CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds

Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, and Linus Gasser, *École polytechnique fédérale de Lausanne* (*EPFL*)*;* Ismail Khoffi, *University of Bonn;* Justin Cappos, *New York University;* Bryan Ford, *École polytechnique fédérale de Lausanne* (*EPFL*)

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Chainiac: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds

Kirill Nikitin[1], Eleftherios Kokoris-Kogias[1], Philipp Jovanovic[1], Linus Gasser[1], Nicolas Gailly[1], Ismail Khoffi[2], Justin Cappos[3], and Bryan Ford[1]

[1]École polytechnique fédérale de Lausanne (EPFL)
[2]University of Bonn
[3]New York University

Abstract

Software-update mechanisms are critical to the security of modern systems, but their typically centralized design presents a lucrative and frequently attacked target. In this work, we propose Chainiac, a decentralized software-update framework that eliminates single points of failure, enforces transparency, and provides efficient verifiability of integrity and authenticity for software-release processes. Independent *witness servers* collectively verify conformance of software updates to release policies, *build verifiers* validate the source-to-binary correspondence, and a tamper-proof release log stores collectively signed updates, thus ensuring that no release is accepted by clients before being widely disclosed and validated. The release log embodies a *skipchain*, a novel data structure, enabling arbitrarily out-of-date clients to efficiently validate updates and signing keys. Evaluation of our Chainiac prototype on reproducible Debian packages shows that the automated update process takes the average of 5 minutes per release for individual packages, and only 20 seconds for the aggregate timeline. We further evaluate the framework using real-world data from the PyPI package repository and show that it offers clients security comparable to verifying every single update themselves while consuming only one-fifth of the bandwidth and having a minimal computational overhead.

1 Introduction

Software updates are essential to the security of computerized systems as they enable the addition of new security features, the minimization of the delay to patch disclosed vulnerabilities and, in general, the improvement of their security posture. As software-update systems [17, 24, 34, 35, 48] are responsible for managing, distributing, and installing code that is eventually executed on end systems, they constitute valuable targets for attackers who might, *e.g.*, try to subvert the update infrastructure to inject malware. Furthermore, powerful adversaries might be able to compromise a fraction of the developers' machines or tamper with software-update centers. Therefore, securing update infrastructure requires addressing four main challenges:

First, the integrity and authenticity of updates traditionally depends on a single signing key, prone to loss [53] or theft [29, 32, 70]. Having proper protection for signing keys to defend against such single points of failure is therefore a top priority. Second, the lack of transparency mechanisms in the current infrastructure of software distribution leaves room for equivocation and stealthy backdooring of updates by compromised [15, 46], coerced [11, 28, 66], or malicious [36] software vendors and distributors. Recent work on reproducible software builds [49, 59] attempts to counteract this deficit by improving on the source-to-binary correspondence. However, it is unsuitable for widespread deployment in its current form, as rebuilding packages puts a high burden on end users (*e.g.*, building the Tor Browser bundle takes 32 hours on a modern laptop [60]). Third, attackers might execute a man-in-the-middle attack on the connections between users and update providers (*e.g.*, with DNS cache poisoning [67] or BGP hijacking [6]), thus enabling themselves to mount replay and freeze attacks [15] against their targets. To prevent attackers from exploiting unpatched security vulnerabilities as a consequence of being targeted by one of the above attacks [72], clients must be able to verify timeliness of updates. Finally, revoking and renewing signing keys (*e.g.*, in reaction to a compromise) and informing all their clients about these changes is usually cumbersome. Hence, modern software-update systems should provide efficient and secure means to evolve signing keys and should enable client notification in a timely manner.

To address these challenges, we propose Chainiac, a decentralized software-update framework that removes single points of failure, increases transparency, ensures integrity and authenticity, and retains efficient verifiability of the software-release process.

First, CHAINIAC introduces a decentralized release sign-off model for developers which retains efficient signature verifiability by using a multi-signature scheme. To propose a software release, a threshold of the developers has to sanity-check[1] and sign off on it to express their approval. Third-party witness servers then validate the proposal against a release policy. These witnesses are chosen by the developers and are trusted collectively but not individually. If the proposed release is valid, the witnesses produce a collective signature [69], which is almost as compact and inexpensive to verify as a conventional digital signature. Although improving security, this approach does not place a burden on clients who otherwise would have to verify multiple signatures per updated package.

Second, CHAINIAC introduces *collectively verified builds* to validate source-to-binary correspondence. CHAINIAC's verified builds are an improvement over reproducible builds, in that they ensure that binaries are not only reproducible in principle, but have indeed been identically reproduced by multiple independent verifiers from the corresponding source release. Concretely, this task is handled by a subset of the witness servers, or *build verifiers*, that reproducibly build the source code of a release, compare the result with the binary provided by the developers, and attest this validation to clients upon success. An additional advantage of this approach is that companies, in order to provide the source-to-binary guarantee to customers, can reveal source code only to third-party build verifiers who sign appropriate non-disclosure agreements.

Third, CHAINIAC increases transparency and ensures the accountability of the update process by implementing a public update-timeline that comprises a release log, freshness proofs, and key records. The timeline is maintained collectively by the witness servers such that each new entry can only be added – and clients will only accept it – if appropriate thresholds of the witnesses and build verifiers approve it. This mechanism ensures the source-to-binary binding to protect clients from compile-time backdoors or malware, and it guarantees that all users have a consistent view of the update history, preventing adversaries from stealthily attacking targeted clients with compromised updates. Even if an attacker manages to slip a backdoor into the source code, the corresponding signed binary stays publicly available for scrutiny, thereby preventing secret deployment against targeted users.

[1]Precise details of this review process depend on the developers' engineering disciplines, which are also security-critical but are beyond the scope of this paper.

Finally, to achieve tamper evidence, consistency, and search efficiency of the timeline, and to enable a secure rotation of signing keys, CHAINIAC employs *skipchains*, novel authenticated data structures inspired by skip lists [55, 61] and blockchains [41, 56]. The skipchains enable clients to efficiently navigate arbitrarily long update timelines, both forward (*e.g.*, to validate a new software release) and backward (*e.g.*, to downgrade or verify the validity of older package-dependencies needed for compatibility). Back-pointers in skipchains are cryptographic hashes, whereas forward-pointers are collective signatures. Due to skipchains, even resource-constrained clients (*e.g.*, IoT devices) can obtain and efficiently validate binary updates, using a hard-coded initial software version as a trust anchor. Such clients do not need to continuously track a release chain, like a Bitcoin full-node does, but can privately exchange, gossip, and independently validate on-demand newer or older blocks due to the skipchain's forward and backward links being offline-verifiable. Although blockchains are well-known tools, to our knowledge the skipchain structure is novel and can be useful in other contexts, besides software updates.

The evaluation of our prototype implementation of CHAINIAC on reproducible Debian packages shows that, in a group of more than a hundred verifiers, the end-to-end cost per witness of release attestation is on average five minutes per package, with the verified builds dominating this overhead. Furthermore, skipchains can increase the security of PyPI updates with minimal overhead, whereas a strawman approach would incur the increase of 500%. Finally, creating a skipblock of the aggregate update timeline for the full Debian repository of about 52,000 packages requires only 20 seconds of CPU time for a witness server, whereas receiving the latest skipblock on a client introduces only 16% of overhead to the usual communication cost of the APT manager [23].

In summary, our main contributions are as follows:

- We propose CHAINIAC (Sections 3 and 5), a software-update framework that enhances security and transparency of the update process via system-wide decentralization and efficiently verifiable logging.
- We introduce skipchains (Section 4), a novel authenticated data structure that enables secure trust delegation and efficient bi-directional timeline traversal, and we discuss their application in the context of CHAINIAC.
- We conduct an informal security analysis (Section 6) of CHAINIAC, justifying its resilience in common attack scenarios.
- We implement CHAINIAC (Section 7) and evaluate (Section 8) a prototype on real-world data from the Debian and PyPI package repositories.

2 Background

In this section, we give an overview of the concepts and notions CHAINIAC builds on, this includes scalable collective signing, reproducible builds, software-update systems, blockchains, and decentralized consensus.

