Simon Hollingshead

Mitigating the Effects of Software Repository Key Compromise

Part II Computer Science Tripos

Queens' College

2013/2014

Proforma

Name:	**Simon Hollingshead**
College:	**Queens' College**
Project Title:	**Mitigating the Effects of Software Repository Key Compromise**
Examination:	**Part II Computer Science Tripos 2013/14**
Word Count:	**11885**
Project Originator:	Simon Hollingshead
Supervisor:	Daniel Thomas

Original Aims of the Project

To create an alternative package manager that does not suffer from well-documented protocol-level flaws in existing solutions, made such that multiple package formats can be distributed using it. It must also be a practical replacement, with minimal overheads and strong foundations on trusted cryptographic libraries.

Work Completed

Generic Package Manager imports multiple package types, currently supporting Debian and Arch Linux. These packages are transformed into a generic metadata format, made available for signing, signature checked, then stored. On the client, the metadata is downloaded, verified, used for dependency resolution, then the original packages are fetched and passed to an installation API.

Special Difficulties

I did not have any special difficulties during this project.

Declaration

I, Simon Hollingshead of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

My project was to create a new package manager less prone to man-in-the-middle tampering than existing package managers have been documented to be. Rather than specialising this to a given package format, my intention was to produce a system operating on an abstract representation for packages. A distribution would be able to specify a transformation into the internal representation without modification to their package layout, then distribute this new package manager as a replacement for their existing executable.

The result, Generic Package Manager (GPM), uses industry-standard asymmetric public key cryptography to sign packages of multiple filetypes entering the system. A chain of trust is built through multiple signatures at multiple stages in the workflow, allowing far more reliable identification of when a connection has been tampered with than many conventional package managers. In the case of key compromise, GPM is at least as resilient as every other package manager tested, in most cases being able to restrict the damage to a far lesser class of attack. For example, at least three separate packagers' keys have to be compromised to successfully install an arbitrary package on a user's machine. The method of multiple key signing only slightly augments the existing workflow of distributions' packagers, using the second and third signatures similar to quality assurance on the package's content.

To make the program a realistic option for distributions, I have made every effort to make the overhead of the added security as small as possible. The package manager not only readily presents signers with a collection of files that are in need of signing, reducing the difficulty of identifying what is due to happen next, but the batch nature in which packages can be signed and imported into the repository reduces some of the more significant overheads through amortisation. On large scale imports, Debian packages import in an average of 1.28 seconds each, the far simpler Arch Linux format taking only 14.7 milliseconds each.

1.1 Principal Motivation

Linux distributions generally retrieve their applications from a software repository run by the distribution. Source code or compiled applications are produced by a party the distribution trusts, placed into archives called 'packages' with additional metadata[1], then uploaded into repositories. These repositories are standard web or FTP servers containing these files for download, copies of which are mirrored by volunteers with spare bandwidth as an act of generosity towards the distribution. Package manager clients come pre-installed on Linux distributions to allow the fetching and installation of software from a mirror.

Communication between a client and a repository passes through three distinct stages:

1. A small (< 200 KB) file listing is fetched from the server to quickly identify if metadata caches on the client are still valid.

2. If the cache is outdated, a large (~ 10 MB) metadata file is fetched, detailing dependency information about all available repository packages. This allows the package manager to identify any additional that will need to be downloaded to satisfy an installation.

3. Finally, the relevant package files are downloaded and installed.

From 2008, researchers have investigated the security of the protocols used between clients and their repositories, and have recognised three main severities of attack [9], here listed from most severe to least:

Package modification The package manager does not detect if the package has been altered or entirely replaced, compared to the original created by the distribution. This allows an attacker to both place arbitrary files anywhere on a victim's system and to write their own post-installation shell script, which will automatically run as the root user.

Metadata modification By altering dependency data, the resolution phase can be coerced to fail to schedule a required install, leaving a broken package, or install too many packages, increasing exposure to vulnerabilities. Alternatively, the download location can be altered to point to an old yet legitimate package, therefore not caught by any package modification defences.

[1] Other packages may need to be marked as either required to be installed for functionality, or marked as incompatible and conflicting.

Freeze attack By repeatedly returning the same file list on every update request, a package manager can be coerced into believing there are no updates. Over weeks or months, vulnerabilities for these now-old packages may surface and become exploitable.

Stork package manager [8] researchers also mention two further attack vectors [6, 7]. By initially being a legitimate mirror, serving malicious packages is far easier than being a man in the middle. This is fixed by package signing. The second, the endless data attack, overwhelms a machine with a package too large for their hard drive. I describe how GPM's implementation fixes it in Section 3.2.5.

It was not until February 2011 that I became aware of package manager insecurity, when blog posts [13] and subsequent news articles [39] reported that Arch Linux's lack of package signing meant it fell vulnerable to all three attacks. Although the package modification problem was fixed in December 2011 [26], it demonstrably remains vulnerable to both other attack classes as of April 2014, as I have performed those exploits in Chapter 2.5.

Other package managers have also been slowly converging on more secure protocols, incrementally adding signatures to each phase of the communication. However, since no package manager is yet 'fully-secure' by these standards, a man-in-the-middle is still able to identify the repository, infer the package manager, then leverage the most devastating attack to which it is vulnerable. The number of people and important machines left without a solution led to my intention to resolve it for my project. By putting effort into repairing it once in a reusable way, rather than allowing each package manager to spend time only on their own fixes, I believe the work will be more useful to the community as a whole.

1.2 Terminology

To ensure clarity for the rest of this dissertation, an issue does exist with the term 'package manager' being used by two classes of application. Firstly, tools like `dpkg` and `rpm`. These are programs that are aware of the full package specification, install and delete packages by manipulating the filesystem, and provide APIs to allow third-party applications to request these same actions. The second is the class into which `apt` and `yum` fall, using the former group's APIs to extend functionality with features such as Graphical User Interfaces, web downloading, and partial upgrades. If the package format were to change, only the former group would need to be altered to handle this, since the exposed API would remain the same, but now perform the new actions.

During the rest of this dissertation, when GPM is being referred to as a 'package manager,' it falls into the latter class, since it has not been designed to know the full specification of any package type, and has been built to use provided APIs extensively so as to focus more on implementing parts that have not been created in this way before.

Chapter 2

Preparation

This section details the work undertaken to fully understand how to implement the project, including algorithm research, security research, exploit creation, and general pre-development design and planning.

2.1 Algorithms and Protocols

The initial insights into the correct protocol to use came from the previously mentioned 2008 paper by Cappos [9] and a second he co-authored two years later [33]. Between these, a relatively complete set of foundations is formed, from exploits to their mitigation. In particular, they discuss the design decisions, benefits and weaknesses of each of three attempts.

The still semi-vulnerable[1] Stork package manager is a first foray into producing a rebuilt protocol to take advantage of added communication security. This includes use of keys on stages, and introduction of a safe order in which to request files from the repository to build up a chain of trust.

The additions made to the concepts behind Stork are given in the specifications of Thandy [24], the updater for the Tor executable, highly specialised yet necessarily designed to be secure, and the TUF protocol, an attempt to serve more arbitrary packages using the mechanisms provided by Thandy. TUF and its implementation of Python `easy_install` package management demonstrated a system close to my intentions for GPM, although in a non-generic form.

By reading the academic papers, I was able to gain an appreciation of the mechanics behind the security in the communication. Firstly, the use of public key cryptography [16, 17] to trust a file, rather than a server. This allows any person to mirror without having to trust them; installation will fail if the package

[1] Stork does not sign its file list, allowing package freezing.

and signature no longer match. Secondly, understanding of the user's 'compliance budget' [36] by including an amount of keys that is sensible within a standard import workflow, reducing the likelihood of users or packagers attempting to circumvent the technology. In addition, having non-expiring package signatures avoids the annoyance of old-but-current packages in the repositories needing their signatures periodically refreshed to ensure they don't expire unnoticed and cause uninstallable periods[2], but allows earlier packages to masquerade as 'current' under metadata replacement.

The repository itself functions like a database, a store that is expected to provide consistency and transactional uploads [11], often including the ACID properties [38], Atomicity, Consistency, Isolation and Durability. Transactions stop a package's uploads appearing before its dependencies also appear, or there will be no possible way to install it.

Finally, investigation was made into implementing the protocol's messages, specifically how to keep data structures valid after an update without a mass-reimport. The Further Java [32] course demonstrates class serialization to produce messages between repository and client, and version numbering to retain the validity of old messages, necessary to support old package managers attempting to update themselves.

2.2 Dependency Resolution

After investigating the protocols, I realised that the most recent proofs of concept, namely Thandy and TUF, did not use nor expose dependency resolution features. Dependency resolution determines which other packages are required in order to install or upgrade a given package. For this reason, I identified algorithms used to schedule a partial order of dependencies. Multiple algorithms exist for this, all variants on identifying nodes with no predecessors and emitting them first. Although most are recursive [5, 19, 29, 35], Knuth's [15] algorithm is far simpler, using only an array of linked lists and a priority queue, shown in Figure 2.1. If x must come before y ($x \prec y$), y is contained in the linked list in element x of the array. The queue holds each element with priority 'how many predecessors?'[3]. By repeatedly removing the minimum from the queue, necessarily zero[4], leaves are found and scheduled. Each element of the removed node's linked list are then decremented in the queue to signify a handled predecessor.

[2]This is similar to administrators who forget to renew their SSL certificate and cause all users warnings and errors for hours.

[3]The number of times the element appears in any linked list.

[4]If all nodes have a predecessor, there is a cycle.

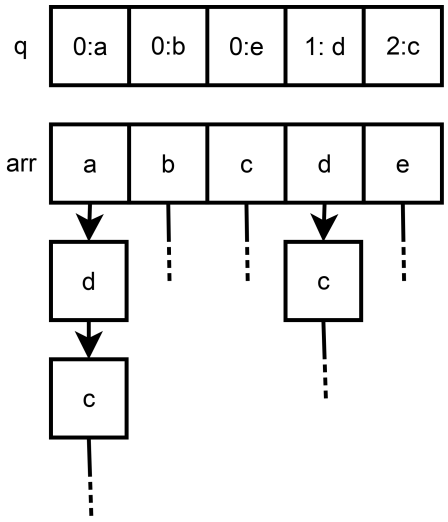

Figure 2.1: Algorithm T storing $a \prec d$, $a \prec c$ and $d \prec c$.

Package managers, however, have extra considerations. For efficiency, the claim that "if a package is installed, all of its dependencies are assumed to be installed" is made. This is based on the assumption that the package manager would have handled those dependencies during the prior install. Another alteration is the inclusion of a second class of edge, for deletions, where any conflicting packages that are installed must be scheduled for removal before the related install occurs.

2.3 Package Formats

A general package is an archive containing a copy of the OS directory structure with the files for the given package, plus extra metadata control files. An example of this can be seen in Figure 2.2. Since the package manager would need to extract multiple package formats, I began by referencing the format specifications for .deb [14], .rpm [2], .ebuild [25] and .pkg.tar.xz [27] to find the common properties in each, and the location in the package in which the relevant metadata to extract was held. I was able to identify features that all formats had (name, version, depends, breaks, provides), and some fields only provided in some distributions (epoch[5], suffix[6]).

[5]The epoch is prepended to the version number such that a lesser version number can still be considered an upgrade: If v16 is followed by v0.17, e1.0.17>e0.16.0.

[6]Some packages may be marked as 'beta' or 'rc' and only considered by some users who accept unstable software.

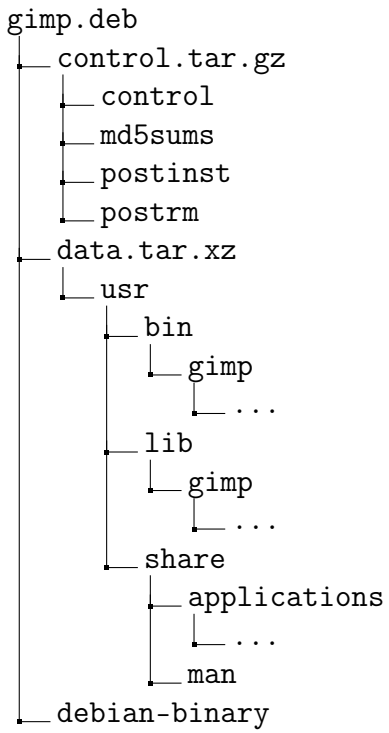

Figure 2.2: The directory structure of a Debian package.

Initially, I believed that most computation would be generic, with a single GPM implementation. However, on comparing specifications, I recognised that version numbers could not be compared generically. As an example, Debian's method (Appendix A) splits the string on each boundary between integer and non-integer substrings, whereas Gentoo's (Appendix B) considers '.' the only boundary character. The comparison of '7r1.6' and '72.0' on Debian would produce '7|r|1|.|6' and '72|.|0'. Since $7 < 72$, the latter string is greater. On Gentoo, the splitting produces '7r1|6' and '72|0' where, since $7r1 > 72$ using standard string comparison, the former is greater. As there is no way around this, each package type must specify its own version comparison function.

2.4 Security Investigation and Threat Model

Next, I investigated implementations of key generation and signing, a crucial part of the system. In September 2013, NSA leaks led cryptography experts to warn of insecurities in Dual_EC_DRBG elliptic-curve cryptography [30]. The claim was that the standard implementation had been deliberately weakened. Due to this, it was at the forefront of my mind that I needed to investigate use of multiple

key types, and specifically which would function well together. RSA and DSA keys can both be generated and used from `openssl`, and if one were found to be weak, the still-strong other keys would still defend against the worst forms of attack until the weak class could be replaced.

