JOHANNES KEPLER
UNIVERSITY LINZ

Author
**Georg Kofler**, BSc
k12006277

Submission
**Institute of
Networks and Security**

Thesis Supervisor
Univ.-Prof. DI Dr.
**René Mayrhofer**

Assistant Thesis Supervisor
**Martin Schwaighofer**, MSc

September 2025

# Reproducible builds of E2EE-messengers for Android using Nix hermetic builds

**Master's Thesis**

**to confer the academic degree of**

**Diplom-Ingenieur**

**in the Master's Program**

**Computer Science**

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

# Abstract

This thesis explores the implementation of verifiable, Reproducible Builds for End-to-End encrypted (E2EE) messaging applications for Android. The topic is part of the research field of Reproducible Builds, which aims to increase trust in software and to improve the security of Software Supply Chains. Reproducible builds aim to enhance trust in software by allowing users to verify that the binary of the software has been created from the exact source code and dependencies specified in the build process. The thesis focuses on providing a reproducible build process for two open-source E2EE messaging applications: Signal and Wire. The motivation to ensure reproducibility—and thereby the integrity—of E2EE messaging applications stems from their central role as essential tools for modern digital privacy. These applications provide confidentiality for private and sensitive communications, and their compromise could undermine encryption mechanisms, potentially leaking sensitive data to third parties. Ensuring the integrity of E2EE apps is not only critical for individual privacy but also for upholding broader principles of freedom, security, and trust in the digital age.

Due to time constraints, only a small subset of the available E2EE messaging apps could be selected for this work. Signal Android and Wire Android were chosen as two of the most popular E2EE messaging applications. However, the intention is to extend the set of reproducibly built applications to include many more in the future. The applications are built in a hermetically isolated build environment, which is configured using a Nix expression. The implementation consists of a CI/CD pipeline, which is called at regular intervals and checks for new versions of the Android applications. If a new version of one of the applications has been released, the pipeline will update the Nix scripts and verify the reproducibility of the new version of the application by building the derivation, which fetches the new source code, retrieves the dependencies, and builds the APK. The resulting APK is compared to the vendor-published APK using the tool Diffoscope to identify any differences. The self-built APK, its hash value, and a build attestation are published as pipeline artifacts of the GitLab instance and logged in a transparency log to allow further external verification of the supply chain. This work contributes to improving the security and integrity of software supply chains, particularly for E2EE applications, by ensuring that no unauthorized modifications occur during the build process.

# Contents

**Abstract** ii

**List of Acronyms** vii

**1 Introduction** 1
- 1.1 Objectives and Approach . . . 1
- 1.2 Outline . . . 2

**2 Background** 3
- 2.1 Android Package . . . 3
  - 2.1.1 Other formats in the Android ecosystem . . . 3
- 2.2 End-to-end encryption . . . 4
- 2.3 Open-Source Software (OSS) . . . 4
  - 2.3.1 Limitations of OSS . . . 4
  - 2.3.2 Relevance for my thesis . . . 4
  - 2.3.3 F-Droid . . . 5
  - 2.3.4 AndroZooOpen . . . 5
- 2.4 Hash Functions . . . 5
  - 2.4.1 Limitations of Hash Functions . . . 5
  - 2.4.2 Relevance for my thesis . . . 6
- 2.5 Signatures . . . 6
  - 2.5.1 Relevance for my thesis . . . 6
- 2.6 Version Control Systems . . . 6
  - 2.6.1 Relevance for my thesis . . . 7
- 2.7 Android build tools . . . 7
- 2.8 Vulnerabilities and Threats . . . 7
- 2.9 Software Supply Chains . . . 9
  - 2.9.1 Introduction to Software Supply Chains . . . 9
  - 2.9.2 Supply Chain Attacks . . . 9
  - 2.9.3 Examples of Supply Chain Attacks . . . 10
  - 2.9.4 Supply Chain Integrity . . . 11
- 2.10 Supply-Chain Levels for Software Artifacts (SLSA) . . . 12
  - 2.10.1 Introduction of the SLSA standard . . . 12
  - 2.10.2 Relevance for my thesis . . . 14
- 2.11 Reproducible Builds . . . 15
  - 2.11.1 Introduction of the concept of Reproducible Builds . . . 15
  - 2.11.2 History of Reproducible Builds . . . 16
  - 2.11.3 Practical Aspects of Reproducible Builds . . . 17
  - 2.11.4 Reproducible build environments . . . 19
- 2.12 Hermetic builds . . . 20
- 2.13 Bootstrappable Builds . . . 20
- 2.14 Build Attestations . . . 21
- 2.15 Reproducible Builds using Nix . . . 22
  - 2.15.1 The Nix deployment system . . . 22
  - 2.15.2 Nix expression language . . . 23
  - 2.15.3 Building a component . . . 23
  - 2.15.4 Nix store . . . 24
  - 2.15.5 The Nix Packages collection . . . 26
  - 2.15.6 The standard environment . . . 26

2.15.7 Nix Flakes . . . 26
2.15.8 Relevance for the thesis . . . 26
2.16 Verifiable Logs . . . 27
2.16.1 Merkle tree . . . 27
2.16.2 Root node . . . 27
2.16.3 Consistency proof . . . 27
2.16.4 Inclusion proof . . . 27
2.16.5 Availability and synchronization . . . 28
2.16.6 Split-view attacks . . . 28
2.16.7 "Bad" entries in a verifiable log . . . 28
2.16.8 Relevance for the thesis . . . 28

**3 Related work 29**
3.1 Projects and tools to improve Supply Chain Integrity . . . 29
3.1.1 Gradle dependency verification . . . 29
3.1.2 Gitian . . . 29
3.1.3 Guix . . . 30
3.1.4 in-toto Framework . . . 31
3.1.5 Extending Cloud Build Systems to Eliminate Transitive Trust . . . 31
3.1.6 Tools that are tailored to improve Android Supply Chain Integrity 32
3.1.7 apksigcopier . . . 32
3.1.8 reproducible-apk-tools . . . 32
3.2 Reproducible Builds on F-Droid . . . 32

**4 Threat model 34**
4.1 Detailed introduction of the fictional people . . . 34
4.1.1 A common user . . . 34
4.1.2 An investigative journalist . . . 35
4.2 Definition of the threat model . . . 36
4.2.1 Source threats . . . 37
4.2.2 Build threats . . . 38
4.2.3 Dependency threats . . . 38
4.2.4 Availability threats . . . 38
4.2.5 Distribution threats . . . 39
4.2.6 Verification threats . . . 39
4.3 Threats in scope . . . 39