2.1 Collective Signing and Timestamping

CoSi [69] is a protocol for large-scale collective signing. Aggregation techniques and communication trees [25,73] enable CoSi to efficiently produce compact Schnorr multi-signatures [64] and to scale to thousands of participants. A complete group of signers, or *witnesses*, is called a collective authority or *cothority*. CoSi assumes that signature verifiers know the public keys of the witnesses, all of which are combined to form an aggregate public key of the cothority. If witnesses are offline during the collective signing process or refuse to sign a statement, the resulting signature includes metadata that documents the event.

In CHAINIAC, we rely on CoSi for efficient collective signing among a large number of witnesses. Furthermore, we use the witness-cosigned timestamp service [69] as a building block in our design for the protection of clients against replay and freeze attacks [15] (where clients are blocked from learning about the availability of new software updates by an adversary). We describe the design of the protection mechanism in Section 5.6.

2.2 Reproducible Builds

Ensuring that source code verifiably compiles to a certain binary is difficult in practice, as there are often non-deterministic properties in the build environment [49,59], which can influence the compilation process. This issue poses a variety of attack vectors for backdoor insertion and false security-claims [36]. Reproducible builds are software development techniques that enable users to compile deterministically a given source code into one same binary, independent of factors such as system time or build machines. An ongoing collaboration of projects [62] is dedicated to improving these techniques, *e.g.*, Debian claims that 90% of its packages in the testing suite are reproducible [22], amounting to ∼21,000 packages. To provide a source-to-binary attestation as one of the guarantees, CHAINIAC relies on software projects to adopt the practices of reproducible builds.

2.3 Roles in Software-Update Systems

The separation of roles and responsibilities is one of the key concepts in security systems. TUF [63] and its successor, Diplomat [44], are software-update frameworks that make update systems more resilient to key compromise by exploiting this concept. In comparison to classic systems, these frameworks categorize the tasks that are commonly involved in software-update processes and specify a responsible role for every category. Each of these roles is then assigned a specific set of capabilities and receives its own set of signing keys, which enables TUF and Diplomat to realize different trade-offs between security and usability. For example, frequently used keys with low-security risks are kept online, whereas rarely needed keys with a high-security risk are kept offline, making it harder for attackers to subvert them. To achieve, for each role, the sweet-spot between security and usability, we follow a similar delegation model in our multi-layered architecture in Section 5.6. However, we decentralize all these roles, use a larger number of keys, and log their usage and evolution to further enhance security and add transparency.

2.4 Blockchains and Consensus

Introduced by Nakamoto [56], blockchains are a form of a distributed append-only log that is used in cryptocurrencies [56, 75] as well as in other domains [41, 74]. Blockchains are composed of *blocks*, each typically containing a timestamp, a nonce, a hash of the previous block, and application-specific data such as cryptocurrency transactions. As each block includes a hash of the prior block, it depends on the entire prior history, thus forming a tamper-evident log.

CHAINIAC uses *BFT-CoSi*, introduced in ByzCoin [42], as a consensus algorithm to ensure a single consistent timeline, *e.g.*, while rotating signing keys. BFT-CoSi implements PBFT [16] by using collective signing [69] with two CoSi-rounds to realize PBFT's prepare and commit phases. CHAINIAC's skipchain structure is partly inspired by blockchains [41]: Whereas ByzCoin also uses collective signatures to enable light-client verification, skipchains extend this functionality with skiplinks to enable clients to efficiently track and validate update timelines, instead of downloading and validating every signature. As a result skipchains can be used for more efficient offline verification of transactions in distributed ledger systems that work with consensus committees [2,42,43].

3 System Overview

In this section, we state high-level security goals that a hardened software-update system should achieve, we introduce a system and threat model, and we present an architectural overview of our proposed framework.

3.1 Security Goals

To address the challenges listed in Section 1, we formulate the following security goals for CHAINIAC:

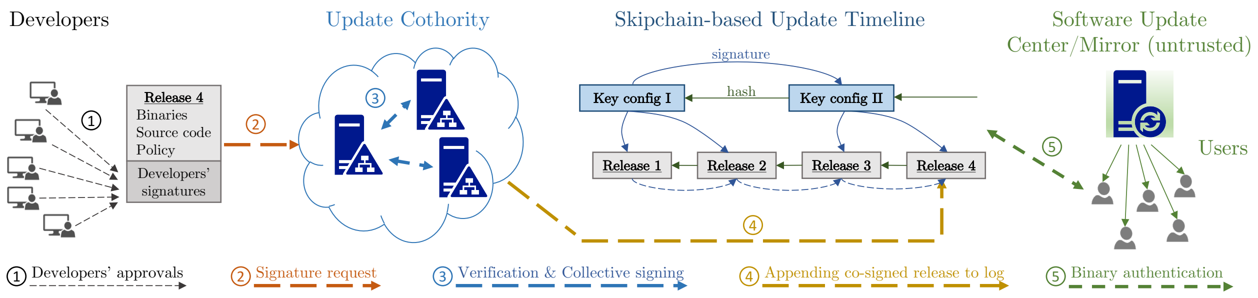

Figure 1: Architectural overview of CHAINIAC

1. **No single point of failure:** The software-update system should retain its security guarantees in case any single one of its components fails (or gets compromised), whether it is a device or a human.
2. **Source-to-binary affirmation:** The software-update system should provide a high assurance-level to its clients that the deployed binaries have been built from trustworthy and untampered source code.
3. **Efficient release-search and verifiability:** The software-update system should provide means to its clients to find software release (the latest or older ones) and verify its validity in an efficient manner.
4. **Linear immutable public release history:** The software-update system should provide a globally consistent tamper-evident public log where each software release corresponds to a unique log entry that, once created, cannot be modified or deleted.
5. **Evolution of signing keys:** The software-update system should enable the rotation of authoritative keys, even when a (non-majority) subset of the keys is compromised.
6. **Timeliness of updates:** Clients should be able to verify that the software indeed corresponds to the latest one available.

3.2 System and Threat Model

In the system model, we introduce terminology and basic assumptions; and, in the threat model, potential attack scenarios against CHAINIAC.

System model. *Developers* write the source code of a software project and are responsible for approving and announcing new project *releases*. Each release includes source code, binaries (potentially, for multiple target architectures), and metadata such as release description. A *snapshot* refers to a set of releases of different software projects at a certain point in time. Projects can have single or multiple packages. *Witnesses* are servers that can validate and attest statements. They are chosen by the developers and should be operated ideally by both developers and independent trusted third parties. Witnesses are trusted as a group but not individually. *Build verifiers* are a subset of the witnesses who execute, in addition to their regular witness tasks, reproducible building of new software releases and compare them to the release binaries. Witnesses and build verifiers jointly form an *update cothority* (collective authority). The *update timeline* refers to a public log that keeps track of the authoritative signing keys, as well as the software releases. *Users* are clients of the system; they receive software releases through an (untrusted) *software-update center*.

Threat model. We assume that a threshold t_d of n_d developers are honest, meaning that less than t_d are compromised and want to tamper with the update process. We further assume that a threshold t_w of n_w witness servers is required for signing, whereas at most $f_w = n_w - t_w$ witnesses can potentially be faulty or compromised. To ensure consistency and resistance to fork attacks, CHAINIAC requires $n_w \geq 3f_w + 1$, hence, $t_w >= 2f_w + 1$. If this property is violated, CHAINIAC does not guarantee single history of the update timeline, however, even then, each history will individually be valid and satisfy the other correctness and validation properties, provided fewer than t_w witnesses are compromised. Furthermore, a threshold t_v of n_v build verifiers is honest and uses a trustworthy compiler [71] such that malicious and legitimate versions of a given source-code release are compiled into different binaries. Software-update centers and mirrors might be partially or fully compromised. Moreover, a powerful (*e.g.*, state-level) adversary might try to target a specific group of users by coercing developers or an update center to present to his targets a malicious version of a release. Finally, we assume that users of CHAINIAC are able to securely bootstrap, *i.e.*, receive the first version of a software package with a hard-coded initial public key of the system via some secure means, *e.g.*, pre-installed on a hard drive, on a read-only media, or via a secure connection.

An attack on the system is successful if an attacker manages to accomplish at least one of the following:

- Make developers sign the source code that they do not approve.
- Substitute a release binary with its tampered version such that the update cothority signs it.
- Trick the update cothority into signing a release that is not approved by the developers.
- Create a valid fork of the public release history or modify/revoke its entries; or present different users with different views of the history.
- Trick an outdated client into accepting a bogus public key as a new signing key of the update cothority.
- Get a client to load and run a release binary that is not approved by the developers or validated by the update cothority.

