A Remote Attestation Infrastructure for Verifying the Application of Software Updates

Karthick Ramachandran and Hanan Lutfiyya
Department of Computer Science
University of Western Ontario
London, Ontario
{kramach, hanan}@csd.uwo.ca

***Abstract*—Cloud computing offers opportunities for organizations to reduce IT costs by using the computation and storage of a remote provider. Despite the benefits offered by the cloud computing paradigm, organizations are still wary of delegating their computation and storage to a cloud service provider due to trust concerns. The trust issues with the cloud can be addressed by a combinatiog of regulatory frameworks and supporting technologies. However, there appears to be little existing work that includes the application of software updates in trust assessment. Not applying software updates that fix security vulnerabilities can lead to serious problems. For example, the Canada Revenue Agency failed to update OpenSSL on their servers resulting in a temporary shutdown of tax filing. To address this issue we have developed a framework based on remote attestation and integrity measurements to address this problem. We showed that this could be effective in ensuring that software updates are done and our initial results suggests little run-time overhead.**

***Index Terms*—Trust, software configuration management, remote attestation**

I. Introduction

Cloud computing represents a model of computing where clients can rent a pool of configurable resources such as network connectivity, storage and services from a cloud provider [5]. From the client's perspective, there are cost savings for the client since the infrastructure does not have to be bought upfront. Due to these benefits, users and enterprises are looking to delegate the storage of data and service execution on resources owned by cloud providers. However, about 46% of respondents in a Forrester survey indicated that a key concern with respect to cloud adoption is the integrity of the software [12].

One of the security threats associated with the cloud is the exploitation of underlying software components. An adversary can inject malicious code in an executing binary by modifying the memory. A malicious administrator has the ability to replace the bios, boot loader, the operating system, and any of the software packages that have been installed with malware. Trusting a cloud server requires that a client can accurately *verify* software binaries that the server executes [14]. This can be done through measurements and attestation. Measurement refers to the creation of a hash of a file [1] which includes software binaries and virtual machines (VMs).

[1]Hash is the unique short digest of a binary, that is more efficient to communicate than the exact software.

Remote attestation [14] is defined as enabling a remote entity (e.g., third party service or cloud client) to test the integrity of a targeted system based on the measurements of the system. Essentially verification requires that a list of software binaries with their measured hash values (*measurement list*) is sent to a third party or client. The third party performs the comparison against known hash values. Based on remote attestation and measurements, the Trusted Computing Group (TCG) [30] provides specifications for securely reporting and verifying a remote platform (i.e., server hardware and software) to support *trusted computing*. Cloud clients need to be able to trust that cloud providers install the right versions of hypervisors and correctly install patches especially when the patch is being used to address a security problem. The latter does not always happen as seen with Heartbleed. Heartbleed refers to a buffer over-read bug in OpenSSL discovered in 2014 that allowed for more data to be read than allowed. This vulnerability was propagated to all Debian machines through the OpenSSL version 1.0.1e-2+deb7u4 [1]. OpenSSL maintainers fixed this vulnerability and released a new version of OpenSSL. The Debian maintainers of the OpenSSL package released a new version with the updated OpenSSL. The servers executing OpenSSL were expected to update their version of OpenSSL with this new version. However, not all sites updated the OpenSSL package including the Canada Revenue Agency (CRA) which reported a theft of Social Insurance Numbers belonging to 900 taxpayers on April 8, 2014 causing the agency to shut down its website and extend the taxpayer filing deadline.

Surveys (e.g., [33] [15]) show that there is a substantial body of work that addresses various cloud-related threats to data confidentiality, data and computational integrity, availability, accountability, and privacy. The threats include (but are not limited to) cross-VM attacks and malicious system administrators. However, to the best of our knowledge there is little work that specifically focusses on verifying that a needed change to software has been installed i.e., trust assessment does not consider if critical software updates are being done. The contribution of this paper is an architecture based on integrity measurements and remote attestation to address the problem where a software needs to be updated.

The organization of the paper is as follows: Section II presents background and related work. Sections III, IV, V

978-3-901882-89-0 @2017 IFIP

present the architecture, scenario and case study related to Heartbleed detection. Section V describes our evaluation. Section VI and VII focus on discussion of strengths and limitations of our approach and conclusions.

II. Background and Related Work

This section describes some of the background and related work.

A. Trusted Computing

The Trusted Computing Group (TCG) [30] has defined a set of standards that describe how to take measurements and store the results in a separate trusted coprocessor referred to as the Trusted Platform Module (TPM). The Trusted Computing Base (TCB) refers to the software components required by a specific service. Trust in a service requires trust of all the components in the TCB. This is done through an *aggregate hash*. Each software binary, S_i, is responsible for properly measuring and recording the hash measurement, M_{i+1} of binary, S_{i+1}, that S_i loads into memory (i, $i+1$ refers to the sequence in which binaries are loaded in the memory). The aggregated hash value at a given stage n (value representing the hash of all the software binaries loaded till this stage, n) is given by $Ag_n = H(Ag_{n-1} \mid\mid H(S_n))$. Every time a software binary, S_n, is loaded into memory, its hash value, $H(S_n)$ is concatenated with Ag_{n-1}, the aggregated hash of all the software up to S_{n-1}. The new aggregated hash Ag_n is given by the hash of the concatenated value, $Ag_{n-1} \mid\mid H(S_n)$. This forms the *chain of trust* where the verifier trusts each software to have properly measured and record subsequently launched pieces of software.