**5 Software Supply Chain of an Android application 40**
5.1 Overview of the Software Supply Chain . . . 41
5.2 Detailed steps of the Supply Chain Model . . . 41
5.2.1 Source code . . . 42
5.2.2 Build parameters . . . 42
5.2.3 Resource Preprocessing . . . 43
5.2.4 Compilation . . . 43
5.2.5 Optimization and Obfuscation . . . 44
5.2.6 DEX Conversion . . . 44
5.2.7 Resource Packaging . . . 44
5.2.8 APK Packaging . . . 45
5.2.9 APK Signing . . . 45
5.2.10 Distribution . . . 45
5.2.11 Verification . . . 45

**6 Contribution of this thesis 47**
6.1 Summary of the proposed approach . . . 47
6.2 Selection of end-to-end encrypted (E2EE) applications . . . 47
6.3 Supply Chains of selected Android applications . . . 48
6.3.1 Existing Supply Chain: Signal Android . . . 48

6.3.2 Existing Supply Chain: Wire Android . . . 49
6.4 Implementation of this thesis . . . 49
6.4.1 Implementation of the supply chain of the Android applications . . . 50
6.5 The CI/CD pipeline . . . 50
6.6 The Flake . . . 52
6.7 Fetching the Source Code . . . 54
6.7.1 Mitigations . . . 55
6.8 Fetching the dependencies . . . 55
6.8.1 Mitigations . . . 56
6.9 Build within the hermetically isolated build environment . . . 57
6.9.1 Build process aspects common to both apps . . . 57
6.9.2 Inclusion of the build tools . . . 57
6.9.3 Build process specifics: Signal Android . . . 58
6.9.4 Build process specifics: Wire Android . . . 59
6.9.5 Mitigations . . . 60
6.9.6 Extra arguments passed to the build command . . . 60
6.10 Signing of the APK . . . 60
6.11 Comparison to the reference APK . . . 61
6.11.1 Using `Diffoscope` to compare the APKs . . . 61
6.11.2 Implementation . . . 62
6.12 Publishing of the resulting artifacts . . . 62
6.13 Update the Reproducible Build process . . . 63
6.13.1 Update references . . . 63
6.13.2 Update dependencies . . . 64
6.13.3 Update build tools . . . 64
6.13.4 Update process trigger . . . 64
6.14 Verification process: Concept . . . 64

**7 Evaluation 68**
7.1 In short . . . 68
7.2 Limitations and Scope . . . 68
7.3 Source threats . . . 69
7.4 Build threats . . . 69
7.5 Dependency threats . . . 69
7.6 Availability threats . . . 70
7.7 Distribution threats . . . 70
7.8 Verification threats . . . 70

**8 Findings 71**
8.1 Versions that were successfully built . . . 71
8.2 Missing dependencies in provided verification-metadata.xml . . . 71
8.3 (Very) long list of dependencies . . . 72
8.4 Unavailable dependencies . . . 72
8.5 Difficulty integrating the Gradle build process in Nix expressions . . . 72
8.5.1 Fetching Gradle dependencies . . . 73
8.5.2 Sandboxing issues: missing tools and other processes failing . . . 73
8.6 Limited performance of the build process . . . 73
8.7 Debug information in release builds . . . 74

**9 Discussion 75**
9.1 List of threats, excluded from the threat model . . . 75
9.2 Differences to other approaches . . . 75
9.3 Use of verifiable logs in the verification process . . . 75
9.4 Beyond Supply Chain Integrity of a application . . . 76
9.4.1 Recursive verification of dependencies . . . 76
9.4.2 Source threats for OSS . . . 76

9.5 Relevance of this work in the current times . . . . . 77

**10 Future work** **78**
10.1 Improve implementation . . . . . 78
10.1.1 Populating the build environment . . . . . 78
10.1.2 Close remaining sources of non-reproducibility . . . . . 78
10.1.3 Improve update process . . . . . 79
10.1.4 Implement distributed verification method . . . . . 79
10.1.5 Recursive Integrity Verification of dependencies . . . . . 79
10.1.6 Additional trust placed in build environments . . . . . 79
10.1.7 Optimization of the distributed verification . . . . . 79
10.2 Extend/improve approach . . . . . 80
10.2.1 Missing build provenance . . . . . 80

**11 Conclusion** **81**

**Bibliography** **83**

# List of Acronyms

**A-B** Attestable Build

**AAB** Android App Bundle

**AAPT** Android Asset Packaging Tool

**AAR** Android Archive

**AGP** Android Gradle Plugin

**APK** Android Package

**ART** Android Runtime

**CI/CD** Continuous Integration/Continuous Development

**DEX** Dalvik Executable

**E2EE** End-to-End encrypted

**FHS** File Hierarchy Standard

**FSO** file system object

**INS** Institute of Networks and Security

**JDK** Java Development Kit

**OSS** Open-source Software

**R-B** Reproducible Build

**SBOM** Software Bill of Materials

**SLSA** Supply-chain Levels for Software Artifacts

**TEE** Trusted Execution Environment

**VCS** Version Control System

# Chapter 1

# Introduction

Recent years have shown an increase in the number of attacks on various links in supply chains of software. The SolarWinds attack [1] in 2020 introduced a backdoor in the attacked software without modifying the source code or tricking users into installing modified binaries with invalid signatures. The attack was applied to the build system, which was used to compile and sign the official binaries of SolarWinds. The attackers abused the large amount of trust placed in the build system and the lack of awareness of the possibility of attacks on it.

Following the SolarWinds attack in 2020, the NotPetya attack in 2017, the CIA's manipulation of Xcode, and many more similar incidents, there was a rise in awareness for the importance and relevance of Supply Chain Security. The concept of Supply Chain Security encompasses all methods to increase the security and integrity of all processes involved in the creation of software—from the source code to the build process and the dependencies thereof, to the integrity verification of the binaries during the software distribution phase.

One method to increase the security and verify the integrity of the build process are Reproducible Builds (R-Bs). As stated on the website of the Reproducible Builds project, R-Bs allow users to verify that binaries are created only using the exact version of the source code and build inputs as stated in the build process. The concept of R-Bs has been around for a while now. It was used already in the 1990s, was first publicly mentioned in 2000, and then more concretely addressed in 2007 [64], but is recently gaining a lot more traction as the advantage of being able to verify that **published binaries** are built from **exactly the published source code**, with **exactly the listed dependencies,** is becoming more appealing.

In the last few years, events like the ongoing efforts of the European Parliament to introduce a law to add content scanners to messaging applications, as well as the policy change of the Google Play Store to compel developers to upload their private signing keys to their servers, theoretically allowing Google to distribute modified Android Packages (APKs) signed with the vendor's official key, increased the need for transparency and verifiability for much more politically motivated reasons. While this trend can be worrying for the public, it should be noted that journalists, activists, and political opponents have been subjects of targeted attacks for a long time already and therefore might benefit the most from tools and methods allowing them to verify that the software they use has not been tampered with. The threat model of this thesis is therefore motivated by the risks faced by two fictional individuals: a common user of E2EE messaging applications and an investigative journalist. The thesis shows that R-Bs are useful for the considered threats as they increase the transparency and thereby trust in build processes. With that, R-Bs can be valuable elements of end-user verifiable software integrity.