NIST, the US Federal Agency responsible for security standards, advises [3] that to generate a key secure against computation-based attacks until at least 2030, RSA private keys should not be less than 2048 bits in length, and DSA private keys should not be under 224 bits (with a corresponding 2048-bit DSA public key). In the interest of enforcing security, my package manager would not permit weaker keys.

The threat model in the academic papers, which GPM must secure against, is incredibly far-reaching, just that any message may be responded to by the malicious party without indication it is not the legitimate party. Examples of attackers that pose a threat in this way include:

- Someone on the LAN performing ARP spoofing to become an intermediary for packets.
- Someone in a coffee shop pretending to be 'free Wi-Fi' to observe communication.
- State-mandated alteration of traffic by ISPs.
- Malicious files inserted into legitimate repositories.
- Malicious repositories.

GPM only needs to secure against these in certain circumstances. For example, it is not possible to guarantee secure communication on an already compromised machine, nor is it possible to guarantee the package manager was downloaded securely given that the download occurs over a less secure protocol[7]. It is also required for users never to install packages through alternate means. If a user manually installs a local package file, even if they check the signature's validity, they have bypassed the file and metadata phases, so a single key compromise allows package replacement attacks.

2.5 Creating an Exploit

Given I had investigated the correct way to do secure communication, I also wanted to perform the attacks hinted at in the papers. In only one day, I was

[7]Careful signature and hash checking is required for a user to trust the download, but the signature and hash specified on the website may also be forged.

able to create a simple, fully-working metadata modification attack capable of installing old versions of Arch Linux packages or inserting alternative dependencies. It took only these steps:

- Download the real 'core.db' metadata file from any repository mirror.
- Open it (it is a .tar.gz archive). It contains one directory per package in the repository, each containing plaintext files such as 'depends' and 'desc'.
- In the case of installing old versions, copy across the directory from an old 'core.db', and also take a copy of the old package. This will replace the newer directory, now causing the files to containing an old version number, filesize, file hash and GPG package signature. In the case of adding packages to install, add extra lines to the 'depends' file[8].
- Upload the altered 'core.db' (and all packages) onto a webserver you control, following the same directory structure as the legitimate server. If using Apache, ensure VirtualHost accepts requests destined for the repository's domain name.
- Run the Arch Linux live CD. This step demonstrates how `pacman` is preconfigured insecure to this attack.
- As a man-in-the-middle, rewrite all requests to the default repository IP (108.59.10.97) to your own server. In my case, I altered routing on the host, not inside the virtual machine, using '`iptables -t nat -I OUTPUT -dest 108.59.10.97 -p tcp -dport 80 -j DNAT -to-dest 178.79.160.57`'.
- Run `pacman -Syy` to check for new metadata. The package manager will now incorrectly calculate results for install requests.

Examples of correct and incorrect operation due to this attack can be seen in Appendices C and D respectively.

2.6 Third Party Tools Used

Next, I turned my attention to exactly how I would carry out my project, beginning with finding existing tools to perform some components that I either did

[8] I only added 'cheese,' however the CLI-only live CD resolved dozens of additional required dependencies.

not wish to, or could not safely, write myself. By selecting Maven as a build tool, I was able to import and keep up to date all libraries automatically.

Rather than attempt to implement my own cryptography functions, something I could very easily do incorrectly, I chose to follow best security practice and use established key generation libraries with many man-hours put into their development, bug-fixing and cryptanalysis. Therefore, all RSA and DSA keys for the system are produced by `openssl` commands and the validation work performed by the `java.security.*` libraries. Before starting, I performed a series of acceptance tests to ensure that only the correct file, signature and public key inputs would ever return confirmation of a valid signature, showing I was using the libraries correctly. These tests highlighted variations between expected and actual behaviour, such as the default key format, PEM, not being recognised by Java and constantly causing the signature check to return false. On altering the `openssl` command to generate DER-formatted keys, the system began working exactly as intended.

To simplify implementation and avoid Java's quirks, I also decided to use the Apache libraries for Java, a set of more simple and, by extension, more useful Java functions. Commons Exec, for example, makes calling the command line easier, and Commons Compress makes extracting compressed packages a single line.

Since most of my project focused on the backend, attempting to create a beautiful user interface was not a top priority. Instead, I located JewelCli [40], a Java library which automatically generates the majority of such code. Twelve lines encapsulates all required command line interaction for my application, including a `--help` command.

```
@CommandLineInterface(application = "gpm")
public interface GPMInterface {
    @Option(shortName = "i")
    List<String> getInstall();
    boolean isInstall();
    @Option(shortName = "u")
    boolean isUpdate();
    @Option(shortName = "m")
    boolean isFetchMetadata();
    @Option(helpRequest = true)
    boolean isHelp();
}
```

Then, a query '`gpm -i bash zsh`', causes `isInstall()` to be true and `getInstall()`

to be the list `[bash, zsh]`. Use of this powerful library saved me many hours of work.

2.7 Software Development Model

Since the concept lends itself well to a modular design, bisected into the client and server side components and then further split into parts regarding the three stages of communication, bottom-up appeared the ideal design for me. This is shown in Figure 2.3. By pre-defining the interface between each, I could work on components in any order, use integration testing to ensure correct operation, then combine them.

Each module could be created in multiple commits to a Git repository on GitHub. This would allow a series of benefits. First, remote backups would be inherent, since copies would exist on the external servers. Secondly, work could be rolled back by undoing the patch sets that added the faulty feature. Finally, it would allow me to follow the changes being made over time to track progress and identify how close to being on schedule I was.

Below, I have provided the simple statements of each module, produced for later functionality testing.

- Trust delegation is correct iff a key signing a phase of the transaction is untrusted until it is placed in the keyring signed with the root key.
- Abstraction is correct iff the stage is able to accept a package file as a single input and output a GPM-defined data file containing all properties of the package required for further computation stages.
- Delta combining is correct iff a set of individually created metadata files for separate packages are combined into a single data structure in a single file after their signatures have been confirmed to be correct.
- Committing is correct iff, for any given import transaction, all packages are imported and represented in the data structure atomically, or none of them are added due to an abort operation at any stage up to the final file copy procedure.
- File downloading is correct iff, when provided a URL, the file is fetched and saved in a file on the local machine.
- Signature checking is correct if, for a given public key, 'true' is returned iff a file and signature are provided that correctly correspond to use of that user's key in a role that they are permitted to perform.

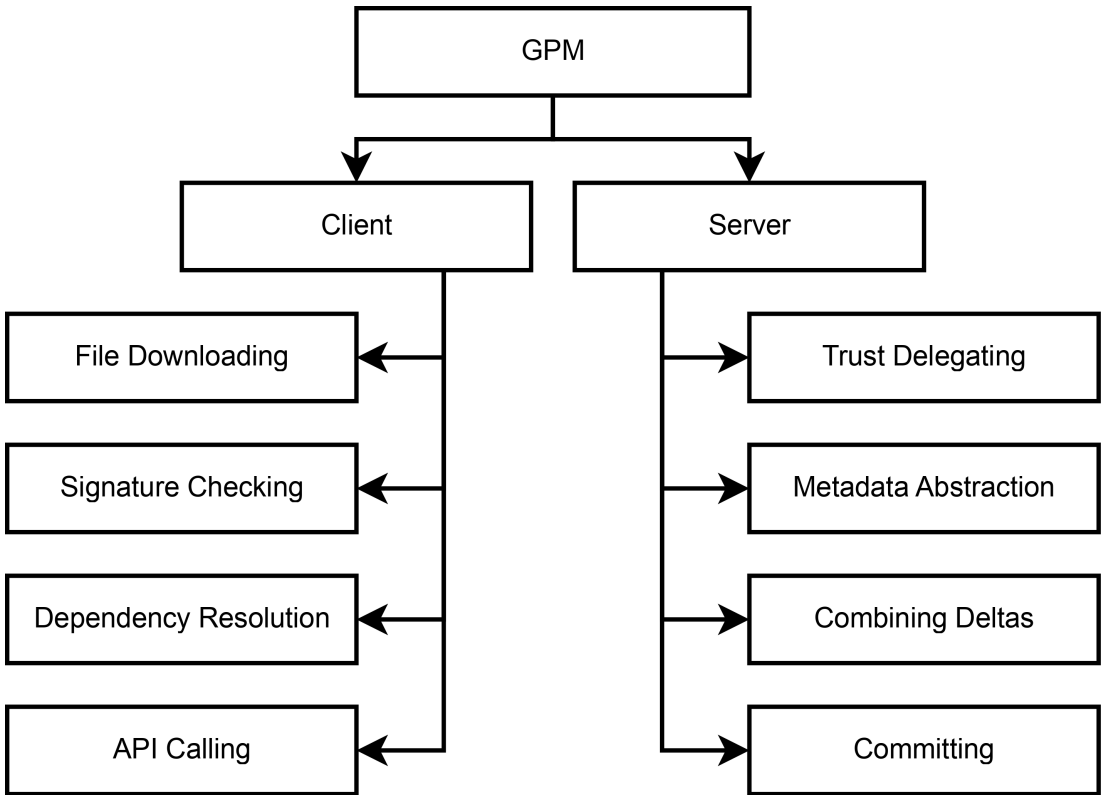

Figure 2.3: The breakdown of GPM into its constituent parts.

- Dependency resolution is correct iff, when provided a package or a set of packages, the returned result is a list of all packages that must also be considered plus those given as input parameters. The list must be provided in an order that correctly respects the partial ordering given in the dependency graph.
- API interaction is correct iff, for a call to install or upgrade a package, the correct system call is invoked to non-interactively perform the local action on behalf of the package manager.

2.8 Requirements Analysis

On a higher level, the requirements that were understood from the start were as follows:

1. The package manager must be capable of installing and upgrading software packages without ever altering their internal representation.
2. The package manager must be capable of throwing errors in cases where any of the communication has been manipulated if no keys have been compromised.
3. In the case of key compromise, the package manager must be at least as capable of identifying attacks as existing package managers.
4. The package manager must have the core of its code written independent of the input format.
5. The transformation must take less time to run on a package than would cause a backlog under standard import rates for distributions.
6. Since a package manager and its dependencies must necessarily be installed on every machine that runs the distribution, the language of choice must not cause an excessive amount of otherwise unneeded packages.

These will be referred back to after development and testing to confirm they have been met.

2.9 Final Test Plan

Although integration testing would identify the suitability of each component, the overall indication of success would be in the results of the entire workflow

and its emergent behaviour. To this end, I devised a series of metrics on which the system's usefulness could be determined. Since it is not possible to identify the time existing distributions take to import packages or to perform only dependency resolution, the results provided have to be given relative to some worst-case scenario where they become unacceptable.

1. Show that the amount of signing and, by extension, thefts required for key compromise is greater than or equal to that of other popular package managers.

2. Packages must be shown to be imported at a rate that does not cause a backlog other than during 'peak' times for clearing during 'off-peak'. Import rates for the most popular packages must be acceptable, whereas poorly performing rare outlier packages can be considered permissible, provided the reason for failure can be justified.

3. A second package format should be implemented and the number of lines of code within package-specific classes counted. This number should be sufficiently low (on the order of hundreds of lines) to justify the effort as less than that of writing a new package manager or retrofitting a new protocol to an old codebase.

4. Dependency resolution should take place, for a legitimate package[9], at an average speed that is unlikely to cause user dissatisfaction.

5. The additional overhead of the increased security should be demonstrated to be an acceptable percentage change from the existing size of a repository storing the exact same files.

In order to test these, I would need access to many packages for a distribution. Since Debian has one of the largest package offerings, I cloned the Debian repository before I had started the project, to ensure I had the files I would require months later, when testing.

[9]Any dependency resolution algorithm will take a time proportional to the size of subgraph it must traverse. Therefore, a pathological case can always be made by adding more dependencies until the time becomes 'unacceptable,' but it is alright for this to only affect manufactured, unrealistic packages.

Chapter 3

Implementation

This chapter provides full details about all components within the GPM executables. This is split into sections on key delegation, on using the server to import packages (corresponding to Figure 3.2), and on using the client to fetch packages (corresponding to Figure 3.4). At the end of each subsection, I mention tests performed to confirm each module's functionality. At the end of both sections, I also mention tests over the assembled workflows.

A full UML class diagram, showing the composition of the classes that make up the serialized files passed between server and client, is given in Figure 3.1.

3.1 Server

3.1.1 Trust Delegation and Key Validation

The user trusts the distribution, or they would not be running it. However, the distribution is not solely responsible for package uploads, therefore they require the ability to delegate their trust onto other users. To do this, the distribution must generate their RSA[1] or DSA[2] root key and store it safely, perhaps even on an airgapped machine to reduce compromise possibility, then use it to sign some collection of trustworthy keys.

