James Tapsell, 2017

Final Year Project Report

Full Unit - Final Report

Securing applications from commit to execution for Android Apps and JAR files

James Tapsell

A report submitted in part fulfilment of the degree of

MSci in Computer Science (YINI)

Supervisor: Jorge Blasco Alis

Department of Computer Science
Royal Holloway, University of London

April 18, 2018

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 5304

Student Name: James Tapsell

Date of Submission: 23/03/18

Signature: 

Table of Contents

Abstract

A lot of applications are now signed to prevent tampering with them, this usually consists of a created executable that is signed by the company or group releasing some software, allowing the end user or their system to verify the source of the software. The signature usually covers the executable code, but also the meta-data, and the resource files, allowing prompts to show who signed a piece of software, and stopping manipulating the resource files to, for example, swap out error messages for security issues with benign looking messages.

For some classes of software (device drivers on windows for example), this is a mandatory step, as 64 bit editions need settings changes to allow the use of unsigned drivers[1]. This was introduced to try to prevent malware from getting deep into the system, as once it is at such a layer not only does it have the ability to take over the system, but also hides itself from any software looking for it.

Unfortunately this is just moving the insertion of malicious payloads to before the signature, This kind of attack generally works by injecting malicious code into the repositories used to build the executables that are then signed for release holding their payload.

Trying to defend against this attack is difficult, but there are some things that can help. One example of this is that git allows for signed commits[2], which can help to prove the author of the commit. This does introduce the issue of validating all commits are signed though, and the storage of keys for developers, although yubikey style devices [3] or smartcards can help reduce the difficulty of this.

This project looks to create a proof-of-concept defence against such an attack with the following features:

1. Developers should not need to install new software on their own devices if possible (although this may be necessary in some cases, this should be avoided if possible).
2. The releases are signed in such a way that end user devices can be sure who committed and released an application, and there should be no breaks in the chain of custody of the source or artefacts created from it.
3. The releases should not be able to be tampered with in an undetectable way.
4. It should not be possible to include an entity (person, group or company) in the list of people who released an application without their permission.

This project does not attempt to solve the issue of rogue developers, not for malicious commits signed by a developer, as these are beyond the scope. It also does not deal with key management for developers, as such tools already exist.

As I would like to gain more experience using Kotlin, and JGit is written in Java, I will write the majority of the code for this project in Kotlin, which is cross compatible with Java (Java code can access all methods and fields of Kotlin code and vice versa, and they can run together in the same JVM). This is possible as Kotlin compiles to Java bytecode, rather than having its own format. Kotlin is also a first class language on Android[4], so the apps that are needed can be developed in it too, without needing to be translated to Java.

By having code that can run on both computers and Android devices, I can make simple versions of the systems, package them as JARs and test them locally, and once they work I can integrate the logic into an Android app, as an APK file. By using this 2 stage pattern, I can test locally, which is more convenient than needing to use a device for testing, and also as a computer can have more resources I can run tests that are more exhaustive in less time.

Chapter 1: **Project Specification**

1.1 Aims

The idea of this project is to create a system that secures executables, with a trust chain that extends all the way from the commit through to the release of the executable. This is designed to defend against the injection of malware at any stage, including to the source control system, or to the built executable.

1.2 Background

Source code is usually compiled to create executables, and the resulting executables are then distributed. This system works well, but it is not very secure, as malware can be inserted into the software at many points, including into the source before a commit, into the source control by modifying a commit, into the source control as a fake commit, added during the build, or added to a genuine build. It can also be difficult to work out at which stage malware was inserted into the system, an example of this was during the CCLeaner attack[5], where it was originally thought that the malware was added to the source, but it was later discovered that the build server had been compromised.

1.3 Deliverables

The main core of the project consists of the JAR part, which has the following deliverables:

- Signature design
  A concete design for the signatures.
- Jar Signer
  An system which can sign JAR files.
- Jar Validator
  A system which can validate signed JAR files.
- Jar Launcher
  A launcher which verifies the launched JAR is signed.

1.4 Extensions

The android extension of the project covers the following deliverables:

- Demo App
  A simple test app.

- App Signer
  A system to sign apps.
- Signed App
  A signed app, using the above deliverables.
- App Validator
  A validator which checks an app is signed and the signature is valid.
- Validating Installer
  An installer which only allows apps which pass the above validation to be installed, and shows metadata before installation.