3.3 Architecture Overview

An illustration of Chainiac, showing how its various components interact with each other, is given in Figure 1. To introduce Chainiac, we begin with a simple strawman design that most of today's software-update systems use, and we present a roadmap for evolving this design into our target layout. Initially, we assume that only a single, static, uncompromisable cryptographic key pair is used to sign/verify software releases. The private key might be shared among a group of developers, and the public key is installed on client devices, *e.g.*, during a bootstrap. To distribute software, one of the developers builds the source code and pushes the binary to a trusted software-update center from where users can download and install it. This strawman system guarantees that users receive authenticated releases with a minimal verification overhead.

This design, though common, is rife with precarious assumptions. Expecting the signing key to be uncompromisable is unrealistic, especially if shared among multiple parties, as attackers need to subvert only a single developer's machine to retrieve the secret key or to coerce only one of the key owners. For similar reasons, it is utopian to assume that the software-update center is trustworthy. Moreover, without special measures, it is hard to verify that the binaries were built from the given (unmodified) source code, as the compilation process is often influenced by variations in the building-environment, hence non-deterministic. If an attacker manages to replace a compiled binary with its backdoored version, before it is signed, the developers might not detect the substitution and unknowingly sign the subverted software.

Eliminating these assumptions creates the need to track a potentially large number of dynamically changing signing keys; furthermore, checking a multitude of signatures would incur large overheads to end users who rarely update their software. To address these restrictions, we transform the strawman design into Chainiac in six steps:

1. To protect against a single compromised developer, Chainiac requires that developers have individual signing keys and that a threshold of the developers sign each release, see step ① in Figure 1.
2. To be able to distribute verified binaries to end users, we introduce developer-signed reproducible builds. Although users still need to verify multiple signatures, they no longer need to build the source code.
3. To further unburden users and developers, we use a cothority to validate software releases (check developer signatures and reproducible binaries) and collectively sign them, once validated: steps ② and ③ in Figure 1.
4. To protect against release-history tampering or stealthy developer-equivocation, we adopt a public log for software releases in the form of collectively signed decentralized hash chains, see step ④ in Figure 1.
5. To enable efficient key rotation, we replace hash chains with skipchains, blockchain-like data structures that enable forward linking and decrease verification overhead by multi-hop links.
6. To ensure update timeliness and further harden the system against key compromise, we introduce a multi-layer skipchain-based architecture that, in particular, implements a decentralized timestamp role.

Before presenting Chainiac in detail in Section 5, we introduce skipchains, one of Chainiac's core building blocks, in Section 4.

4 Skipchains

Skipchains are authenticated data structures that combine ideas from blockchains [41] and skiplists [55, 61]. Skipchains enable clients (1) to securely traverse the timeline in both forward and backward directions and (2) to efficiently traverse short or long distances by employing multi-hop links. Backward links are cryptographic hashes of past blocks, as in regular blockchains. Forward links are cryptographic signatures of future blocks, which are added retroactively when the target block appears.

We distinguish *randomized* and *deterministic* skipchains, which differ in the way the lengths of multi-hop links are determined. The link length is tied to the height parameter of a block that is computed during block creation, either randomly in randomized skipchains or via a fixed formula in deterministic skipchains. In both approaches, skipchains enable logarithmic-cost timeline traversal, both forward and backward.

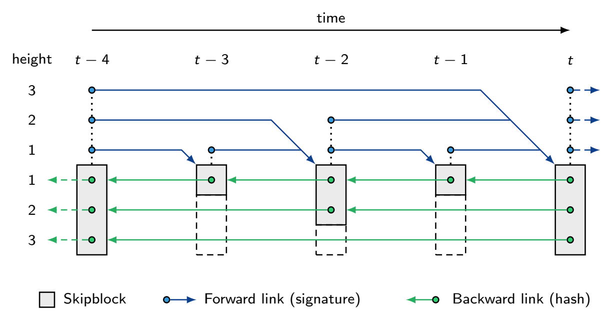

Figure 2: A deterministic skipchain $\mathcal{S}_2^3$

4.1 Design

We denote a skipchain by $\mathcal{S}_b^h$ where $h \geq 1$ and $b > 0$ are called *skipheight* and *skipbasis*, respectively. If $0 < b < 1$ we call the skipchain randomized; and if $b \geq 1$ (b integer), we call it deterministic. The elements of a skipchain are *skipblocks* $\mathcal{B}_t = (\mathsf{id}_t, h_t, D_t, B_t, F_t)$ where $t \geq 0$ is the block index. The variables id_t, h_t, D_t, B_t, and F_t denote block identifier, block height, payload data, list of backward links, and list of forward links, respectively. Both B_t and F_t can store exactly h_t links and a reference at index $0 \leq i \leq h_t - 1$ in B_t (F_t) points to the last (next) block in the timeline having at least height $i+1$. For deterministic skipchains this block is $\mathcal{B}_{t-j}$ ($\mathcal{B}_{t+j}$) where $j = b^i$.

The concrete value of h_t is determined by the dependency of the skipchain's type: if $\mathcal{S}_b^h$ is randomized, then a coin, with probability b to land on heads, is repeatedly flipped. Once it lands on tails, we set $h_t = \min\{m, h\}$ where m denotes the number of times it landed on heads up to this point. If $\mathcal{S}_b^h$ is deterministic, we set

$$h_t = \max\{i : 0 \leq i \leq h \wedge 0 \equiv t \bmod b^{i-1}\} .$$

Fig. 2 illustrates a simple deterministic skipchain.

During the creation of a block, its identifier is set to the (cryptographic) hash of D_t and B_t, both known at this point, *i.e.*, $\mathsf{id}_t = \mathsf{H}(D_t, B_t)$. For a backward link from $\mathcal{B}_t$ to $\mathcal{B}_{t-j}$, we simply store id_{t-j} at index i in B_t. This works as in regular blockchains but with the difference that links can point to blocks further back in the timeline.

Forward links [41], are added retroactively to blocks in the log, as future blocks do not yet exist at the time of block creation. Furthermore, forward links cannot be cryptographic hashes, as this would result in a circular dependency between the forward link of the current and the backward link of the next block. For these reasons, forward links are created as digital (multi-)signatures. For a forward link from $\mathcal{B}_t$ to $\mathcal{B}_{t+j}$, we store the cryptographic signature $\langle \mathsf{id}_{t+j} \rangle_{E_t}$ at index i in F_t where E_t denotes the entity (possibly a decentralized collective such as a BFT-CoSi cothority [41, 42, 69]) that represents the head of trust of the system during time step t. To create the required signatures for the forward links until all slots in F_t are full, in particular, E_t must "stay alive" and watch the head of the skipchain. Once this is achieved, the job of E_t is done and it ceases to exist.

4.2 Useful Properties and Applications

Skipchains provide a framework for timeline tracking, which can be useful in other domains such as cryptocurrencies [42, 43, 56], key-management [41, 51], certificate tracking [1, 45] or, in general, for membership evolution in decentralized systems [68, 69]. Beyond the standard properties of blockchains, skipchains offer the following two useful features.

First, skipchains enable clients to securely and efficiently traverse arbitrarily long timelines, both forward and backward from any reference point. If the client has the correct hash of an existing block and wants to obtain a future or past block in the timeline from an untrusted source (such as a software-update server or a nearby peer), to cryptographically validate the target block (and all links leading to it), the client needs to download only a logarithmic number of additional, intermediate blocks.

Secondly, suppose two resource-constrained clients have two reference points on a skipchain, but have no access to a database containing the full skipchain, *e.g.*, clients exchanging peer-to-peer software updates while disconnected from any central update server. Provided these clients have cached a logarithmic number of additional blocks with their respective reference points – specifically the reference points' next and prior blocks at each level – then the two clients have all the information they need to cryptographically validate each others' reference points. For software updates, forward validation is important when an out-of-date client obtains a newer update from a peer. Reverse validation (via hashes) is useful for secure version rollback, or in other applications, such as efficiently verifying a historical payment on a skipchain for a cryptocurrency.