In order to create a keyring, a user needs to create a `SigningKeysRW` instance, an object that maps unique user identifiers to their `SigningKey`. This contains a public key, keytype, expiry and set of permissible roles, along with a function to validate if a <file, signature> pair was created by the key. In its current state, the

[1] `openssl genpkey -algorithm RSA -out priv.pem; openssl rsa -in priv.pem -pubout -outform DER > pub.der`

[2] `openssl dsaparam -out param.pem 2048; openssl genpkey -out priv.pem -paramfile param.pem; openssl dsa -in priv.pem -pubout -outform DER > pub.der`

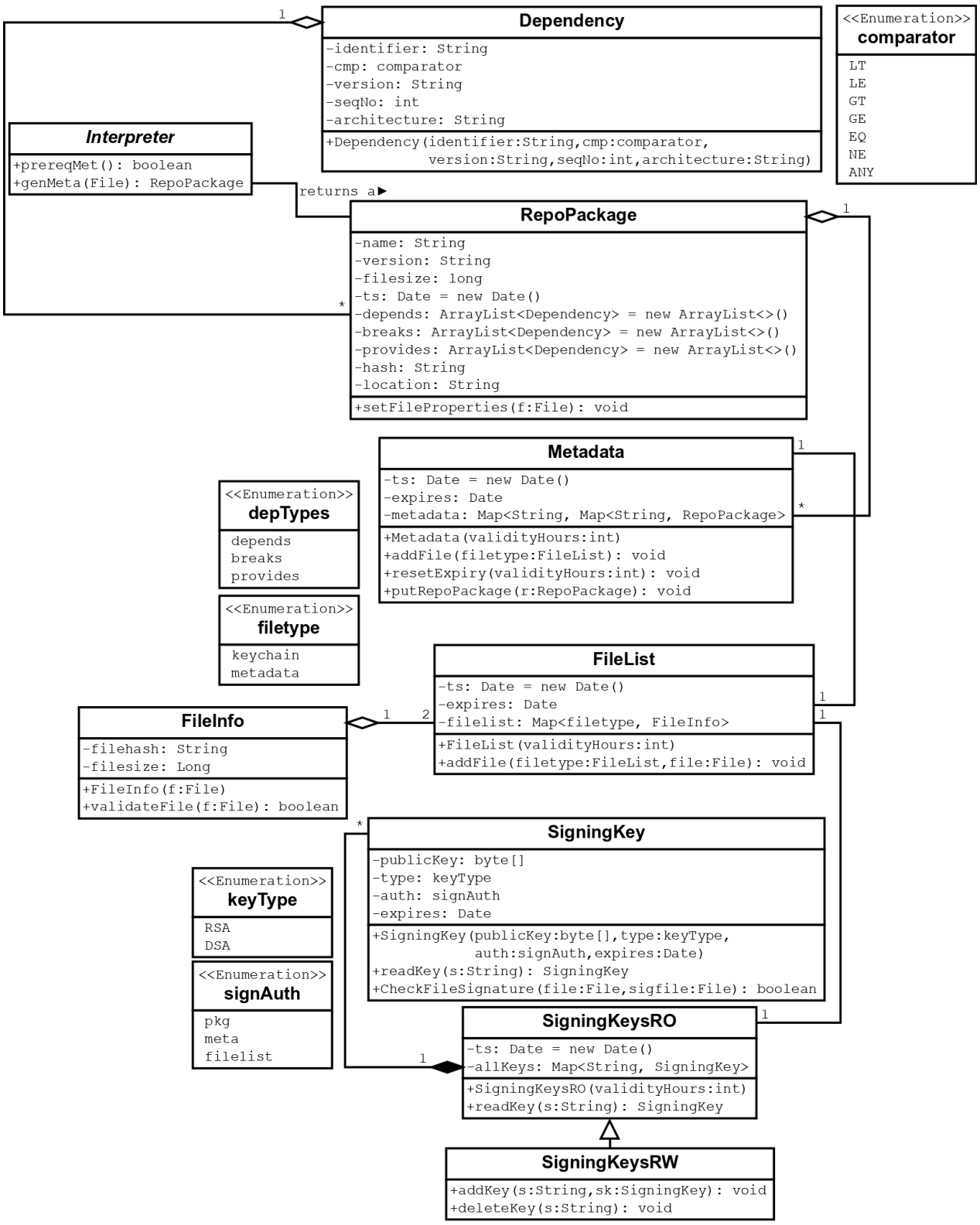

Figure 3.1: The UML class diagram for the serialized constructs.

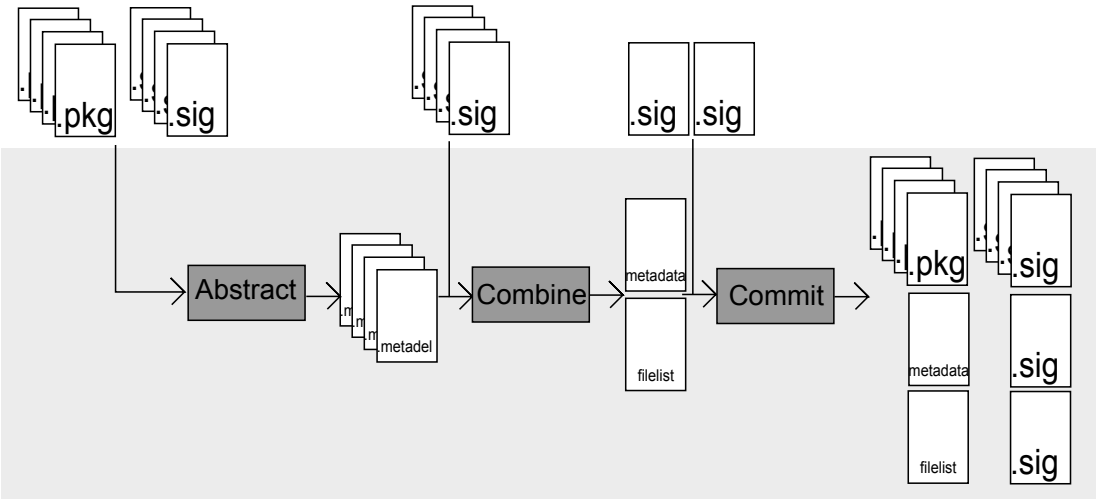

Figure 3.2: The GPM server-side workflow.

enumeration of trust levels allows package, metadata or filelist signing, however it has the ability to be arbitrarily extended to groups of packages. This would restrict compromise to only some subset of packages for which the key could be reused if stolen.

Once the distribution has inserted the correct keys they trust, they can cast this to a `SigningKeysRO`, a parent class missing modification functions since no other phase has any right to be altering the keyring, then serialize it. Using `openssl`, this serialized file needs to be signed[3], and both files need to be uploaded to the root level of the repository. These two files, 'keyring' and 'keyring.sig,' provide both the server and client knowledge of which files are trustable.

Testing of this module was similar to Section 2.6, checking the function correctly returned if the key signed a given file and signature, but also confirmed key expiry correctly made it return false.

3.1.2 Repository Structure

An example of GPM's repository structure can be found in Figure 3.3. In the repository, there is a single writeable directory, called 'incoming,' into which maintainers can upload packages and signatures. All files created by the executable during a transaction are placed in a second directory, called 'interim,' to which signers have read access only. This is necessary so they can see any generated files they are required to sign. Every accepted package is stored in 'packages,' under three directories, one with only the first letter of the package name, then

[3]e.g. `openssl dgst -dss1 -sign priv.pem -out keychain.sig keychain` for DSA.

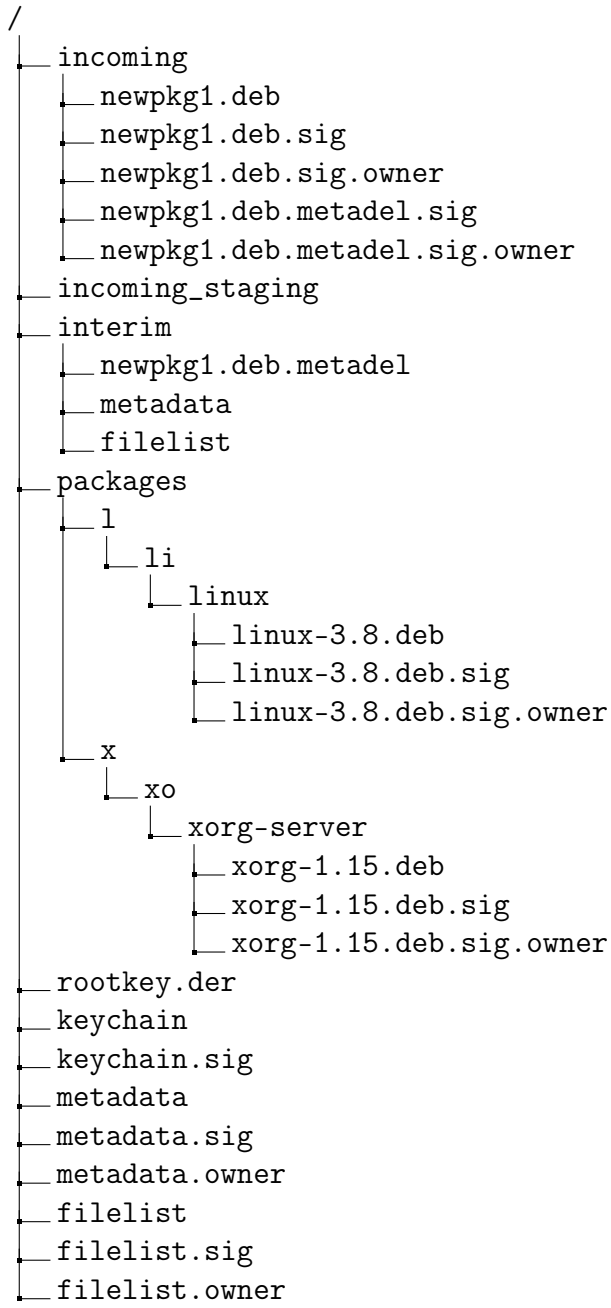

Figure 3.3: An example filesystem layout for the GPM repository while in use.

one with the first two, then one with its full name.

Using nested subdirectories containing progressively larger substrings of the original filename is a speed optimisation for accessing contents of the directory. This is due to the fact the Linux filesystem using indirect, doubly indirect, and triply indirect pointers to store the block locations of progressively longer files and directories [18], meaning larger directories take longer to read.

The fact that all state for this repository structure is held in the special files at the root level and inside the 'packages' directory means the system is very easily mirrorable using `rsync` commands. This is identical to the general practices most distributions use to synchronise mirrors with their canonical 'up-to-date' mirror already.

3.1.3 Abstracting

The first stage converts the distribution specific packages into the GPM internal representation. To do this, package maintainers upload packages into 'incoming,' with a corresponding '.sig' signature file created using their trusted key, and a '.sig.owner' text file containing only their unique identifier. The owner file is required, as signatures do not contain information about the related private key, so an index into the `SigningKeysRO` structure is required.

When GPM's server importer is run, the list of incoming files are read and, using the Strategy design pattern, the correct `Interpreter`-implementing class is run based on the package suffix[4]. This is a loosely coupled design, since the main function makes use of the `Interpreter` without any consideration for which specialisation is made. In fact, all returned values from the `Interpreter` are never related back to which concrete instance created them; it would be possible to create two different packages in two different formats that produce identical metadata.

The first function expected of all `Interpreter`s is `prereqMet()`, a boolean returning function that states if the `Interpreter` is capable of running, such as after a check for required external executables or the JVM version.

Assuming prerequisites are met, the only other required function is called, `genMeta(File f)`. It performs all the work; emitting a `RepoPackage` that adequately describes the input package by parsing out the relevant information and calling setters on the `RepoPackage` object.

To add depends, breaks and provides, `Dependency` objects are instantiated and added to arrays within the `RepoPackage` object. These contain the depen-

[4] A `file`-like method for identifying a package would be more useful, except that many package formats only look like a standard compressed archive from the outside.

dency identifier[5], a comparator and version for constraints on the dependency[6], and a sequence number[7] to indicate if any dependencies are equivalent.

Rather than setting the hash or filesize manually, however, GPM will insert this information. This is because it is not a property of the package format; no `Interpreter` should need to be changed if the hashing algorithm used within GPM is altered.

The automatically generated timestamp on each of the `RepoPackage`, `Metadata` and `FileList` classes is crucial; if a signer submits a signature then wishes to retract it, deleting the uploaded '.sig' file is insufficient. A second user can take a copy of the signature and reupload it, and it will still be valid. By the revoker pushing a file named 'reset' into the incoming directory, GPM purges the interim files. On GPM's next run, the files will be regenerated in a new transaction, however the differing timestamp will invalidate the signature on the file.

In Appendix E, I have provided the core of the Java code for extracting metadata and dependencies from a .deb package. In order to test its operation, I performed white box testing, attempting to form DEB packages that I did not believe it would be able to handle, and observing its operation. These included packages with no dependencies, multiple equivalent dependencies, and specifically versioned dependencies. After confirming the contents of multiple `RepoPackage`s were correct for the corresponding archive, I considered this section complete.

3.1.4 Combining

Once `RepoPackage` instances have been serialized onto disk in '.metadel' files (so called because these are delta additions to the metadata file), the system refuses to import any more packages, considering the serialized `RepoPackage`s to form a single transaction. It must atomically succeed or be reset in the manner described in the previous section.

Since differing people may have the prerequisite knowledge to sign off the different packages making up a single transaction, signatures are collected on these '.metadel' files from those with keys with metadata signing permissions. This barely alters existing workflows for package updates, in that rather than marking acceptance on a bug tracker, the same person uploads the signature for the package in agreement.

[5]Dependencies may be on a package name or, in the RPM case, on files within a package.

[6]A dependency constraint of '<=3.9.6' would become a comparator of 'LE' and a version of '3.9.6'.

[7]If the distribution contains AND and OR relations, the dependencies must be converted to Conjunctive Normal Form. (`python2` $\vee$ `python3`) $\wedge$ (`mysql` $\vee$ `mariadb`) will assign `python2` and `python3` sequence number 1, and `mysql` and `mariadb` sequence number 2.

On running the executable again, if all '.metadel' files have been signed correctly, a new 'metadata' and 'filelist' are emitted that have the correct changes to take into account the new transaction. Any person able to sign metadata can sign the new metadata file as a formality, as others have signed each constituent part of it to form consensus that the entire package database is acceptable. A gatekeeper responsible for considering validity of transactions as a whole then signs the file listing, a small file that references the newest version of the metadata and keyring files by hash and total length.

In this way, the three signers have successively given oversight to a broader view of the transaction. First, the package creator signed to say they believed the sources or compiled application in the package was correct. Next, the metadata signer ensured the package creator was not acting maliciously, and also that all distribution-specific additions, such as the dependency lists, had been created correctly. Finally, the file listing signer had validated that the transaction of multiple packages would be acceptable for import, catching errors like uninstallable packages, dependency cycles, and any scheduling considerations, in the case where transactions occur at given intervals[8].

