ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Enhancing Debian Update Service

Gaspard Zoss

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

January 2017

| **Responsible** | **Supervisor** | **Supervisor** |
|---|---|---|
| Prof. Bryan Ford | Eleftherios Kokoris Kogias | Kirill Nikitin |
| EPFL / DEDIS | EPFL / DEDIS | EPFL / DEDIS |

# Contents

# Chapter 1

# Introduction

Software updates are an essential element in securing any software running on devices going from small embedded devices to computer clusters. In this project, we will study how security of software update process in large projects can be improved and apply it to Debian APT [1].

When a new feature is added or when a bug is fixed in one of the software available through the Debian package manager, the package's maintainer, often one of the project's developers, generates a binary by compiling the code, hashes it and creates various scripts used to install, upgrade or remove the software. These scripts and the binary are then packed together and placed inside a repository, from which the end user may download the update. The repository itself is signed, usually using a single private cryptographic key and the binaries are verified by using checksums. Additionally, the software can be made reproducible by the developers allowing users to verify if the given binary was produced using the publicly available source code. But as of today, not all Debian packages are reproducible [5].

The described system has multiple attacks vectors; first, if reproducible builds aren't used, a malicious developer could distribute a binary containing some spyware or hidden functionality not present in the public code repository without informing the user and sign the binary with his private key. Then, the end-user's only choice would be to trust the package maintainer and the only certitude he has is that the binary he received is the one he was intended to receive. Secondly, even if the package maintainer is assumed to be trusted, a powerful third party opponent could steal the maintainer's private key and sign a modified binary which he could then substitute without noticing the end-user. The single signing key is thus a single point of failure in this architecture.

On the other hand, even if reproducible builds are available for a given package not all code details are monitored and each end-user should review every parts of the code to have the absolute certainty that the binary is safe. Furthermore, reproducing a package may take a lot of time, for example it takes on average 32 hours to build Tor on a modern laptop [9], and most end-users don't want to spend time on reproducing packages.

We propose a new way of verifying Debian update by introducing a wrapper around *APT* that verifies the validity of the updates against a public cothority service that keeps an updated *log* of the Debian updates on a per-repository basis.

First, we will present the background in a first part. Then we will present the main ideas of a Collectively Signed APT and describe the Debian release system in the second part. In the third part, we will discuss our implementation and, in the fourth part, how we evaluated the results. Finally we will discuss the future work and formulate a conclusion.

## 1.1 Background

Aside of introducing CHAINIAC, a decentralized software-update framework [10], authors also propose to extend it to large projects. They built CHAINIAC on top of CoSI, a scalable witness cosigning protocol [12]. They introduce the concept of *skipchains*, a data structure built by combining ideas from blockains and skiplists. Such *skipchains* are useful as a way of maintaining a traversable public log of all updates for a given software.

A *skipchain* enables the client to traverse the timeline in both backward and forward direction, by, in the case of backward links, including a cryptographic hash of the previous block to the current one (similar as the blockchain case) and, in the case of forward links, a cryptographic signature of future blocks is retroactively appended to the previous block. Additionally, *skipchains*' blocks have links to several other blocks to enable a faster traversal. The height of the *skipchain* determines the number of links and how far the backward and forward links are going.

These *skipchains* constitute a public log and they are maintained by a COTHORITY, a decentralized collective authority. This COTHORITY consists of an arbitrary number of hosts that exchanges informations via various protocols and runs custom services on each hosts.

# Chapter 2

# Collectively Signed APT

## 2.1 The Debian release system

As Debian is under continual development, the repositories organization is carefully crafted in a way that enables every user to get updates as frequently as he desires. For example, the administrator of a server used in production may want to update the packages only when the update is really necessary as it fixes a major security vulnerability or when the update has been successfully tested and is safe to deploy on a production server. On the opposite, a developer may want to receive the latest updates as soon as they are available, even if a few bugs are still present.