## 1.1 Objectives and Approach

This thesis will focus on the reproducibility of open-source E2EE instant messaging applications for Android. The goal is to provide an online, accessible archive of binaries

that have been built from their publicly available source code. Along with those binaries, the instructions of the reproducible build process are provided for every built application. The archive also provides the results of the comparison of the self-built APKs with the official binaries published by the vendors to transparently document the differences between the two applications, if there happen to be any. To allow for accountability and traceability of the results of the reproducible builds, the APKs and their hash value are published on a GitLab server operated at the Institute of Networks and Security (INS). This allows the approach to also provide trust and accountability in cases where reproducible builds are not yet fully achievable.

The build process is made reproducible by executing it within the build phase of a Nix derivation. This provides a hermetically isolated build environment that ensures that no other software binary is accessible to the build process than the exact ones that are specified by the build process. The comparison between the APK resulting from the build process and a vendor-published reference APK is done using the tool `Diffoscope`. The build is executed by a Continuous Integration/Continuous Development (CI/CD) pipeline on GitLab that regularly checks for a new version of the E2EE messenger apps. If a new version is found, it updates the references, starts the build process, compares the resulting APK with the officially published one, and publishes the result of the comparison, the self-built APK, and its hash value as GitLab pipeline artifacts. It is necessary for the source code for the Android application to be publicly available to allow for R-Bs and therefore this thesis is only concerned with open source software.

## 1.2 Outline

Section 2 explores the topics and concepts that will be referred to in this thesis. The work continues in Section 3 by highlighting related research and projects, aiming to provide similar contributions.

Section 4 then defines the threat model considered in this thesis. Section 5 gives an overview of the general software supply chain of an Android application and points out which threats can occur along it. In Section 6, the approach to mitigate the threats occurring along the supply chain model of the Android applications, and the following sections will explain the implementation part of this thesis.

Section 7 evaluates which threats are mitigated by the approach and which threats remain open. Section 8 highlights the findings of this thesis, and Section 9 discusses questions regarding the presented work. Section 10 gives an outlook on potential future research topics and extensions of this work, and Section 11 closes the thesis with a conclusion.

# Chapter 2

# Background

This section explores the concepts, topics, and tools that are referenced and used in this thesis.

## 2.1 Android Package

The APK file format is used to distribute and install applications on the Android platform. It is a ZIP-archive format with a predetermined structure that contains all necessary components to run the application on Android devices. It contains core functionality in the binary `classes.dex` file in the top-level directory and required native libraries as precompiled binaries in the `lib/` directory. The `.dex` files are bytecode, optimized for Android Runtime or the older Dalvik Virtual Machine, and the native libraries are platform-specific binaries that are typically used for performance critical tasks or hardware specific functionality. APKs then typically also contain resources like icons, images, and other UI elements in the `res/` directory and raw asset files like fonts and sounds in the `assets/` directory. Another essential file in an APK is the `AndroidManifest.xml`, which contains the metadata about the application, including its version, package name, components, required permissions, and required hardware and software features. The directory `META-INF/` contains metadata about the archive itself and the signature files `CERT.SF` and `CERT.RSA`, given that the APK is signed. [37, 42]

Modifying the files of an APK allows to change the UI of applications and to inject custom code into the applications. In order to prevent such unauthorized modifications, applications are signed by hashing the contents of the archive except for the APK signing block. Any changes to the APK outside of the APK signing block will therefore invalidate the signature. [63]

### 2.1.1 Other formats in the Android ecosystem

Android App Bundle (AAB) and Android Archive (AAR) are file formats that serve slightly different purposes compared to the APK format:

- **AAB:** Is a new format introduced by Google, and since August 2021 it is the required format to upload applications to the Google Play Store [20]. AABs cannot be directly installed, but rather provide a bundle of resources from which optimized APKs can be generated. These generated APKs are then called *split APKs* and are tailored to a specific device configuration, which reduces the application's download size and improves its performance. [19, 20]
- **AAR:** This file format is also not directly installable and is instead used to package reusable Android libraries. It contains compiled code, resources, and a manifest file to allow developers to include the library in their projects. AAR files are typical dependencies in Android projects. [21]

## 2.2 End-to-end encryption

This thesis focuses on Android applications providing end-to-end encrypted messaging functionality. E2EE ensures that the content of messages is encrypted in such a way that only the intended recipient can decrypt it. Essentially, this means that the message is not decrypted at any point once it is sent—neither on the server of the messaging applications nor anywhere else along the communication path between sender and receiver. This elevates the privacy guarantees that such an application can provide, as no third party and not even the vendor of the application itself can read the content of the message. However, recent trends in increased digital surveillance worldwide, as well as legislative proposals such as the ongoing discussions of the EU's Child Sexual Abuse Regulation [7], have raised serious concerns about the future of E2EE [6]. The proposal introduces mechanisms that could undermine E2EE by compelling messaging application vendors to scan communications for illegal content. This would effectively require vendors to provide plaintext access to user data, thereby compromising the core privacy and security guarantees of E2EE. [70]

## 2.3 Open-Source Software (OSS)

Open-source Software (OSS) is software that is built from source code, which is freely available to anyone. This allows anyone to build their own applications from the source code, as well as to analyze the source code and look for weaknesses and vulnerabilities in it:

- **Benefits for developers:** The free access to open source software allows for several benefits, including the ability to learn from existing projects and to be able to build applications faster and more easily by including open source libraries in the development and build process of the software. It is common practice nowadays for projects to include OSS libraries to save the time and effort of programming it oneself. This does, however, require the developer to audit or trust such libraries to work as intended and to not include any unwanted logic.
- **Security benefits:** One of the original promises of OSS was that the public availability of the code would allow for distributed auditing of the software to ensure the benevolence of the logic of the software and to find and patch vulnerabilities in the source code of the application to enhance the end users security [56, 67].

### 2.3.1 Limitations of OSS

While the idea of being able to audit the code of OSS was shown to work well if a big community of users is interested and motivated in verifying the source code, it can also give a false sense of security if there are not enough users to perform audits on the software [81]. OSS can even introduce additional threats, as many OSS projects do not only allow for audits and feedback on the software but encourage or even rely on users from the community to contribute to the development of the source code. Depending on the existing security mechanisms to verify the benevolence of contributed source code, this could allow attackers to submit code that contains backdoors [50], vulnerabilities, or other malicious flaws and enable a malicious actor to perform a Supply Chain Attack (see Section 2.11).

### 2.3.2 Relevance for my thesis

The aspect of auditability of source code is very central to the approach described in the thesis, because the source code of the applications needs to be freely and openly accessible

to be able to reproduce the build process (see Section 2.11 for more information). Not all the applications within the current landscape of E2EE messengers are open source, but because that is a necessity for reproducible builds to work, this thesis will only focus on applications for which the source code is available. Table 6.1 lists a selection of E2EE messengers for which the source code could be found.