5 Design of CHAINIAC

In this section, we present CHAINIAC, a framework enhancing security and transparency of software updates. For clarity of exposition, we describe CHAINIAC step-by-step starting from a strawman update-system that uses one key to sign release binaries, as introduced in Section 3. We begin by introducing a decentralized validation of both source code and corresponding binaries, while alleviating the developer and client overhead. We then improve transparency and address the evolution of update configurations by using skipchains. Finally, we reduce traversal overheads with multi-level skipchains and demonstrate how to adapt CHAINIAC to multi-package projects.

5.1 Decentralized Release-Approval

The first step towards CHAINIAC involves enlarging the trust base that approves software releases. Instead of using a single (shared) key to sign updates, each software developer signs using their individual keys. At the beginning of a project, the developers collect all their public keys in a *policy* file, together with a threshold value that specifies the minimal number of valid developer signatures required to make a release valid. Complying with our threat model, we assume that this policy file, as a trust anchor, is obtained securely by users at the initial acquisition of the software, *e.g.*, it can reside on a project's website as often is the case with a single signing key in the current software model.

Upon the announcement of a software release, which can be done by a subset or all developers depending on the project structure, all the developers check the source code and, if they approve, they sign the hash of it with their individual keys, *e.g.*, using PGP [14], and they add the signatures to an append-only list. Signing source code, instead of binaries, ensures that developers can realistically verify (human-readable) code.

The combination of the source code and the signature list is then pushed to the software-update center from where a user can download it. For simplicity, we first assume that the update center is trusted, later relaxing this assumption. When a user receives an update, she verifies that a threshold of the developers' signatures is valid, as specified in the policy file already stored on user's machine. If so, the user builds the binary from the obtained source code and installs it. An attacker trying to forge a valid software-release needs to control the threshold of the developers' keys, which is presumably harder than gaining control over any single signing key.

5.2 Build Transparency via Developers

The security benefits of developers signing source-code releases come at the cost of requiring users to build the binaries. This cost is a significant usability disadvantage, as users usually expect to receive fully functional binaries directly from the software center. Therefore in our second step towards CHAINIAC, we transfer the responsibility of building binaries from users to developers.

When a new software release is announced, it includes not only the source code but also a corresponding binary (or a set of binaries for multiple platforms) that users will obtain via an update center. Each developer now first validates the source code, then compiles it using reproducible build techniques [49, 59]. If the result matches the announced binary, he signs the software release. Assuming a threshold of developers is not compromised, this process ensures that the release binary has been checked by a number of independent verifiers. Upon receiving the update, a user verifies that a threshold of signatures is valid; if so, she can directly install the binary without needing to build it herself.

5.3 Release-Validation via Cothority

Although decentralized developer approval and reproducible builds improve software-update security, running reproducible builds for each binary places a high burden on developers (*e.g.*, building the Tor Browser Bundle takes 32 hours on an average modern laptop [60]). The load becomes even worse for developers involved in multiple software projects. Moreover, verifying many developer-signatures in large software projects can be a burden for client devices, especially when upgrading multiple packages. It would naturally be more convenient for an intermediary to take the developers' commitments, run the reproducible builds and produce a result that is easily verifiable by clients. Using a trusted third party is, however, contrary to CHAINIAC's goal of decentralization. Hence to maintain decentralization, we implement the intermediary as a collective authority or *cothority*.

To announce a new software release, the package developers combine the hashes of the associated source-code and binaries in a Merkle tree [52]. Each developer checks the source code and signs the root hash (of this tree), that summarizes all data associated with the release. The developers then send the release data and the list of their individual signatures to the cothority that validates and collectively signs the release. Clients can download and validate the release's source and/or any associated binary by verifying only a single collective signature and Merkle inclusion proofs for the components of interest.

To validate a release, each cothority server checks the developer signatures against the public keys and the threshold defined in the policy file. Remembering the policy for each software project is a challenge for the cothority that is supposed to be stateless. For now, we assume that each cothority member stores a project-to-policy list for all the projects it serves for. We relax this assumption in Section 5.5. The build verifiers then compile the source code and compare the result against the binaries of the release. The latter verification enables the transition from reproducible builds to *verified builds*: a deployment improvement over reproducible builds, which we introduce. The verified builds enable clients to obtain the guarantee of source-to-binary correspondence without the need to accomplish the resource-consuming building work, due to the broad independent validation.

5.4 Anti-equivocation Measures

Many software projects are maintained by a small group of (often under-funded or volunteer) developers. Hence, it is not unreasonable to assume that a powerful (state-level) attacker could coerce a threshold of group members to create a secret backdoored release used for targeted attacks. In our next step towards CHAINIAC, we tackle the problem of such stealthy developer-equivocation, as well as the threat of an (untrusted) software-update center that accidentally or intentionally forgets parts of the software release history.

We introduce cothority-controlled hash chains that create a public history of the releases for each software project. When a new release is announced, the developers include and sign the summary (Merkle Root) of the software's last version. The cothority then checks the developers' signatures, the collective signature on the parent hash-block, and that there is no fork in the hash-chain (*i.e.*, that the parent hash-block is the last one publicly logged and that there is no other hash-block with the same parent). If everything is valid, it builds the summary for the current release, then runs BFT-CoSi [42] to create a new collective signature. Because the hash chain is cothority controlled, we can distribute the witnessing of its consistency across a larger group: for example, not just across a few servers chosen by the developers of a particular package, but rather across all the servers chosen by numerous developers who contribute to a large software distribution, such as Debian. Even if an attacker controls a threshold of developer keys for a package and creates a seemingly valid release, the only way to convince any client to accept this malicious update is to submit it to the cothority for approval and public logging. As a result, it is not possible for the group to sign the compromised release and keep it "off the public record".

This approach prevents attackers from secretly creating malicious updates targeted at specific users without being detected. It also prevents software-update centers from "forgetting" old software releases, as everything is stored in a decentralized hash chain. CHAINIAC's transparency provisions not only protect users from compromised developers, but can also protect *developers* from attempts of coercion, as real-world attackers prefer secrecy and would be less likely to attack if they perceive a strong risk of the attack being publicly revealed.

5.5 Evolution of Authoritative Keys

So far, we have assumed that developer and cothority keys are static, hence clients who verify (individual or collective) signatures need not rely on centralized intermediaries such as CAs to retrieve those public keys. This assumption is unrealistic, however, as it makes a compromise of a key only a matter of time. Collective signing exacerbates this problem, because for both maximum independence and administrative manageability, witnesses' keys might need to rotate on different schedules. To lift this assumption without relying on centralized CAs, we construct a decentralized mechanism for a trust delegation that enables the evolution of the keys. As a result, developers and cothorities can change, when necessary, their signing keys and create a moving target for an attacker, and the cothority becomes more robust to churn.

To implement this trust delegation mechanism, we employ skipchains presented in Section 4. For the cothority keys, each cothority configuration becomes a block in a skipchain. When a new cothority configuration needs to be introduced, the current cothority witnesses run BFT on it. If completed successfully, they add the configuration to the skipchain, along with the produced signature as a forward link. For the developer keys, the trust is rooted in the policy file. To enable a rotation of developer keys, a policy file needs to be a part of the Merkle tree of the release, hence examined by the developers. Thus, the consistency of key evolution becomes protected by the hash chain. To update their keys, the developers first specify a new policy file that includes an updated set of keys, then, as usual during a new release, they sign it with a threshold of their current keys, thus delegating trust from the old to the new policy. Once the cothority has appended the new release to the chain, the new keys become active and supersede their older counterparts. Anyone following the chain can be certain that a threshold of the developers has approved the new set of keys. With this approach, developers can rotate their keys regularly and, if needed, securely revoke a sub-threshold number of compromised keys.

5.6 Role Separation and Timeliness

In addition to verifying and authenticating updates, a software-update system must ensure update timeliness, so that a client cannot unknowingly become a victim of freeze or replay attacks (see Section 2.1). To retain decentralization in CHAINIAC, we rely on the update cothority to provide a timestamp service. Using one set of keys for signing new releases and for timestamping introduces tradeoffs between security and usability, as online keys are easier compromisable than offline keys, whereas the latter cannot be used frequently. To address the described challenges, we introduce a multi-layer skipchain-based architecture with different trust roles, each having different responsibilities and rights. We distinguish the four roles ROOT, CONFIG, RELEASE, and TIME. The first three are

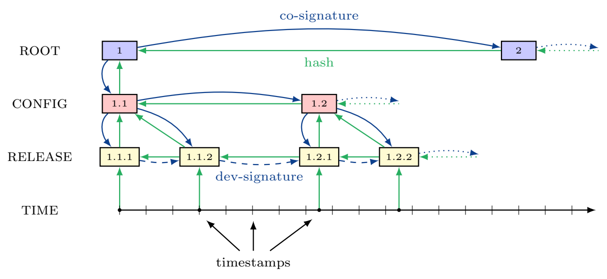

Figure 3: Trust delegation in CHAINIAC

based on skipchains and interconnected with each other through *upward* and *downward* links represented as cryptographic hashes and signatures, respectively. Figure 3 shows an overview of this multi-layer architecture.