For a server platform the software components required typically include the operating system. The first program is the BIOS (S_0). This leads to a condition where the BIOS should be trusted. However, a BIOS cannot be trusted as the server administrator, who has physical access to the server, can easily overwrite a BIOS (BIOS flashing) of the compromised server through a BIOS rootkit attack [10]. Therefore, there is a need to verify the BIOS before measuring the software stack that is loaded subsequently. The BIOS is verified by trusted hardware, *root of trust*, Trusted Platform Module (TPM). On the first system boot, the TPM calculates the hash value of the BIOS and records it in its storage before launching the BIOS. The BIOS measures and loads the boot loader and the boot loader measures and loads the operating system and so on, hence forming a chain of trust. Such a boot is referred to as *trusted boot* [14]. The TPM has special registers called platform configuration registers (PCR) that hold the aggregated hash of integrity values of loaded applications.

The aggregated hash property is not commutative and hence a change in the loading order of software causes a change in hash values. For example, measuring S_i and S_{i+1} is not the same as measuring S_{i+1} and S_i. The other hash property is onewayness, where it is cryptographically impossible for an attacker to determine the input message (hash values of software) given a PCR value.

B. Real-time Measurements

The aggregated cryptographic hash allows the client to verify the code identity[2] of the software binary. However, the measured cryptographic hash does not control the behavior of the binary, as it just verifies if the binary is not modified or corrupted before loading it into the memory. When an administrator modifies the part of the loaded program memory with malicious code, the client will not be able to detect it, since the static measurement is made only before the loading of the binary. This can be countered by reporting real-time hash measurements of the binary to the client. The real-time hash measurements are performed by calculating the hash over the memory in which the program is loaded rather than the binary image in the disk [6], [9], [34].

C. Remote Attestation Infrastructures

This section provides examples of remote attestation infrastructures. Specifically this section presents mechanisms of collecting and reporting integrity values and a framework for verifying if measurements emerge from a valid TPM.

1) Integrity Measurement Architecture (IMA): Sailer et al. [25] proposed the Integrity Measurement Architecture (IMA) as an implementation of TCG standards in GNU\Linux. IMA is supported from Linux 2.6 onwards. The IMA is designed to collect the integrity measurements of all the files before each file is loaded into the memory and executed by the operating system and then store integrity measurements in a secure location. Figure 1 presents the architecture of IMA.

The IMA architecture is hosted on the server that needs to be remotely attested. The IMA has the following components: Challenger, Attestation Service, Measurement Agents, Measurement List and TPM. The hash of the BIOS (trusted BIOS measurements), along with the hash values of all the files are collected and stored in the measurement list by the measurement agents. To secure the measurement list against tampering, the aggregated hash associated with the loading of the last software binary is stored in one of the Platform Configuration Registers (PCR).

When a client (challenger) wants to verify the state of the server, it requests the integrity values from the server by invoking the attestation service hosted in the server through the *integrity challenge* (Figure 1). The attestation service retrieves the measurement list along with the aggregated hash from the PCR and sends it back to the challenger. The aggregated hash is cryptographically signed by the TPM and is verifiable by the client. The client validates the integrity values against its local data store of known secure values and then makes a decision on whether or not the server's state is "healthy", i.e., not corrupted by any malicious software.

2) Privacy Certificate Authority (CA): The remote attestation infrastructure as implemented in IMA requires the server (attestant) to send a detailed description of its system state to the client (attester). The attestant also provides the attester

[2]Code identity of a binary allows the user to identify if the binary is modified before loading into the memory.

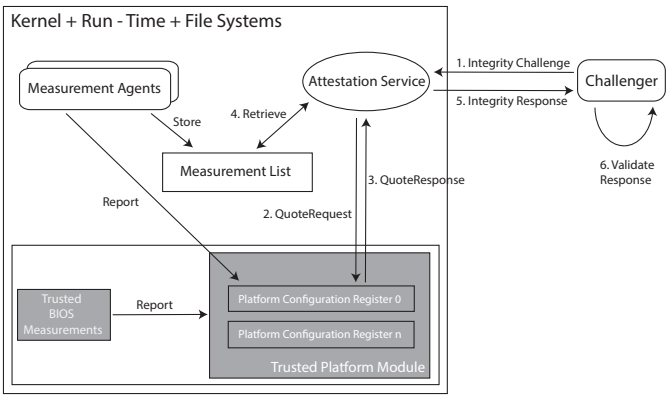

Fig. 1: Integrity Measurement Architecture

with the Attestation Identification Key (AIK)[3] certificate which guarantees that the AIK is owned and secured by a TPM that enforces TCG specified policies. The TCG specifications [30] define the PrivacyCA service to issue these AIK certificates to the attestant.

