Department Informatik

Technical Reports / ISSN 2191-5008

Lamya Abdullah, Sebastian Hahn, Felix Freiling

Achieving Consistency of Software Updates against Strong Attackers

Technical Report CS-2019-03

August 2019

Please cite as:

Lamya Abdullah, Sebastian Hahn, Felix Freiling, "Achieving Consistency of Software Updates against Strong Attackers,"
Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2019-03, August
2019.

Achieving Consistency of Software Updates against Strong Attackers

Lamya Abdullah, Sebastian Hahn, Felix Freiling

Chair of Computer Science 1 (IT Security Infrastructures)

Dept. of Computer Science, University of Erlangen, Germany

lamya.abdullah@fau.de, uni@sebastianhahn.net, felix.freiling@fau.de

Abstract—Update systems that regularly distribute updates for installed software systems to end users are an essential part of modern security. Problems arise when the update system is misused and *malicious updates* are sent to a small set of end users only. Such situations can occur if the software supplier has been successfully attacked or is coerced by government agencies to distribute hand-crafted updates to a set of “suspects” containing promiscuous functionality like back-doors. In this paper, we define a set of general security requirements for update systems that encompass protection against malicious updates. We then introduce the design of an update system that satisfies all requirements and present an implementation as an extension to the *advanced package tool* (APT) for the *Debian* operating system. We evaluate the strengths and weaknesses of the system and discuss its large-scale applicability with respect to security and performance overhead.

Index Terms—Malicious Update, Software Update, Consistency, Debian, Merkle Tree.

I. INTRODUCTION

One of the most important security recommendations for end users is to regularly update software applications and their operating system. The reason for this recommendation is clear: software updates often patch security vulnerabilities, and regular and timely updates thereby decrease the chance of a known software vulnerability being exploited [1]. However, users have been reported to be annoyed and confused by update messages they received [2], and because of past experiences, e.g., with severe changes in functionality, users often feel reluctant to update software [3].

There also exists another way in which update systems can threaten the security of a system: If attackers can “hijack” a software update and insert their own code, they can gain control over a system in the same way as by exploiting an un-patched vulnerability. Fortunately, today, most update systems associated with widely deployed software systems like APT for Debian Linux, Ap-

ple’s AppStore or Mozilla’s Update System for Firefox, Thunderbird and Tor Browser can be considered secure in the sense that end users regularly receive updates in a timely and authentic way. However, the ability of software suppliers to target individual users by abusing their software update systems remains and is of concern to users, since it can also make the supplier a target for legal pressure or blackmail.

Attacking individual users can have many different motivations. Documented cases include the attack on system administrators’ computing infrastructure to gain access to the administered resources at a later time [4], [5], the case of the San Bernadino attacker’s iPhone [6] as well as *Stuxnet* [7], in addition to a recent case of ShadowHammer update attack against ASUS laptops [8]. Generally, if a user’s computing device cannot be accessed directly and hacking attempts fail, software suppliers that provide software updates to their users become a target for collusion with the attacker. Effectively, that means the software supplier becomes the attacker.

The security of update systems today heavily relies on the Internet’s authentication infrastructure which mostly relies on SSL/TLS certificates [9], [10]. The entities that issue certificates are called *Certificate Authorities* (CAs). By construction, CAs are completely trusted third parties that, in case of a compromise, can issue certificates for any domain name, whether the owner of the domain initiated the process or not.¹ To ensure such a breach would not go undetected, Google created the system of *Certificate Transparency* (CT): CT aims at providing a complete archive of all public certificates produced by any browser-trusted CA in order to increase transparency and accountability of CAs on the Internet [12].

The central element in CT is the *Log*, a publicly

¹This became evident in 2011, when the CA *DigiNotar* was breached resulting in the widespread mal-issuance of certificates for domains such as “.google.com”, “.windowsupdate.com” and “.torproject.org” [11].

accessible service providing a verifiable append-only data structure (a Merkle tree) storing TLS certificates. It accepts all certificates signed by a valid CA and provides *inclusion* and *consistency proofs* as well as a complete listing of all stored contents to facilitate public verification of its correct operation. Using the CT-Log, clients (like web browsers) can check the inclusion of a certificate before they make a connection. The task of detecting anomalies like issuance of certificates by a non-authorized CA is taken over by a set of *monitors* which do not have to be trusted and can be run by any number of interested third parties.

While CT has been criticized for infringing the privacy of users (since every access to a SSL-protected site by a browser is reported to a Log), it ensures that mal-issued certificates will eventually be detected. However, it does not per se protect against a *strong* attacker, i.e., an attacker who has taken over *both* a CA and the Log. To prevent such attacks, monitors regularly verify the append-only property of the Log and exchange information about the state of the Log they have seen. This prevents a Log from providing two different views of its contents to different clients.

So overall, while CT can be utilized to detect attacks on update systems that are based on mal-issued certificates, it cannot detect attacks by the update system itself. If targeted (i.e., *malicious*) software updates involve updates to the operating system, installing a software update can result in a *total compromise* of the system, a situation which — if performed in a stealthy manner — may remain undetected for a long time.

A. Related Work

Prior work has focused on different security aspects of update systems. For example, *The Update Framework* (TUF) [13] provides a way for software update systems to manage cryptographic key compromise without user interruption and without compromising update security properties [14]. So like CT, TUF protects against key compromise but not against situations in which the software supplier becomes the attacker.

It is often claimed that targeted attacks through software updates can be easily detected by comparing update files and raising an alarm if they differ on two machines. However, this is problematic for update systems that distribute binary code [15] because compilers are non-deterministic regarding software build processes. Even recompilation on the same machine can lead to binaries with significant differences, meaning that a user has no way to trust that some software is the result of an un-

tampered build process. To resolve this issue, the concept of *reproducible builds* was introduced. The first attempts at reproducible builds were undertaken by the developers of the *Bitcoin Core* [16], [17] software, later followed by the Tor Project [18], [19].