### 2.3.3 F-Droid

F-Droid is a repository of open source Android applications[1]. It is also a community driven application store project, a collection of tools to set up and run an app store, and an app store client to install applications from the central repository or from 3rd party repositories[2]. Part of the collection of tools are build and release tools that support developers in turning app source code into published application builds. F-Droid also provides tools to upload apps as reproducible builds, which will be covered in more detail in Section 3.2. [11]

### 2.3.4 AndroZooOpen

AndroZooOpen [52] is another large-scale collection of open source Android applications that can be the starting point for future work to apply the concepts of this thesis to numerous applications to verify the integrity of the build process and the resulting APK of them.

## 2.4 Hash Functions

Cryptographic hash functions are software algorithms that map arbitrary strings of data to fixed length outputs, the so-called *hash value*. The functions are deterministic, which means that for the same inputs they will always provide the same outputs. However, changing the inputs even just in the slightest will result in a very different *hash value*. Other properties of modern hash functions are [24]:

- **Pre-image resistance:** Given a hash value, it is very difficult to find the input that led to this value.
- **Weak collision resistance:** Given an input value, it is very difficult to find another input value that results in the same hash value if the hash function is applied to them.
- **Strong collision resistance:** It is very difficult to find two input values that result in the same hash value, if the hash function is applied to them.

These properties allow hash functions to be used to perform integrity checks: “digest” large amounts of data (for example, a large file) to check its validity by comparing the resulting hash value of the file to the expected hash value. [24]

### 2.4.1 Limitations of Hash Functions

The only reason why the hashes of the differing files could be the same is if a so-called *hash collision* occurs, which is very unlikely with modern hashing algorithms like SHA-256, SHA-512, or SHA-3 variants. *Hash collisions* can, however, be crafted for older hashing algorithms like MD5 or SHA-1, which means that getting the same hash from two files using an old hash algorithm does not guarantee that the files are identical, and therefore it is necessary to use secure algorithms instead.

[1] https://f-droid.org/repo/
[2] https://forum.f-droid.org/t/known-repositories/721

### 2.4.2 Relevance for my thesis

In this thesis, hashes are mainly used to check whether two files are bitwise identical by applying a secure hash function to the files and verifying whether the resulting hash values are identical. This concept is used

- by Nix to verify the equivalence of a derivation and an existing entry in the Nix store (see Section 2.15 for more information).
- by the conceptual verification method described in Section 6.14 to ensure the equivalence of the Android application to be installed and the APK that was built by the protected build process described in Section 6.
- in the form of checksums, to reference a specific version in, Version Control Systems (VCSs) like *git*.
- in various other steps of the contribution to verify the integrity of some artifact or piece of digital information (e.g., when fetching dependencies, see Section 6.8).
- as a fundamental building block of transparency logs (see Section 2.16 for more information).

## 2.5 Signatures

Signatures are a cryptographic concept that allows to ensure the authenticity in addition to the integrity of digital information. In this thesis, signatures will be mainly relevant in the signing step of Android applications (see Section 5.2.9). In this step, the developer of an Android application takes an unsigned APK and their private signing key to sign the application. This signed application then allows users to verify that the application was built by the developer. [35, 53, 63]

### 2.5.1 Relevance for my thesis

This thesis will make use of the authenticity that signatures provide. It will also use the fact that signatures are only valid for the exact binary that they are created for. This allows for a mechanism to verify the integrity of an application after it was built and to verify the bit-wise equivalence of a signed and an unsigned APK by copying the signature from the signed APK and patching it into the unsigned application. If the two applications have differed by more than just the signature, the patched application will not be signed in a valid way.

## 2.6 Version Control Systems

VCSs are a very common tool in modern software development. They allow developers to collaboratively extend and improve the functionality of software in small changes that are submitted in so-called *commits* and to combine or revert changes as needed due to the recording of these *commits* in the so-called *commit history*. In combination with the concept of, OSS (mentioned in Section 2.3) VCSs allow for the concept of *accountability* towards the public of every change to the source code, because every *commit* will show which changes have been applied to the software and by whom. This concept can be leveraged to trace back malicious source code, once it is found, to the *commits* and the author of the *commits* that introduced the unwanted changes.

### 2.6.1 Relevance for my thesis

Every *commit* in modern VCSs can be referenced by a *checksum* (see Section 2.4) and describes a specific state of the source code. Many VCSs also allow labeling commits with a so-called *tag* to allow for human-readable references to a specific *commit*. The thesis uses the concept that referencing a specific *commit* via its *checksum* or *tag* describes the same state of the source code independent of who accesses the code and thereby allows using the very same code as input to build processes.

## 2.7 Android build tools

Among the build systems used for Android development, Gradle is the most widely used one. It is the default build tool for Android Studio and supports complex build configurations, dependency management, and multi-module projects. There are many other build tools, most of which are specialized tools or modern tools that allow for cross-platform development. Some of these tools are:

- **Maven, Apache Ant** are tools that were commonly used before Gradle became the standard. There exist some legacy projects that still use them, but in general they are less commonly used now.
- **Meta's Buck2 and Google's Bazel** can be used to build Android projects, but are less commonly used than Gradle.
- **Apache Cordova, Reactive CLI, Flutter, and Xamarin** are a few examples of build systems that are based on different technologies, but all provide cross-platform compilations.

This thesis focuses on providing isolated reproducible builds for existing projects of Android applications and tries to provide an approach (see Section 6) that creates minimal changes to the existing build workflow. Therefore, the approach is designed to wrap around existing build workflows and such that it executes Gradle builds within a hermetically isolated build environment. Future work could extend the approach to support hermetic builds for other technologies.

## 2.8 Vulnerabilities and Threats

In this section, the terms "vulnerabilities" and "threats," used in this thesis, will be described:

- **Vulnerability:** A *vulnerability* in general refers to a weakness or flaw, for example, in a system, some software, a piece of hardware, or a process, that can be exploited by an attacker to compromise the confidentiality, integrity, or availability of a system. Vulnerabilities can arise from various sources, such as coding errors, misconfigurations of systems or software, outdated applications, or inadequate security measures.
  In this thesis, vulnerabilities mainly refer to exploitable weaknesses of either the application itself, the build process of the application, or another link in the Software Supply Chain (see Section 2.9) that lead to the compromise of the confidentiality or integrity of the application.
- **Threat:** A *threat* is any potential event, actor, or condition that could exploit a vulnerability to compromise a system. In general, threats can be intentional, such as cyberattacks by hackers, or unintentional, such as natural disasters or human errors. In the context of this thesis, threats are mainly referring to malicious actors that attempt to compromise the integrity or confidentiality provided by E2EE messaging applications to filter the shared content or steal sensitive information shared in those

applications. In Section 4 a Threat Model is defined that includes all the threats that are considered in this thesis.