D. Virtual Machine Based Architectures

Server attestation (e.g., [16], [23], [26], [27], [30]) is used to ensure that the server is free from malicious system files, faulty configuration files and that the boot is secure. Attestation is either done by a third party or the user. Virtual Machine (VM) Attestation (e.g., [6], [7], [13], [19], [20], [22], [28], [31], [32], [35]) focuses on ensuring that a VM can be trusted. There is work (e.g., [7], [8]) that combines both VM and server attestation. The seminal work in VM attestation is Perez et al.'s virtual TPM (vTPM) [22] with its focus on sharing TPM across multiple VMs. The vTPMs multiplex a physical TPM as multiple virtual TPMs. This provides each vTPM with an illusion of trust for enabling remote attestation. A modified implementation of vTPM is adopted by Xen and KVM.

E. Other Work

Recent work on remote attestation has focussed on instrumented things, embedded devices and mobile phones e.g., [21].This work is not related to ours. On the surface, property attestation [24] may seem similar. With property attestation, hash values are converted to property certificates (PC) which are managed by a TTP. There could be a property that for confidence in data transmission a specific version of SSL is used. A different SSL would require a different PC. There is a need to notify the TTP that the old PC is obsolete and a new one is required.Basically a similar architecture to that introduced in the next section is needed.

III. Architecture

The proposed infrastructure is graphically depicted in Figure 2. The arrows in Figure 2 denote the flow of data. For each virtual machine, the client registers the Infrastructure as a Service (IaaS) server hosting the virtual machine with the Trusted Third Party (TTP). The TTP polls the servers registered with it in a predefined interval in order to get the measurement list (software binaries that need their integrity verified).

We note that the TCG model [3] for attestation provides specifications for schemas and security requirements to be used for measurements to be stored. The repository that uses the specification is referred to as the Remote Integrity Measurement Manifest (RIMM) repository. The Resource Integrity Measurement Manifest (RIMM) Software as a Service (SaaS) cloud is the repository of the valid integrity values (hash) of all the software binaries that are allowed to be executed in a server hosted in the Infrastructure as a Service (IaaS) Cloud. The TTP uses the integrity values from the RIMM SaaS Cloud and verifies the values against the integrity values obtained from its RIMM repository. If the verification fails, the TTP notifies the client. The integrity values of RIMM SaaS are updated by authenticated software vendors. The proposed infrastructure assumes the following cardinality of the entities.

- The client and IaaS server have a many to many relationship. There can be many clients transacting with one IaaS server.
- The client and a TTP have a many to one relationship and the TTP and the IaaS server has a one to many relationship.
- The TTP and RIMM SaaS cloud have a many to one relationship. There is only one RIMM SaaS cloud.
- There is one RIMM SaaS cloud to ensure a centralized auditing and verification system of software.

In this section, we detail the functionality of each component and the protocols used in between them for communication.

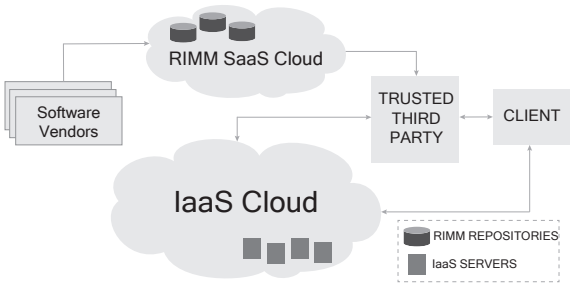

Fig. 2: Architecture

A. IaaS Cloud

The components of an IaaS server are presented in Figure 3. Along with the standard hardware such as processors, hard disk and memory, each IaaS server has a Trusted Platform Module (TPM) that serves as the root of trust. The IaaS server uses the Integrity Measurement Architecture (IMA) (see Section II-C1). The IMA kernel measures all the files before they are mapped (`mmap` Linux system call) into the

[3]Attestation Identification Keys are special keys produced by the TPM for the purpose of securing the integrity values while transmitting to the clients.

memory for execution. The hash values are stored in a secure measurement list in the server's file system. The aggregated hash value (See section II-A) of the measurements are stored securely in a PCR of the TPM. The IaaS Trust Component is responsible for relaying the measurement list along with the TPM PCR values to the trust verifier, hosted by the TTP. Essentially the TTP is a challenger (see section II-C1). Each server has a domain controller and other supporting applications that are used for managing the VMs i.e., the domain controller is responsible for creating and managing the virtual machines hosted by the server.

