaczmarek and M. R. Wrobel, "Operating system security by integrity checking and recovery using write-protected storage," *IET Information Security*, Vol. 8, 2014, pp. 122-131.
18. H. Altwaijry and S. Algarny, "Bayesian based intrusion detection system," *Journal*

*of King Saud University – Computer and Information Sciences*, Vol. 24, 2012, pp. 1-6.
19. Y. Y. Wee, W. P. Cheah, S. C. Tan, and K. Wee, "Causal discovery and reasoning for intrusion detection using bayesian network," *International Journal of Machine Learning and Computing*, Vol. 1, 2011, pp. 185-192.
20. L. Xiao, Y. Chen, and C. K. Chang, "Bayesian model averaging of bayesian network classifiers for intrusion detection," in *Proceedings of IEEE 38th Annual International Computers*, *Software and Applications Conference Workshops*, 2014, pp. 128-133.
21. W. Hu, J. Gao, Y. Wang, O. Wu, and S. Maybank, "Online adaboost-based parameterized methods for dynamic distributed network intrusion detection," *IEEE Transactions on Cybernetics*, Vol. 44, 2014, pp. 66-82.
22. S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Proceedings of IEEE 40th Annual Computer Software and Applications Conference*, 2016, pp. 577-582.
23. F. Mira, W. Huang, and A. Brown, "Novel malware detection methods by using LCS and LCSS," in *Proceedings of the 22nd International Conference on Automation and Computing*, 2016, pp. 1-6.
24. T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of Internet Society Symposium on Network and Distributed System Security*, 2003, pp. 1-16.
25. A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Al, "CloudSec: A security monitoring appliance for virtual machines in the IaaS cloud model," in *Proceedings of IEEE 5th International Conference on Network and System Security*, 2011, pp. 113-120.
26. J. Hizver and T.-C. Chiueh, "Real-time deep virtual machine introspection and its applications," *ACM SIGPLAN Notices*, Vol. 49, 2014 , pp. 3-14.
27. G. Tal, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing." *ACM SIGOPS Operating Systems Review*, Vol. 37, 2003, pp. 193-206.
28. J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing security of virtual machine images in a cloud environment," in *Proceedings of ACM Workshop on Cloud Computing Security*, 2009, pp. 91-96.
29. A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 119-134.
30. N. Santos and N. P. Lopes, "Leveraging trusted computing and model checking to build dependable virtual machines," in *Proceedings of the 10th Workshop on Hot Topics in System Dependability*, 2014, pp. 1-6.
    T. Y. Win, H. Tianfield, and Q. Mair, "Virtualization security combining mandatory access control and virtual machine introspection," in *Proceedings of IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 1004-1009.
31. N. Viswanathan and A. Mishra, "Dynamic monitoring of website content and alerting defacement using trusted platform module," in N. Shetty, N. Prasad, N. Nalini, eds., *Emerging Research in Computing*, *Information*, *Communication and Applications*, Springer, Singapore, 2016, pp. 117-126.

**Gwan-Hwan Hwang (黃冠寰)** is a Professor in Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. He received the B.S. and M.S. degrees while in the Department of Computer Science and Information Engineering at National Chiao Tung University, in 1991 and 1993, respectively, and the Ph.D. degree while in the Department of Computer Science at National Tsing Hua University, Hsinchu, Taiwan, in 1998. His research interests include cloud trust, information security, blockchain technology, software engineering.

**Kun-Yih Huang (黃鯤義)** is currently a Ph.D. candidate in Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. His research interests include cloud security, blockchain technology and medical informatics.

**Bo-Siang Liao (廖柏翔)** received his M.S. from Department of Computer Science and Information Engineering at National Taiwan Normal University in 2016. His research interests include cloud computing, cloud trust, and blockchain technology.

**Yi-Ling Yuan (袁儀齡)** received his M.S. from Department of Computer Science and Information Engineering at National Taiwan Normal University in 2016. His research interests include cloud trust, blockchain technology, and software engineering.

**Hung-Fu Chen (陳虹甫)** received his M.S. from both Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan, and Department of Information Technology at Uppsala University, Sweden, in 2016. His research interests include cloud trust and software engineering.