Chapter 2: **Introduction**

2.1 Motivation

The idea of this project is to create a system which defends against injection of malware into executables. There have been many cases of malware being injected into tools which are then distributed. This class of malware is seeing increases, and so defending against this is important, as unless defences are put in place the situation will only get worse.

2.2 Objectives

This project looks to create a proof-of-concept defence against such an attack with the following features:

1. Developers should not need to install new software on their own devices if possible (although this may be necessary in some cases, this should be avoided if possible).
2. The releases are signed in such a way that end user devices can be sure who committed and released an application, and there should be no breaks in the chain of custody of the source or artefacts created from it.
3. The releases should not be able to be tampered with in an undetectable way.
4. It should not be possible to include an entity (person, group or company) in the list of people who released an application without their permission.

2.3 Limitations

SHA1 limitations
: Git only supports SHA1 commit IDs, despite Shattered breaking SHA1[6], which means that for identifying a commit this project only allows the use of SHA1. There is some argument [6][7] about if this attack is practical against a git repository, but as a defensive measure the systems designed in this project are designed to be able to be used with the output of different hash algorithms, so it will support the new algorithms as soon as git does.

Mallicious developers
: This project only deals with trying to verify that the author of the code is the entity that the user believes it is, and that they can trust them, it does not deal with trying to make sure the author is not writing malicious code.

Build Server compromise
: We assume the build server is a secure environment, as otherwise we can't guarantee our code is running, as the build system could be replaced with one that signs a trojan with the securely generated chain.

Trust Establishment
: This project uses Keybase as a secure link from identity to keys. This can be used for

either a social user, or a website, which can relate to a single person, a group or another kind of entity like a company.

The use of keybase is abstracted away, so in theory any other trust mechanism could be used, such as one of:

- The local GPG keychain
- A keychain relayed over a secure medium (such as via a diplomatic bag, or pre-installed into a device)
- TXT records relayed via DNSSEC[8] (for linking a key to a website)

2.4 Structure of report

- Background Theory
  This describes the background information which this project uses.
- Professional Issues
  This describes the professional issues faced during this project.
- Software Engineering
  This covers the software engineering processes used in this project.
- Running the software
  This explains how to run the code produced by this project.
- Issues faced
  This describes the issues faced during this project, and explains the impact of them.
- Software completed
  This enumerates the pieces of software that have been completed so far.
- Usage
  Describes how I have used the base of this project myself, and the issues faced while trying to do so.
- Git Internals
  Describes the git internals.
- GitHub
  Describes GitHub.
- Evaluation
  Evaluates the project against the aims set out.

Chapter 3: **Background Theory**

3.1 Git and GitHub

Git is a source control system, which is based on a merkle tree. This is the same kind of tree used for bitcoin[9], and in git this means that if you trust the hash of the last commit you can trust that all of the ones before it have not been modified, assuming the hash function is secure (all objects in git are identified by their SHA1 hash, which may cause issues, although no attack against git specificaly has been made public at this point, and git uses the hardened version of SHA1 now).

Another feature of git is that it is distributed, so any machine can have a full working copy, this is good, as it allows for offline working, and speeds up working on slow connections, where repeatedly downloading branches would slow down workflows. There is a problem with faked commits however, as there is no centralised node which can be used to link a commit to an account, leading to the ability to spoof commits and push them to a repository.

GitHub is a git repository hosting provider, which has a web UI, and supports showing signature status of commits and tags (branches cannot be signed).

3.2 Keybase

Keybase is a service which allows for linking keys (mainly PGP) and online identities, in a way which can be proven without having to rely on Keybase.

For online accounts, this is normally created by publicly posting a signed proof, like the one I posted to link my Facebook to my Keybase[10]. These have the effect of showing that the person who controls an account on that site also controls the keybase account. As the proof is signed neither the third party nor Keybase cannot link people to accounts that they do not control, as they would need to have access to both the user's key, and also their keybase account to link the signed proof to their account.

There is theoretical attack where Keybase could delete a user's merkle tree, and then create a new fake user with different accounts, in an attempt to spoof someone, but this is defended against because people who *follow* a person on Keybase pin their tree, so if a new one appears all of the old followers would get a warning that the new tree is not the same one they pinned. This could be defeated by a malicious client, but as the client is open source users can check the code, and can build it themselves if they do not trust the pre-built executables.