Confirming the correctness of this module was relatively simple. Given a metadata file and a series of `RepoPackage` instances, I only had to confirm that the result was a metadata file with the new entries also inserted. All tests related to the transactional nature of the system were performed later.

3.1.5 Committing

The final stage of importing a package into the repository performs all computations again for safety reasons, with all incoming files moved into a read-only staging area. The package and package signature are confirmed to match, the `RepoPackage` is regenerated with the old timestamp to ensure the package hasn't been replaced since its initial import, the signature for the delta is tested, then finally the signatures for the combined files are tested[9].

Any failure on this stage causes the transaction to abort and the staged files to return to the incoming directory, although the transaction will not reset itself. If the phase succeeds, however, each package, its signature and its owner file move to the correct subdirectory in the package tree. Since the metadata and file listings remain unchanged at this point, all client requests still see the old data

[8] A company with a repository for quality-checked packages for employee machines may have a release schedule.

[9] If all stored '.metadel' files are valid, the merge stage must be correct, since the operation was on files in a read-only directory.

and can download the old packages, which are not removed.

The alterations to metadata and filelist will indicate to clients a new transaction has occurred, however moving in either order provides a period of potential failure. If the filelist is moved first, clients will attempt to update to the new metadata mentioned, but find the one they download to be for the prior transaction and therefore have the wrong hash. If the metadata is moved first, any person that has an even older metadata file will see the prior transaction's in the filelist, attempt to fetch it, but retrieve the new metadata with the wrong hash. All clients with the previous metadata file will still believe it is the newest, though, and skip the download phase. The latter is the method GPM takes, specifically because of the fewer number of users affected.

A possible addition, not included in GPM currently, would be to retain multiple metadata files. This would provide perfect atomicity; the second metadata file would exist with a different name, then the pointer in the metadata file would swap to indicate this new file. This, however, was intended to be part of a larger extension of pruning: the deletion of old packages, the cleanup of the Map inside the serialized `Metadata` object, and the ability to keep n old metadata files.

Once the files have been moved into place and the metadel files have been purged, the system is ready to start another transaction using the new files as the basis. It is important to note that all four ACID properties are provided to some level by the commit stage. The act of transactions produces atomicity. The system applies `RepoPackage` instances to a valid metadata file to produce another valid metadata file, providing consistency. Isolation is achieved by never allowing simultaneous transactions, forcing a sequential order. Durability is intended to be provided by the underlying filesystem, Java's move function accepting a parameter[10] that states the entire file must be moved atomically and permanently.

Tests related to committing were minimal, simply that moves never caused unintended overwrites, a situation fixed through renaming the package to its name and version, and only overwriting in the case where these are identical[11].

3.1.6 Integration Testing

All of the pieces of the server were finally fitted together to ensure packages flowed through from initial input to result. All tests passed, the system allowing me to import hundreds of packages at a time in transactions.

Multiple attempts at breaking the workflow were made. These included deleting signatures after they had been checked, altering the packages, and replacing

[10] `StandardCopyOption.ATOMIC_MOVE`

[11] In this case, the `Metadata` is also overwritten, perhaps to fix a faulty package commit.

the package and its signature to another valid pair, all caught by the regeneration step. It was possible to break the repository by sending the process a SIGKILL while it was performing the transaction, however in the case where only packages had been moved, uploading the same packages again and carrying out the transaction was sufficient to overwrite the package and insert the metadata into place. In the case where the filelist and metadata were from different runs, because only one had been copied, the repository required manual intervention to revert to a consistent state. This issue has not been fixed, since it would be an incredibly rare occurrence.

3.2 Client

The GPM client interacts with the repository to install packages or upgrade a system[12] using the generic metadata file produced by the repository. After computation has been performed, the required packages are downloaded and handled by the relevant distribution-specific API. In this manner, GPM's final downloads are of binary files that it cannot interpret, but that it has resolved are necessary through the GPM serialized data structures that it is able to read. A full run of GPM is given in Figure 3.5, and its output is described in each of the following sections.

3.2.1 File List

On issuing a request to update metadata, the file listing is downloaded and checked for a valid signature using the client's locally cached copy of the key store. The root public key is hard-coded into the application; it is not transferred during the protocol as this would give an opportunity for an attacker to attempt to alter that key with his own.

If the trusted keyring mentioned in the file list is different from the one cached on the machine, the remote copy is downloaded and signature checked, then the entire protocol resumes from the start. The restart is not strictly required, since attacks that could be leveraged with a stolen key would seek to stop clients downloading the information about the revocation, therefore never leading to the situation where the system learns of a revocation then immediately finds that the file listing is signed by a now-invalid key. However, it is safer than optimising it away to avoid the potential that additions to the start of the protocol will be

[12]Package queries and deletions have not been implemented, since they would uninterestingly call a CLI command and output it to the screen with no extra computation.

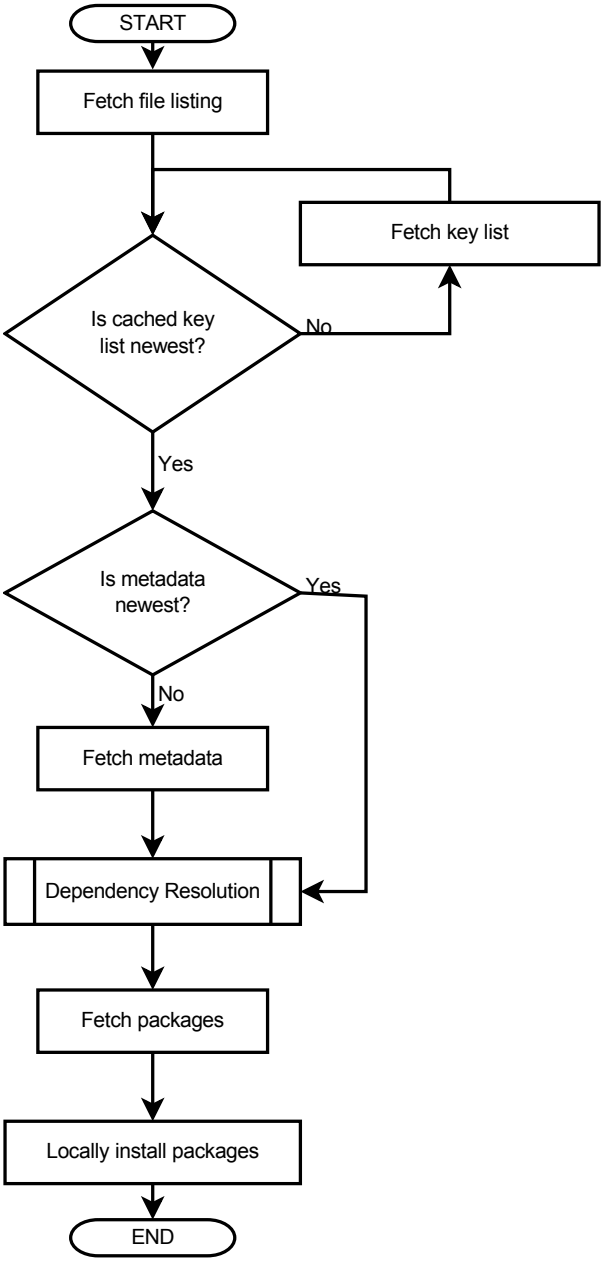

Figure 3.4: The GPM client-side workflow.

```
# gpm -m
Checking file list... Metadata not up to date.
Downloading metadata... Newer metadata file cached.

# gpm -i openbsd-netcat
The following schedule has been created:
- Install libbsd
- Delete gnu-netcat
- Install openbsd-netcat

Downloading libbsd... OK
Downloading openbsd-netcat... OK

Installing libbsd
Deleting gnu-netcat
Installing openbsd-netcat
```

Figure 3.5: Installing `openbsd-netcat` using GPM.

skipped because of this, leaving opportunities for security failures.

Once the key list in the cache matches that reported by the correctly signed filelist, the metadata hash is checked to see if the metadata requires downloading. If so, it is fetched and validated, otherwise the existing metadata file is used.

Integration tests were performed on this part to confirm that the system could identify the inconsistency between local and remote copies of cached files after new transactions had taken place.

3.2.2 Dependency Resolution

The metadata file, as generated by the server, is a map of maps of package information. By selecting a package name, then a version of this package, all metadata is returned as an object, as illustrated in Figure 3.6.

```
(linux -> (3.14   -> [RepoPackage],
           3.13.7 -> [RepoPackage],
           ...),
 gcc   -> (4.8.2  -> [RepoPackage],
           4.8.1  -> [RepoPackage],
           ...),
 ...)
```

Figure 3.6: A visualisation of a `Metadata` map.

To perform dependency resolution for a known package name, a valid version to install must be automatically selected (e.g. the highest version), then its dependencies and conflicts must be satisfied earlier in the schedule. Dependencies are often visualised in a graph of the form given in Figure 3.7. As an example from the figure, it would not be acceptable to install `libncurses5` before installing `libc-bin`, since although there is no direct dependency joining the two, `libc6` would either not be installed yet, or may not have installed correctly.

The key algorithm, initially based upon Knuth's "Algorithm T" mentioned in Section 2.2, was converted into a recursive algorithm[13] by the end due to computation issues. Whereas Algorithm T requires a queue counting the number of 'depended on by' relations, the recursive variant uses calls to travel down towards the leaves, and `return`s to move back towards the root. By marking each leaf when it has been placed into the schedule, two paths to the same dependency[14] do not cause it to be scheduled twice.

At a high-level, the function to produce a queue of actions in a valid order is:

- For each package provided as input:
  - If the package is marked as handled or is already installed at an acceptable version, mark the package[15] and do nothing for this iteration.
  - For the set of dependent packages, call this function again.
  - For every conflicting package, add its removal to the queue.
  - Add this package's install to the queue.
  - Mark the package as already handled.

[13] Knuth references an unpublished chapter during his explanation of the algorithm, stating a 'better' recursive variant. I believe mine to be similar, since it almost matches the four recursive algorithms mentioned by other sources.

[14] The figure shows `libncurses5` and `libtinfo5` both explicitly requiring `libc6`.

[15] In case it is already installed but not marked.

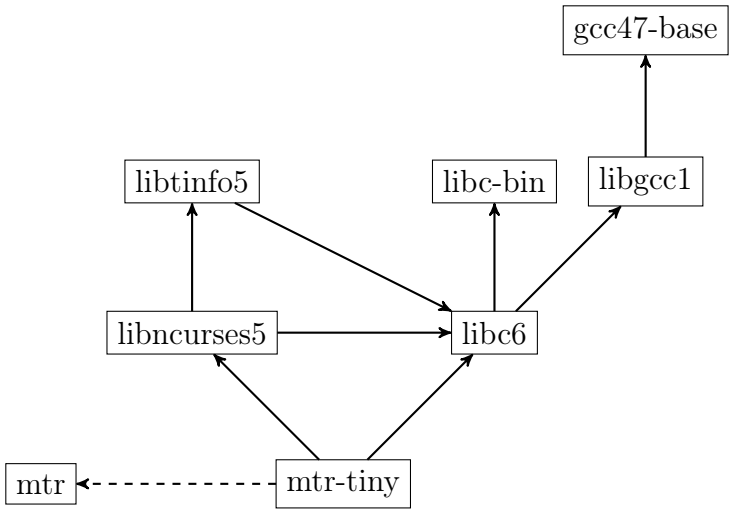

Figure 3.7: A standard dependency tree. Arrows point in the direction of a package that must be considered first. Dotted lines indicate deletion prerequisites, solid lines installation prerequisites.

The greater complexity is in correctly implementing each of the above points, such as identifying which version to install, identifying which dependencies are permitted to be missing (if two dependencies have the same sequence number, the first resolution may fail if the second succeeds), and on identifying if local packages require alteration, upgrade, or already satisfy given conditions. In fact, this complexity caused me to have to remove and rewrite the whole of dependency resolution three times, possibly only due to being able to undo Git commits, and already having defined the input and output formats such that no other code had to change.

White box testing on the final incarnation, by deliberately crafting difficult package metadata, showed two cases in which the function does not act exactly as intended. The first is in the case where a package conflicts with one of its dependencies, making it uninstallable. As an example, Figure 3.8.

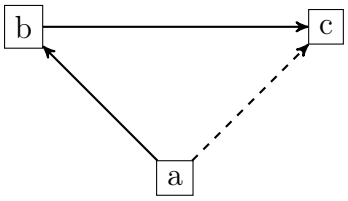

Figure 3.8: A problematic dependency tree.

Assuming `b` is already installed, the system will not check what `b` depends

upon. Therefore, a request to install `a` will attempt to delete `c`, then install `a`. This will leave `b`, and `a` by extension, broken.

The solution to this is not to place dependency cycle-containing packages into the repository. The filelist signer has the responsibility to check this at the moment, but the real solution is to identify it automatically, as Debian EDOS [22] does, and refuse to allow the transaction.

The second is the issue of tracking all packages that will also be uninstalled on a given uninstallation request. Since the metadata only has relations in the 'depends on' direction, identifying 'depended on by' is hard. Either the repository must perform the inversion, increasing download sizes and making delta merging non-append-only[16], or each client must compute this identical data, taking many seconds. This remains an open investigation[17], the temporary fix being to issue API requests with the 'and delete all dependencies' flag. This is dangerous, however, since GPM cannot fully state the result of the API request it is about to make.