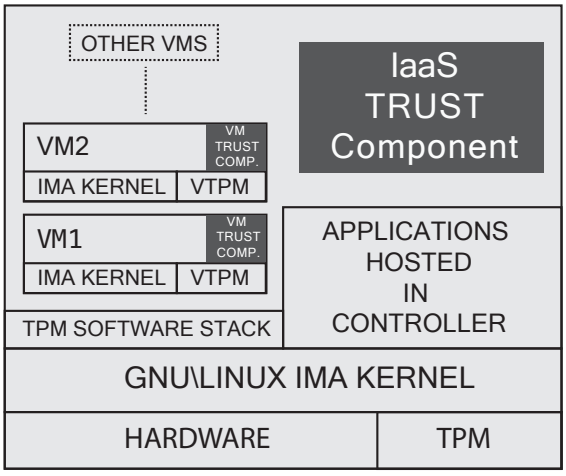

Fig. 3: IaaS Server Components

IaaS Trust Component: The IaaS trust component hosts a service, `request_ml()`, that is used to retrieve the measurement list of the domain controller. This service reads the measurement log created by the IMA kernel of the domain controller and sends it to the TTP along with the aggregated hash from the PCR of TPM signed by the TPM's Attestation Identification Key (AIK). The measurement list is a collection of tuples where each tuple consists of the following information: $\{id, hash\_value\}$, where id is the unique identifier representing the software binary and the $hash\_value$ is the integrity measurement. Attestation Identification Keys [23] are special keys produced by the TPM for the purpose of securing the integrity values while transmitting to the clients.

VM Trust Component: The VMs are allowed to host only IMA enabled kernels. This can be verified through remote attestation. The virtual TPMs [22] act as the root of trust device for the VM-IMA. Each instance of the VM has a VM Trust Component, that relays the measurement list from the VM to the same trust verifier, hosted by the TTP. The services hosted by the VM trust component are used to validate the state of the VM.

B. Trusted Third Party

The trusted third party hosts the trust verifier and the Resource Integrity Measurement Manifest (RIMM) data store. The trust verifier registers the IaaS server's domain controller and the VM, and verifies the state of the controller and the VMs in predefined intervals. The verification of the state is performed by comparing the software measurements from the IaaS server and VM against the reference software integrity values stored in the RIMM data store.

The RIMM data store is updated with valid integrity values of safe software either from the RIMM SaaS cloud or by the client. The domain controller of the IaaS server is verified only using the values provided by the RIMM SaaS cloud. In other words, the IaaS is only allowed to execute software that is approved by the community managed RIMM SaaS cloud. The implication is that if a software vendor has upgraded its software then the hash values provided by the RIMM SaaS cloud and that retrieved from the IaaS server are no longer the same.

The verification of a VM can be done by the TTP using values provided by the RIMM SaaS cloud or by the client. This allows the clients to deploy their own custom trusted applications on the VM.

C. RIMM SaaS Cloud

The RIMM SaaS Cloud (RSC) provides clients with a list of all verified valid integrity values of the software that could be executed safely in the IaaS servers and VMs. The RSC is also responsible for the auditing of the software to ensure they are not vulnerable or malicious. The auditing functionality of the RSC is beyond the scope of this work. In a typical workflow, the SaaS cloud audits the newly updated software binary and checks for any security vulnerabilities. Once the software's audit is successful, the new update is published to all the clients (this assumes the client does the comparison of hash values) and the TTP.

D. Client

The client in this architecture registers the VM and the domain controller with their custom measurement list policy with the TTP. The client also subscribes for notifications from the TTP when the verification fails or there are recommended updates that needs to be performed in the VM, as specified in the VM policy.

E. VM Policy

The VM policy is used by the client to specify advanced software verification configuration and the update requirements of the client. The policy allows the client to denote a list of allowed software (*whitelist*) that can be installed in the VM along with the software's update configuration. Platform Trust Services (PTS) specification [2] proposes a static whitelist management of software. We present a policy with a dynamic whitelist with the following components:

- **Integrity value origin:** The integrity values can be either provided by the RIMM SaaS cloud or by the client (The client may trust the software that they deployed in the VM and do not want to take the RSC route). To accommodate both the integrity values from the client and the RIMM data store, we introduce the $origin$ attribute.

- **Application Chaining:** For some deployments, there is a need to specify the applications that have to be loaded before other applications (e.g., the client may want the database server and the logging server to be loaded before the web server) to avoid any error requests. We propose the use of run levels for achieving application chaining (Section III-E1).
- **Alert levels:** This functionality allows users to define and customize various notification levels and associate policy violations to a particular level, e.g., For some VMs the user may not want aggressive notifications when there are policy violations. They may desire in a daily/weekly digest mail a list of violations. However, for violations in mission critical applications, the user may expect automated phone calls, SMS or high priority emails.

1) Formalisms: The formalism for the policy based on the functionalities described is presented here. This is an extension of the PTS specification [2]. The policy P is expressed as a set of the software configurations (S), notification levels (N) and time interval (T).

$$P = \{S, N, T\} \tag{1}$$

S is a list of individual software configurations, s, where:

$$s = \{id, build\_info, origin, r\_level, n\_level\} \tag{2}$$

The variable id represents the unique identifier of the software binary. The variable $build\_info$ denotes the details of the build (expanded in Equation 3). The variable $origin$ represents if the client is presenting the hash value or the hash value has to be fetched from RIMM repository. The variable $origin$ has valid values: `global` or `client`. The variable r_level is the run level of the software, indicated by integers. This will aid in establishing the application chain in the system. The variable n_level represents the notification level associated with the policy violation. This notification level maps to the variable $level$ of Equation 4.