The idea of reproducible builds usually involves setting up a virtual machine according to a requirement list to precisely specify the used versions of software required for a deterministic build, then using that virtual machine to execute the build process. The resulting binary can then be signed and, if it matches the binary that others have independently produced, will become the next released version. More than 90% of all packages included in the Debian project's repository can be reproducibly built [20], [21].

CHAINIAC [22] aims to defend against attacks on the software supply chain. The design relies on reproducible builds to be executed by a *collective authority* (*cothority*) [23], a set of independent authorities that collectively signs the release. Releases are logged into a distributed ledger to enable a verification chain [24]. While the concept of cothority minimizes the dangers of a single party offering malicious updates, operating a cothority is an operationally complex task as different organizations have different requirements for developers' signatures and the build process. So while the code of CHAINIAC is publicly available, we know of no established infrastructure using it.

Application Transparency (AT) [25] is a framework for Android app markets that enforces transparency of app installations and updates. Inspired by CT, AT applies the same core concept of an append-only log to which the app market/developer submits the applications and receives proofs to be attached to the applications when sent to the users. Similar to CT, the framework utilizes Merkle trees to implement the logs. The AT implementation avoids attacks by the log by replicating it and allowing users to trigger a proof-fetching request to a (set of) "preferred" log(s) before app download takes place. The verification overhead is justified in the context of mobile apps installation, as installation process takes time so that the additional overhead of proof-fetching does not extend the installation process notably. AT focuses solely on the update of user applications while updates to the operating system are assumed to be trusted. Being built on top of the Google Play app market, any update using AT reveals the identity of the client giving rise to privacy concerns.

The advanced package tool (APT) is the default way to install packages for a variety of Linux distributions, most

notably *Debian* and its derivatives. APT installs software from a user-defined list of *software repositories* and also distributed updates for the operating system itself. It uses a three-tiered system to ensure package integrity. It has three components: the *Release*, *Package* files, and the packages themselves.

Release files provide the highest level of package integrity protection. They are signed using PGP and are used to verify all other package information, thus it guarantees update authenticity. The used keys are published to the developer announcement mailing list `debian-devel-announce` [26]. *Package* files contain the metadata about all packages available from a repository. Each available combination of repository, release, architecture and suite has its own *Package* file, to conserve bandwidth during download. Nevertheless, these files can reach sizes of 50 MB. The individual packages are stored in files with names ending in *.deb*. These files include additional metadata that is relevant to the installation process required at first installation/update.

Debian developers submit packages they have built and signed with a personal key to a central repository. A developer's signature is stripped and the package is resigned according to the rules laid out in Debian's policy [27]. Once accepted into a release, the package gets included in a *Package* file and distributed to the download mirrors. When a mirror operator and the release signing team collude and sign a second version of a *Release* file, the mirror operator can choose to distribute that file to anyone, thus APT does not protect from malicious updates. Furthermore, Debian currently requires the installation of the *apt-transport-https* package to allow the option of HTTPS-protected file transfer. But, although not enabled by default, Debian APT does provide the possibility of anonymous usage (through the *apt-transport-tor* package).

B. Contributions

As the example of anonymous usage shows, there appears to be no common understanding as to which properties an update system must satisfy to be called "secure" and whether these properties must only hold for updates to software applications or also for operating system updates. Furthermore, the attacker models in prior work range from rather weak (e.g., malicious CA as in TUF [13]) to strong (e.g., malicious coalition between software supplier and log as in CT [12] and AT [25]) and therefore make these solutions hard to compare.

With this paper we wish to unify the field of secure update systems by (1) providing a common definition of

what it means for an update system to be secure, and (2) by focusing on the most general case, e.g., operating system updates and strong attackers. Based on the deficiencies of current update systems, we propose the design of a *malicious update detector* (MUD). Briefly spoken, MUD has similar goals as CHAINIAC but results in a simpler and more practical design which is inspired by CT: It uses a lightweight distributed infrastructure of clients to check for inconsistencies between installed binaries. In contrast to AT, MUD adds privacy aspects and reduces the complexity of the log infrastructure by introducing the concept of *attestor logs*, i.e., lightweight secondary logs that only replicate the state of the primary log and not its full contents.

More specifically, this paper makes the following contributions:

- 1) We focus on the security properties of update systems, especially their resistance against attacks that target individual users by having them install a different version of a software package compared to everyone else. We unify and formalize these properties in the context of a strong and precisely defined attacker model (Section III).
- 2) We introduce the design of a generic *malicious update detector* (MUD) framework that by design satisfies all security properties of secure update systems (Section IV).
- 3) We present an implementation of MUD, i.e., a general-purpose software updating and installation tool as an extension of APT for the *Debian* operating system (Section V).
- 4) We evaluate the strengths and weaknesses of the system and discuss its applicability with respect to security and performance (Section VI). We thereby show that the implementation provided here is flexible and modular and its performance is satisfactory.

II. BACKGROUND: MERKLE TREES

A *Merkle tree* is a binary hash tree data structure in which leaf nodes are the cryptographic hash of the data elements/content blocks represented by the tree, and each non-leaf node is the hash of the concatenation of its children nodes. Thus, the root uniquely characterizes the data represented by the tree, i.e. the root summarizes all of the data in the leaves. An example is shown in Figure 1: Given a cryptographic hash function H for which collisions cannot be found, the data elements e_i determine the values of the leaf nodes l_j by computing $l_j = H(e_i)$. Each intermediary node n_k is the result of the hash

of its (two) children, i.e., $n_k = H(\text{child}_{k1} || \text{child}_{k2})$. Insertions into the tree add the element to the left-most unoccupied entry, increasing the height of the tree if required and updating all affected parent hashes. Note however, that the left sub-tree of any inner node (which is a full binary tree) will not be changed after insertion. From this construction principle it can be shown [28] that the tree always has a minimal height.