The ROOT role represents CHAINIAC's root of trust; its signing keys are the most security-critical. These keys are kept offline, possibly as secrets shared among a set of developer-administrators. They are used to delegate trust to the update cothority and revoke it in case of misbehavior. The ROOT skipchain changes slowly (*e.g.*, once per year), and old keys are deleted immediately. As a result, the ROOT skipchain has a height of one, with only single-step forward and backward links.

The CONFIG role represents the online keys of the update cothority and models CHAINIAC's control plane. These keys are kept online for access to them quicker than to the ROOT keys. Their purpose is to attest to the validity of new release-blocks. The CONFIG skipchain can have higher-level skips, as it can be updated more frequently. If a threshold of CONFIG keys is compromised, the ROOT role signs a new set of CONFIG keys, enabling secure recovery. This is equivalent to a downward link from the ROOT skipchain to the CONFIG skipchain.

The RELEASE role wraps the functionality of the release log, as specified previously, and adds upward links to ROOT and CONFIG skipchains, enabling clients to efficiently look up the latest trusted ROOT and CONFIG configurations required for verifying software releases.

Finally, the TIME role provides a timestamp service that informs clients of the latest version of a package, within a coarse-grained time interval. Every TIME block contains a wall-clock timestamp and a hash of the latest release. The CONFIG leader creates this block when a new RELEASE skipblock is co-signed, or every hour if nothing happens. Before signing it off, the rest of the independent servers check that the hash inside the timestamp is correct and that the time indicated is sufficiently close to their clocks (*e.g.*, within five minutes). From an absence of fresh TIME updates and provided that clients has an approximately accurate notion of the current time[2], the clients can then detect freeze attacks.

5.7 Multiple-Package Projects

To keep track of software packages, users often rely on large software projects, such as Debian or Ubuntu, and their community repositories. Each of these packages can be maintained by a separate group of developers, hence can deploy its own release log. To stay updated with new releases of installed packages, a user would have to frequently contact all the respective release logs and follow their configuration skipchains. This is not only bandwidth- and time-consuming for the user but also requires the maintainers of each package to run a freshness service. To alleviate this burden, we further enhance CHAINIAC to support multi-package projects.

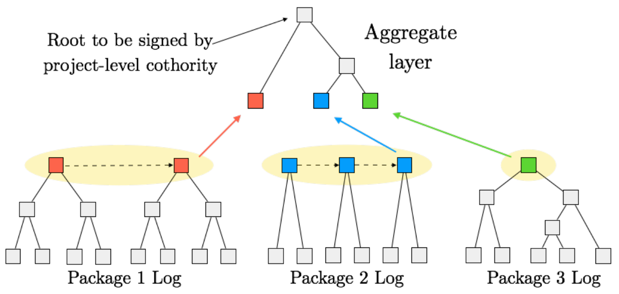

Figure 4: Constructing an aggregate layer in CHAINIAC

We introduce an *aggregate* layer into CHAINIAC: this layer is responsible for collecting, validating and providing to clients information about all the packages included in the project. A project-level update cothority implements a project log where each entry is a *snapshot* of a project state (Figure 4). To publish a new snapshot, the cothority retrieves the latest data from the individual package skipchains, including freshness proofs and signatures on the heads. The witnesses then verify the correctness and freshness of all packages in this snapshot against the corresponding per-package logs. Finally, the cothority forms a Merkle tree that summarizes all package versions in the snapshot, then collectively signs it.

This architecture facilitates the gradual upgrade of large open-source projects, as packages that do not yet have their own skipchains can still be included in the aggregate layer as hash values of the latest release files. The project-level cothority runs an aggregate timestamp service, ensuring that clients are provided with the latest status of all individual packages and a consistent repository state. A

[2] Protecting the client's notion of time is an important but orthogonal problem [50], solvable using a timestamping service with collectively-signed proofs-of-freshness, as in CoSi [69, Section V.A.].

client can request the latest signed project-snapshot from the update cothority and check outdated packages on her system using Merkle proofs. If there are such packages, the client accesses their individual release logs, knowing the hash values of the latest blocks.

A multi-package project can potentially have several aggregate layers, each representing a certain distribution , *e.g.*, based on the development phase of packages, as *stable*, *testing*, and *unstable* in Debian. Individual packages would still maintain a single-view linear skipchain-log but the project developers would additionally tag each release with its distribution affiliation. For example, the stable distribution would then notify clients only when correspondingly tagged releases appear, and would point to the precise block in the package skipchain by providing its hash value, whereas the developers might move ahead and publish experimental versions of the package to its release log. The timeliness is then ensured by maintaining a separate timestamp service for each distribution.

6 Security Analysis

In this section, we informally analyze the security of CHAINIAC against the threat model defined in Section 3.2. We thereby assume that an adversary is computationally bound and unable to compromise the employed cryptosystems (*e.g.*, create hash collisions or forge signatures), except with negligible probability.

Developers. The first point of attack in CHAINIAC is the software-release proposal created by developers. An attacker might try to sneak a vulnerability into the source code, compromise the developers' signing keys, or intercept a release proposal that the developers send to the update cothority, and replace it with a backdoored version. If developers carefully review source-code changes and releases, and fewer than the threshold t_d of developers or their keys are compromised, the attacker alone cannot forge a release proposal that the update cothority would accept.[3] As developer-signed release proposals are cryptographically bound to particular sources and binaries, the update cothority will similarly refuse to sign a release proposal whose sources differ from the signed versions, or whose binaries differ from those reproduced by the build verifiers. If a sub-threshold number of developer keys are compromised without detection, a regular signing key rotation (Section 5.5) can eventually re-establish full security of the developer keys.

Update cothority. The next point an adversary might attack is the update-cothority's witness servers. The witnesses and build verifiers should be chosen carefully by the software project or repository maintainers, should reside in different physical locations, and be controlled by trustworthy, independent parties. For a successful attack, the adversary must compromise at least t_w witnesses to violate the correctness or transparency of the release timeline, and must compromise t_v build verifiers to break the source-to-binary release correspondence. As with developer keys, the regular rotation of cothority keys further impedes a gradual compromise.

If a threshold of online cothority keys are compromised, then, once this compromise is detected, the developers can use the offline ROOT keys to establish a new cothority configuration (see Section 5.6). Non-compromised clients (*e.g.*, those that did not update critical software during the period of compromise) can then "roll forward" securely to the new configuration. An unavoidable limitation of this (or any) recovery mechanism using offline keys, however, is an inability to ensure timeliness of configuration changes. Old clients, whose network connectivity is attacker controlled, could be denied the knowledge of the new configuration, hence remain reliant on the old, compromised cothority configuration. "Fixing" this weakness would require bringing the offline ROOT keys online, defeating their purpose.

Update timeline. An attacker might attempt to tamper with the skipchain-based update timeline containing the authoritative signing keys and the software releases, *e.g.*, by attempting to fork either of the logs, to modify entries, or to present different views to users. The skipchain structure relies on the security of the underlying hash and digital signature schemes. Backward links are hashes ensuring the immutability of the past with respect to any valid release. An attacker can propose a release record with incorrect back-links, but cannot produce a valid collective signature on such a record without compromising a threshold of witnesses, as honest witnesses verify the consistency of new records against their view of history before cosigning. An attacker can attempt to create two distinct successors to the same prior release (a fork), but any honest witness will cosign at most one of these branches. If the cothority is configured with a two-thirds supermajority witness-threshold ($t_w \geq 2n_w + 1$), forks are prevented by the BFT-CoSi consensus mechanism.