The variable $build\_info$, either has the $version\_number$ or tag.

$$build\_info = \{version\_number \mid tag\} \tag{3}$$

The variable $version\_number$ specifies the fixed version number (build version, major version and minor version with patch levels) of the software that is allowed to be hosted. The variable tag allows extra flexibility in specifying the current software version. Instead of fixed number, the variable has one of the following values: `stable`, `unstable` or `testing`. The values `stable`, `unstable` and `testing` denotes the latest stable, unstable and the testing builds of the given software respectively. The trust verifier fetches the hash values of the specified tag branch of the software from the RIMM for verification.

The variable $origin$ has one of the following values: `global` or `client` where the variable `global` represents that the hash value is updated by the RIMM SaaS cloud and the variable `client` denotes that the hash value is provided by the client.

The notification levels, N from Equation 1 is represented as

$$N = \{level, options\} \tag{4}$$

where the variable $level$ is the identifier that is used in the software configuration s (Equation 2) and the variable $options$ represents the type of notification (e.g., notification through email, phone etc.).

IV. Scenarios

In this section we present the orchestration of the different units in our architecture, for the registration, verification and update RIMM operations in the system.

A. Registration

The client (user) that needs to use the services of the TTP for verification of an IaaS server, registers the IaaS server with the `register_vm` call to TTP. The algorithm of the `register_vm` service is presented in Algorithm 1. The algorithm takes as input the invoker or $client$, $controller$ and vm that needs to be registered along with the variable vm_policy. The TTP checks if the domain controller (denoted by $controller$) of the IaaS server was registered in the TTP by some other client (Line 1). If the controller is not registered, the TTP invokes the function get_ml (function get_ml calls the corresponding controller's $request\_ml$) to receive the measurement list from the IaaS trust component (Line 2). The function get_ml returns the measurement list to the TTP and the TTP verifies the measurement list against the known software values. If verification is successful then the controller is added to the verification list of the TTP (Line 6). The policy parameters, S, N and T, described in Section III-E1, are extracted from vm_policy (Line 9). The software binary configuration of each software in S is stored in the RIMM repository (Line 11). The control is returned and the TTP waits from initialization of VM from the user at this point (Line 13).

If the verification of the IaaS server succeeds the client initiates preparation of the VM in the IaaS server and deploys the application environment[4] in the VM. After deployment, the client invokes the service `init_vm_ml` of the TTP from the user's VM in the IaaS provider by passing the measurement list of the VM to the TTP. The TTP verifies the measurement list with allowed values specified in the variable `vm_policy` that is passed to the TTP by the client during the invocation of the service `register_vm`. The client is notified if the verification fails.

Although not explicitly stated in the algorithm, the TTP verifies if the client registering the VM owns the corresponding VM.

[4]Typically an application environment will involve the application that the user wants to host along with all the supporting applications

Algorithm 1 Registration of Controller

Input: $client$, $controller$, vm, vm_policy

```
1: if not controller_list.contains(controller) then
2:     controller_ml = get_ml(controller)
3:     if verify_ml(controller, ml) = false then
4:         notify_client(CONTROL_VERIFY_FAIL)
5:     else
6:         controller_list.add(controller)
7:     end if
8: end if
9: Extract policy parameters {S, N, T}
10: for each s in S do
11:     Store s in RIMM repository
12: end for
13: return WAIT_FOR_VM_INITIALIZATION
```

B. Verification

Verification is performed by the TTP in predefined intervals specified by the client.

The TTP, in predefined intervals specified by the client, invokes get_ml service (trust component) of IaaS server to get the measurement list of the server. The TTP verifies the measurement list against the known values of the IaaS server and notifies all the users that have registered with the IaaS controller, if the verification fails.

The verification of the VM is performed by the TTP by invoking the service get_ml of the virtual machine to retrieve the measurement list of the VM. The measurement list is verified against the valid measurement list specified by the client during registration. The user is notified when the verification fails.

Algorithm 2 Verification of ML: $verify\_ml$

Input: $target$, ml

```
1: for each s in ml do
2:     hash_value = s.hash_value
3:     ref_value = NULL
4:     if is_controller(target) then
5:         ref_value = get_global_hash(s.id, s.build_info)
6:     else
7:         origin = get_origin(s.id, target)
8:         if origin is global then
9:             ref_value = get_global_hash(s.id, s.build_info)
10:        else
11:            ref_value = get_global_hash(s.id, s.build_info,
12:                                        target
13:        end if
14:    end if
15:    if hash_value != ref_value then
16:        n = get_notification_options(s.id)
17:        notify_client(VERIFICATION_FAIL, s, n)
18:    end if
19: end for
```

Algorithm 2 presents the logic for the $verify\_ml$ function in TTP. The function takes as input the variable $target$ (controller or VM) for verification and the measurement list, ml from the target. Each software binary represented in the variable ml is verified against known values through a loop counter (Line 1). The variable $hash\_value$ is extracted from the ml and stored in a variable (Line 2). If the variable $target$ is a controller then the reference value for verification is retrieved from the global hash for that software binary and build, that is updated by the RIMM SaaS cloud (Lines 4 to 5). If the variable $target$ is a VM, the variable $origin$ for the software is checked in the RIMM repository of TTP. If the variable $origin$ is `global` the reference value is retrieved from the global hash and if the variable $origin$ is `client` the reference value is retrieved from the values provided by the client during registration (Lines 8 - 11). The variable $hash\_value$ of the software is checked against the variable ref_value and if they are not equal the notification options for that software binary is retrieved by the function $get\_notification\_options$ (Line 15). The client is notified based on the notification options (Line 16).