3.2.3 Installation and Removal

My decision to use the underlying features of the specialised package managers allows installing and deleting to be incredibly simple. Rather than perform the installation, the system saves the downloaded, validated-correct package to a temporary directory and then invokes an `APICaller`, again using the Strategy design pattern. It is analogous to the `Interpreter` on the server, except provides an interface between GPM and the relevant API. The client can then call `install(File f)` for each package in turn to install them without user interaction or 'dependency not met' errors.

The only required tests for this section were that the commands used in the `APICaller` did perform the expected actions, generally very easy given the manual pages for the package manager.

3.2.4 Integration Testing

To complete testing on the client, all the components were fitted together and a series of random install requests and upgrade requests were made to demonstrate

[16] When adding a package, all its dependencies would have to have the package inserted in their 'depended on by' collection.

[17] During testing, the idea to record the relation only for installed packages, and to add to it only on new install requests, was conceived. This amortises the work, keeps only a subset of the relation, and tracks partial upgrades properly. However, I neither had time to test this properly nor create a proof of concept implementation.

the system's functionality. In all cases, I first selected some packages to resolve, checked the computed schedule, and compared it to that of `debtree` [31], an application that draws Debian dependency trees. In all cases, this showed the schedule to be safe and valid. I did find instances of failure. In some, where no valid resolution existed, EDOS confirmed this. In some, the system overflowed the stack due to cyclic dependencies, which EDOS also mentioned. I believe both of these failures to be justified as being the fault of the user who permitted that transaction.

3.2.5 Download Nuances

Where authentication occurs through signatures, a Denial of Service is possible. This is because signature verification cannot occur until the download is complete, not even to fail early. Supposing a victim automates updating, a man in the middle is capable of returning a bad file to the client, with no intention of trying to trick the package manager into believing it is correct. Instead, the attacker seeks to provide the largest possible file, forcing an excessive download. If on the LAN, the attacker can attempt to exhaust the filesystem or RAM. If the attacker is remote and serving this file to a user with a bandwidth cap, the victim could end up being billed by the ISP or disconnected[18].

One possible solution would be to arbitrarily impose limits on the sizes of the files, but this is likely to become obstructive. Limits would have to be incremented gradually as files grew, and certain limitations would be entirely ineffective. A limit of 10 GB per package to permit some full-sized games would still allow 10 KB shell scripts to be grown to 100,000 times their original size.

Instead, the solution I provided was to arbitrarily pick a 1MB limit for the filelist. This is because it is only intended to hold a couple of entries of an enum indicating a filetype, a hash, and a length. Then, every stage states the length of the next stage. The file listing states the length of the key list and of the metadata file. The metadata file specifies the length of the package. Then, if a file that is overly long is fetched, the download is aborted at the expected length, rather than when the file finishes being sent.

[18]Queens' College would bill £100 if my 1 TB drive was exhausted by this attack. The same attack on BT's uncapped fibre package would cause overages of £1,060.

Chapter 4

Evaluation

In this chapter, I aim to provide a series of investigations into the resultant behaviour of my package manager, and judge to what level it is suitable as an alternative to those already established in the field. This comprises both qualitative analysis of its feature set and quantitative benchmarking results, as was given in my test plan in Section 2.9.

4.1 Overall Security

The initial purpose of this package manager was to mitigate key compromise. To this end, it is important to confirm exactly what benefits the implemented protocol has when compared to existing package managers.

4.1.1 Comparative Amount of Communication Security

Firstly, I seek to identify how much of the communication between GPM and the repository is signed and compare this to other package managers.

This is done by providing Table 2 from Samuel et al. [33] as Table 4.1, containing entries for whether certain phases of package managers' communication is authenticated. At the end, I have inserted a line showing GPM's implementation.

In the table, downloading over SSL implies end-to-end encryption. However, a secure link to a malicious mirror is of no benefit. The standard Java libraries allow GPM to perform SSL connections, but in no way do I consider it an advertised security feature.

This figure demonstrates, at a high level, that I have attained more security during communication than any other package manager surveyed by Samuel et al. I believe this to be a very positive result. However, it is important to analyse the purpose of the security, so I do so in the next section.

Name	Package	Metadata	File List	Over SSL?
`yum` (Fedora)	Signed			Yes
`yum` (Red Hat)	Signed		Signed	Yes
`apt` (Ubuntu)	Signed		Signed	
`yast` (OpenSuSE)	Signed			Yes
`slackpkg` (Slackware)	Signed			
`pacman` (Arch Linux)	Signed			Optional
Sparkle (OS X apps)	Optional			Optional
Adobe AIR app updater	Yes		Yes	Yes
GPM	Yes	Yes	Yes	Optional

Table 4.1: The signedness of each stage of communication for many popular package managers.

4.1.2 Resilience to Key Compromise

As mentioned in the previous section, a more pertinent question is "To what level does GPM's signing prevent against attacks when some subset of keys have been compromised?" To answer this, I am replicating a reformatted Table 5 from Samuel et al.

Roles compromised	Fake package?	Old/Extra packages?	Freeze updates?	Until?
None	No	No	Yes	[a]
File List	No	No	Yes	[b]
File List + Package				
File List + Metadata	No	Yes	Yes	[c]
File List + Package + Metadata	Yes	Yes	Yes	
Root	Yes	Yes	Yes	[d]
Metadata	No	No	No	N/A
Package				
Metadata + Package				

[a] Until the signature on the file listing expires, usually set to the longest period between two consecutive transactions. May be 24 or 48 hours, not enough time for a vulnerability to be found in a package.

[b] Until the stolen key expires, the signature on the metadata file expires, the signature on the trusted keyring expires, or the root key expires.

[c] Until a stolen key expires or the root key expires.

[d] Until the root key expires (usually set to never expire for a given OS release, a new release using a new repository and having a new root key).

For each instance of 'No,' this means that not only will the attack not cause the package manager to exhibit unwanted behaviour, but it will also be detected.

In the case where a stolen key expires, the new exposure to attack is given by the row containing any still-compromised keys. In the case of comparing another package manager that does not secure a given stage, such as one from Section 4.1.1, simply assume all keys of that class are compromised, and that they have no expiry date.

As a result of its protocol, GPM therefore requires a filelist key theft for extended periods of update freezing, a filelist and metadata key theft for metadata modification, and one of each key theft for package replacement.

4.2 Performance testing

To analyse the performance of the system, separate evaluations of the server and the client are made in the following subsections. This is due to the nature of a package manager; the server and client operations do not occur sequentially, nor do they have the same set of timing requirements.

4.2.1 Server

The features required of the server relate to the speed of import. Using the full mirror of Debian packages downloaded at the start of the project, I performed a test that my implementation was able to import every package, and recorded the import times for each package. The results of this also produced a GPM repository usable for the client testing components.

4.2.1.1 Acceptable Daily Rate

The first important property the server must have is to be able to import packages faster than they are likely to be uploaded.

To test this, I first had to find the daily rate of package uploads, a number which varies depending on how release schedules of different projects happen to coincide. Peaks tend to occur on mass-rebuilds, where packages require recompilation under a new library or need a 'soname bump'[1].

Debian provides a list of the last seven days of package updates [20]. Over the month of December, I counted the number of new packages that entered their

[1]Where a dynamically loadable library changes its name, such as `libreadline.so.5` to `libreadline.so.6`, all dependant packages will attempt to read a non-existent file if they are not recompiled.

unstable repository using this page, and got 983. This is only 32 per day, on average. However, on the day `gcc` was recompiled, 94 packages were admitted. Ambitiously, if a day's imports are targeted to take less than ten minutes, they must complete in $\frac{10 \text{ minutes}}{94} = 6.38$ seconds. However, this must be halved, since abstraction occurs twice, once for file creation and once for staging.

Figures 4.1 and 4.2 show the concentration of import speeds and the outliers for the mass-abstraction, a test that lasted 199234 seconds (an average of 1.28 seconds per package for the 155065 packages imported).

Since the graphs show that not only did the average package import in far under the required time, but 98.45% of all packages did so, I conclude that the import occurs at a satisfactory rate for even large distributions handling many package uploads per day.

The three packages that were the slowest had 211 (libmono-cil-dev), 87 (openoffice-core) and 117 (ia32-libs) dependencies. I believe the increased amount of parsing and regular expressions used to extract these dependencies is likely what caused the slower imports, up to 21.9 seconds in the worst case.

However, I recognise that not every package should be considered equal for importing importance, and therefore it should instead be verified that packages relevant to the majority are fast to upload and distribute.

4.2.1.2 Most Popular Packages

To resolve the issue recognised as the end of the previous section, I formed a plan to identify the most important of all of the packages imported in the batch run to more accurately understand GPM's speed.

In order to determine which packages classify as 'important,' I used the Debian Popularity Contest [1], 'popcon'. This provides lists as to the most used, the most often upgraded, and the most installed packages. These help identify packages that people rely on and enjoy having up to date, and those that are installed but forgotten, probably only required as dependencies of the base install rather than user invoked.

Figure 4.3 shows the results of importing the top 500 packages in each section. In each case, only the very uppermost result exceeds the 3.2 second target, 499 of the 500 importing in the ambitious target time. I believe this to be very promising, and show that users will be satisfied with the distribution speed for uploaded packages they are interested in.

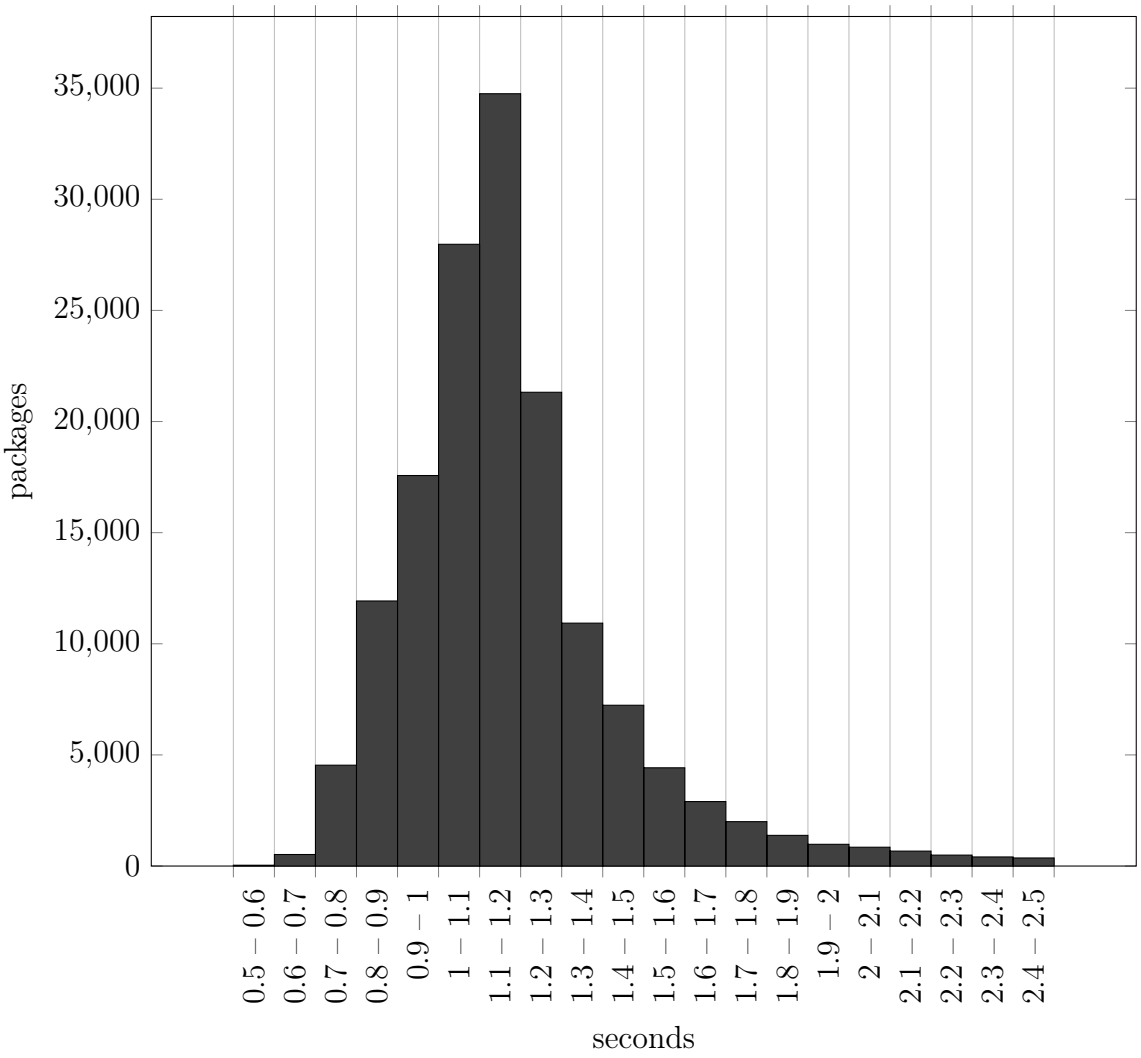

Figure 4.1: A visualisation of the peak of the import times, and the narrow spread of results.