Forward links are signatures that can be created only once the (future) target blocks have been appended to the skipchain. This requires that witnesses store the sign-

[3]Of course there is no guarantee that even honest, competent developers will detect all bugs, let alone sophisticated backdoors masquerading as bugs. CHAINIAC's transparency provisions ensure that even compromised releases are logged and open to scrutiny, and the freshness mechanisms ensure that a compromised release does not remain usable in rollback or freeze attacks after being fixed and superseded.

ing keys associated with a given block, until all forward links from that block onwards are generated. This longer key-storage, gives the attacker more time to compromise a threshold of keys. To mitigate this threat, we impose an expiration date on signing keys (*e.g.*, one year), after which honest witnesses delete outdated keys unconditionally, thereby imposing an effective distance limit on forward links. Note that the key expiration-time should be sufficiently long so that the direct forward links are always created to ensure secure trust delegation.

In summary, to manipulate the update timeline managed by the update cothority, an attacker needs to compromise at least a threshold of t_w witness servers. Note that one purpose of the update timeline in CHAINIAC is to ensure accountability so that even if the attacker manages to slip a backdoor into a release, the corresponding source code stays irrevocably available, enabling public scrutiny.

Update center. An adversary might also compromise the software-update center to disseminate malicious binaries, to mount freeze attacks that prevent clients from updating, or to replay old packages with known security vulnerabilities and force clients to downgrade.

Clients can detect that they have received a tampered binary by verifying the associated signature using the public key of the update cothority; the key can be retrieved securely through CHAINIAC's update timeline. The clients will also never downgrade, as they only install packages that are cryptographically linked to the currently installed version through the release skipchain. Finally, assuming the clients have a correct internal clock, they can detect freeze and replay attacks by verifying timestamps and package signatures, because an attacker cannot forge collective signatures of the update cothority to create valid-looking TIME blocks (see Section 5.6).

7 Prototype Implementation

We implemented CHAINIAC in Go [31] and made it publicly available[4], along with the instructions on how to reproduce the evaluation experiments. We built on existing open-source code implementing CoSi [69] and BFT-CoSi [42]. The new code implementing the CHAINIAC prototype was about 1.8kLOC, whereas skipchains, network communication, and BFT-CoSi were 1.2k, 1.5k, and 1.8k lines of code (LOC), respectively. Although the implementation is not yet production quality, it is practical and usable for experimental purposes.

We rely on Git for source-code control and use Git-notes [30], tweaked with server hooks to be append-only, for collecting developer approvals in the form of PGP signatures. For the build verifiers, we use Python to extract the information about the building environment of the packages, and Docker [26] to reproduce it.

[4]https://github.com/dedis/paper_chainiac

8 Experimental Evaluation

In this section, we experimentally evaluate our CHAINIAC prototype. The main question we answer is whether CHAINIAC is usable in practice without incurring large overheads. We begin by measuring the cost of reproducible builds using Debian packages as an example, and we continue with the cost of witnesses who maintain an update-timeline skipchain and the overhead of securing multi-package projects.

8.1 Experimental Methodology

In the experiments of Sections 8.2, 8.3 and 8.4, we used 24-core Intel Xeons at 2.5 GHz with 256 GB of RAM and, where applicable, ran up to 128 nodes on one server with the network-delay set between any two nodes to 100ms with the help of Mininet [54]. Because we had not yet implemented a graceful handling of failing docker-builds, we measured building time in a small grid of 4 nodes and extrapolated this time to the bigger grids in Figure 6. In Section 8.5, we simulated four collectively signing servers on a computer with a 3.1 GHz Intel Core i7 processor and 16 GB of RAM and did not include any network-latencies, as we measured only CPU-time and bandwidth.

To evaluate the witness cost of the long-term maintenance of an update timeline, we used data from the Debian reproducible builds project [22] and the Debian snapshot archive [19]. The former provides checksums and dependency information for reproducible packages. Unfortunately, the information was not available for older package versions, therefore we always verified each package against its newest version. We used the latter as an update history to estimate average cost over time for maintaining an individual update timeline and the overhead of running an aggregate multi-package service. In Section 8.4, we used real-life data from the PyPI package repository [17]. The data represented snapshots of the repository of about 58,000 packages. There were 11,000 snapshots over a period of 30 days. Additionally, we had 1.5 million update-requests from 400,000 clients during the same 30-day period. Using this data, we implemented a simulation in Ruby to compare different bandwidth usages.

8.2 Reproducing Debian Packages

To explore the feasibility of build transparency and to estimate the cost of it for witnesses, we ran an experiment on automatic build reproducing. Using Docker containers, we generated a reproducible build environment for each package, measured the CPU time required to build a binary and verified the obtained hash against a pre-calculated hash from Debian.

We tested three sets of packages: (1) *required* is the set of Debian required packages [21], 27 packages as of today; (2) *popular* contains the 50 most installed Debian packages [20] that are reproducible and do not appear in *required*; (3) *random* is a set of 50 packages randomly chosen from the full reproducible testing set [22]. Figure 5 demonstrates a CDF of the build time for each set.

10 packages from the random set, 8 from required and 2 from popular produced a hash value different from the corresponding advertised hash. 90% of packages from both the random and required sets were built in less than three minutes, whereas the packages in the required-set have a higher deviation. This is expected as, to ensure Debian's correct functioning, the required packages tend to be more security critical and complex.

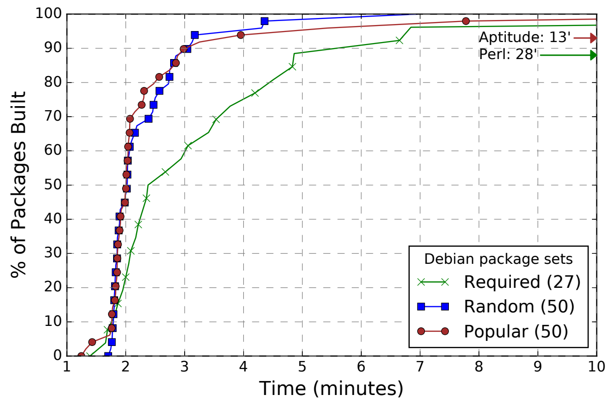

Figure 5: Reproducible build latency for Debian packages

8.3 End-to-End Witness Cost

In this experiment, we measured the cost for a witness of adding a new release to an update timeline. We took a set of six packages, measured the cost for each one individually and then calculated the average values over all the packages. The build time was measured once and copied to the other runs of the experiment, which enabled us to test different configurations quickly and to break out results for each operation. The operations included verifying developers' signatures, reproducible builds, signing off on the new release and generating a timestamp. The witness cost was measured for an update cothorities composed of 7, 31, and 127 nodes.

Figure 6 plots the costs in both CPU time and wall-clock time used. The CPU time is higher than wall-clock time for some metrics, due to the use of a multi-core processor. The verification and build times are constant per node, whereas the time to sign and to generate the timestamp increases with the number of nodes, mostly due to higher communication latency in a larger cothority tree. As expected, the build time dominates the creation of a new skipblock. Every witness spends between 5 and 30 CPU-minutes for each package. Current hosting schemes offer simple servers for 10-US$ per month, enough to run a node doing reproducible builds for the Debian-security repository (about eight packages per day).