The online identities supported are:

- Facebook
- GitHub
- Twitter
- Reddit
- DNS

- HTTPS
- HackerNews
- Bitcoin
- ZCash

3.3 Trust

This project allows for trusting the releasing entity for a given project, this means that a company can, for example lock down their machines, only allowing software from approved vendors to be executed. There are a set of properties that may want to be asserted about an executable:

- Who released an executable?

  This would be used to assert that a trusted company made a piece of software such as a driver, rather than it being a fake version, which may be used for malicious purposes.

- Which developers have been involved in the creation of an executable?

  This is used for example to say who made an open source library, so that if a malware author injects code into a project they cannot then claim to have not done it.

- That a developer has not been involved in the creation of an executable

  This would be used so that a malicious party cannot create malware, and then claim it was created by an uninvolved party instead of themselves.

- That an executable has not been modified since its release.

  This is used to check that malware has not been injected into an executable after it was built.

3.4 Digital Signatures and PKI

Digital signatures allow an entity to sign or encrypt a message with a public key. This can be used to sign code, so that the entity which released the code can be asserted. It can also be used to chain trust, so you can trust a root certificate, which can then trust intermediates, which can then trust leaf certificates which are used. This is how certification authorities are able to sign certificates for sites which browsers will trust.

3.4.1 GPG

GPG is a tool which allows for signing and encrypting of messages, and can armor the resulting message, making it suitable for sending via email.

3.5 Code signing

Code signing is the process by which an entity can assure that an executable was released by them, and has not been modified after that point. There are many uses of this in real world situations, but some of the more common are Android apps, IOS apps, Authenticode and signed JARs.

3.5.1 Android Apps

For android apps a certificate is used to sign an application, but the certificate does not need to be part of a trust chain, it is just trusted the first time it is seen, and while the app is installed this certificate cannot change. This does reduce the overhead needed to create an app, as the developer does not need to get their certificate signed, but it has 2 main downsides, firstly the app could be a fake, and as long as it is not already installed the fake certificate will be trusted, and second if the certificate is lost the app can no longer be updated, and to make a new version the old version must be uninstalled and an app with a new name and certificate must be released.

3.5.2 Authenticode

Authenticode works similarly to Android apps, but the cerificate must be signed and trusted. This causes issues for developers of open source projects, as code signing certificates usually cost at least $100 a year, and can be a lot more expensive. Another issue is that the amount of CAs that are trusted for code signing and issue code signing certificates is relatively small, so there is not much competition.

3.6 Reproducible builds

There are multiple projects which are trying to make their builds reproducible, such as debian[11], and even some which aim to allow multiple parties to sign their build result and then use this to try to validate the build, one example of which is gitian[12].

While these try to solve the same issue I am trying to solve, they rely on the end user being able to compile the project, which in some environments may not be possible (embedded devices and mobiles for example). It is possible a future project could look at a hybrid of the 2 strategies, with an aim of getting the best of both worlds, where multiple releasing entities could sign a reproducible build with the signature scheme designed in this project.

3.7 Proprietary Software

Proprietary software adds difficulty to this project, as the default assumption is that the user wants to check that the executable running on their machine was produced from only the code which they trust. In a proprietary project, however, only the releasing entity can be trusted, as the end user is not allowed access to the original source code, only the executable that is produced after compilation, possibly obfuscation, and then packaging.

Chapter 4: **Professional Issues**

There were very implies more then one but your list only have one issue professional issues raised by this project, as the only aim is to stop malicious code being injected into a binary, the main one was prevention of legal insertion of malware.

This is a difficult issue, as computer systems cannot tell the difference between good and bad software, so an injected bit of software added to a download by a malicious actor cannot be differentiated from a banking trojan added to a banking app. This leads to issues, particularly in jurisdictions where groups are either allowed or required to be able to inject code.

One example of such an instance was the request by the FBI that Apple design a backdoor to allow reading of encrypted devices[13].

The problem with a secure system like the one designed in this project is that it would block such injection, leading to problems with it being banned in certain jurisdictions. This would lead to fragmentation, which would increase the attack surface for malicious actors.