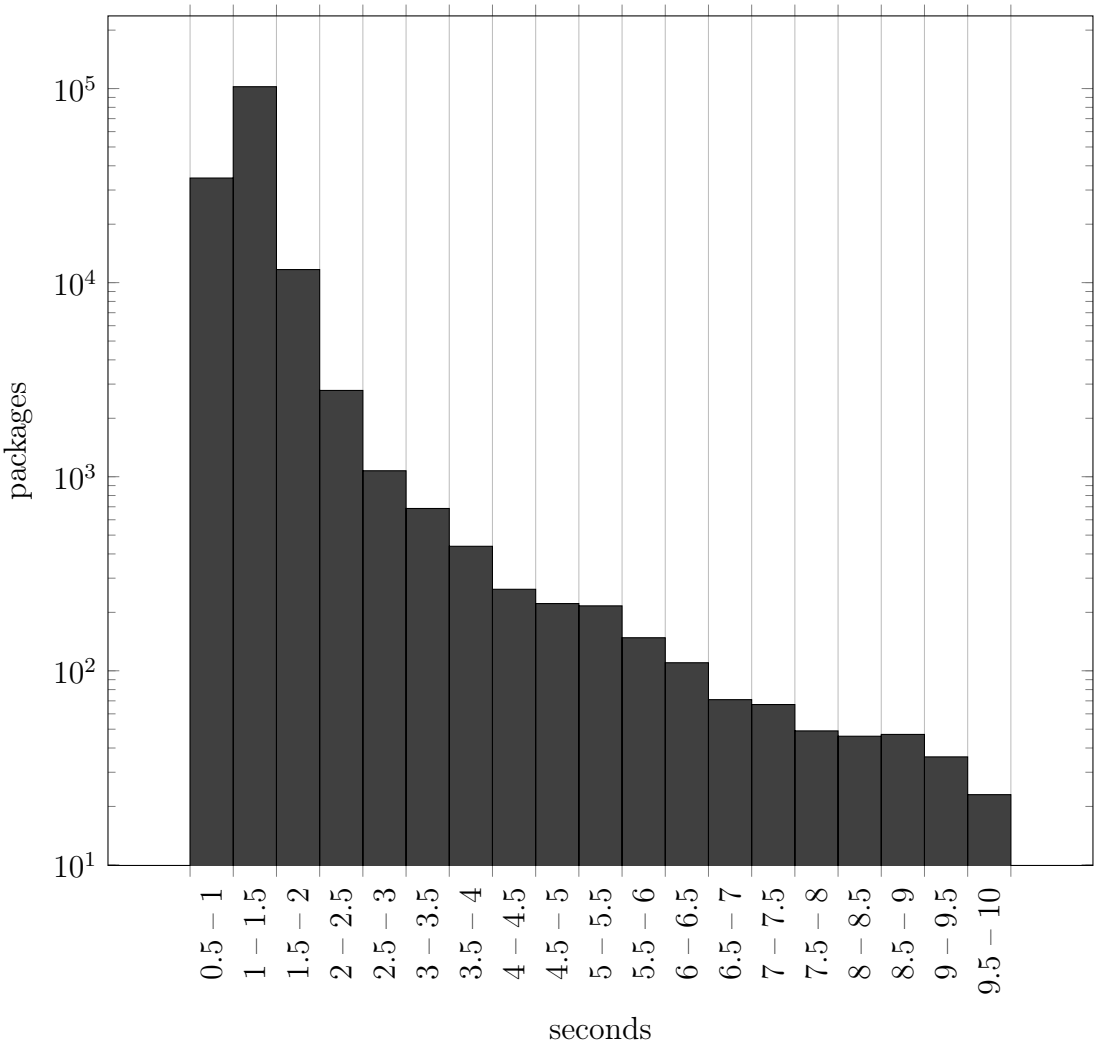

Figure 4.2: A log graph of results for the mass-import.

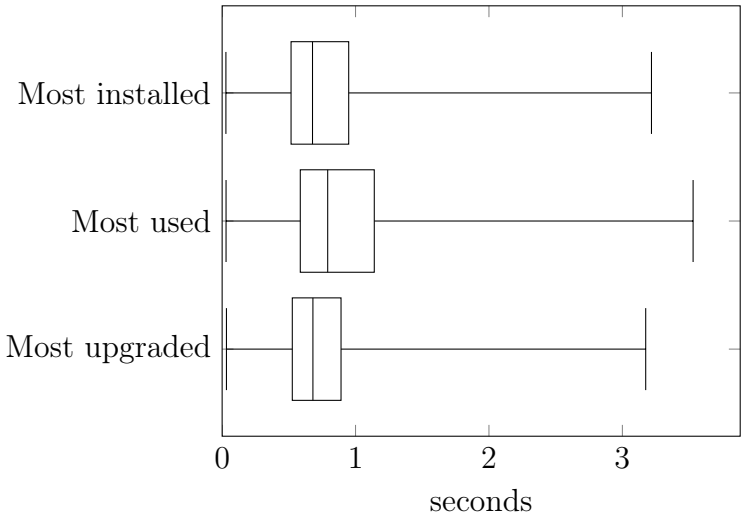

Figure 4.3: A box plot showing the distribution of import times for packages deemed 'relevant' to users based on Debian popcon.

4.2.1.3 Ease of Implementation

Finally, it is necessary to check that distributions would accept writing the number of lines of code required to implement their package format in GPM, rather than rewriting their own codebase to provide the same features.

Depending on the distribution's existing security, different levels of work would be needed. At the very least, extra signatures, plus any code to produce a chain from these multiple signatures. For systems signing only one phase, their implementation may be even more naïve, missing the concept of roles entirely.

Therefore, in order to make GPM a realistic choice, the amount of code required to specialise it must be on the order of hundreds of lines, not thousands. As such, I sought to write a second `Interpreter` class, capable of abstracting Arch Linux's '.pkg.tar.xz' package format.

The entirety of the `Importer_Arch` file is 92 lines of code (including comments and blank lines for formatting), none more complex than use of a decompression library and a set of regular expressions. The abstraction of my laptop's package cache, containing a representative sample of packages that a general system runs, was 55.08 seconds for 3744 packages, or an average of 14.7 milliseconds per package. This is more than likely due to the far simpler to parse packaging format[2], and I believe these results to be more than satisfactory.

[2]Arch Linux uses a less nested archive structure which makes it faster to extract, and uses a less complex metadata format to parse through regular expressions.

4.2.2 Client

Unlike the server, where delays are acceptable for batch importing, the client has a far greater reliance on speedy responses, since a user will be looking at the screen and expecting instantaneous output. Unlike servers, client machines are also often far more bandwidth constrained. The tests in this section seek to identify if the alternative protocol causes unacceptable drawbacks for users.

4.2.2.1 Dependency Resolution

The main algorithm the client uses is dependency resolution. Therefore, it is important that this part performs well; it is the only intensive computation required. The aim is to test that a user will not become annoyed by the algorithm's execution time, particularly as the algorithm cannot display an accurate progress bar[3].

Attempting to measure the time dependency resolution takes as part of another package manager is near-impossible, since the phases are not clearly separated. Therefore, I had to use the rather informal determination that, at rare times on both my Arch Linux laptop and my Debian webserver, I have experienced dependency resolution times of around six or seven seconds, and considered them onerous. Therefore, dependency resolution of GPM must be less.

To test resolution times, I directly called the resolution function with the top 1750 packages installed on Debian from popcon, since these packages constitute those most users see over a general lifetime of use of their system, and therefore will want to be fast. Every time, GPM was told no packages were installed on the system, so as to perform the maximum traversal possible.

Figure 4.4 shows the single 1750 package test. I found the results of it fascinating; the packages had partitioned themselves into two distinct sets, although I could not find an exact formula to separate them. Regardless, I have added two regression lines for the datapoints corresponding to shallow, broad trees (faster, traversal requires mainly iteration) and deep, narrow trees (slower, traversal requires more recursion)[4] which appear to fit the data particularly well.

Extrapolating both lines, packages of 2098 'shallow' dependencies and 590 'deep' dependencies would be installable in five seconds, numbers considerably

[3]If the root has two dependencies, one with one dependency, one with one hundred, the best approximation possible would be that completing one of these dependencies causes 50% progress, which is false. Until the package manager inspects the child nodes, it does not know how many grandchildren nodes there will be, and thus how much work there will be to perform.

[4]The fifty packages with highest and lowest average branching factor for their subtrees were used to generate the regression lines.

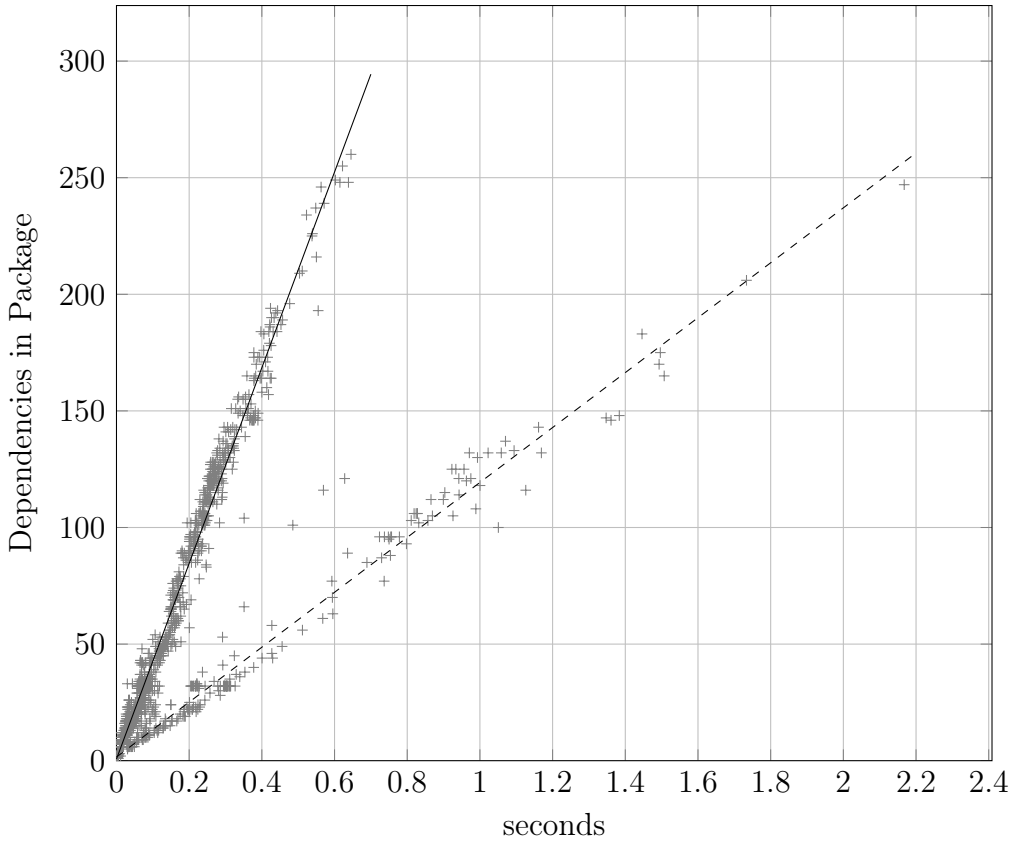

Figure 4.4: Dependency resolution times for each of the top 1750 installed Debian packages.

greater than for any package I've encountered. This suggests the client to far exceed dependency resolution speed expectations, let alone requirements. However, it is also important to note that based on a user's bandwidth, this resolution period may not be the longest component of the overall install.

4.2.2.2 Comparative Filesizes

As the previous section hinted, a user on a slow broadband connection may not be willing to accept excessive filesize overheads for security, download times of which may be far greater than that of even the most inefficient dependency resolution algorithm. The downloads Debian provide are approximately 10 megabytes, an operation that already takes $\frac{10 \text{ MB}}{3.6 \text{ mbps}} = 22.2$ seconds at the global average connection speed [4], as an example. Therefore, the download time needs to barely increase under GPM, perhaps a couple of seconds at most.

For this, I am using GPM's metadata for the full import of all packages as produced during server benchmarking. This is being compared to the metadata files on the repository from which I fetched those packages. However, since GPM misses some of the features `apt` has, such as the ability to reference multiple metadata files in the filelist[5], or to hold information about a package's description, the '`apt` (stripped)' column accurately represents the amount of these files that contains information that GPM also represents. Since `apt`'s metadata is transmitted compressed, I have also calculated the filesize if GPM's metadata object were also compressed in transit with `gzip`.

	`apt`	`apt` (stripped)	GPM	GPM w/ compression
File List	154 kB	1.2kB	850B	
Metadata	6.3 MB	5.7 MB	21 MB	7.1 MB
Packages	Identical			

Although nearly 33% of the `apt` metadata is omitted in the stripped version, it is only data that compresses well like descriptions in plain English. When recompressed, this is only an 11% decrease.

In summary, there is a filesize overhead associated with this new system. If GPM has no compression, it causes 34 seconds of additional downloading. If GPM files are compressed for transit, it is 3.1 seconds extra. I feel three seconds is defensible, however the file must be extracted on the local machine

[5]Debian has thirteen architecture ports, and multiple metadata listings for each, making a 1600-line file listing. GPM's, by comparison, has two entries.

and stored, and 20 MB is a significant increase from 6.3 MB. Using a more succinct serialized form than Java's, such as those provided by the Kryo [37] libraries, would hopefully fix this issue. Others' benchmarks indicate that the serialized files Kryo provides are only about 25% of that of Java natively [34].

4.3 Testing Summary

In the previous sections, I have completed an extensive evaluation of GPM. Firstly, I showed GPM to be more secure, and to be more resilient to key theft. Then, I performed benchmarks over the server and client. I showed that the server would be unlikely to ever take over ten minutes of processing time to import an entire day's workload. Finally, I showed that the client is fast enough for users, with dependency resolution of the majority of tested packages not exceeding one second, and download times potentially only about 10% slower if compression is implemented. On all fronts, GPM has performed above any expectations I had when producing the software, and I am proud of the data shown.

Chapter 5

Conclusion

In summary, I have completed my project to produce a more secure package manager. The result, GPM, uses a modified form of the TUF and Thandy protocols to sign the three stages of communication with a repository; the file listing, the metadata, and the package itself. Unlike other package managers, GPM is not constrained to a given package format, instead using the Strategy design pattern to import and handle packages of multiple different specifications with minimal code writing. The system is versatile, capable of being altered to support different hashing and signing algorithms to ensure it stays secure over time as cryptanalysis shows weaknesses in existing work.