To tackle with these very different requirements, the Debian community has separated the repositories into various categories; The *stable* release with major version approximately every two years and minor versions, called Point Release, every few months. On top of that, *stable-update* gives early access to updates which will be added to the next Point Release for some softwares that have a much smaller release cycle, such as *clamav* (an antivirus software) or the Linux kernel. Security-critical updates go through a different repository, *stable/updates* on *security.debian.org*, which enables the updates to be pushed to the user on the same day the update is released. This security-critical updates are then merged in the *stable* release. The *testing* release contains the latest version of the packages which is stable enough (i.e. the packages have to pass different automatic requirements). Additionally, the *stable-proposed-updates* repository contains packages which are proposed for the next *stable* release but are not fully tested yet. Figure 2.1 shows the overall Debian repository structure.

In this work, we will focus on the *stable*, the *stable-updates*, the *testing*

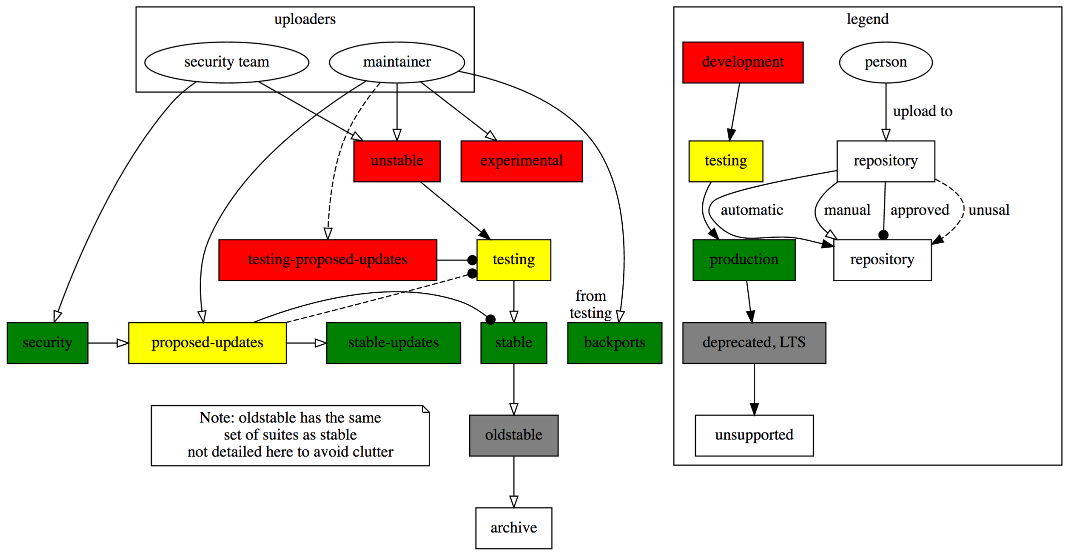

Figure 2.1: Debian Repository Workflows [11]

and the *security* repositories, as these are the default way to receive updates on a standard installation of Debian. Adapting the project to other repositories will be discussed at the end of this document.

### 2.1.1 Update frequency

To ensure the feasibility of creating a skipchain per Debian repository we extracted statistics from the Debian Snapshot project [2]. Table 2.1 summaries the update frequencies of packages from the various Debian repositories (only for *binary-amd64*).

| repository | approximative update frequency |
|---|---|
| *stable* (Point Release) | $2 - 3$ months |
| *stable-update* | 1 month |
| *security* | 2 days |
| *testing* | less than 1 day |

Table 2.1: Update frequencies for some of the Debian repositories

We also looked at the number of packages updated per repository release, which are summarized in Table 2.2

| repository | Average number of packages per release |
|---|---|
| *stable* (Point Release) | 70 |
| *stable-update* | 2 |
| *security* | 3 |
| *testing* | 100 |

Table 2.2: Average number of packages per release

## 2.2 Collectively Signed Debian Update Service

### 2.2.1 Extending CHAINIAC

Chainiac [10] is a decentralized software-update framework working closely with Cothorities [12]. Authors proposed a way to extend Chainiac to large projects. We study here the feasibility and efficiency of the proposed ideas, and design their implementation into Debian. Using this method could also enable a smooth transition from the actual state of Debian updates to a fully verified and reproducible update-framework.