Another example was the use of SHA1, which git mandated, as it is the only hash function that git currently supports. Although this at the moment is secure, throughout the project I had to bear in mind that SHA1 has been broken and that as a result I had to make the code able to be changed when git changes to a new hash algorithm. However, git currently uses a hardened version of SHA1, so it is possible that this will not happen until a version of the attack found for SHA1 is found which cannot be hardened against.

The main reasons that git does not currently support multiple hash functions are the extra overhead of supporting 2 different hashes, and also the difficulty of maintaining backwards compatability in a way that is also secure. This is important because even if the new implementation is secure, if the old implementation is still exploitable people will just attack it instead. This was how an exploit against SSH worked, where connections using the more secure V2 protocol would be downgraded to the V1 protocol, which was then attackable.[14].

Chapter 5: **Software Engineering**

5.1 Patterns

5.1.1 Singleton

The use of Kotlin simplifies using patterns, the following classes objects, which in Kotlin are implemented as thread safe singletons[15].

- uk.co.jrtapsell.fyp.gpgWrapper.GpgWrapper
- uk.co.jrtapsell.fyp.processTools.extensions._Thread
- uk.co.jrtapsell.fyp.keybaseConnector.Keybase
- uk.co.jrtapsell.fyp.keybaseConnector.dataclasses.KeybaseMapper

Listing 5.1: Reduced example singleton.

```
object Keybase {
    // Code removed for simplification
}
```

This is an example of an `object` in Kotlin, which is how singletons are written in Kotlin.

5.1.2 Factory

Using companion objects in Kotlin factories can be implemented in relatively small amounts of code, one example of how this works is the companion object of uk.co.jrtapsell.fyp.gitWrapper.data.Commit, which creates Commits from their JSON representation.

Listing 5.2: Reduced example of a factory.

```
data class Commit {
    /* Reduced to simplify.*/
    companion object {
        private fun convert(/* Reduced to simplify.*/): Commit {
            /* Reduced to simplify.*/
            return Commit(
                    Hash.fromString(commitHash),
                    parents,
                    subject,
                    author,
                    committer,
                    signIdent)
        }

}
```

5.2 Automation

This project makes heavy use of automation, all of the modules have automatic tests which are called on every commit by travis, which allows for validation and detection of regressions.

Normally, this would be used with GitHub protected branches to prevent broken code getting onto master, but as the merge commits need to be signed, this cannot be used, in the final project there will be a web UI for GitHub which allows for fast forward merging only passing commits, which will prevent broken code ending up on master, without requiring its own signing key, as branches are not signed in git.

5.3 TDD

I used TDD throughout the project, with tests being written first and then the code being written to pass the tests. This meant that it was easy to tell if the code was broken or not. I also used travis to allow commits on GitHub to be signed.

5.4 Signed Commits

All commits to the repository were signed with a GPG key stored in a smart card, which allows me to be certain no party could inject fake commits into the GitHub repository without somehow getting them signed with my key, which should not be possible, unless they can create a commit with a matching hash. This would be important in a real environment as attackers have been known to attack suppliers in order to get access to their clients.

Chapter 6: **Running the software**

6.1 Getting the software

The following repositories are related to the project:

- `https://github.com/jrtapsell/android_app_verifier`
  This containes the majority of the code.
- `https://github.com/jrtapsell/KGit`
  This contains the Kotlin implementation of git.
- `https://github.com/jrtapsell/usernameSpoof`
  This is an example repo that shows how GitHub handles spoofed usernames in commit (performed with the permission of the user in question).
- `https://github.com/jrtapsellProject18/mergeStrategies`
  This shows how GitHub handles signatures for merges performed using the various different options available on GitHub.