5.1 Completion Of Requirements

Section 2.8 contains a series of initial requirements that were identified from the investigation and preparation. Here, I seek to show that I have met them.

1. The package manager must be capable of installing and upgrading software packages without ever altering their internal representation: This is shown from use of an `Interpreter` in Section 3.1.3 to read data from the package, rather than manipulating its contents.

2. The package manager must be capable of throwing errors in cases where any of the communication has been manipulated if no keys have been compromised: This is shown in Section 4.1.2, where attacks are explicitly stopped by the presence of certain keys, since the manipulation is then visible. In all of these cases, the package manager halts immediately for safety.

3. In the case of key compromise, the package manager must be at least as capable of identifying attacks as existing package managers: Section 4.1.1

and 4.1.2 show exactly how secure GPM is. GPM uses more keys at more stages than all other surveyed systems, and is shown to be at least as resilient to attack for exactly this reason.

4. The package manager must have the core of its code written independent of the input format: The only specialised components are an `Interpreter` on the server, detailed in Section 3.1.3, and an `APICaller` on the client, detailed in section 3.2.3.

5. The transformation must take less time to run on a package than would cause a backlog under standard import rates for distributions: This was shown during the server benchmarking in Section 4.2.1.1, where even under extreme circumstances, a day of imports could occur in 10 minutes of processing time.

6. The language of choice must not cause an excessive amount of otherwise unneeded packages: This one was provided by using Java. Java headless[1] is three Debian packages[2] with an install footprint under 30 MB. Libraries in Java are stored in the JAR archive rather than as system packages like Perl or Python, avoiding a situation whereby upgrading these libraries for other applications' sake may simultaneously break the package manager if it is not fully tested first.

5.2 Areas Needing Improvement

In order to keep the project running to schedule, edge cases and inoptimal implementations were overlooked in favour of larger amounts of progress[3]. At this point, I believe GPM has shown itself to meet its aims, however I do not believe that it is releaseable to the community for use. Given more time, the following areas require alteration:

- The inefficiency of serialization, in Section 4.2.2.2, demonstrates the greatest barrier to adoption. At the very least, it is clear metadata must be compressed, but altering the serialized format altogether using Kryo may also be required.

[1] A release of Java created to only provide command-line interaction, no GUI components are included.

[2] `jre-headless`, `jre-headless-lib` and `ca-certificates-java`

[3] Dependency resolution overran by approximately two weeks before I had to accept the drawbacks of the provided implementation.

- The lack of tree inversion, in Section 3.2.2, means users are not necessarily sure what side effects a deletion will have.
- Partial upgrades, where some packages on a system are updated and some are left at their old version, have explicitly not been included because not all distributions endorse them as an update method [28]. However, since some distributions allow it, I should add a boolean to `APICaller` and implement it only to be available if true is returned.
- The project proposal describes implementations for Debian and Red Hat packages, however this was altered to Debian and Arch Linux after a few weeks of investigation. The control files of Debian and Arch Linux packages are similar and straightforward to locate and parse. By comparison, the RPM metadata is held in a C struct prepended to the archive, in network byte order, not necessarily host byte order. The metadata itself is held as an array of dependencies, array of comparators and array of versions, where identical indices in each array indicate portions of the same dependency. The only possible way to parse this would be to pass the data over the JNI, read it in C, then pass the result back. I felt this work would take too long for the week allocated to creating `Interpreter`s. However, I feel that having an `Interpreter_RPM` before release is necessary.

5.3 Future Work

Other than fixes, looking forward on the future of GPM, I believe there are three areas for further research and implementation.

I believe that, in its current state, the client's interface is not ready. Although a seasoned command line user may be able to operate it, they are currently unable to get an increased verbosity mode, a quiet mode, and other expected features. Furthermore, a new Linux user would have significant difficulty using it. I believe the solution to this is to provide Gnome PackageKit [12] bindings. PackageKit provides an API to allow any GUI to be attached to any compatible backend consistently. This would then allow a third-party package manager GUI to sit over the top, potentially the exact same user interface as for the package manager GPM replaces in an upgrade scenario. This would give a seamless transition and not cause any learning curve for users.

Secondly, I believe a system should be made to prompt signers more effectively as to what needs to be signed. An application could scour the interim directory, check which files do not have signatures in incoming, and present to packagers

only the phases that they have keys they can sign, all metadata about that package, and allow them to download and inspect the contents. Maintainers could then one-click 'sign and upload' their support.

Finally, I believe investigation could be done into applying GPM over the top of an existing repository structure. This would require additional `Interpreter` function calls to identify where a package should be located in a repository, since it would have to follow the conventions of the package manager it is attempting to coexist with. However, the result would then be one file structure from which an old package manager could update, since it would not know to look for the new files, and one from which this new package manager could also run, since it would just also check those additional signatures. This would avoid storing the packages twice in two different repositories on the same host, which may otherwise duplicate required storage space during a transition.

5.4 Closing Thoughts

I believe I have demonstrated that TUF, and by extension GPM, are viable for real world use. The protocol itself is more secure than those already used, and GPM implements it with the potential for others to develop upon the foundations I have laid. GPM fills a real need for secure software updates on Linux machines, from stable servers to personal computers in untrusted networks, and does so with a core design that can be contributed to by all those who write `Interpreter`s, gradually forming an increasingly stable and trusted code base. It also encourages the creation of new package formats, using only a few lines to specialise GPM for it. I am incredibly enthusiastic about the potential of this, and intend to remove the 'private' restriction from my GitHub repository and open-source the code under a permissive license after the academic year is over so any interested parties can contribute and assist.

Bibliography

[1] Bill Allombert. Debian Popularity Contest. http://popcon.debian.org.

[2] Edward C. Bailey. Maximum RPM: Taking the Red Hat Package Manager to the Limit, Appendix A: Format of the RPM File. http://www.rpm.org/max-rpm/s1-rpm-file-format-rpm-file-format.html.

[3] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for Key Management – Part 1: General (Revision 3). Technical Report 800-57, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2012.

[4] David Belson, editor. *The State Of The Internet, Volume 6*. Number 3. Akamai Technologies, 2013.

[5] Ferry Boender. Dependency resolving algorithm. http://www.electricmonk.nl/log/2008/08/07/dependency-resolving-algorithm/.

[6] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. Attacks on Package Managers. http://www.cs.arizona.edu/stork/packagemanagersecurity/.

[7] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. Other Attacks. http://www.cs.arizona.edu/stork/packagemanagersecurity/otherattacks.html.

[8] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. Stork Package Manager. http://www.cs.arizona.edu/stork/.

[9] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. A Look in the Mirror: Attacks on Package Managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 565–574, New York, NY, USA, 2008. ACM.

[10] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. Package Management Security. Technical Report 02, Department of Computer Science, University of Arizona, 2008.

[11] Timothy G. Griffin. *Databases*, pages 7, 11, 96. Lecture notes, 2013.

[12] Richard Hughes. What is PackageKit? `http://www.freedesktop.org/software/PackageKit/pk-intro.html`.

[13] IgnorantGuru. ArchâĂŹs Dirty Little Not-So-Secret. `http://igurublog.wordpress.com/2011/02/19/archs-dirty-little-notso-secret/`. Posted 2011-02-19.

[14] Ian Jackson, Christian Schwarz, and David A. Morris. *Debian Policy Manual v3.9.5.0*, pages 9–14, 23–33, 41–48, 99–100. The Debian Policy Group, 2013.

[15] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, pages 261–271. Addison-Wesley, 3rd edition, 2000.

[16] Markus Kuhn. *Security I*, pages 75–81. Lecture notes, 2013.

[17] Markus Kuhn. *Security II*, pages 59, 85–87, 89. Lecture notes, 2014.

[18] Ian Leslie. *Operating Systems I*, pages 121–127, 132–142. Lecture notes, 2011.

[19] Eric Lippert. I'm putting on my top hat, tying up my white tie, brushing out my tails – in that order. `http://blogs.msdn.com/b/ericlippert/archive/2004/03/16/90851.aspx`.

[20] Debian-WWW maintainers. New Packages in "sid". `https://packages.debian.org/sid/newpkg`.

[21] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Fabio Mancinelli, Ralf Treinen, and Stefano Zacchiroli. Debian Quality Assurance. `http://edos.debian.net/`.

[23] J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[24] Nick Mathewson. Thandy: Automatic updates for Tor bundles. `https://git.torproject.org/checkout/thandy/specs/thandy-spec.txt`.

[25] Ciaran McCreesh, Petteri Räty, and Ulrich Müller. Gentoo Development Guide: Ebuild Writing. `http://devmanual.gentoo.org/ebuild-writing/index.html`.

[26] Allan McRae. Pacman Package Signing âĂŞ 4: Arch Linux. `http://allanmcrae.com/2011/12/pacman-package-signing-4-arch-linux/`. Posted 2011-12-17.

[27] Allan McRae, Dan McGee, Dave Reisner, Judd Vinet, Aurelien Foret, Aaron Griffin, Xavier Chantry, and Nagy Gabor. Pacman Manual, PKGBUILD(5). `man PKGBUILD` on an Arch Linux system, or `https://www.archlinux.org/pacman/PKGBUILD.5.html`.

[28] Daniel Micay. Arch Linux Wiki: Pacman - Partial Upgrades Are Not Supported. `https://wiki.archlinux.org/index.php/pacman#Partial_upgrades_are_unsupported`.

[29] Alan Mycroft. *Optimising Compilers*, pages 36–37. Lecture notes, 2013.

[30] Nicole Perlroth. Government Announces Steps to Restore Confidence on Encryption Standards. `http://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/`. Posted 2013-09-10.

[31] Frans Pop. debtree – package dependency graphs on steroids. `http://collab-maint.alioth.debian.org/debtree/`.

[32] Andrew Rice and Alastair Beresford. *Further Java Workbook 2*, pages 1–3. Lecture notes, 2012.

[33] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable Key Compromise in Software Update Systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 61–72, New York, NY, USA, 2010. ACM.

[34] Eishay Smith. JVM Serializer Benchmarking. https://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking.

[35] Frank Stajano. *Algorithms II*, pages 7–39, 43–45, 45–46, 57–58. Lecture notes, 2012.

[36] Frank Stajano. *Security II*, pages 11, 39. Lecture notes, 2014.

[37] Nathan Sweet. Kryo. `https://github.com/EsotericSoftware/kryo`.

[38] Robert Watson. *Concurrent Systems*, pages 3.16–22, 3.29–4.04, 4.16–4.17, 4.25. Lecture notes, 2013.

[39] Nathan Willis. Arch Linux and (the lack of) package signing. `http://lwn.net/Articles/434990/`. Posted 2011-03-23.

[40] Tim Wood. JewelCli. `http://jewelcli.lexicalscope.com/`.

Appendix A

Excerpt of Debian Policy Manual Chapter 5

When comparing two version numbers, first the epoch of each are compared, then the upstream_version if epoch is equal, and then debian_revision if upstream_version is also equal. epoch is compared numerically. The upstream_version and debian_revision parts are compared by the package management system using the following algorithm:

The strings are compared from left to right.

First the initial part of each string consisting entirely of non-digit characters is determined. These two parts (one of which may be empty) are compared lexically. If a difference is found it is returned. The lexical comparison is a comparison of ASCII values modified so that all the letters sort earlier than all the non-letters and so that a tilde sorts before anything, even the end of a part. For example, the following parts are in sorted order from earliest to latest: ~~, ~~a, ~, the empty part, a.

Then the initial part of the remainder of each string which consists entirely of digit characters is determined. The numerical values of these two parts are compared, and any difference found is returned as the result of the comparison. For these purposes an empty string (which can only occur at the end of one or both version strings being compared) counts as zero.

These two steps (comparing and removing initial non-digit strings and initial digit strings) are repeated until a difference is found or both strings are exhausted.

Note that the purpose of epochs is to allow us to leave behind mistakes in version numbering, and to cope with situations where the version numbering scheme changes. It is not intended to cope with version numbers containing strings of letters which the package management system cannot interpret (such as ALPHA or pre-), or with silly orderings.

Appendix B

Excerpt of Portage Documentation

When implementing a database backend for portage, the versions of packages need to be directly comparable. An integer sized 64 or 128 bit is the way to go. This page describes a scheme to encode the original version strings into compacted integers.

The format of version strings used in ebuilds can be categorized as:

```
version ::= <basic> <suffix>
basic   ::= <number> {'.' <number>} [char]
suffix  ::= ['_' <symbol> [number]] ['-r' <number>]
number  ::= an arbitary non-negative number
symbol  ::= one of "alpha", "beta", "rc", "pre", "p"
char    ::= 'a' to 'z'

NOTE: {'a'} means 'a' can appear none or arbitary times.
```

Provided their LSBs (least significant bits) are aligned, they are still comparable. So we can align the LSB of numbers to some fixed positions. If we use 4 bits per number we can get:

```
1.0      => 0001 0000 0000 0000
1.0.0    => 0001 0000 0000 0000
1.1      => 0001 0001 0000 0000
3.4.1.10 => 0011 0100 0001 1010
```

This method has a big limitation that the size of an integer and the number of integers are fixed. To overcome these limitations, we can prepend a length marker before the number and use the following encoding:

```
0xx for 0-3
1xxxxx for 4-31
```

Losing one bit for a number, now we can encode more complex versions:

```
1.0         => 001 000               => 0010 0000 0000 0000
1.0.0       => 001 000 000           => 0010 0000 0000 0000
1.1         => 001 001               => 0010 0100 0000 0000
1.3         => 001 011               => 0010 1100 0000 0000
1.4         => 001 100100            => 0011 0010 0000 0000
2.6.20      => 010 100110 110100     => 0101 0011 0110 1000
3.0         => 001 000               => 0010 0000 0000 0000
3.1         => 001 001               => 0010 0100 0000 0000
3.2.3.3     => 011 010 011 011       => 0110 1001 1011 0000
3.2.3.3.1   => 011 010 011 011 001   => 0110 1001 1011 0010
```

The point here is to use Huffman coding. We just have to make sure the length markers are comparable. In other words, markers for shorter length must be less than markers for longer length, when aligned to the MSB. Also, no length marker can be a prefix of another, or we can't distinguish them.