Our implementation is composed of three distinct parts; a server part, running on a Cothority, responsible for maintaining the Debian skipchains, another server part used by the repository maintainers to track changes in the repository and to update corresponding skipchain and a client which is a modified version of $APT$ to verify updates before installing them.

### 2.2.2 The Cothority and server parts

The server part runs as a Cothority service. For each Debian update repository configured to do so, it *maintains* a Skipchain where each block is the root of a *Merkle tree.* This Merkle tree is formed by having the head of an individual package's skipchain as leaf (see Figure 2.2 for the overall scheme). Additionally, some packages can be added to the repository Merkle tree even if they do not possess an individual Skipchain (for example during a transition period or if having a Skipchain for this package is not relevant). For those packages, the root of the Merkle-tree formed by its files hashes is added to the Repository Merkle tree instead of the latest Skipblock. This service is accessed by both the server and the client through an API.

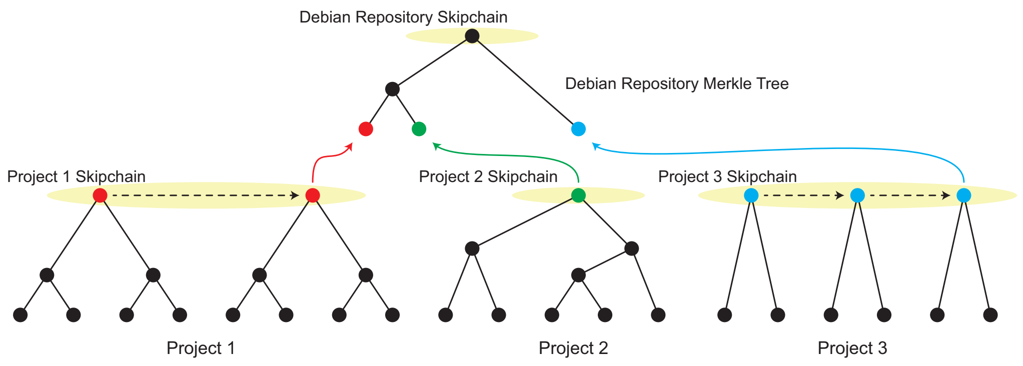

Figure 2.2: The big picture of a Skipchain for some Debian Repository and 3 fictional packages

The server part which runs on the Debian repository maintainer's server is triggered multiple times per day by a cron job. Its main role is to request the latest's skipchain block from each individual package's software update service. After receiving all the replies, it aggregates all the data into a single message sent to the Debian Update Cothority service using an API. The later then creates a Merkle-tree from this aggregated answer by having packages' hashes as leaves in the tree. If the new root of this tree differs from the previous ones (which implies that some of the packages have changed), it is collectively signed and added to the Debian Update Skipchain. To ensure the information's freshness in the Debian Update Skipchain, CHAINIAC's role-based trust delegation model is applied to the Debian Update project. In the case where a package doesn't have an individual Skipchain, the server part can also be configured to include some packages' hash in the aggregated answer by building a Merkle-tree of its files and including the root of this Merkle-tree in the Debian Update Skipchain.

### 2.2.3 The user client part

The client part is a modified version of the *Advanced Packaging Tool* which verifies the downloaded update against a public Update Cothority. When the user wants to update its distribution, the *APT* wrapper performs the following tasks :

1. The wrapper calls *apt-get update* to download the latest package lists and their headers.
2. It then requests the latest signed Skipblock from the Cothority's release skipchain.

3. After having downloaded the Skipblock, the wrapper verifies the signature of the block.
4. It then verifies the packages needed to be updated by computing the Merkle proof verification for each of them using the informations it received from the Cothority.
5. The wrapper then uses *apt* to download the latest version of the packages that requires an upgrade but do not install them yet.
6. For each of the outdated packages installed, the client then goes to individual package's Skipchains and verify them individually using the method described in CHAINIAC.
7. On success, the client can be sure that the given package is the correct one and install it.