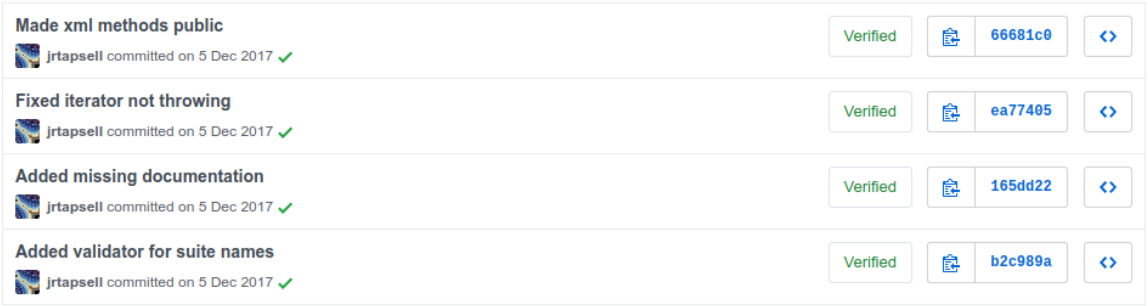
6.2 Tests

All of the tests can be run automatically using gradle, for example to run the `GitWrapper` tests:

```
./gradlew gitWrapper::test
```

The tests can be validated by looking on the repository, which looks like the following screenshot:

Figure 6.1: Github screenshot showing passing tests



The commits with a tick passed the tests, which are automatically run on TravisCI[16].

Chapter 7: **Issues faced**

- Keybase issue — [GitHub issue]
  The keybase client has a UI issue where users who only have smartcard GPG keys cannot decrypt messages sent to them.

  I have created a PR[17] for this issue, which is still open.

- Keybase API issue — [18][GitHub issue]
  The official Keybase API docs had issues getting signers for a given domain, which I raised an issue about, this was then fixed.

  This caused issues for checking an entity's keys, as lookup by URL had to be performed twice, to get around this issue.

- Runtime JAR signing
  This is not possible using the JDK tools, so I use jarsigner, but this required making a wrapper for it.

- Git issues

  – Git inconsistencies
    Although there is an unknown key return value, git returns unsigned for unknown keys, which breaks checking.
  – Git only allows 1 signature
    So if the committer and author are different they cannot both sign.
  – JGit limitations
    JGit does not support signatures so a custom version was required.
  – Git format
    Git has many oddities in its disk representation, such as the way that pack files use a mix of big endian, litte endian, and even custom integer representations, which complicated the code and slowed down development.

Chapter 8: **Software completed**

demoJar
: This is a simple project used to make a clean testing JAR, it is an application written in Kotlin, which simply prints a message and exits. It is built into a jar, which can then be signed, injected into and can be used for other tests.

gitWrapper
: Wraps git, and allows for getting repo status and commit information back without each part that uses git having to call out to the git executable manually.

gpgWrapper
: This module wraps the GPG executable, and allows the system to sign messages with keys stored in memory, on disk, or on a smart card. It can also validate messages against either the default keyring, or against a set of known keys.

jarInfo
: Allows for reading of information from JAR files, including from their manifest and other files, and it also allows for getting the signature status of the files from the commit.

jarInjector
: This tool allows for manipulation of jar files that are stored on disk, this includes being able to sign them, unsign them, and add and remove files. This will be used by the signature system to clean any existing signatures from the jar file, before it injects the relevant information into the jar and signs it with the RSK.

keybaseConnector
: Connects to Keybase via their API and allows for details of an identity to be retrieved. This acts as a trust anchor for the project, and also means that if a key is compromised the entity can just remove it from keybase, and releases signed with that key will then become invalid, which reduces the amount of harm that a compromised key can cause to a company.