- **Backdoor:** *Backdoors* are hidden methods or mechanisms that allow attackers unauthorized access to a system, bypassing normal authentication or security controls. They are one of the main vulnerabilities considered in this thesis, as they could allow attackers to compromise the confidentiality of the considered E2EE messaging applications.
  This thesis will mainly consider *bug doors*, a subtype of *backdoors*, as they can be created unintentionally by developers or intentionally by malicious actors by introducing a software bug or coding error that, if exploited correctly, provides unauthorized access to a system. *Bug doors*, since they can be as challenging to detect as other bugs of software, often go unnoticed for a long time if the source code of the application is not rigorously tested and reviewed.

## 2.9 Software Supply Chains

In this section the concept of Software Supply Chains will be explored. First an introduction of the concept is given in Section 2.9.1, then in Section 2.9.2 Software Supply Chain Attacks are introduced, and in Section 2.9.3 examples of recent Supply Chain Attacks are listed. Section 2.9.4 will then provide the description of the concept of Supply Chain Integrity.

### 2.9.1 Introduction to Software Supply Chains

In general *Supply Chains* are interlinked steps that describe the process that converts raw materials into finished products and that distributes the final product to the end customer. Every product in the commerce sector has a supply chain, and likewise, software products in the e-commerce sector have a *Software Supply Chain*. In the following sections of this thesis, the term *Supply Chains* will refer to *Software Supply Chains* and describe the collection of all individual steps from the creation of the source code of an application to the distribution of an installable binary. The term *Software Supply Chain* will also include the verification steps, confirming the authenticity and integrity of the application.

Similar to supply chains of physical products, the execution of steps of a software supply chain involves a set of different parties. The parties involved in a software supply chain are the developers of the software, the party hosting the build systems, the parties providing the dependencies and resources included in the build process, and the distributors providing the applications to the end-user.

### 2.9.2 Supply Chain Attacks

*Supply Chain Attacks* are a class of cyberattacks that, instead of directly attacking the infrastructure of their target, aim to indirectly compromise the system or organization by attacking third parties that are involved in the software supply chain of some application that the target is using. For this, supply chain attacks exploit vulnerabilities in software and processes involved in the software development, compilation, distribution, or verification. Supply chain attacks are a growing threat, and due to their indirect attack vector, they can be used to compromise well secured targets that do not sufficiently verify the integrity and benevolence of the tools and applications they use.

#### History of Supply Chain Attacks

Ken Thompson's 1984 Turing Award lecture, "Reflections on Trusting Trust" [79] already emphasized the idea that the security of a program extends beyond the logic in its source code. To ensure the security and integrity of a program, every single step of its software supply chain needs to be protected. This is necessary to prevent not only vulnerabilities and backdoors in the source code but also the addition of malicious logic during the build process and the distribution of malicious binaries as the result of a compromised supply chain.

#### Supply Chain Attacks during build time

Supply chain attacks in the build phase of an application work by tampering with the steps involved in building an application and result in a modified version of the software product containing unwanted or malicious logic. An attacker might, for example, compromise the development system executing the build process and add malicious logic to

the software binaries at build time. Thereby the resulting application will have been built and signed by the vendor but will contain unwanted logic. In his lecture [79] Thompson described an example of a sophisticated *Supply Chain Attack* that allowed him to inject malicious logic into a binary that had been compiled with a compromised compiler. This compromised compiler would propagate malicious logic transparently, without changes to the source code, and could, for example, also propagate its malicious logic to the next generation of compilers if they are compiled using this compromised one. This theoretical attack was known since 1984 but was not considered in many threat models until the recent attacks on Software Supply Chains gained more attention.

#### Factors that enable *Software Supply Chain Attacks*

In recent years attackers recognized the potential impact of attacks on Software Supply Chains. Compromising a well selected tool or library, which is used in numerous Software Supply Chains can impact numerous systems or people.

The recognition of Software Supply Chain Attacks as high-value targets and the therefore increased number of such attacks have mounted into a pressing concern for governments and organizations worldwide. The large scale availability and benefits of open source software (see Section 2.3) have resulted in the trend to include OSS tools and libraries in steps along the Software Supply Chains of many modern software projects. The use of OSS speeds up development and is to be found in the whole software life cycle. Dependency managers facilitate the use and inclusion of third-party libraries. They automatically resolve, download, and install hundreds of open source packages, as software projects commonly depend on multiple libraries, and they introduce numerous transitive dependencies themselves. Only a few dependency managers, however, provide functionalities to verify the integrity of the dependencies.

It was observed, however, that some libraries and other software dependencies are part of the dependencies of thousands of projects, which results in the risk that if one of those software artifacts is compromised, the attack will compromise the large number of projects that depend on it. [61]

In addition to attacks on OSS artifacts, Supply Chain Attacks can also arise if software artifacts are developed by paid third parties. It should be noted that outsourcing the software development is not an issue by itself, but rather the tendency to buy software from cheap providers, which, to save time and costs, might not sufficiently review the developed source code.

The risk of Software Supply Chain attacks can be reduced if projects include libraries and dependencies that are of good quality, well maintained, and audited.

### 2.9.3 Examples of Supply Chain Attacks

Any software can introduce vulnerabilities into a supply chain. As a system gets more complex, it's critical to already have checks and best practices in place to guarantee artifact integrity, that the source code you're relying on is the code you're actually using. In the past, build systems were either generally trusted or not even considered in the Threat Models, but the 2020 SolarWinds attack marked a tipping point in the awareness of software supply chain security, and it became evident that such attacks can occur at every link along the Software Supply Chain. This attack was only one of many - in recent years these kinds of attacks have increased in number and damage that they caused. In the following, some examples of recent attacks are given, as the SolarWinds attack was only one of many, and literature documents at least 174 similar incidents [61].

- **SolarWinds attack [61]**
  In December 2020, SolarWinds' Orion software was attacked. In this attack, malicious code was injected into the build system, and subsequently the build process

injected malicious code into the executables it was building, resulting in compromised binaries signed with the company's official code-signing keys. This attack affected numerous U.S. government agencies and private organizations and prominently shows the devastating potential of supply chain attacks.