# Chapter 3

# Implementation

We implemented the clients and the server in Go [3]. The following subsections describe their implementations and the strategical choices undertaken. Figure 3.1 shows the general architecture described in the following sections.

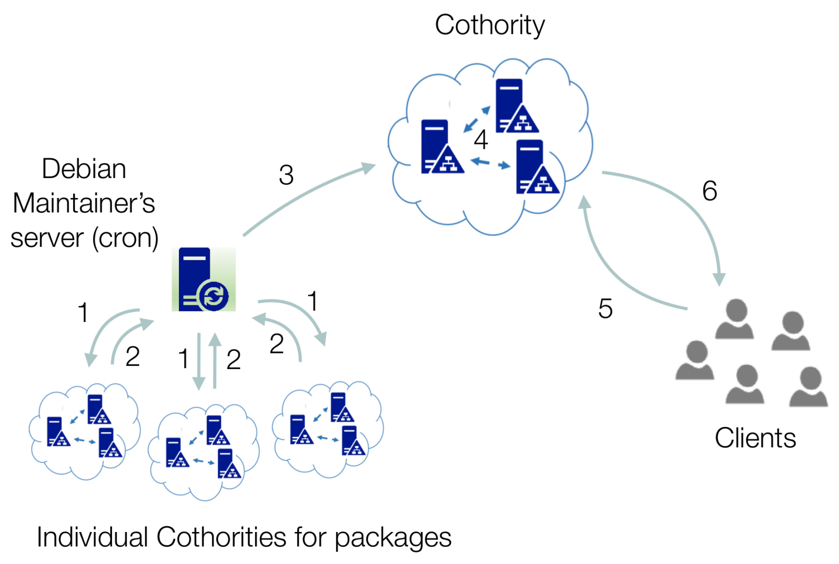
Cothority
4
Debian Maintainer's server (cron)
3
6
1
1
5
2
1
2
2
Clients
Individual Cothorities for packages

Figure 3.1: General architecture of the project. (1) are the latest block requests from the Maintainer's server to the individual Cothorities, (2) are the answers to those requests. (3) is the aggregated answer sent to the Debian Repository Cothority. (5) and (6) are the request and answer from the client to the Cothority.

## 3.1 The Cothority Service

The Debian Update Cothority Service part follows closely the existing implementation of the Software Update Cothority Service with a few differences. Instead of having a service that maintains Skipchains on a per-package basis, our implementation now has a service that maintains Skipchains on a per-repository basis. To simulate the aggregated message received by the service, we decided to use Debian Snapshots [8] as an aggregated message to construct the Debian Repository Merkle-tree.

The first step for the Debian repository maintainer's server is to ask the service to create a new Skipchain for a given Snapshot. This is done by providing the service with the initial *Release* file, which describes the repository name, version and various parameters and the initial *Packages* file, which describes all the packages available at that time in the repository, their hashes, versions and other informations. To simplify the use of the service, we have chosen that both of those files are exactly the files downloaded from the Snapshot project without requiring any modifications. Alongside, other parameters are passed to the Cothority service, such as the skipchain parameters and the precomputed Merkle tree root and proof. The service then verifies this initial state by verifying the root of the Merkle tree against the proof. On successful verification, the service sends back the latest block of the Skipchain. Figure 3.2 shows a simplified version of the request and answer to the service.

```
CreateRepository  {
   Repository
   RootID
   Proofs
   Base
   Height
}
```

```
CreateRepositoryReturn {
   LatestBlock
}
```

Figure 3.2: The informations sent and received on creation of a new Skipchain

When a new Debian Snapshot is produced, the Skipchain has to be updated with these new informations. When receiving new datas from the Debian maintainer's server, which consist of a repository new snapshot, the service first checks if some packages where updated or if the snapshot is the same as the previous one. Depending on the repository, the later case can happen often, for example the stable repository only had 13 changes during the past year. The service then updates the Timestamp Skipchain, even if the new Snapshot does not contain changes since the previous one, and updates the Data Skipchain only if the Snapshot is different than the previous one. The service then sends back the latest block of the Skipchain. Figure 3.3 shows a simplified version of the request and answer to the service.

```
UpdateRepository  {
   LatestBlock
   Repository
   RootID
   Proofs
}
```

```
UpdateRepositoryRet {
   LatestBlock
}
```