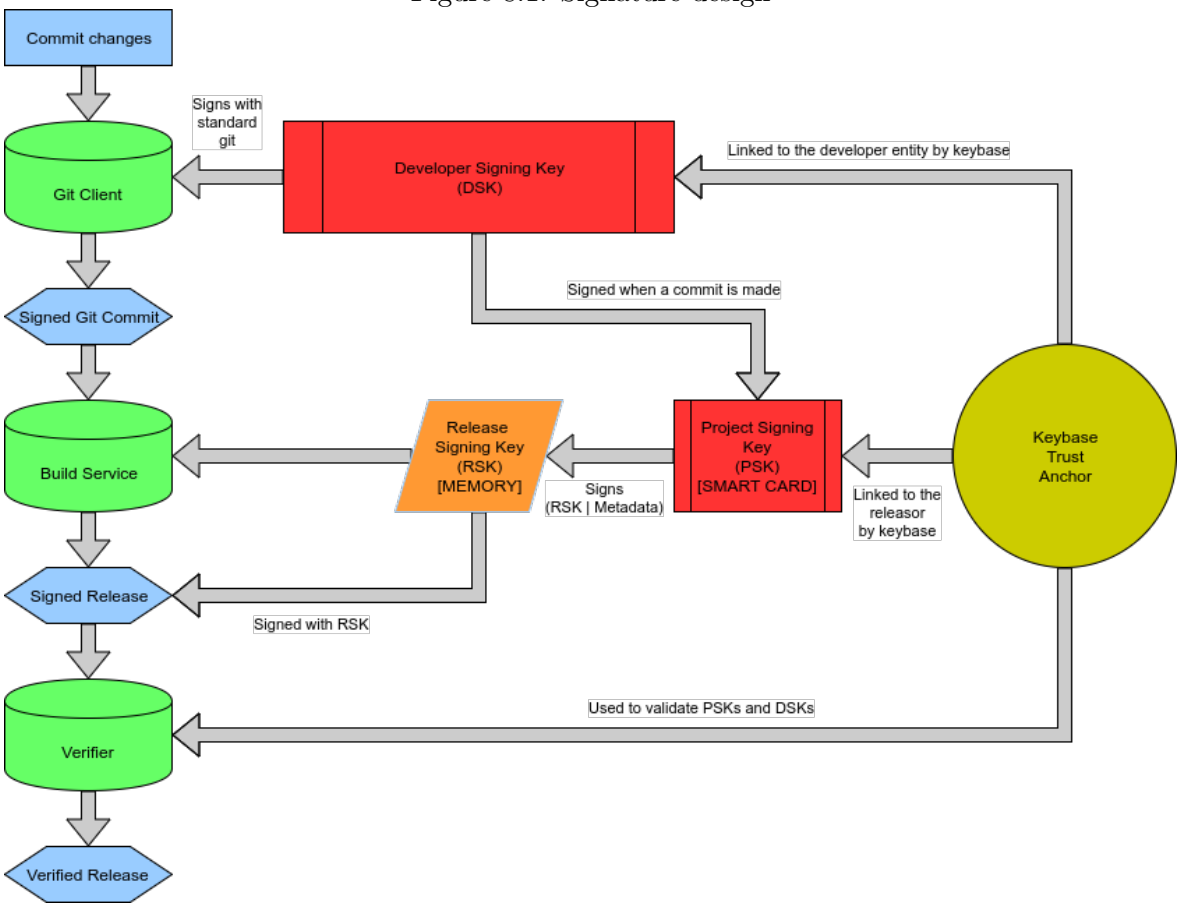
processTools
: This helps with calling external processes, communicating with them, and checking exit codes, so that all the modules do not need to repeat this, causing duplicated code and allowing for inconsistent behaviour.

projectValidator
: This checks the project is setup correctly, all tests are hooked correctly and that the packages are correct. This makes sure that there are no configuration errors that would make the project appear to be fine, while it is actually broken.

Signature design

I have designed the signature scheme below, so that the entities involved in the release of an artefact can be proven to be involved, and so that uninvolved entities cannot be spoofed into a project.

Figure 8.1: Signature design



KGit

This is an implementation of a git client written in Kotlin, which is similar to JGit, except that it supports signatures.

Chapter 9: **Usage**

The code from the main repository is signed at each commit, which meant I had to use the workflow I would impose upon users. I found that it works well, the only minor difficulties are that if the smartcard is not present, and a commit is started then the commit message will be lost, and when the commit is retried it has to be retyped, and that a lot of git clients, like GitKraken do not support signatures, so they cannot be used, however the GUI can be used to stage all the files, and then only the actual commit command needs to run from the command line.

For this, my developer key is stored on a Yubikey 4C, the key for which is available on my Keybase profile.

Chapter 10: **Git Internals**

Git stores data on the disk very compactly, but this leads to a very complex layout. By reading a writeup[19] online, I managed to find how the git internals work.

10.1 Raw files

At the start, git stores objects in a file with the name:

```
.git/X/Y
```

Where X is the first 2 hexidecimal letters of the object hash, and Y is the rest of the hash. These files are compressed, and have a small header which describes their type and size.

10.2 Packfiles

This can cause performance issues when there are a lot of objects, so packfiles are used to compact lots of small objects into a few larger objects.

Firstly, there are index files, these are not strictly necessary, but as packfiles can get large, they can help with finding objects quickly.