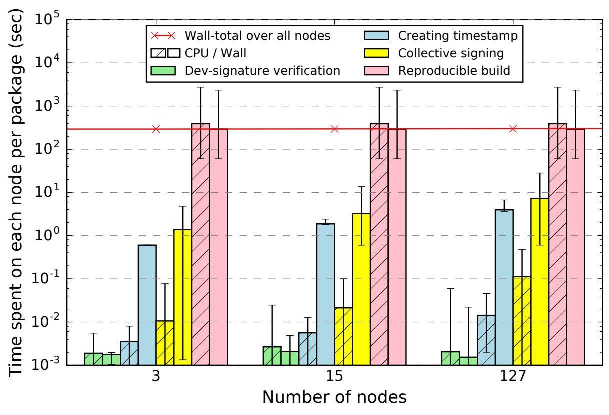

Figure 6: CPU cost of adding a new block to a timeline

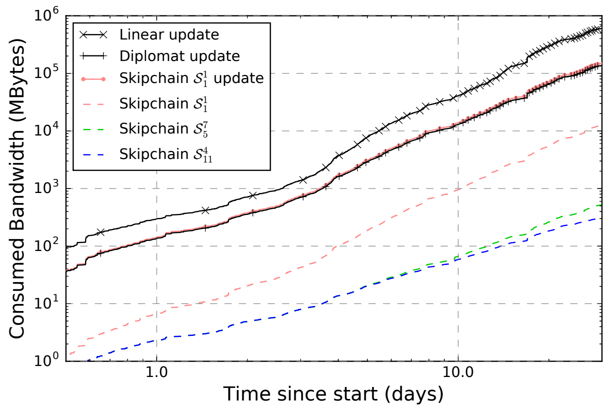

Figure 7: Communication cost for different frameworks

8.4 Skipchain Effect on PyPI Communication Cost

To evaluate the effect on communication cost of using skipchains for update verification, we compare it with two other scenarios using data from the PyPI package repository. The scenarios are as follows:

1. **Linear update:** When a client requests an update, she downloads all the diffs between snapshots, starting

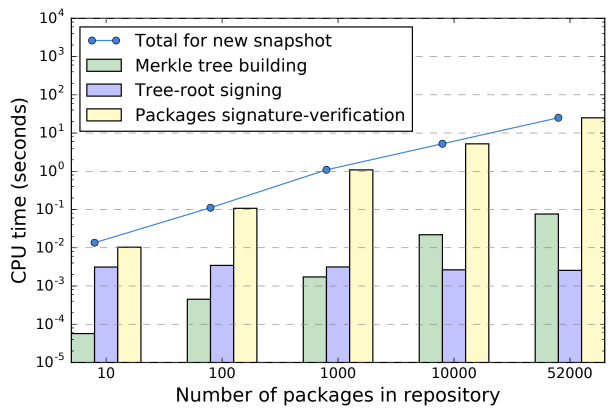

Figure 8: CPU time on server for repository-update

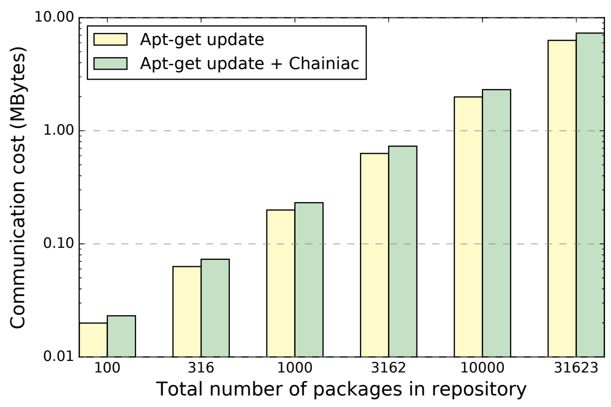

Figure 9: Communication cost to get new repository state

from her last update to the most recent one. This way she validates every step.

2. **Diplomat:** The client only downloads the diff between her last update and the latest update available.
3. **Skipchain S_1^1:** The scenario is as in Diplomat, but every skipblock is also sent to prove the correctness of the current update. The skipchains add security to the snapshots by signing it and by enabling users to efficiently track changes in the signers.

The results over the 30-day data are presented in Figure 7. The straight lines correspond to the aforementioned scenarios. Linear updates increase the communication cost, because the cumulative updates between two snapshots can contain different updates, which are only transferred once, of the same package, as in the case of Diplomat or skipchains. As it can be seen, the communication costs for Diplomat and skipchain are similar, even in the worst case where a skipchain has height-1 only, which corresponds to a simple double-linked list.

To further investigate the best parameters of the skipchain, we plotted only the skipchain overhead using the same data. In Figure 7, the dashed lines show the additional communication cost for different skipchain parameters. We observe that a skipchain with height > 1 can reduce by a factor of 15 the communication cost for proving the validity of a snapshot. Using the base 5 for the skipchain can further reduce the communication cost by another factor of 2.

8.5 Cost of Securing Debian Distribution

In our final experiment, we measured the cost of a witness server that deploys an aggregate-layer skipchain in a multi-package project (Section 5.7) and a client who uses it. . We took the list of all the packages from the snapshot archive of the Debian-testing repository and created one skipchain per package over 1.5-year history, such that each skipblock is one snapshot every five days. We then formed the aggregate Debian-testing skipchain over the same period.

In the first experiment, a witness server receives a new repository-state to validate, verifies the signature for all the packages, builds a Merkle tree from the heads of the individual skipchains and signs its root, thus creating a new aggregate skipblock. Figure 8 depicts the average costs of the operations, over the whole history, against the size of the repository. For a full repository of 52k packages, which corresponds to the actual Debian-testing system, the overall CPU-cost is about 20 seconds per release. This signifies that CHAINIAC generates negligible overhead on the servers that update a skipchain.

The second experiment evaluates the overhead that CHAINIAC introduces to the client-side cost of downloading the latest update of all packages. In order to maintain the security guarantees of CHAINIAC, the client downloads all package hashes and builds a full Merkle tree to verify them, thereby not revealing packages of interest and preserving her privacy. Figure 9 illustrates that CHAINIAC introduces a constant overhead of 16% to the APT manager. This modest overhead suggests CHAINIAC's good scalability and applicability.

9 Related Work

We organize the discussion topically and avoid redundancy with the commentary in Section 2.

Software-update protection. The automatic detection and installation of software updates is a common operation in computer and mobile systems, and there are many tools for this task, such as package- and library-managers [18, 23, 33, 76], and various app stores. There are several security studies [10, 15, 57] that reveal weaknesses in the design of software-update systems, and different solutions are proposed to address these weaknesses. Solutions that reduce the trust that end users must have in developers by involving independent intermediaries in testing are shown [3, 4] to be beneficial in open-source projects and content repositories. Several systems, such as Meteor [7], DroidRanger [77] and ThinAV [37], focus on protecting the infrastructure for mobile applications and on detecting malware in mobile markets. Other systems [38, 47, 58] use overlay and peer-to-peer networks for efficient dissemination of security patches, whereas Updaticator [5] enables efficient update distribution over untrusted cache-enabled networks.

Certificate, key, and software transparency. Bringing transparency to different security-critical domains has been actively studied. Solutions for public-key validation infrastructure are proposed in AKI [40], ARPKI [9] and Certificate Transparency (CT) [45] in which all issued public-key certificates are publicly logged and validated by auditors. Public logs are also used in Keybase [39], which enables users to manage their online accounts and provides checking of name-to-key bindings by verifying ownership of third-party accounts. This is achieved via creating a public log of identity information that third-parties can audit. EthIKS [12] provides stronger auditability to CONIKS [51], an end-user key verification service based on a verifiable transparency log, by creating a Smart Ethereum Contract [75] that guarantees that a hash chain is not forked, as long as the ethereum system is stable and correct. Application Transparency (AT) [27] employs the idea of submitting information about mobile applications to a verifiable public log. Thus, users can verify that a provided app is publicly available to everyone or that a given version existed in the market, but was removed. However, AT can protect only against targeted attacks but leaves attacks against all the users outside of its scope. Finally, Baton [8] tries to address the problem of renewing signing keys in Android by chaining them but this solution does not help in the case of stolen signing keys.

Blockchains. The creation of Bitcoin [56] was first perceived as an evolution in the domain of financial technology. Recently, however, there has been an increasing interest in the data structure that enables the properties of bitcoin, namely, the blockchain. There is active work with blockchain in cryptocurrencies [13, 65], DNS alternatives [74] and even general-purpose decentralized computing [75]. All of these systems secure clients in a distributed manner and with a timeline that can be tracked by the clients. However, these systems force the clients to track the full timeline, even if the clients are interested in a very small subset of it, or to forfeit the security of decentralization by trusting a full node.

10 Conclusion

In this work, we have presented CHAINIAC, a novel software-update framework that decentralizes each step of the software-update process to increase trustworthiness and to eliminate single points of failure. The key novel components of CHAINIAC's design are multi-level skipchains and verified builds. The distinct layers of skipchains provide, while introducing minimal overhead for the client, multiple functionalities such as (1) tamper-evident and equivocation-resistant logging of the new updates and (2) the secure evolution of signing keys for both developers and the set of online witnesses. Verified builds further unburden clients by delegating the actual reproducible building process to a decentralized set of build verifiers. The evaluation of our prototype has demonstrated that the overhead of using CHAINIAC is acceptable, both for the clients and for the decentralized group of witnesses, by running experiments on real-world data from Debian. Furthermore, we have replayed 30 days of actual client requests to the PyPI repository and shown that the use of skipchains limits the verification overhead.