For an arbitary version string, we just encode each part of the version and concatenate them from MSB to LSB using:

```
01000xxxxx          a-z
01xxx except 01000  0-6 (small version numbers, 0 -> 01001, etc.)
100xxxxxx           7-70
101xxxxxxxxxxxx     71-4166
110{27x}            4167-134221894, for DDMMYYYY
1110{40x}           134221895-1099645849670, for timestamps
1111{48x}           1099645849671-282574622560326, for anything else
```

As 0-6 are already encoded as 01xxx, 100000000-100000100 are illegal and so 100000000 encodes 7 allowing an upper bound 6 greater than usual (64+6=70). The same trick applies in every following range.

Suffixes are also given their own encoding. For example, 'alpha', 'beta', 'pre' and 'rc' should be considered to be before the real version numbers, and therefore their value must be less than the usual version number.

```
000 prerelease
001 general release
```

Then if there are suffixes, we encode the suffixes and append the sequences according to this table:

```
0 + NUM     ebuild has a revision number as a variable size integer.
10xxx       ebuild has a suffix but no revision.
11xxx + NUM ebuild has both a suffix and a revision.
```

The suffix is then from:

```
000 _alpha
001 _beta
010 _rc
011 _pre
```

This produces a final value that can be compared from MSB to LSB to version check the ebuild.

Appendix C

Correct attempt to install ‘kernel’ and ‘bc’

```
root@archiso ~ # pacman -S linux
resolving dependencies...
looking for inter-conflicts...

Packages (1): linux-3.10.10-1

Total Download Size:   52.84 MiB
Total Installed Size:  69.47 MiB
Net Upgrade Size:      4.49 MiB

:: Proceed with installation? [Y/n]

root@archiso ~ # pacman -S bc
resolving dependencies...
looking for inter-conflicts...

Packages (1): bc-1.06-8

Total Download Size:   0.08 MiB
Total Installed Size:  0.18 MiB

:: Proceed with installation? [Y/n]
```

Appendix D

Hijacked attempt to install ‘kernel’ and ‘bc’

```
root@archiso ~ # pacman -S linux
warning: downgrading package linux (3.10.9-1 => 2.6.32.61-1)
resolving dependencies...
looking for inter-conflicts...

Packages (1): linux-2.6.32.61-1

Total Download Size:   49.87 MiB
Total Installed Size:  68.93 MiB
Net Upgrade Size:      3.95 MiB

:: Proceed with installation? [Y/n]
```

If the package had not already been installed at a greater version, the warning on the first line would not have been generated since the package manager would have no means by which to notice this inconsistency.

```
root@archiso ~ # pacman -S bc
resolving dependencies...
:: There are 3 providers available for libgl:
:: Repository extra
   1) mesa-libgl  2) nvidia-304xx-utils  3) nvidia-libgl

Enter a number (default=1): 1
looking for inter-conflicts...
```

```
Packages (167): aalib-1.4rc5-10  alsa-lib-1.0.27.2-1 ...
                cdparanoia-10.2-5  celt-0.11.3-2 ...
                ...
                ...
                sqlite-3.8.2-1  taglib-1.9.1-1 ...
                xproto-7.0.25-1  xvidcore-1.3.2-3  bc-1.06-8

Total Download Size:   73.43 MiB
Total Installed Size:  372.13 MiB

:: Proceed with installation? [Y/n]
```

If I had not chosen a package which depended on a virtual package, one with multiple options for what can provide it (`libgl`), then the user would not have had to type a number.

Appendix E

Excerpts from Interpreter_DEB

Extracting metadata into a string named 'result' by unwrapping the '.ar' archive, then the '.gz' archive, then the '.tar' archive:

```
String result = null;
RepoPackage pkg = new RepoPackage();
boolean success = false;
try {
    InputStream is = new FileInputStream(f);
    ArArchiveInputStream aais = (ArArchiveInputStream) new
                  ArchiveStreamFactory().createArchiveInputStream("ar", is);
    ArArchiveEntry aae;
    outerloop:
    while ((aae = (ArArchiveEntry) aais.getNextEntry()) != null) {
        if (!aae.isDirectory() && aae.getName().equals("control.tar.gz")) {
            TarArchiveInputStream tais = new TarArchiveInputStream(
                                                new GZIPInputStream(aais));
            TarArchiveEntry tae;
            while ((tae = (TarArchiveEntry) tais.getNextEntry()) != null) {
                if (!tae.isDirectory() && tae.getName().equals("./control")) {
                    StringWriter w = new StringWriter();
                    IOUtils.copy(tais, w);
                    result = w.toString();
                    success = true;
                    break outerloop;
                }
            }
```

```
        }
    }
} catch (ArchiveException | IOException e) {
    throw new CannotContinueException("The archive was not found to
                        contain metadata corresponding to a .deb package.");
}
```

Extracting dependencies from a comma separated list 'ds' with optional version comparison or optional architecture exclusions, with the regular expression broken over multiple lines only to ensure it is all visible on the page:

```
String[] split = ds.split(",");
int epoch = 0;
for(String eqdeps : split) {
    String[] split2 = eqdeps.split("\\|");
    for(String dep: split2) {
        // This will loop multiple times only if the OR symbol is used.
        // The epoch will be the same for each ORed package.
        Pattern p = Pattern.compile("^\\s*([^\\s\\(\\)\\[\\]]+)
                                            (\\s+\\(([^\\s]+) ([^\\)]+)\\))?
                                            (\\[([^\\s\\]]+)\\])?\\s*$");
        Matcher m = p.matcher(dep);
        m.find();
        // 1. Name
        // 3. Comparator
        // 4. Version
        // 6. Architecture Restrictions
        for(int i = 0; i<=m.groupCount(); i++){
            System.out.println(i+" "+m.group(i));
        }
        // Finds the correct enum element to match the symbol (> -> GT)
        Dependency.comparator c = toComparator(m.group(3));
        Dependency d = new Dependency(m.group(1), c,
                                    m.group(4), epoch, m.group(6));
        switch (t) {
            case depends:
                r.setDepends(d);
                break;
            case breaks:
                r.setBreaks(d);
```

```
                break;
            case provides:
                r.setProvides(d);
                break;
        }
    }
    epoch++;
}
```

Appendix F

Project Proposal

Computer Science Project Proposal

Mitigating the Effects of Software Repository Key Compromise

S. Hollingshead, Queens' College

Originator: S. Hollingshead

25th October 2013

Project Supervisor: Daniel Thomas

Director of Studies: Dr. R. D. H. Walker

Project Overseers: Dr. A. C. Rice & Dr. T. G. Griffin

Introduction

In Linux, the predominant method of installing and updating software is through distribution-provided repositories, the download taking place using a package manager on the user's machine. The fact these packages generally comprise of code compiled externally, with the intention of being installed locally, requires a large amount of trust from end users. In the case of trusting the source code, it is possible for a user to examine it. The user also implicitly trusts the packager, since they must have trusted the distribution and the decisions it makes, otherwise they could change to another distribution. Trusting the communication between computer and repository, however, is fraught with potential issues and a needless source of exploits.

Academic papers [9] [33] outline methods that malicious actions, leveraged from a man-in-the-middle position, can interfere with the installation of packages. The malicious actions rely on fundamental mistakes in the interaction with the repository, such as failure to require signed metadata, or failure to limit the scope of damage a single stolen key can do. The only solution to this is to alter the protocol and repository files in a backward-incompatible manner. There must not be a fallback mode, otherwise this would continue to be exploited with the exact same methods.

My project, therefore, aims to consider an alternative to waiting for each existing package manager to rewrite their protocols from scratch. I will provide a generic package manager and repository framework capable of handling the secure retrieval of any object that can loosely be considered to be a package[1]. The project's result could be used as a temporary replacement for existing tools, where a distribution would provide a separate copy of their files signed adequately. It could be proposed as a replacement for the existing package management systems, or a base upon which a new distribution could create their own package format and handle package retrieval.

Starting Point

A large amount of work in the field already exists:

- Package formats and installation of packages are already well defined, and I do not seek to interfere with this. My only concern is in secure package

[1] A single file capable of describing other 'packages' it depends upon, or conflicts with, and version information to indicate when it is a candidate for an upgrade.

retrieval and upgrade, dependency resolution from metadata, parsing and handing off the files to the underlying system[2].

- Similarly, I do not intend to write my own implementations of any key generation algorithm, since the existing work in this area is far more well tested than I could hope to write myself.

- Test cases for attacks to consider can be found in papers [9] [33], and ongoing research projects are continuing to generate new reports [10] and information on their related project homepages [6]. I can first implement the features to fix these, then compare and contrast with the existing implementations.

- Package dependencies can already be represented as a satisfiability problem, and existing work already has conversion from .deb and .rpm packages to this generic format [21].

- I can perform dependency resolution either using a satisfiability solver [23] or by using other existing algorithms [5].

- Comparing existing package managers and the directory structure of their repositories can provide guidance for mine and indicate special cases I may otherwise have not foreseen.

Resources Required

I intend to use my own laptop, an Acer 5750G running Arch Linux. This is to allow me to run my own local web server for repository hosting, and to give me far more control over configurations than the MCS can.

A git repository will be created to hold the project, located in Dropbox. The files will be pushed to a private GitHub account. If my computer fails, most of the files can be restored to the MCS. The SRCF and DTG are both willing to run the repositories in this case.

I will need to download multiple versions of different distributions' packages to test both package install and package upgrade. Old packages are provided by both CentOS[3] and Debian[4].

[2]The project intends to implement an `APT`-like, or `yum`-like tool, not `dpkg` or `RPM`.
[3]http://vault.centos.org/
[4]http://snapshot.debian.org/

Work to be done

The project breaks down into the following sub-projects:

1. The production of a well-documented specification of the communication between the client and the repository.
2. The construction of a mechanism for signing keys as trusted using a root key, and designating them only valid for signing certain files.
3. The creation of a repository management tool, able to handle the inflow of new packages, facilitate signing, and promote them to be publicly downloadable only when given signing criteria are met.
4. The addition of a bulk-import tool, able to mass-import and mass-sign packages from a prior system.
5. A client side tool which can communicate with the repository, securely retrieve packages and their dependencies, and resolve conflict with existing files on the system by considering required uninstallations.

Success Criteria for the Main Result

Since the project's inception was to reduce the severity of attacks outlined in literature as causing package managers to operate in potentially dangerous manners other than those expected when issued certain commands, the success criteria must be based on collecting a list of all attacks that it is possible to locate.

For any install or upgrade command issued to the package manager, it must perform the expected action, resolve dependencies, validate the security of the connection, download the packages, and hand them off to be installed. When any attack is performed from the collected list, the system must at least resist attacks to an identical level to that of the existing systems[5], although the target is to evaluate how many more repository maintainers must be compromised before the given attack can succeed.

Since this tool is considered to be generic, a demonstration of the exact same successes and failures above occurring on two Linux distributions, each with a different package format, would be adequate to show success.

[5] As with every system dependent on multiple keys, sufficient thefts would appear as a valid signed chain, but the ease of stealing such an amount of keys without their revocation is highly unlikely, and is nonetheless far more difficult than existing man-in-the-middle attacks.

Possible Extensions

Adding a method to make roles more or less granular. Initial implementations [33] use very fine permissions, a key corresponding to a package. It would be good to extend this to teams, such as the GNOME team, each owning a key that could sign any package within their collective remit.

The downside to adding roles and signing file listings is the requirement for long hashes in file listings, increasing the filesize greatly despite the fact only a few full lines change at once. Investigating the potential to provide signed linediffs for delta updates of metadata as an inherent feature could prove useful.

Another useful feature would be the ability for signatures to be added to an existing repository's hierarchy, relying on the fact the existing insecure package manager would not look at the signatures, but this new tool would. This would allow a transitional rollout without requiring separate copies of packages in two repositories.

Timetable

Planned starting date is 24/10/2014.

1. **2013M Wk. 3–4:** Create extensive list of existing vulnerabilities from academic sources. Collect information about existing repository structures and viable signing algorithms. Collect detailed information about the format of .deb and .rpm packages. Store all such information locally, so it cannot disappear before the end of the project.

2. **Wk. 5–6:** Produce repository protocol document. Create preliminary repository to specification. Import in packages, signed manually.

3. **Wk. 7–8:** Create both tool to give keys authority, and repository tool for accruing package signatures and promoting relevant packages into main repository.

4. **Vacation:** Implement most of the package manager, including communication, key validation, metadata analysis, downloading, and system package version checking for upgrades. Dependency resolution may not be complete, particularly in edge cases such as needing to uninstall packages.

5. **2014L Wk. 0–1:** Finish package manager, audit system to ensure correct operation, tweak as needed. Produce progress report.

6. **Wk. 2–3:** Begin evaluation. Examine what added resilience to key compromise the new software offers over existing package managers, and evaluate any time trade-off in the new system. Perform a stress test to examine potential worst-case performance.

7. **Wk. 4–5:** Based on results, make improvements in inadequate areas and re-evaluate cases until considered acceptable. Prepare the framework of the dissertation to ensure acceptable content exists for all sections.

8. **Wk. 6+:** These weeks are intentionally left empty to allow project overrun. After this period, the project must be feature-complete.

9. **Vacation:** Entirely complete and proofread dissertation with feedback. Easter will be solely for examinations.