Once the index file has been parsed, the location in the packfile is known. Some objects in the packfiles will be stored as a delta to another object, although this only applies to objects which are of the same type[20]. The files also mix integer representations, containing all of:

- Big endian
- Little endian
- Custom encoding

where the custom encoding uses the first byte to say which other bytes should be read in.

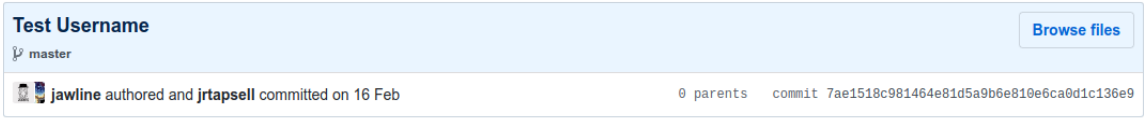
All of these optimisations lead to a very compact storage format, but make parsing in languages where bit shifting is generally avoided more difficult.

Chapter 11: **GitHub**

11.1 Spoofed commits

I tested various ways of spoofing commits (with the consent and presence of the person having commits spoofed), and managed to create a commit which appeared to be created by someone else, but which was not. However this didn't show up on their profile on GitHub. A screenshot of how a faked commit looks on the web UI is provided below.

Figure 11.1: Example of a spoofed commit



Using PGP signatures to protect commits would protect against this, but while the web UI shows that the targeted user authored the commit, and shows their icon over the top of the person who pushed the commit it will still possible to use unsigned commits to try to spoof users. This is due to the decentralised nature of git as discussed above, as it is not impossible for a user to push another user's code from a remote repository, and without signatures it is impossible to check if this code has been modified or spoofed.

11.2 Merge effect on signatures

I also checked how different ways of merging branches affects the state of the signatures.

- GitHub merge
  Signs the resulting commit with a GitHub key.
- GitHub rebase
  Strips signatures from the commits.
- Github squash
  Signs the squashed commits with a GitHub key.
- Local no-ff
  Signs the merge commit with the author's key.
- Local ff
  Keeps the original signatures.

ff stands for fast forward, and refers to if a new commit must be generated, even when uneeded.

This means that if the GitHub UI is used, the GitHub key must be trusted, and the rebase option cannot be used to merge in changes.

With local merging, no such problems occur as the original author's keys are used.

Chapter 12: **Evaluation**

12.1 Summary

Although the full project was not completed due to issues faced, there was a large amount of progress that was made, including producing many of the libraries following the software design principles taught on the course. There were also external issues that were fixed, which helped not only this project, but other projects and users using these external tools and libraries.

12.2 Parts Completed

- Kotlin
  At the start of this project, I had only used Kotlin at a basic level. Now I have a better understanding of it.
- Gradle
  I have learned more about build automation and testing. I had used it previously, but I have also learned to use the Kotlin gradle DSL, although I did not use if for this project, as it would require rewriting the build system for the whole project.
- LaTeX
  I have improved my ability to write LaTeX. I had previously only used LaTeXin a very basic capacity, and the documents that I had written were not very well written. Through learning LaTeXI have learnt to properly lay out my documents, and have found a set of tools which support my preffered LaTeXworkflow.
- Git
  I now have an understanding of how the internals of git work. This includes both raw and packed files. This will help not only when I work with Git, but also when I use git, as many of the limitations now make sense, as I can see why they exist, and how to work around them. Some examples of this are the restriction on multple signaturs, and the fact that which branch a commit was made to may not be known.
- Keybase
  There were 2 issues with Keybase that were found during this project:
  1. The API issue.[18]
     This caused issues requesting data from the API, and was different to what the documentation said. It was solved, so the issue will no longer affect users.
  2. The client issue.[21]
     This causes issues if all of a user's keys are on smartcards, I created a pull request[17] for it, but that is still open.

12.3 Further work

Further work for this project could include the android elements, and also implementing different trust anchors. It could also look at how commit to release security could work for

projects where information about who is commiting cannot be revealed, and for proprietary code. It could also create a git wrapper which deals with the smarcard issues found in git, so that the commit messages are not lost should signing the commit fail.

Bibliography

[1] Driver signing policy. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy–windows-vista-and-later-, 2017.
This lists the signing policy for windows drivers, which shows it is mandatory on recent versions of windows using secure boot, which is a mandatory part of the windows hardware specifications[22].