Figure 3.3: The informations sent and received on update of the Skipchain

## 3.2 The server part

Our prototype implementation of the Debian Repository maintainer's server part consists on a script which downloads the Debian Snapshots [8] (both the Package and Release files) given the name of the repo, a starting date and an ending date. Simulations then use these files and a Cothority's service API to create and update the Skipchains for each repository. These two files are parsed by the service and the relevant informations are stored in the Skipchain. Figure 3.4 shows the datas extracted from the *Package* and *Release* files.

```
Repository {
   Origin
   Suite
   Version
   []Packages
   SourceUrl
}
```

```
Packages {
   Name
   Version
   Hash
}
```

Figure 3.4: Datas extracted from the *Release* and *Package* files

## 3.3 The client part

While having a modified version of *APT* would be a production ready software, we decided to implement a wrapper for *APT* in Go. Doing so allows us to avoid re-implementing most of the networking and CoSi functions already available. Our wrapper handles the extra communication with the Cothority and the verification of the repository's state validity. It then performs calls to the installed *APT* version. We took as an example the idea of *yaourt* [7], a *pacman* [6] front end used by *Arch Linux* users which adds the *Arch User Repository*'s packages to *pacman*. To simplify our simulation, this client was abstracted using only the API calls instead of launching the full client every time. When a client makes a request to the Cothority through the API, the service sends the latest Skipblock containing the signed root and a list of all the packages in the repository containing the package's hash and proofs. It then verifies the root's signature and can check the proof for all the installed packages.

# Chapter 4

# Results and Evaluation

To evaluate our proposed solution, we used the simulation framework included in the *Cothority* codebase. This framework allowed us to run the simulations on multiple platforms using the same code. We implemented various simulations to cover most of the test cases and thus extract various metrics. The following sections show the different simulations we ran using the *localhost* simulation (as the simulations required only a limited CPU and memory power, we did not need to run them on DetherLab or Mininet).

## 4.1 Cothority: Adding new release

To test the CPU load on the Cothority depending on the number of packages in the repository added to the Skipchain, we implemented a simulation where a Cothority with 4 nodes is created, and a new release for a repository containing a certain number of packages was added to the Skipchain. Figure 4.1 shows the average CPU time in second against the number of packages in the repository. Three metrics were individually measured; the time it took to verify the individual packages signatures (coming from each individual Cothority's Skipchains), in our test it was one single signature verification times the number of packages, the time it took to sign the root of the produced Merkle tree and the time it took to verify all the proofs from the individual packages. It is interesting to read on the graph that signing the root is nearly constant, as excepted and that both the proofs and signature verifications are linearly dependent on the number of packages.

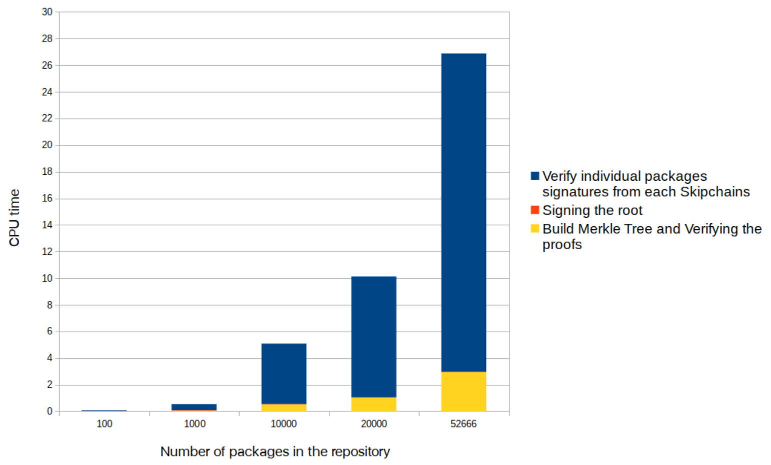

Figure 4.1: The CPU time usage in second against the number of packages in the repository.