- **NotPetya attack [61]**
  In 2017, a ransomware was concealed in a malicious update for a Ukrainian accounting software. This attack harmed Ukrainian and global companies, creating losses worth multiple billion dollars.
- **Malicious version of CCleaner [61]**
  Also in 2017, the popular maintenance tool for Microsoft Windows systems, CCleaner, listed a version containing malicious logic on their official website. In the time frame of more than one month until the malicious version was detected, the binaries had been downloaded 2.3 million times.
- ***event-stream* take-over [38]**
  In 2018, the maintainer of the npm package *event-stream* was offered by the alleged attacker to take over the maintenance of the package. After gaining the original author's trust, the alleged attacker then proceeded to modify the npm package by making it depend on a malicious package. At that time almost 1,600 other packages depended on *event-stream* and were thereby hit by this attack.
- **Straw horse [68]**
  In 2015, *The Intercept* published a story on leaked secret CIA documents on the strategies of an attack, based on Ken Thompson's attack [79], to inject arbitrary changes into compiled binaries of applications for iOS. The described attack was based on the manipulation of the software development kit *Xcode*, but the documents did not disclose any indications of the actual execution of the attack or its success rate.
- **XZ Utils [50]**
  In 2024, a critical vulnerability in the widely used XZ Utils has been found [2]. A developer gained the trust of the project maintainer until they were promoted to co-maintainer and could inject a backdoor into the tarballs with valid signatures that are distributed to a wide range of package repositories for Linux distributions. The backdoor was not included in the source code to prevent its discovery by reviewing the code, and only in the binary archives was the file included that activated the backdoor.

### 2.9.4 Supply Chain Integrity

The examples mentioned in Section 2.9.3 are just an excerpt of the many supply chain attacks that have occurred in recent years. They show that a compromised supply chain can lead to the distribution of software that is validly signed and also not suspicious in any other way but contains malicious logic. This is why ensuring the integrity of software supply chains has become a critical concern in recent years, as highlighted by Lamb et al. [46]. They go on to say that reviewing the source code of applications can increase confidence in the security of the software product but is not sufficient to guarantee the integrity of the executable counterpart. Trusting the source code does not equate to trusting the binaries that are supposedly derived from it.

In addition to reviewing the source code, the integrity of included libraries and the tools used throughout the software supply chain need to be rigorously verified as established by Alkhadra et al. [1]. The advice that organizations should gain an overview of their entire software supply chain. The supply chain should generally be added to the threat model, and a "third party risk" analysis should be established: vendors of applications should be carefully chosen and audited. The use of unapproved or unaudited software should be

eliminated. Alkhadra et al. further advise companies to also carefully audit open-source software that is used by their software developers and contribute to the maintenance of the source code by finding and patching bugs. The last piece of advice they mention is to also monitor the dark web to know about security breaches earlier in case the breached company does not notice or disclose the breach immediately.

These actions do not replace other threat mitigations but are to be applied alongside general security measures, like, for example, the concept of least privilege, monitoring the security status of systems, implementing advised measures by policies and guides (e.g., NIST standards), having a trained incident response team, deploying intrusion detection mechanisms, and limiting the number of access points.

#### Establishing trust using software supply chains

The primary challenge in securing software supply chains lies in the amount of trust that is placed in third-party software and build systems. Instead of trusting those dependencies and tools blindly, they should be audited and protected against tampering or unauthorized modification at any stage.

Furthermore, users of an application have to trust the vendor of the application to ensure the supply chain integrity and have no insight into which resources have gone into building that application.

The topic of ensuring supply chain integrity has also been recognized to be important by the US government. In 2021, an executive order [60] was issued by the US President to protect the integrity of Software Supply Chains. This executive order mentions the requirement to distribute an Software Bill of Materials (SBOM) alongside the binaries of the software to allow users to know the resources used to build the software and to allow them to determine whether they trust the application to be benevolent or not.

#### Linking sources and binaries together

The list of resources and dependencies, however, is not sufficient yet to guarantee that (only) these inputs have been used in the build process of the software. To achieve guarantees that only a certain set of inputs has been used in the build process, a transparent link between the source code and the compiled binary must be established.

A promising approach to address the establishment of this link is the concept of *Reproducible Builds* [64], as they provide a mechanism to verify that a binary corresponds exactly to its original source code by allowing anyone to reproduce the build process and allowing them to obtain bitwise identical build outputs from the reproducible builds. The exact method of how the reproducibility of the build process is ensured depends on the implementation, but in general requires the software source code to be publicly available, as well as the description of the exact build process to be distributed. Section 2.11 will further expand on the topic and advantages of reproducible builds.

## 2.10 Supply-Chain Levels for Software Artifacts (SLSA)

In this section the security standard Supply-chain Levels for Software Artifacts (SLSA) will be introduced. This standard provides general guidelines to improve *Supply Chain Integrity* and concrete mitigations against threats to *Software Supply Chains*.

### 2.10.1 Introduction of the SLSA standard

The SLSA framework [16] is a checklist of standards and controls to strengthen the integrity and security of supply chains and their resulting artifacts. The goal of the

framework is to improve the resilience against supply chain attacks at any link in the chain. The secondary focus of the standard is to ensure the availability of the package, its build process, and all its code and change history to allow for future maintainability of the artifact and to enable future investigations or incident responses.

The framework uses common language to describe measures to harden all the links in the software supply chain against tampering. It recommends mitigations classified into 4 levels of increasing integrity protection.

The authors of the standard provide the measures for producers, consumers, and infrastructure providers to increase trust across the entire *Supply Chain* of software artifacts:

- **Producers:** The mitigations help to better protect their source code against tampering and insider threats.
- **Consumers:** The standard provides guidelines to verify the software they are using, instead of just relying on its claims of authenticity and integrity.
- **Infrastructure providers:** SLSA provides guidance for hardening the infrastructure they provide (e.g., a build platform or package ecosystem).

### Tracks

The SLSA levels are split into *tracks*. Each track has its set of requirements and goals for the four levels and focuses on a particular aspect of the Supply Chain Integrity. At the time of writing this thesis, the only existing track is the build track, focusing on the transparency and integrity of a package artifact's build process. An SLSA track providing increasing levels of trust in source code[3] is currently at the stage of final review. The focus of this thesis is on the build process, and therefore the levels of the build track are described in the following list:

Level 0: Does not provide any guarantees. It represents the lack of SLSA. Build processes of this level include development or test builds that are run on the same machine. It should be noted that Level 0 might not be considered a proper level of the SLSA standard, which is supported by the fact that the SLSA Source track is made of levels 1 to 4.

Level 1: Requires build information to be recorded as the so-called provenance, which includes information about the entity that built the software artifact, the used build process, and the top-level build inputs. The build information must then also be distributed to consumers. Another requirement is to have the build process be consistent to allow consumers to form expectations about what the build process should typically look like. The requirements of SLSA Build level 1 require only small adaptations that are easy to integrate into existing build workflows without changing them. The level already provides a list of benefits, including easier reviewing and rebuilding of the software and of the build process, preventing software builds from a publicly unavailable state of the source code if the provenance is verified, and providing data to create an overview of software and platforms used in build processes. The provenance of this level does not provide mechanisms against tampering, as it may be incomplete, unsigned, or both.

Level 2: Extends the benefits of level 1 by ensuring that an artifact has not been tampered with after it has been built by requiring the provenance to be signed and its authenticity validated by the consumer. In addition to the requirements of level 1, it is also necessary to run the build on a hosted platform (dedicated infrastructure that is not an individual's workstation), which also generates the provenance and performs the signing of it. This reduces the attack surface of

[3] https://slsa.dev/spec/v1.2-rc1/source-requirements

the build process, as the dedicated build platform can be especially audited and hardened. These requirements also provide the advantage that it requires an explicit attack to evade the verification or to forge the build information, which might still be easy to perform but will also deter unsophisticated adversaries who face legal or financial risks when evading the security controls.