Fig. 1. Merkle Tree: Basic Structure

The value of the root node is called the *tree hash*. It changes with every insertion into the tree and so can be used to prove that a certain element has been inserted into the tree. If the tree hash is protected by a digital signature, it authenticates the contents of the entire tree. In CT, the root is signed with the private key of the CT log provider and called the *signed tree root*. Additionally, a timestamp and the number of contained elements are typically included in the signed data. The timestamp can be used to quickly indicate whether the tree is kept updated with new entries and that it does not go back in time, by comparing two different tree heads.

A Merkle tree can be used to authenticate data: For example, it enables an efficient *proof of inclusion* (also known as *proof of presence*) by returning $\log(n)$ hashes, where n being the count of layers of the tree. A verifier simply has to follow the sequence of hashes down to the leaf node to verify that some data item is represented in the tree. Similarly, a Merkle tree provides the possibility of efficient verification known as *proof of consistency* by proving that a sub-tree has a specific hash value and that therefore one snapshot of the tree is the successor of another snapshot. This means that it is not possible to include a back-dated element into an already established part of the tree.

III. DEFINING SECURE UPDATE SYSTEM

A *malicious update* appears to be an intuitive concept but, to the best of our knowledge, it has not been precisely defined yet. A precise definition, however, is the basis for correct detection. We therefore define what a secure update system is and derive general conditions, based on which a malicious update can be identified. For simplicity, we focus on software distribution as pre-compiled binaries, the standard way to distribute software today.

1) *Update System.*: Abstractly, an update system assumes the following main actors:

- *Software suppliers* regularly produce new versions of software that shall be distributed by the update system to users.
- *Users (or clients)* of the update system are interested to receive latest software versions from software suppliers.

When using the update system, a user first requests the most recent metadata on available software versions. In a second step, the user then selects appropriate versions (e.g., “all software which is newer than the currently installed versions”) and then downloads and installs the corresponding binary programs.² It is worthy to notify that further entities might be involved in real practical software supply chain. However, for the generality of the formal definition, we strict the main actors to the software suppliers and the users to focus on the exact wished properties.

To formalize software versions, a *version vector* is a vector v of update parameters. What exactly constitutes an update parameter can be specific to the used update system, however, the following list applies to general update system as considered in this paper:

- Software version number, usually assigned in increasing order, often referring to different categories of releases (major/minor), e.g., 15.6.
- CPU architecture.
- Features specially requested by the user (like the supported language).

An *update system* provides a *mapping* of version vectors to specific binaries. By providing this mapping, the update system ascertains that the specific binary is valid at the given time instance. More formally, let V denote the set of all possible version vectors, B be the set of all installable binary programs (updates), U be the

²In Debian’s APT, the first step corresponds to the command `apt-get update` and the second to `apt-get upgrade`.

set of users and let T denote the range of real time which we for simplicity take as the range of natural numbers. Therefore, for any point in time an update system can be formalized as a partial function with the following signature:

$$\mu : T \times U \times V \mapsto B$$

Let v be a concrete version vector and u be a user. A query by u of the update system for version vector v at time 10 can then be formalized as follows:

$$\mu(10, u, v) = b$$

The resulting value b means that the update system provides b as the binary belonging to version vector v . Since μ is a partial function, it may be undefined for certain version vectors. This models the situation in which a particular version of a software is not available yet or not supported anymore. We indicate an undefined value with the symbol $\perp$.

Continuing the example above, if v' is the next version of v and not supplied yet by the update system, then $\mu(10, u, v') = \perp$. Note that at a later time, version v might become invalid while v' is available, e.g., $\mu(15, u, v) = \perp$ and $\mu(15, u, v') = b'$.

2) *Update Consistency*.: Intuitively, a malicious update attack is any attempt by a software supplier to install software that is not the same as what all other users who provided the same update vector are receiving. A malicious update therefore exists if an update system maps the same update vector to two different binaries.

More precisely, a *malicious update* occurs if, for a specific version vector v , a first user installs binary b and later a second user installs binary $b' \neq b$. The following *update consistency condition* therefore ensures that no malicious updates occur: $\forall t, t' \in T : \forall u, u' \in U : \forall v \in V : \mu(t, u, v) = b \neq \perp \wedge \mu(t', u', v) = b' \neq \perp \Rightarrow b = b'$. A malicious update implies that the same version vector v has been mapped to two different binaries over time. To detect malicious updates, it is therefore necessary to keep track of all such mappings and regularly check them for inconsistencies.

Note that preventing malicious updates does not prevent situations in which a particular user is the *only one* that installs some binary b . This can happen for special update vectors (e.g., if the MAC address of the computer is used as an update parameter).

3) *Freeze Protection*.: Ensuring the malicious update condition may prevent different users receiving different versions. It does not, however, guarantee that an update system provides updates to all users in the system. If

the update system provides a particular binary to a user at a specific point in time, it must also provide that binary to all other users at the same point in time. We formalize this as the *freeze protection condition* as follows: $\forall t \in T : \forall u, u' \in U : \forall v \in V : \mu(t, u, v) = \perp \Rightarrow \mu(t, u', v) = \perp$. While update consistency is a condition that applies when μ is defined, freeze protection covers situations in which μ is undefined. So while both conditions appear structurally similar, neither implies the other.

4) *Requirements for Secure Update System*.: While the focus of this work is the resistance to malicious update attacks, many deployed update systems fail to protect against much simpler threats. We therefore now define five general security requirements for an update system to be called secure. The first three are general requirements of a secure update system, while the last two are specifically aimed at malicious software suppliers (as defined above).