## 4.2 Client: CPU usage

In order to measure the additional load on the clients, we created a simulation where a client requests and verifies the latest block of the Skipchain and measured the average CPU time in seconds it took to verify the proofs and to verify the signatures. We included in the graph the time it took to verify a single GPG signature and the hashes for all those packages in $APT$ as a baseline. Figure 4.2 shows the average CPU usage in second against the number of packages installed on the client machine. Again, this simulation was run with a Cothority of 4 nodes.

Computing the proofs on the client also depends on the number of packages in the repository in the Skipchain. We used subsets of the *testing* repository with 1, 10, 100, 1000, 10000 and 52666 (the entire *testing* repository) packages and measured the time it took to compute the proofs and to verify the signature. Figure 4.3 shows those metrics. Again, as a baseline we used the time it took for APT to run. This simulation was run with a 4 node Cothority.

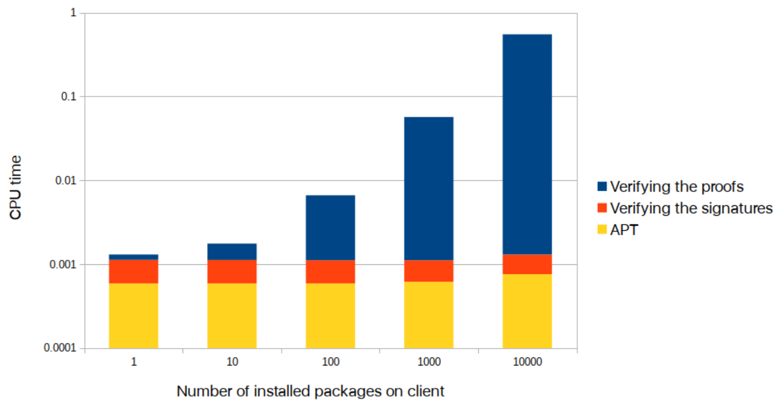

Figure 4.2: The average CPU time (in second) usage against the number of installed packages on a client.

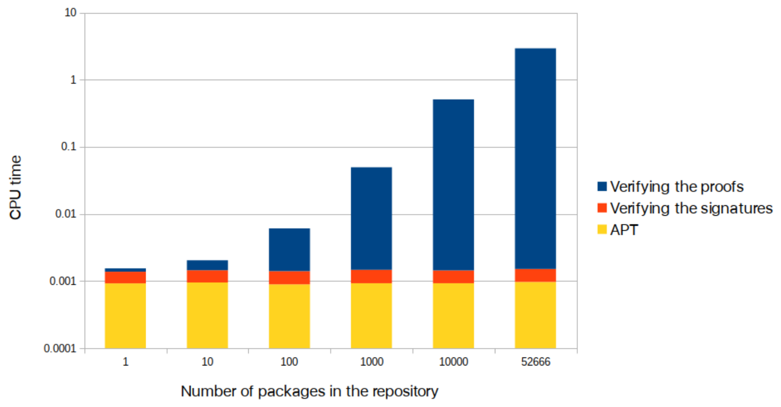

Figure 4.3: The average CPU time (in second) usage against the number of packages in the repository.

## 4.3 Client: Bandwidth usage

Using the same setting to as in the later case to also measure the bandwidth overhead on the client compared to plain APT. This result is shown on figure 4.4.