Level 3: Requires builds to be run on hardened platforms that offer strong tamper protection. They must implement strong controls to isolate builds from external influences, which extends to builds within the same project. They must also prevent signing material used to authenticate the provenance from being accessible to the user-defined build steps to protect credentials from being compromised. Build processes protected according to this level prevent tampering during build time and thereby prevent insider threats, credentials being compromised, and other attacks originating from the same host. They provide strong confidence that the software artifacts were built from the official source and build process and ensure that forging build information or evading verification of a build process of SLSA level 3 would require attacks that are beyond the capabilities of most adversaries.

### 2.10.2 Relevance for my thesis

Section 4 describes the threat model that is considered in this thesis. The underlying model of the software supply chain in this thesis is based on the supply chain model of the SLSA standard. The contribution of this thesis aims at providing Supply Chain Integrity measures equivalent to the SLSA build track level 3. The aim is to provide authenticated build information about the inputs and execution of a consistent and simple build process, which is run on a dedicated hosted platform in a hermetically isolated build environment, ensuring isolation from any external influences during build time. The contribution should also prevent user-defined build steps from having access to the credentials used to authenticate the build information.

## 2.11 Reproducible Builds

This section explains the concept of *Reproducible Builds (R-Bs)*. Section 2.11.1 gives an introduction to the concept, and Section 2.11.2 takes a brief look at the history of the concept of *Reproducible Builds*. Section 2.11.3 highlights the practical considerations of *Reproducible Builds* and explains common causes of non-reproducibility. Section 2.11.4 provides insight into the concept of*Reproducible build environments* and lists technologies that provide those.

### 2.11.1 Introduction of the concept of Reproducible Builds

In general, software is installed by the end-user in its binary form, and these binaries are distributed by the vendor after compiling the source code using a certain set of build tools and by including a list of dependencies and libraries in the build process. A general user is unable to retrace the transformation of the source code of software to the binary that is distributed by the vendor or an app store. This might seem logical for proprietary software, for which no source code is publicly available, but it is in general also true for OSS. OSS, though it allows reviewing the source code, does not necessarily provide any mechanisms to guarantee the correspondence of the source code to the binary of the software.

#### Supply Chain Security benefits

A solution to provide a link between source code and the compiled binary is to enable anyone to reproduce the build process by providing a description of the build environment and all the software artifacts involved in compiling the application. The description should be detailed enough to allow anyone to rebuild bit-by-bit identical binaries. This allows trust to be established that the described build process is actually the very same build process executed by the vendor to obtain the APK. Given that the build is reproducible (i.e., bitwise identical build outputs are obtained), an independent party can audit the inputs of the build process and the build process itself. They can analyze the source code, the list of dependencies, and the build tools involved and verify their benevolence.

Fourné et al. [29] emphasize the importance of the concept of R-Bs, as they can provide protection mechanisms for modern *Software Supply Chains*. They allow establishing a strong foundation for defending against attacks on the build system, as any changes in the build process that result in modified build outputs can easily be detected during the verification step. [29]

#### Requirements of Reproducible builds

The definition of reproducible builds by the Reproducible Builds project [64] is as follows:

> "A build is **reproducible** if, given the same source code, build environment, and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts.
>
> The relevant attributes of the build environment, the build instructions, and the source code, as well as the expected reproducible artifacts, are defined by the authors or distributors. The artifacts of a build are the parts of the build results that are the desired primary output."

There exist many more definitions of *Reproducible Builds*, but based on the definition of the Reproducible Builds project [64] it can be summarized that for a build to be reproducible, the following requirements need to be met:

- The whole source code of the application must be publicly accessible.
- All the libraries and dependencies included in the build process need to be available.
- The software providing the build environment needs to be available.
- The description of the exact configuration of the build environment must be provided.

This described build environment, to yield bit-by-bit identical outputs, generally needs to be designed to be deterministic and isolated from the host system it is running on. The build process, given the same inputs, should always provide the same outputs. The concept of reproducible build environments is further described in Section 2.11.4.

#### Advantages of Reproducible Builds

Along with the security benefits, *Reproducible Builds* also provide a few non-security related benefits [46]:

- **Quality assurance:** Reproducible builds complement quality assurance (QA) efforts by identifying issues that may affect the reliability of software, because problems with reproducibility often hint at symptoms of larger issues in the build process or software design. By addressing these issues, developers can improve both the reproducibility and overall quality of their software.
- **Development environments:** The adoption of Reproducible Builds in software development processes allows to ensure that build processes behave consistently across systems, which improves the stability and speed of the development process.
- **Ensure licensing compliance:** distributing the information to make builds reproducible also enforces license compliant usage of libraries and tools, as any abuse could be detected by analyzing the build process.

#### Drawbacks of Reproducible Builds

Making build processes reproducible will also introduce operational overhead, as every input to the build process must be precisely defined to guarantee identical results. This increase in efforts has led some large projects, such as Kubernetes [25], to stop supporting reproducibility of their build processes.

### 2.11.2 History of Reproducible Builds

The widespread adoption of Reproducible Builds as a standard practice in software development and thereby a core part of *Software Supply Chains* is driven by projects and institutions such as the *Reproducible Builds project* [64]. The Reproducible Builds project began in 2014 with the goal of making the Debian operating system fully reproducible [46]. This initiative was particularly significant because Debian is one of the most mature Linux distributions and maintains one of the largest curated collections of free and open-source software (FOSS). Seven years after the beginning of the Reproducible Builds project, approximately 95% of over 30,000 packages in Debian's development branch could be built reproducibly [46]. Lamb and Zacchiroli [46] emphasize that these achievements are thanks to cross-community efforts that enabled such a large scale adaptation of the concept. Outside the Reproducible Builds project, many other software distributors started to adapt the concept of R-Bs (e.g. F-Droid, see Section 3.2 for more information). Researchers like Fourné et al. [29] aim to identify the remaining challenges that need to be overcome, such that R-Bs can become more commonly adapted.

### 2.11.3 Practical Aspects of Reproducible Builds

#### Testing the reproducibility of a build process

Due to the scale of the project, developers at Debian needed to create an automated way of testing build processes for reproducibility. They therefore developed a CI/CD system that builds each package in the Debian archive twice in a row on two independent build systems. Those build systems were deliberately configured to differ as much as possible. In total, more than 30 variations can be applied to the systems; among those variations are changing the system time, having a different host name, running on different kernel versions, or using a custom file system to have varying file ordering [46]. With this infrastructure in place, the reproducibility of the artifacts can be verified with a very high degree of confidence.