- *Transport Security*. Observers of network traffic cannot see the content of updates or the version vectors sent over the network by the update system. This requirement is usually satisfied by using encryption/authentication over a secure transport layer.
- *Update Authenticity*. Authenticity of updates means that users will only install supplier-provided software. More formally, users of the update system requesting version vector v will only install binary b if the update system maps v to b at that point in time. This task will usually involve cryptographic routines to check the origin of files (digital signatures).
- *Anonymized Usage*. Anonymized usage is an often underestimated requirement, but it is included here because it is a powerful tool in preventing targeted attacks through malicious updates. It states that the software supplier learns nothing about which user requested which version vector and installed which update.
- *Update Consistency Protection*. Ensures the update consistency condition defined above.
- *Freeze Protection*. Ensures the freeze protection requirement defined above.

IV. DESIGN OF A MALICIOUS UPDATE DETECTOR (MUD)

This section presents the design of malicious update detector (MUD).

A. Entities and Attacker Model.

The design of MUD considers the following entities:

- *Software suppliers* regularly produce new versions of a software to be installed by users. Software in this paper, commonly, can be applications software, firmware, operating systems services and resources.
- *Archive/mirrors*: Software archive, where software versions usually are stored, managed and distributed to users.
- *Primary Log*: This is the main append-only log in MUD to include the pair (b, v) of software binary and version vector for all software packages provided by the software supplier and thus by the archive.
- *Attestor Log*: This is a special log that includes every signed tree root of the primary log and is used to control the behavior of the primary log.
- *Clients (or users)* of the software, interested to receive latest software binaries (given a version vector) from software suppliers.

We assume a strong attacker model meaning that the software supplier, the archive *and* the primary log can collude and together act maliciously. Possible actions can be, for example:

- The software supplier and/or the archive can provide different software versions to different users (*malicious update attack*).
- The software supplier and/or the archive may fail to provide up-to-date version of the software to certain users (*freeze attack*).
- The primary log provider may try to split the log to create special log for targeted users (*split reality attack*).

1) *Secure First Use*: A major problem of update systems is how to secure its *first use*, i.e., the installation of the update system itself. Assuming that the distribution mechanism is secure, the user can actually rely on having the software installed that they were intended to get. Hence, the possibility to ship a device with a pre-installed update system considerably strengthens the security of a system. However, a device pre-loaded with a version of the update system that is made to target that user is impossible for the user to detect. For example, it is well known that the NSA has a unit for computing hardware pre-infection during shipment, the so called *Tailored Access Operations* team [29]. In the following, we assume that an update system has been installed correctly such that a secure first use is ensured. Moreover, the MUD framework design is not affected by

the practical fact that a user may wish to roll back to a previous version of a software. Hence the update system stores all software versions, user shall be able to request a previous version (v_{i-k}) and MUD framework ensures that (v_{i-k}) is not malicious.

B. MUD Framework

We now give an overview of the MUD framework. We note in advance that the framework is designed to always allow the use of anonymous communication for all protocol steps. We assume a total of m attester logs. An overview of the interaction of all entities is shown in Figure 2.

When a software update b regarding version vector v is submitted to the archive (step 1), then (b, v) is also submitted to the primary log (step 2) which updates the Merkle tree and generates a new signed tree root (step 3). It then distributes the signed tree root to all m attester logs (step 4). Only binaries signed with the supplier's key are accepted into the log.

When the client, i.e., the user, wishes to update software, it considers the most recent software metadata information provided by the archive and then sends a request for an update regarding version vector v to the archive (step 5). She then receives a certain software binary b (step 5.1). Before the client installs the software, it requests an inclusion proof for (b, v) from the primary log (step 6). It also requests an inclusion proof of the signed tree root of the current primary log from at least $n > m/2$ attester logs (steps 8 and 9). Only if all these inclusion proofs are valid, the client installs the received version (step 10). Note that while steps 6 to 9 can be performed in parallel, it is vital not to allow software installation based on the proof provided by only the primary log. This ensures that the tree root of the log containing every installed binary is recorded in a majority of attester logs.

Inclusion proofs do not necessary proof the uniqueness of entries of the tree, i.e. a version vector v might be inserted to the tree paired with two different binaries" (v, b) and (v, b') . So that inclusion proofs can be provided for each pair correctly hence the hash root is assigned to all the attester logs. The primary log must be implemented to guarantee the uniqueness of the entities.

C. Monitoring the Primary Log

While attester logs are used to verify the state of the primary log, attester logs do not necessarily check for inconsistencies in the stored data. Similar to CT, the MUD framework assumes independent parties called *monitors*

Fig. 2. MUD Framework Design: All Parties and Steps.

that regularly evaluate the contents of the primary log for violations of policy. Monitors verifying the integrity of a log must, for example, verify the append-only property of the log, ensure all logged documents are available from the log, and ensure all logged documents are signed by a valid key. Any suspicious activity must be flagged for further review and analysis.

D. Ensuring Main Properties

1) *Update Consistency*: An obvious issue with logging software versions is that a targeted user might be presented with an inclusion proof for a different Merkle tree than everyone else. This attack requires a collusion of the primary log with the software supplier, but it must be prevented. In our framework, the client needs to receive correct inclusion proofs from a majority of attester logs. For efficiency reasons, we cannot require each client to query *all* attester logs. Similar to the mechanism that is used in atomic commit protocols to prevent conflicting updates in distributed databases [30], update consistency is ensured via receiving proofs from majority of attester logs. Majority is mandatory, hence it defines what is considered as a correct view, i.e. if there is no majority then a correct view of the update versions can be uncertain (not clear which view is the malicious one).