[2] 7.4 git tools - signing your work. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work, 2017.
This shows that git supports signed commits, proving their author or committer.

[3] Git signing. https://developers.yubico.com/PGP/Git_signing.html, 2017.
This shows yubikeys can be used to store keys used for signing commits, proving that they were created by the person that was expected, this helps to simplify managing keys, and excluding an attack against the yubikey keeps the key safe even when inserted into malicious machines.

[4] Maxim Shafirov. Kotlin on android. now official. https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/, 2017.
This shows that Kotlin is now a first class language on android, which allows it to be used for this project, although before it would have required additions to the toolchain it is now fully supported.

[5] Andy Greenberg. Software has a serious supply-chain security problem. https://www.wired.com/story/ccleaner-malware-supply-chain-software-security/, 2017.
This was a good example of malicious code being injected before signatures, getting around the current protections.

[6] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. Shattered. https://shattered.io/, 2017.
SHAttered attack, demonstrating a pair of colliding PDFs.

[7] Linus Torvalds. Re: Sha1 collisions found. https://marc.info/?l=git&m=148787047422954, 2017.
Linus Torvalds' thoughts on the effects of the SHA1 collision on git.

[8] D. Atkins and R. Austein. Threat analysis of the domain name system (dns). RFC 3833, RFC Editor, 2004.
DNSSEC RFC, explains how DNSSEC works.

[9] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *N/A*, 11 2008.
The original bitcoin paper (author name is a pseudonym).

[10] Linking my facebook and keybase accounts. (i have invites if anyone tries to join and cannot), howpublished = `https://www.facebook.com/jrtapsell/posts/1362460587209034`, month = , year = , note =
My facebook post linking my keybase to my facebook account (accessed on 03/27/2018).

[11] Reproduciblebuilds/about - debian wiki. `https://wiki.debian.org/ReproducibleBuilds/About`. (Accessed on 03/28/2018).

[12] Gitian: a secure software distribution method. https://gitian.org/.
A project that allows for signing and trusting signatures to validate a build.

[13] Customer letter - apple. https://www.apple.com/customer-letter/.
The public letter from Apple describing the request to design a backdoor.

[14] Marco Valleri Alberto Ornaghi. Man in the middle attacks demos (blackhat 2003). https://blackhat.com/presentations/bh-usa-03/bh-us-03-ornaghi-valleri.pdf.
A BlackHat 2003 presentation explaining various downgrade attacks. (Accessed on 03/28/2018).

[15] Objects in kotlin: Create safe singletons in one line of code (kad 27). https://antonioleiva.com/objects-kotlin/. This shows how Kotlin objects are thread safe Singletons.

[16] jrtapsell/android_app_verifier - travis ci. https://travis-ci.org/jrtapsell/android_app_verifier.
Travis CI page for the main repository (Accessed on 03/27/2018).

[17] Improved handling of keys where the private key is not accessible by jrtapsell - pull request #10570 keybase/client. https://github.com/keybase/client/pull/10570.
Github issue for the pull request to fix the smartcard issue [21]. (Accessed on 03/27/2018).

[18] Api documentation issue - issue #3122 keybase/keybase-issues. https://github.com/keybase/keybase-issues/issues/3122.
GitHub issue for the documentation issue. (Accessed on 03/27/2018).

[19] Unpacking git packfiles. https://codewords.recurse.com/issues/three/unpacking-git-packfiles.
Git pack files explained. (Accessed on 03/27/2018).

[20] git/pack-heuristics.txt at master git/git. https://github.com/git/git/blob/master/Documentation/technical/pack-heuristics.txt#L165.
Git documentation on packfiles. (Accessed on 03/27/2018).

[21] Unable decrypt messages from web ui if keys are all stored on smart cards - issue #3126 keybase/keybase-issues. https://github.com/keybase/keybase-issues/issues/3126.
GitHub issue for the smartcard UI issue. (Accessed on 03/27/2018).

[22] Hardware compatibility specification for systems for windows 10, version 1607. https://docs.microsoft.com/en-gb/windows-hardware/design/compatibility/systems#systemfundamentalsfirmwareuefisecureboot, 2017.
This shows that secure boot is a mandatory part of the requirements for windows 10.