C. RIMM policy update

The TTP invokes the $subscribe$ service of the RIMM SaaS cloud by passing the software list as a parameter. The software list contains the identifier of all the software binaries that the TTP is interested in getting the update information. The RIMM SaaS cloud adds the client to the subscription list of all the software denoted by the TTP. When a particular software in the software list gets an update the software vendors invokes the $notify\_update$ service of the RIMM SaaS cloud with the new hash value. This value is published to all the clients that have subscribed to get the updates of that software. The TTP then adds that value to its data store. Any future verifications by the TTP will use the new updated version value.

V. Heartbleed Detection

The Heartbleed bug scenario is simulated and the workflow that represents the Heartbleed detection is presented in Figure 4. When a new Debian package is released with updated binaries of OpenSSL, v5, `notify_update` of the RIMM SaaS is invoked by the software vendor with the variable $update\_type$, `critical`. The RIMM SaaS updates its data store with u5's hash values. The service `publish_client` of each TTP that has subscribed to the updates of OpenSSL is invoked by the RIMM SaaS with the variable $update\_type$, `critical`. The TTP updates its RIMM data store with new hash values. These hash values are used for verification in the next verification cycle of the IaaS controller and VM. If the controller or the VM is not running the new version of OpenSSL the client is immediately notified.

VI. Experiments

The TTP, server and client are implemented using the Twisted networking framework [11] in Python. Twisted is a platform for developing event driven networking applications. Twisted enables developers to implement non blocking

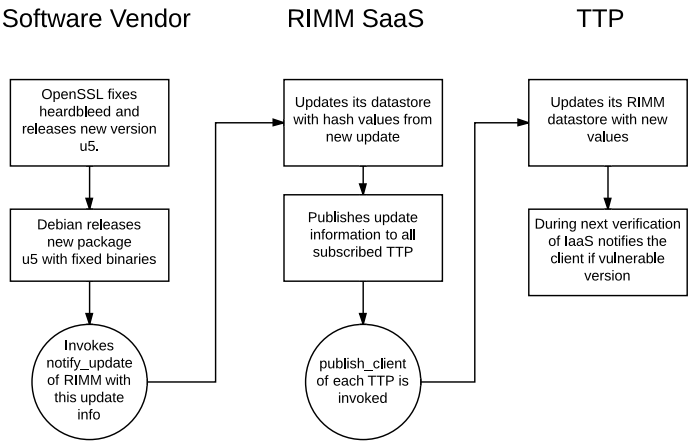

Fig. 4: Heartbleed Detection Workflow

servers using callback programming model. The services in all the servers use serialized Python objects (Pickle) to transfer messages in between them. Synchronized queues are used to regulate data flow during slow connections. The RIMM SaaS cloud and RIMM data store of TTP use redis key/value database for storing the policy information.

A. Experimental Environment

For our experiments we extracted the packages with priority, *required*, of the Debian. There were around 336 required packages in Debian Sid version. Each package has an average of 5 to 10 files. The hash values of each file were extracted and saved to form the RIMM SaaS cloud. We also installed the Xen VM machine and the supporting software for it and added to the measurement list in RIMM SaaS cloud.

Experiments were conducted to analyze the effectiveness of the proposed infrastructure and to study the overhead. The effectiveness of the proposed infrastructure is illustrated by the Heartbleed bug scenario, where the clients are alerted when the server is running a vulnerable version of the OpenSSL software. The overhead of the system is measured against plain vanilla setup without remote attestation infrastructure.

For the experiments, we established a TCB that contained the vulnerable OpenSSL u4. The following servers were setup based on the TCB:

- A RIMM server is deployed with the hash values of the vulnerable OpenSSL software binaries.
- An IaaS server is set up with software based on TCB, hosting the vulnerable OpenSSL binary.
- The TTP server was also deployed and the client was made to register the IaaS with the TTP.

The workflow described in the previous section was deployed in this environment. The system successfully detected that there was a need to update the OpenSSL software.

B. Overhead

The overhead of the infrastructure is analyzed by comparing it with an IaaS server that does not run the remote attestation infrastructure. The booting overhead of the system and overhead during computational work is studied and analyzed.