As mentioned earlier, in order to guarantee the update consistency, the client must query the inclusion proof from a quorum of n attester logs out of m total attester log, where $n > m/2$. Otherwise, with only $n \leq m/2$ attester logs, there would be some uncertainty. Hence, a malicious primary log may inject different views to a group of attester logs from which the user is likely to inquire the proofs.

Fig. 3. Merkle Tree Roots over Time - Views maintained by Attestor Logs.

One main observation we make is that the primary log cannot include two different views within the MT of one attester log. This is because the primary log has to sign all tree roots with the same key. Therefore, attester log shall maintain a view for the MT (the hashed roots hr_i over time), demonstrated by Fig 3a. Given by the properties of MT, it is easy to verify that later versions of a log include everything from an earlier version and that all data is recorded and presented in the log. Thus, if the Primary log attempts to inject different views to different attester logs, i.e. split reality by forking the log view as demonstrated by Fig 3b. Due to the properties of MT, it is not possible re-unite the branches of the view, i.e. hr_{i+1} will not be the same hash for later versions insertion to the MT. A split reality attack will therefore be detectable if a mismatch in the hashes stored at attester logs is recognized, i.e. the view maintained by attester logs will be consistent.

This means that each set of responses from any subset of attester logs received by a client will overlap in at least one element. Therefore, if one client installs a specific update, any other client will also install the same update.

2) *Freeze Protection*: To ensure the timeliness of updates it is important to regularly update metadata about available packages on the archive. Failure to receive a sufficiently recent package list is an indication of a freeze attack.

E. Summary

The design of MUD ensures update consistency and freeze protection. If the communication system is setup correctly, MUD also guarantees transport security and anonymous usage. If the appropriate digital signatures are available and match, MUD also ensures update authenticity. Therefore, MUD fulfills all requirements of secure update systems described in Section III. If the

software supplier and the primary log collude to issue a specific version v' to a targeted user, by attempting to split the log (fork view) and trying to submit different log views to subset of attessor logs, will result in non-matched inclusion proofs. MUD shall notify the user via what-so-called an error-condition and installation is aborted.

V. IMPLEMENTATION

As a proof-of-concept, we have implemented the MUD framework as an extension to the Debian package manager APT. The system prototype is called the *MUD Demonstration Implementation* (MUDDI). For the implementation we used the *Rust* systems programming language developed by *Mozilla* [31]. The implementation of MUDDI utilizes Tor to achieve the anonymity property. MUDDI completely supports anonymous use of the logs. Any general purpose anonymization technology that allows connecting to the Internet can be used for all communication with the primary log and the attessor logs. An overview of logging software and verification processes flow is depicted by Figure. 4.

A. Primary Log

MUDDI is the primary log which regularly downloads package archives from Debian mirrors and produces a signed Merkle tree of the software versions. The signed tree root is regularly distributed to several attessor logs. Before a user installs an update, the MUDDI client requests a proof that the to-be-installed software version has been logged in the primary log. Furthermore, it waits for the inclusion proof of the signed tree root from a sufficiently large quorum of attessor logs before it starts the installation.

Although the protocol specification includes all necessary requirements to implement a monitor, we did not yet implement one, yet. So while all required log-side functionality is available, our focus with the implementation was mainly to measure the efficiency and the practical overhead in a real world prototype. While monitors are important for security analysis, they are not crucial for evaluating the performance of log implementation.

B. MUDDI Client.

The program *muddi-client* supports two different operations: upgrading all installed packages and installing a previously not installed package. The usage is very simple: if a package name is given at the command line that package is installed, otherwise an upgrade is performed.

Fig. 4. Process Flow overview of Prototype Implementation of MUD for Debian.

The implementation is a single binary which first calls out to *apt-get* to update local state information about available updates. This step fetches the *Release*, *Release.gpg*, *InRelease*, and *Package* files. After fetching, *apt-get* verifies the integrity of these files and the process is aborted if verification fails.

Afterwards, depending on the chosen operation, either *apt-get upgrade* or *apt-get install* is called. For this operation the `--download-only` option is used to ensure installation does not proceed. The `--assume-yes` flag is also passed to include all dependencies in the download automatically. The user has the option of declining installation after download verification. Again, all downloaded files are integrity-checked by *apt-get*.

All downloaded resources are hashed and for each hash, inclusion in the primary log is checked by fetching and verifying an inclusion proof. The signed tree root is hashed as well and an inclusion proof is retrieved from each of the attessor logs. MUDDI defaults to requiring all configured attessor logs to have logged the signed tree hash that the primary log presents.

After verification of all resources required for installation is completed, the *apt-get upgrade* or *apt-get install* command is repeated. The `--assume-yes` option is not used to allow the user to abort the process if *apt-get* queries them, and the `--no-download` option is specified to ensure no resources but those already

verified can be fetched.

Any errors are reported to the user and the update process is aborted to allow the user to investigate the situation.

C. Primary Log and Attestor Logs Operations.

Primary log operation can be split into four separate tasks for which we created individual utilities:

- 1) Information about a newly available version of a package in *Debian* needs to be collected. To achieve this, we maintain a local mirror of the *debian* and *debian-security* repositories. *Debian* provides tools to automatically verify the files in local mirrors which means that once a mirror synchronization has completed, all files in the local mirror need to be logged. Two shell scripts, *mirrorbuild.sh* and *mirrorbuildsecurity.sh*, take care of wrapping the *debmirror* tool available in the Debian repository.
- 2) The on-disk data needs to be analyzed for changes that need to be applied to the existing tree. To make this process quicker and prevent recalculating hashes for all locally available packages, the tool *muddi-seed_update* keeps a cache of all previously hashed files. It provides the override flag `--redo-handled` to ignore the cache and re-scan the entire mirror directories. Any hash newly included in the tree also triggers a copy of the original file to a special output directory to be served to monitors doing a full tree traversal.
- 3) The *muddi-sign_tree* program signs, with the primary log's secret key, the previously created Merkle tree thus producing a signed tree root. The attestor logs need to be notified so they can trigger the inclusion process for that new tree head on their end. Any running server processes need to be notified to terminate and re-launch to make use of the updated tree.
- 4) The signed tree is loaded and a port is opened by *muddi*, to listen for client and monitor requests. These requests can be for inclusion proofs, consistency proofs, or tree traversal.