Acknowledgments

We thank Stevens Le Blond and our anonymous reviewers for valuable suggestions, and Gaspard Zoss for help with the experiments.

References

[1] Joe Abley, David Blacka, David Conrad, Richard Lamb, Matt Larson, Fredrik Ljunggren, David Knight, Tomofumi Okubo, and Jakob Schlyter. DNSSEC Root Zone – High Level Technical Architecture, June 2010. Version 1.4.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An Incentive-compatible Cryptocurrency Based on Permissionless Byzantine Consensus. *CoRR*, abs/1612.02916, 2016.

[3] Khalid Alhamed, Marius C. Silaghi, Ihsan Hussien, Ryan Stansifer, and Yi Yang. "Stacking the Deck" Attack on Software Updates: Solution by Distributed Recommendation of Testers. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT) 2013*, 2013.

[4] Khalid Alhamed, Marius C. Silaghi, Ihsan Hussien, and Yi Yang. Security by Decentralized Certifica-

tion of Automatic-Updates for Open Source Software controlled by Volunteers. In *Workshop on Decentralized Coordination*, 2013.

[5] Moreno Ambrosin, Christoph Busold, Mauro Conti, Ahmad-Reza Sadeghi, and Matthias Schunter. Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution over Untrusted Cache-enabled Networks. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Proceedings, Part I*, 2014.

[6] Hitesh Ballani, Paul Francis, and Xinyang Zhang. A Study of Prefix Hijacking and Interception in the Internet. In *ACM SIGCOMM Computer Communication Review*, volume 37. ACM, 2007.

[7] David Barrera, William Enck, and Paul C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *Mobile Security Technologies 2012*. IEEE, 2012.

[8] David Barrera, Daniel McCarney, Jeremy Clark, and Paul C. van Oorschot. Baton: Certificate Agility for Android's Decentralized Signing Infrastructure. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, New York, NY, USA, 2014. ACM.

[9] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *1st USENIX Workshop on Hot Topics in Security (HotSec)*, July 2006.

[11] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. Duqu: Analysis, Detection, and Lessons Learned. In *ACM European Workshop on System Security (EuroSec)*, 2012.

[12] Joseph Bonneau. EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log. In *Financial Cryptography and Data Security 2016*. Springer Berlin Heidelberg, 2016.

[13] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, and Bryan Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *1st IEEE Security and Privacy On The Blockchain*, April 2017.

[14] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 4880: OpenPGP Message Format, 2007.

[15] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. A Look In the Mirror: Attacks on Package Managers. In *15th ACM Conference on Computer and Communications Security (CCS)*, October 2008.

[16] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[17] Python Community. PyPI - the Python Package Index, September 2016.

[18] Python Community. EasyInstall Module, May 2016.

[19] Debian. snapshot.debian.org.

[20] Debian. Debian Popularity Contest, September 2016.

[21] Debian. Reproducible Builds for Required Packages (AMD64), September 2016.

[22] Debian. Reproducible Builds for Testing Packages (AMD64), September 2016.

[23] Debian. Advanced Package Tool, May 2016.

[24] Debian. Debian Repository, August 2016.

[25] Stephen E. Deering and David R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[26] Docker. What is Docker?, September 2016.

[27] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew 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*. ACM, 2014.

[28] Bryan Ford. Apple, FBI, and Software Transparency. *Freedom to Tinker*, March 2016.

[29] Frields, Paul W. Infrastructure Report, August 2008.

[30] Git-notes Documentation, September 2016.

[31] The Go Programming Language, September 2016.

[32] Red Hat. Critical: OpenSSH Security Update, August 2008.

[33] Max Howell. Homebrew – The Missing Packet Manager for macOS, May 2016.

[34] Google Inc. Google Play, September 2016.

[35] Microsoft Inc. Windows Apps - Microsoft Store, September 2016.

[36] The Intercept. Strawhorse: Attacking the macOS and iOS Software Development Kit, March 2015.

[37] Chris Jarabek, David Barrera, and John Aycock. ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, New York, NY, USA, 2012. ACM.

[38] Håvard Johansen, Dag Johansen, and Robbert van Renesse. FirePatch: Secure and Time-Critical Dissemination of Software Patches. In Hein Venter, Mariki Eloff, Les Labuschagne, Jan Eloff, and Rossouw von Solms, editors, *New Approaches for Security, Privacy and Trust in Complex Environments: Proceedings of the IFIP TC-11 22nd International Information Security Conference (SEC 2007)*, may 2007.

[39] Keybase – Public Key Crypto for Everyone, Publicly Auditable Proofs of Identity, 2016.

[40] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *International Word Wide Web Conference (WWW)*, 2014.

[41] Eleftherios Kokoris-Kogias, Linus Gasser, Ismail Khoffi, Philipp Jovanovic, Nicolas Gailly, and Bryan Ford. Managing Identities Using Blockchains and CoSi. Technical report, 9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016), 2016.

[42] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[43] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger. Cryptology ePrint Archive, Report 2017/406, 2017.

[44] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2016.

[45] Ben Laurie. Certificate Transparency. *ACM Queue*, 12(8), September 2014.

[46] E. Levy. Poisoning the software supply chain. *IEEE Security Privacy*, 1(3):70–73, may 2003.

[47] Jun Li, P. L. Reiher, and G. J. Popek. Resilient Self-organizing Overlay Networks for Security Update Delivery. *IEEE J.Sel. A. Commun.*, 22(1), September 2006.

[48] Canonical Ltd. Ubuntu Repositories, September 2016.

[49] Lunar. How to make your software build reproducibly. Chaos Communication Camp 2015, August 2015.

[50] Aanchal Malhotra, Isaac E. Cohen, Erik Brakke, and Sharon Goldberg. Attacking the Network Time Protocol. Cryptology ePrint Archive, Report 2015/1020, October 2015.

[51] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing Key Transparency to End Users. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 383–398. USENIX Association, 2015.

[52] Ralph C Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology (CRYPTO)*, 1988.

[53] Michael Mimoso. D-Link Accidentally Leaks Private Code-Signing Keys. *ThreatPost*, September 2015.

[54] Mininet – An Instant Virtual Network on your Laptop (or other PC), September 2016.

[55] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic Skip Lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[56] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[57] Null Byte. Hack Like a Pro: How to Hijack Software Updates to Install a Rootkit for Backdoor Access. WonderHowTo, 2014.

[58] E. Palomar, J.M. Estevez-Tapiador, J.C. Hernandez-Castro, and A. Ribagorda. A Protocol for Secure Content Distribution in Pure P2P Networks. In *Proceeding of the 17th International Conference on Database and Expert Systems Applications*, 2006.

[59] Mike Perry, Seth Schoen, and Hans Steiner. Reproducible Builds. Moving Beyond Single Points of Failure for Software Distribution. Chaos Communication Congress 2014, December 2014.

[60] Tor Project. Building the Tor Browser Bundle (TBB) Using the Gitian Build System, May 2015.

[61] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[62] Reproducible Builds – Provide a Verifiable Path From Source Code to Binary, 2016.

[63] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable Key Compromise in Software Update Systems. In *17th ACM Conference on Computer and Communications security (CCS)*, October 2010.

[64] Claus P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[65] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, page 5, 2014.

[66] Christopher Soghoian and Sid Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, FC'11. Springer-Verlag, 2012.

[67] Sooel Son and Vitaly Shmatikov. The Hitchhikers Guide to DNS Cache Poisoning. In *International Conference on Security and Privacy in Communication Systems*, pages 466–483. Springer, 2010.

[68] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable Bias-Resistant Distributed Randomness. In *38th IEEE Symposium on Security and Privacy*, May 2017.

[69] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *37th IEEE Symposium on Security and Privacy*, May 2016.

[70] The FreeBSD Project. Security Incident on FreeBSD Infrastructure, November 2012.

[71] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, August 1984.

[72] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities. In *25th USENIX Security Symposium (USENIX Security 16)*, August 2016.

[73] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In *14th International Conference on Network Protocols (ICNP)*, November 2006.

[74] Vincent Durham. Namecoin, 2011.

[75] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 2014.

[76] YUM. Yellowdog Updater Modified, May 2016.

[77] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Hey, You, Get Off of My Market: Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12. ACM, 2012.