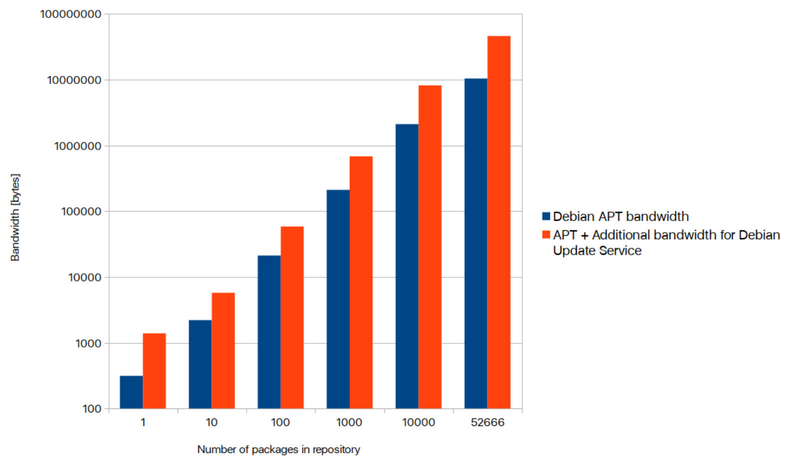

Figure 4.4: The average bandwidth (in bytes) usage against the number of packages in the repository.

## 4.4 Results and limitations

As we saw in the previous sections, the overhead CPU cost on the client depends on both the number of installed packages and number of packages in the Skipchain, as for each installed packages the client must verify the Merkle Proof. This verification depends on the size of the Merkle Tree and has complexity that is proportional to the logarithm of the number of leaves in the tree [4]. We also saw that the bandwidth overhead for the clients doubles compared to the bandwidth used only to download the latest release and packages informations by *APT*.

Also, we saw that in order to setup a Cothority service for the repository they maintain, the administrators of this repository must have powerful enough servers to run it. Hopefully the requirements are proportional to the quantity of packages in the repository and they are only a small overhead compared to the infrastructure required to run a Debian Repository.

# Chapter 5

# Conclusion and Future Work

In this report, we presented a solution to various issues present in the Debian package update infrastructure such as the single cryptographic signing key used to signed the releases of a repository and described the prototype implementation of a new framework we built, solving these issues using a *Cothority* service based on *Skipchains* and the *Software Update Service* presented in previous work [10]. As we saw in the section 4.4 Results and limitations, the lack of reproducible builds for the current Debian release limits the possibility of using this new framework in a production environment but we hope that reproducible builds will be a standard for Linux packages in the following years.

We built our framework in a way that adding a new repository to the *Debian Update Skipchain* is automatically handled by the framework and thus, using it on other repository such as *stable-proposed-update* or *unstable* is already possible.

A possible future work on this project would be to separate the message sent by the cothority service to the client containing both the signed root and the packages proofs, by first requesting only the root and the proofs for the installed packages instead of requesting for all the proofs. This could lower significantly the bandwidth used by the clients.

An extension of this project could be to port the framework to handle other distribution such as *Arch Linux* or *OS X's homebrew* but, due to the lack of reproducible builds usage, for which the Debian project is a leader on, greatly limits the inpact of using our framework.

Another extension of our framework could be the implementation of the client part directly inside the *APT* codebase instead of having a wrapper.

# Bibliography

[1] Apt - Debian Wiki. https://wiki.debian.org/Apt.

[2] Debian Snapshot project. http://snapshot.debian.org/.

[3] The Go Programming Language. https://golang.org/.

[4] Merkle tree - Wikipedia. https://en.wikipedia.org/wiki/Merkle_tree.

[5] Overview of various statistics about reproducible builds. https://tests.reproducible-builds.org/debian/reproducible.html.

[6] Pacman - ArchWiki. https://wiki.archlinux.org/index.php/Pacman.

[7] Piec/yaourt. https://github.com/piec/yaourt.

[8] Snapshot.debian.org. http://snapshot.debian.org/.

[9] Tor Bug Tracker & Wiki. https://trac.torproject.org/projects/tor/wiki/doc/TorBrowser/BuildingWithGitian#BuildTimes.

[10] Back to the Future: Transparent Software-Updates with Collectively Signed Skipchains and Build Verification. 2016-2017.

[11] Debian. DebianReleases - Debian Wiki. https://wiki.debian.org/DebianReleases?action=show#Workflow.

[12] 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. 2015.

# Chapter 6

# Guide for installation and running

To install and run the final version of the code, one should use the following instructions

```
git clone git@github.com:gaspardzoss/cothority-private.git
cd cothority-private
git checkout nsdi_branch
go get -u -t ./...
# To run the simulations
cd simul/
go run simul.go runfiles/debianupdate_1_client.toml
```