Attestor logs need to follow only three steps: When a new primary log signed tree root becomes available the *muddi-update_attestor* program takes care of verifying the signature and updating the tree. The next step corresponds to step 3 above, but without notifications. The third step is exactly the same as step 4 above.

D. Summary

Overall, MUD (and its prototype implementation MUDDI) ensures that only binaries signed with the supplier's key are accepted into the primary log, and only logged binaries can be installed. The schedule of monitors ensures that the contents of the primary log are analyzed frequently. It allows to configure a minimum number of required attestor log answers before installation can proceed in order to prevent split reality attacks. Finally, the usage of error conditions whenever a resource of sufficient recency cannot be acquired effectively prevents freeze attacks. Error conditions are reported to the user and installation is aborted. Logs with known-compromised keys or for which monitors have found a proof of malbehaviour are immediately treated as not trusted by all other parts of the update system and must be replaced. Therefore, MUDDI — being based on APT — ensures all security requirements of a secure update system.

VI. EVALUATION

The implementation of the MUD design has been successfully used to both install new packages and update existing packages to newer versions on multiple architectures.

A. Performance of Log Implementation

The primary log was set up on a Debian Linux computer (version 9.3) with an Intel Core i7-6700K CPU and 48 GB of memory. The system installation resides on a pair of *Crucial* CT250BX100SSD1 disks (capacity 256 GB), configured as a software *RAID* level 1, encrypted, and using *LVM*. All data is stored on an extra partition residing on two *Seagate* ST8000VN0022-2EL disks (capacity 8 TB), also configured as a level 1 software *RAID*, encrypted, with *LVM* on top. File system caches are dropped before a benchmark unless otherwise indicated. The amount of data stored in the file system exceeds system memory by more than an order of magnitude.

All Rust programs used to produce benchmark data were compiled using *rustc 1.24.0-nightly (4a7c072fa)* in the *release* configuration. All other compilers and benchmark tools are using versions available in Debian 9.3.

1) *Initial import of Debian archive.*: The time the initial import took as measured by the POSIX.2 [32] *time* utility is given in Table I. Performance is heavily dominated by the involved disks' read and write speeds, as can be seen by the relatively low reported *user* and *sys*

TABLE I
IMPORT DURATION AS REPORTED BY *time*.

Type	Initial Release	Point Release	i386-addition	Security Update	No changes
real	89m 51s	3m 7s	33m 57s	0m 30s	0m 18s
user	28m 22s	3m 29s	11m 1s	0m 3s	0m 3s
sys	11m 59s	0m 57s	4m 30s	0m 27s	0m 8s

values (specifying how long MUDDI spent in user-space and kernel code) compared to total wallclock execution time (reported as *real*). This is exacerbated by the fact that the file hashing in MUDDI is embarrassingly parallel and implemented using 8 threads. The reported *user* value should therefore surpass the wall-clock execution time if disk read speeds were of no concern.

2) *Rescan of import files.*: Because MUDDI relies on the Debian mirroring system to include new packages, a re-scan of a synced mirror directory has to happen frequently. These scans generally complete in less than a minute, unless many packages have changed.

The longest-running import operation was when the *i386* architecture was added. The total time for that operation was a bit less than 34 minutes. These import times do not pose a significant problem as security updates can be prioritized and adding those was very fast, as shown in Table I.

To demonstrate the effects of slow disks, an experiment on a smaller dataset was made: Only the debian-security packages were imported with a total file size of 14 GB. No copy to the output directory is performed. The results shown in Table II confirm the positive effects of parallelizing the hashing operations as well as the dependency on disk speed.

3) *Attestor Logs.*: A total of three attestor logs has been set up, on the same computer as the primary log. A real deployment must not mirror that set up, but it is acceptable for performance evaluation benchmarks as different network latencies to the individual logs have been tested using Tor (see also Section VI-B1). The initial import is a trivial operation for an attestor log, because only a single signed tree root has to be included.

4) *Other Server-Side Performance Statistics*: All parts of the Merkle tree data structure need to stay in memory. Exact space requirements depend on the used hash function (for MUDDI it is 32 Bytes per hash value). Storing n elements requires storing at least $2^{\lceil \log n \rceil + 1} + n$ hashes and n array indices (the tree structure itself, as well as a map from inserted element to insertion position). Creating a tree with 100,000,000 entries resulted in a memory use of 17.4 GB while

Fig. 5. Insertion time for a single element into a tree of size $x - 1$.

Fig. 6. Insertion time for a single element into a hash map of size $x - 1$.

still performing well (creating an inclusion proof of maximum length required just $508\text{ ns} \pm 1\text{ ns}$).

5) *Insertion time patterns.*: A benchmark was run measuring the insertion time of an element into a tree of size 0 to 2999. The result is shown in Figure 5. The result shows that the cost of insertion is dominated by the amount of hash operations required to perform, except in the case where a tree is grown from 2^n to 2^{n+1} contained elements. This directly reflects the choice of data structure, because the underlying vector has to grow to accommodate more entries and data has to be copied to be in the correct place. Some other peaks can be

TABLE II
CACHE AND MULTI-THREADING BENCHMARK FOR INITIAL IMPORT OF REDUCED DATA SET.