For the experiments we set up two different infrastructures: Setup A and Setup B. Setup A, the remote attestation infrastructure, has three servers: IaaS server, TTP and the client. Setup B without remote attestation has two entities: IaaS server and client. The IaaS servers of both A and B run the same version of the Debian operating system, Sid with kernel version 3.7.1. The kernel of setup A's IaaS is installed with IMA enabled kernel and the IMA kernel code is instrumented to log the time taken for hashing of each binary file that is loaded into the memory.

1) Booting: The booting process of IaaS server involved loading of 84 files to the memory. Both the IaaS in setup A and setup B are booted around 5 times to measure the average difference in booting. In our experiments setup A, that ran the IMA enabled IaaS took 1772ms longer to boot than the setup B without the remote attestation setup. This is because at the boot time many binary files are loaded into the memory for the first time, hence they need to be hashed before loading. In a complete boot process which takes around 30 seconds, 1772ms is negligible.

2) Computations: To measure the overhead on processing, Unixbench [29] was executed in the IaaS server on both Setup A and B. UnixBench executes a number of computationally intensive operations on the server and returns indicator values, that denote the performance of the system. Higher scores in Unixbench indicate better performance. The experimental output of both the operations are tabulated in Table I. Setup A took 251.266s to complete the operations specified in Table I and Setup B took 252.521s.

VII. Discussion, Conclusions and Future Work

In this work, we presented the technologies behind remote attestation infrastructures, surveyed some of the state of art in remote attestation infrastructures and showed how remote attestation addresses problems related to software updates.

Operation	Setup B	Setup A
Numeric Sort	813.44	836.32
String Sort	523.56	532.32
Bitfield	2.59E+08	2.59E+08
FP Emulation	275.8	275.76
Fourier	21142	18937
Assignment	26.544	26.576
Idea	5167.7	5165.8
Huffman	2340.3	2340.5
Neural Net	46.526	46.483
LU Decomposition	1267.7	1264.7

TABLE I: Iterations per second for Setup A and Setup B

This section describes the strengths and limitations of the work. Future work will focus on addressing the limitations.

Static Load-time Measurements: Our infrastructure is based on static load-time measurements ("measure before load"). Inherently these measurements cannot completely predict the dynamic properties of the software. Once the software is loaded in the memory, there could be variety of reasons why the software could fail or behave insecurely (e.g., buffer overflow). This behavior is not captured by static measurements as the behavior is also dependent on other unpredictable parameters. There is very little research on how to guarantee these dynamic properties of the software. The current scope of our work does not address this issue.

Use of Existing Building Blocks. Our architectural framework uses existing building blocks as described in Section II. There are two advantages: (i) We can address the challenge of dealing with software updates within the context of frameworks that address other security challenges that benefit from attestation. For example, our architecture is based on IMA which provides nonces and thus allows our architecture to deal with attacks based on simple replay attacks; (ii) Improvements in the efficiency of the building blocks would benefit the proposed architectural framework. We added to the existing bulding blocks as needed to reduce security vulnerabilities. For example, there are no required restrictions on polling by trusted third parties. It is possible that if the TTP polls the servers in a predefined interval, then a malicious server can prepare itself in advance knowing when it is going to be attested. We address this issue by allowing clients to choose the intervals through a policy.

Security Vulnerabilities. Verifying the software for security vulnerabilities and malwares is non-trivial. An open source disk encryption software, TrueCrypt was commissioned for auditing in September 2013. It took two security engineers 6 months to complete the audit of a part of the software [17]. This experience show that auditing of all the software by RIMM SaaS cloud is an extremely challenging task. Tools are needed to speed up this task.

The Use of a Trusted Third Party. The advantages of using a TTP include the following: (i) The TTP authenticates any entity that interacts with it including clients. Since clients are not authorized to update entries in the TTP RIMM data store then it would be difficult for a client to maliciously update values; (ii) The TTP simplifies interactions for the client e.g., they do have to deal with software update alerts. On the otherhand, trusted third parties have security vulnerabilities. One approach is to have the software stack of a TTP can be limited to known software as the TTP has a well-defined API. For example, if the TTP exposes its services using XML web-services, the TTP can be restricted to run a minimal GNU\Linux OS with Java Runtime Environment and Tomcat Apache web server. Therefore the state of the TTP can be in turn verified by any external entity. However, to avoid complexity at the TTP level, we assumed that the TTP does not need verification and is trusted by the client and IaaS. Future work will study the performance of having the TTP periodically verified.

TPM Vulnerability: The proposed infrastructure is vulnerable to hardware attacks. There are known attacks on TPM [4], [18] that need to be resolved before TPM becomes truly tamper resistant.

Overhead. We used Unixbench which is a heavily used benchmarking tool used for the Linux operating systems. The benchmark programs are representative of typical workloads. The experiments show little difference between the use of our architecture and not using our architecture with Table I showing negligible results. This provides confidence that our architectxure does not impact performance.