Category	Duration			
	Multi-threaded		Single-threaded	
	Cold cache	Hot cache	Cold cache	Hot cache
real	1m 13.357s	0m 15.363s	1m 47.709s	0m 56.463s
user	1m 18.413s	1m 45.834s	1m 11.636s	0m 54.100s
sys	0m 4.366s	0m 3.307s	0m 4.044s	0m 2.360s

observed in the data which are not caused by growing the tree itself. Instead, the hash map that is used to look up insertion position for a given element has to reorganize its hash function. This can be clearly seen from Figure 6 which shows performance of hashmap insertion.

The first benchmark resulted in an average insertion time of $11.123\text{ ms} \pm 0.050\text{ ms}$, the second benchmark in $68.570\text{ ms} \pm 0.301\text{ ms}$. Even in the case where a reallocation has to occur the time required for the tree operation is of no concern with regard to the timeliness of updates.

6) *Insertion into a large tree.*: Even for large trees with more than 1,000,000 entries, insertion performance is quite good. Two benchmarks were run, one inserting 1,000 elements into a tree with 1,000,000 entries, the other inserting 1,000 elements into a tree with 1,048,000. The values are chosen this way to ensure the second benchmark will start out with a tree with less than 2^{20} entries and grow to contain more than that, whereas the first benchmark will not cross any 2^n boundaries. The importance of including both types of benchmarks can be seen in Figure 5.

Signing the tree is very cheap since it only affects the tree head. That also means the time this operation takes can be implemented to be completely independent of tree size. In MUDDI, this is not true, because of the overhead of deserialization of the unsigned tree and serialization of the signed tree, which grows linearly with tree size. Signing a tree without serialization takes $27745\text{ ns} \pm 182\text{ ns}$.

B. Client Performance

The *muddi-client* was tested on three different computers with vastly different hardware capabilities. The first device used was the aforementioned host also used for the logs, nicknamed *khon*, the second an ASUS Eee PC with an Intel Atom CPU N280 @ 1.66 GHz and 1 GB of memory (*eeepc*), and the third a LeMaker Banana Pi using an ARMv7 Processor rev 4 (v7l) with 1 GB of memory (*bananapi*). Network latencies

(round-trip time as measured by *ping*) to the log server were $3.814\text{ ms} \pm 7.286\text{ ms}$ from *eeepc* (wifi link) and $0.336\text{ ms} \pm 0.044\text{ ms}$ from *bananapi* (cable-bound connection). To simulate high-latency connections, a Tor hidden service was used as well.

To set up the experiment, the package *tor-arm* was first made available in apt's local package cache, where it remained unchanged for the duration of the experiment. This was to reduce some network latency jitter from the results. The package is also very small (250 kB) and architecture-independent. All dependencies were installed to ensure all measurements would do the same amount of work during each iteration.

For the measurement, *tor-arm* was first uninstalled (including a purge of all configuration files) from the system and then reinstalled. This process was repeated 50 times. For each iteration, the amount of time the installation took was recorded. The install process was modified in three ways: First, *muddi-client* was not used at all, the installation process consisted of the commands

```
apt-get update
apt-get install tor-arm --download-only
apt-get install tor-arm --no-download
```

This mirrors what *muddi-client* does internally. The second measurement used *muddi-client* to make a direct connection to the logs. On *khon*, a third measurement was added which used a Tor hidden service connection to contact all logs (see Section VI-B1). The results can be seen in Fig. 7. Unsurprisingly, raw installation time heavily depends on computer performance. But high network latency plays an even bigger role in the results, as can most easily be seen by the results for *khon*. The overhead of *muddi-client*'s cryptographic operations are negligible on a powerful computer, but the installation process took very long when a Tor hidden service was used for the connection to the logs.

Nevertheless, the performance when using Tor hidden services is good enough to ensure timely updates are

Fig. 7. Time to install the *tor-arm* package.

possible even with the current implementation. Consistently less than two minutes’ time to install an update is a small price to pay compared to the benefits of malicious update protection. Unfortunately, for lack of an embedded device suited to run Rust applications, *muddi-client* was not evaluated on such a device.

1) *Privacy*: To test the full supports of anonymous use of the logs, a Tor [19] hidden service [33] has been used to make all logs accessible. The way this experiment was set up, a total of seven randomly chosen Tor relays would be used to establish a connection between *muddi-client* and *muddi*.

Due to the hidden service setup a log’s location can be hidden as well, making it harder to physically attack the log. The only thing required on the log’s side is a Tor client that can serve as the hidden service endpoint.

VII. CONCLUSIONS AND FUTURE WORK

Malicious updates are software updates that misuse regular update systems to affect a single user only. Given today’s security landscape, malicious update attacks are a relevant threat. We argue that contemporary software update system do not provide satisfactory solutions to the problem. Defending against malicious updates is also in the interest of the software supplier as it removes them as a potential coerced aide in a targeted attack initiated elsewhere. With MUD we presented the design for a malicious update detector that can be integrated with many different kinds of update systems. We also provided an experimental implementation of *MUD* in the form of *MUDDI*. It has been used to perform multiple successful update and installation processes.

The introduced overhead due to *MUDDI* is negligible and can be further reduced with additional optimizations.

The required server infrastructure for *MUD* is not overly complex but does have high availability and data integrity requirements. *MUDDI*’s design lends itself well to separating trusted components from the server components exposed to the internet. While inspired by CT, the concept of attestor logs is a cheap way to prevent a requirement of operating many primary logs while providing the same configurable security properties. Gossiping is not required of clients and all information required for the detection of malicious updates has been collected before the installation proceeds.

To prevent attacks where certain user groups are targeted while the installed program is executing it is important to use reproducibly built software. The design of *MUD* is easily extended to include a provision to accomplish this without overburdening developers or significantly slowing down the release process. This requires cooperation on the software vendor side, as they have to guarantee that only reproducibly built binaries are distributed and that the source code is available.³ The submission process gets changed in the following way: Before a binary can be accepted into the primary log, the log verifies that the binary can actually be produced by executing a reproducible build. While executing this build, the primary log ensures that all additional resources required to complete the reproducible build are logged as well. The implementation demo (*MUDDI*) shows a proof of concept that such a framework can be applied practically with acceptable overhead, so, scalability shall be provided by efficient implementation and deployment.

VIII. FUTURE WORK

Future work involves extending the monitoring side of *MUD*, i.e., investigate how update systems can be efficiently monitored. Only if monitors can reliably detect update system tampering in an acceptable time frame are attackers deterred from performing update hijacking. An efficient monitoring protocol is needed to guarantee the uniqueness property within the logs. Another reasonable research idea could be a study of performance and efficiency of *MUDDI* compared to *CHAINIAC*. Last but not least, additional research into areas where transparency through the usage of logs can be achieved has the chance

³This does not require that the product actually be open source or available free of cost. The vendor is free to choose any license. However, as of today, we are unaware of any proprietary software product vendor making reproducible built binaries available.

to uncover exciting new applications. How these systems can be used to strengthen user privacy will need to be a big focal point of any such research.

1) *Remark::* Source code of MUDDI can be available by request to the authors.

REFERENCES

- [1] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitras, "Some vulnerabilities are different than others - studying vulnerabilities and attack surfaces in the wild," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, 2014, pp. 426–446. [Online]. Available: https://doi.org/10.1007/978-3-319-11379-1_21
- [2] M. Fagan, M. M. H. Khan, and R. Buck, "A study of users experiences and beliefs about software update messages," *Computers in Human Behavior*, vol. 51, pp. 504–519, 2015.
- [3] K. Vaniea and Y. Rashidi, "Tales of software updates: The process of updating software," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, 2016, pp. 3215–3226. [Online]. Available: <http://doi.acm.org/10.1145/2858036.2858303>
- [4] R. Gallagher and G. Greenwald, "How the NSA plans to infect millions of computers with malware," *The Intercept*, vol. 12, 2014.
- [5] The Intercept, "I hunt sys admins," <https://theintercept.com/document/2014/03/20/hunt-sys-admins/>, 2017, accessed: 2017-12-26.
- [6] United States District Court, "Order compelling Apple, Inc. to assist agents in search," https://cdn2.vox-cdn.com/uploads/chorus_asset/file/6053155/in-the-matter-of-the-search.0.pdf, 2016, accessed: 2017-12-26.
- [7] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.
- [8] "Shadowhammer: Malicious updates for asus laptops," <https://www.kaspersky.com/blog/shadow-hammer-teaser/26149/>, March 2019, accessed: 2019-04-29.
- [9] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the https certificate ecosystem," in *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 2013, pp. 291–304.
- [10] E. Rescorla, "HTTP over TLS," RFC 2818, 2000.
- [11] J. R. Prins and B. U. Cybercrime, "Diginotar certificate authority breach operation black tulip," *Fox-IT, November*, 2011.
- [12] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," RFC Editor, Tech. Rep., 2013.
- [13] The Update Framework Project, <https://theupdateframework.github.io>, 2017, accessed: 2017-11-15.
- [14] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *Proceedings of the 17th ACM conference on Computer and Communications Security*. ACM, 2010, pp. 61–72.
- [15] M. J. Karels, "Commercializing open source software," *Queue*, vol. 1, no. 5, p. 40, 2003.
- [16] <https://github.com/bitcoin/bitcoin>, 2017, accessed: 2017-11-15.
- [17] X. de Carné de Carnavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 16–25.
- [18] "Tor project — privacy online," <https://torproject.org>, 2017, accessed: 2017-12-26.
- [19] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," Naval Research Lab Washington DC, Tech. Rep., 2004.
- [20] "reproducible-builds.org," <https://reproducible-builds.org>, 2017, accessed: 2017-11-16.
- [21] Debian, "Overview of various statistics about reproducible builds," <https://tests.reproducible-builds.org/debian/reproducible.html>, 2017, accessed: 2017-11-16.
- [22] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1271–1287. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin>
- [23] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, and B. Ford, "Decentralizing authorities into scalable strongest-link cothorities," *CoRR*, abs/1503.08768, 2015.
- [24] L. Kokoris-Kogias, L. Gasser, I. Khoffi, P. Jovanovic, N. Gailly, and B. Ford, "Managing identities using blockchains and CoSi," in *9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016)*, 2016.
- [25] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith, "Hey, NSA: stay away from my market! future proofing app markets against powerful attackers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 1143–1155. [Online]. Available: <https://doi.org/10.1145/2660267.2660311>
- [26] "Debian Archive Keys," <https://ftp-master.debian.org/keys.html>, 2017, accessed: 2017-12-26.
- [27] "Debian policy manual," <https://www.debian.org/doc/debian-policy/>, 2017, accessed: 2017-12-26.
- [28] B. Laurie, "Certificate transparency," *ACM Queue*, vol. 12, no. 8, pp. 10–19, 2014. [Online]. Available: <https://doi.org/10.1145/2668152.2668154>
- [29] B. Gellman and E. Nakashima, "US spy agencies mounted 231 offensive cyber-operations in 2011, documents show," *Washington Post*, vol. 31, 2013.
- [30] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. [Online]. Available: <http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx>
- [31] "Rust," <https://www.rust-lang.org>, 2017, accessed: 2017-11-23.
- [32] S. R. Walli, "The POSIX family of standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.
- [33] R. Dingledine, "Tor hidden services," *Proc. What the Hack*, 2005.