Scalability. The overhead is promising but the number of operations was small. We have concerns about scalability resulting from the need to have what could be massive measurement lists. As noted in Berger et al **??** it may be feasible to maintain a list for a fixed software but in an environment where a diverse and potentially frequently changing applications this may not be feasible. Berger et al **??** have proposed an approach that looks to be a promising approach to this problem. Basically it assumes that only a subset of software components are used to calculate the hash value. Our future work includes determining will futher study that subset of softare components for which hash values are calculated. We believe this is important to ensure real-world applicability.

REFERENCES

[1] Heartbleed Bug Info. In link http://heartbleed.com.

[2] TCG IWG Integrity Report Schema Specification.

[3] Trusted Computing Group - TNC Architecture for Interoperability Specification.

[4] Security Failures in secure devices. In *Black Hat DC Presentation*, 2008.

[5] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[6] Ahmed M Azab, Peng Ning, Emre C Sezer, and Xiaolan Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *2009 Annual Computer Security Applications Conference (ACSAC)*, pages 461–470. IEEE, January 2009.

[7] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Certicloud: a novel tpm-based approach to ensure cloud iaas security. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 121–130. IEEE, 2011.

[8] Ge Cheng and Alex K Ohoussou. Sealed storage for trusted cloud computing. In *Computer Design and Applications (ICCDA), 2010 International Conference on*, volume 5, pages V5–335. IEEE, 2010.

[9] Xue Dongliang, Wu Xiaolong, Gao Yunwei, Song Ying, Tian Xinhui, and Li Zhaopeng. TrustVP: Construction and Evolution of Trusted Chain on Virtualization Computing Platform. In *2012 Eighth International Conference on Computational Intelligence and Security (CIS)*, pages 623–630. IEEE, January 2012.

[10] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[11] Abe Fettig and Glyph Lefkowitz. *Twisted network programming essentials*. O'Reilly Media, Inc., 2005.

[12] Inc Forrester Research. Hybrid Cloud Future: What it Looks Like And How to Get There, 2012. Accessed: 2016-09-16.

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, Dan Boneh, Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. *Terra: a virtual machine-based platform for trusted computing*, volume 37 of *a virtual machine-based platform for trusted computing*. ACM, December 2003.

[14] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital distributed system security architecture. In *Proceedings of the 1989 National Computer Security Conference*, pages 305–319, 1989.

[15] Lorena González-Manzano, Gerd Brost, and Matthias Aumueller. An architecture for trusted paas cloud computing for personal data. In *Trusted Cloud Computing*, pages 239–258. Springer, 2014.

[16] Xin Huang, Tingting Zhang, and Yifan Hou. ID management among clouds. *Future Information Networks, 2009. ICFIN 2009. First International Conference on*, pages 237–241, October 2009.

[17] iSecPartners. Open Crypto Audit Project: Truecrypt.

[18] B Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the USENIX Security Symposium*, 2007.

[19] Chunwen Li, Xu Wu, Chuanyi Liu, and Xiaqing Xie. An Implementation of Trusted Remote Attestation Oriented the IaaSCloud. In *link.springer.com.proxy1.lib.uwo.ca*, pages 194–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] Qian Liu, Chuliang Weng, Minglu Li, and Yuan Luo. An In-VM Measuring Framework for Increasing Virtual Machine Security in Clouds. *Security & Privacy, IEEE*, 8(6):56–62, 2010.

[21] Yong-Hyuk Moon and Yong-Sung Jeon. Cooperative remote attestation for iot swarms. In *Information and Communication Technology Convergence (ICTC), 2016 International Conference on*, pages 1233–1235. IEEE, 2016.

[22] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vtpm: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[23] Martin Pirker, Ronald Toegl, Daniel Hein, and Peter Danner. A privacyca for anonymity and trust. pages 101–119, 2009.

[24] Jonathan Poritz, Matthias Schunter, Els Van Herreweghen, and Michael Waidner. Property attestationscalable and privacy-friendly security assessment of peer computers. 2004.

[25] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. 13:223–238, 2004.

[26] N Santos, R Rodrigues, K P Gummadi, and S Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security*, pages 175–188, 2012.

[27] Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 1–5, May 2009.

[28] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 43–46. ACM, 2010.

[29] Ben Smith, Rick Grehan, and Tom Yager. Byte-unixbench: A Unix benchmark suite. Technical report, 2011.

[30] TCG. Trusted Computing Group (TCG) and the TPM 1.2 Specification. In *Trusted Computing Group*, 2005.

[31] Michael Velten and Frederic Stumpf. Secure and privacy-aware multiplexing of hardware-protected tpm integrity measurements among virtual machines. In *International Conference on Information Security and Cryptology*, pages 324–336. Springer, 2012.

[32] David Wallom, Matteo Turilli, Andrew Martin, Anbang Raun, Gareth Taylor, Nigel Hargreaves, and Alan McMoran. mytrustedcloud: Trusted cloud infrastructure for security-critical computation and data managment. pages 247–254, 2011.

[33] Zhifeng Xiao and Yang Xiao. Security and privacy in cloud computing. *IEEE Communications Surveys & Tutorials*, 15(2):843–859, 2013.

[34] A Yu and D Feng. Real-Time Remote Attestation with Privacy Protection - Springer. *Trust*, 2010.

[35] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.