#### Root-cause analysis of non-reproducibility

In the case that build processes are not reproducible, their resulting binaries will have differences caused by sources of non-reproducibility. While detecting mismatches between builds is straightforward (e.g., by comparing checksums of artifacts), identifying the root cause of differences can be challenging at times. To address this, the Reproducible Builds project developed *Diffoscope* [10], a tool that analyzes the differences between two provided artifacts. If the artifacts are archives of some sort (e.g. APKs or Zip archives), it will recursively unpack the artifacts and compare the contained files. This allows exact pinpointing of the differences between two artifacts instead of just providing the information that the two artifacts differ. *Diffoscope* supports many file and archive formats and can translate binary formats into human-readable forms. It highlights differences between the analyzed files and provides the found differences as text or HTML output. It can, for example, highlight build dates embedded in binaries and will provide the context of found differences to help developers identify the source code responsible for the variation. While the tool is highly effective in finding the differences, it still requires the judgment of software engineers to attribute the differences to certain causes. [46]

Lamb and Zacchiroli [46] established two main causes for non-reproducibility:

- **Uncontrolled build inputs:** Many toolchains used in conventional build processes can be affected by attributes of the host system they are running on. Such attributes include system time, environment variables, and the build path, and, according to Lamb and Zacchiroli [46] is analogous to breaking the encapsulation concept in software design, as it corresponds to high-level processes being influenced by low-level implementation details.
- **Build nondeterminisms:** The second class of root causes for non-reproducibility can be summarized as build nondeterminisms: aspects of the build process that behave nondeterministically and therefore encode random outcomes in the resulting binaries. These nondeterminisms occur if outputs depend on the state of a pseudo-random number generator or on the arbitrary order of process scheduling.

In the following, a list of typical examples of sources of non-reproducibility is provided, and a list of advice on how to mitigate these sources of non-reproducibility is provided. Section 2.11.4 will introduce reproducible build environments, which are designed to mitigate a high number of the non-reproducibility sources. Section 8 highlights sources of non-reproducibility that were encountered in this thesis.

#### Typical sources of non-reproducibility

Debian hosts a repository [9] containing sources of non-reproducibility they encountered. The repository contains a list of more than 430 different problems, and in the following list, the most common ones will be briefly introduced:

- **System time and timestamps:** Many build processes include the system time or include the timestamps of source files during the compilation phase. The system time is in general different for every invocation of the build process, and also source file timestamps might be unintentionally changed. The *Reproducible Builds project* found that the best solution is to extend build tools to take the build time from the environment variable *SOURCE_DATE_EPOCH* and to then distribute the value of this variable in the recorded build information.
- **File order:** Another common source of non-reproducibility is the processing order of files in a build process. If the order is not explicitly set, they might show up in a different order in the build artifacts.
- **(Pseudo-) randomness:** This class of root causes includes (pseudo-) random values (like temporary file paths and unique identifiers) that can occur during a build process and might be included in the output artifact.
- **CPU-related:** The scheduler might change the order of execution of the task in the build process. If the output binaries are optimized for the current CPU class, they will differ depending on the architecture the binary was compiled on.
- **Memory related:** Without corresponding measures in place, the memory addresses in the binary result might differ between builds. If the memory is not properly initialized by the build process, the built artifacts might also contain the residual values.
- **Build paths:** Depending on the path in which the build is executed, build paths that are included in the results of the build process will differ from system to system.
- **Locale and timezone settings:** These settings might be included in the build output if parts of the build process include this information.

#### Hints and solutions to counteract sources of non-reproducibility

This section provides an exemplary selection of methods that can be applied to build processes to eliminate sources of non-reproducibility [46]:

- **"Jail" builds:** The most fundamental measure is to sanitize the build environments, for example, by running the build process inside a container, such that the build process is always presented a canonical interface to the underlying build system. This method is applied in the reproducible builds of Bitcoin and Tor [46]. Most variants of this method bring some overhead with them and therefore can slow down build speeds. They might also impose technical or societal restrictions on developers, as, for example, to not have a free choice in the tools they use. Most of the build "jailing" methods do not address non-deterministic aspects of the build process either. [46]
- **Provide build parameters:** Environment variables and build parameters that affect the outputs should be explicitly set and provided in the recorded build information or the source code.
- **Stable ordering for inputs:** Listing inputs explicitly or applying sorting algorithms to ensure that multiple inputs are always processed in the same order. This is necessary as directory listings are not stable across file systems.
- **Controlled value initialization:** The problem of accidentally recording memory in the build output can be mitigated by initializing memory regions to a known value.
- **Use deterministic versioning:** Instead of generating the version number of the artifact, the information used to determine the version should be extracted from the source code. It can, for example, be retrieved from the VCS information computed from the hash of the source code or obtained from a changelog entry.

- **Avoid timestamps:** Timestamps are the biggest source of non-reproducibility. Archives, for example, keep the modification times in the metadata of files. These timestamps need to be either avoided by providing arguments to the archiving command or by using tools that remove the timestamps of existing archives. Including timestamps in the build output should be fully avoided if possible, and if they are absolutely needed, the source for setting the timestamp should be explicitly set such that anyone can set the sources to the same value. This can be done, for example, by setting the environment variable SOURCE_DATE_EPOCH, as proposed by the Reproducible Builds project, to a time that can be extracted from the changelog or from the commit metadata in the VCS. [45]
- **Avoid (pseudo) randomness:** To avoid randomness in the outputs of build processes, sources of randomness like pseudorandom number generators should be seeded to a known value, which could, for example, again be provided in the source code. Results of other sources of randomness, like, for example, iterations over hash tables and parallel execution of processes (which can be completed in an arbitrary order due to processor scheduling), should be sorted.
- **Don't record build system information:** Naturally, recording information about the build system in the build outputs will result in non-reproducibility. Therefore, the recording of information like the date and time of the build, the hostname, build path, network configurations, CPU architecture, or environment variables, should be avoided, or if it is needed to be recorded, then these records should be stored outside the binaries.
- **Signatures:** Signing an artifact introduces non-reproducibility because of the confidentiality of the private key that is used to sign. The Reproducible Builds project proposes three different ways to handle such embedded signatures:
  - Ignoring the signature: Using comparison tools, which do not consider missing signatures a difference in the artifacts.
  - Stripping the signature: Removing the signature from the reference binary and then comparing the two artifacts.
  - Copying the signature: In case the signature is desired, it can in many scenarios be copied to the self-built artifact by extracting the signature from a reference binary and patching it into the right place of the self-built binary.

### 2.11.4 Reproducible build environments

The outcome of a general computational process depends on numerous factors, including the hardware, operating system, and software libraries used. Reproducible build environments are tools to abstract these dependencies and decouple the build process from the underlying host system. The reproducibility of the build environment is achieved by eliminating sources of non-determinism. They define the tools that are available to the build process and their specific versions. Depending on the exact type of build environment, they can also specify the operating system, the build architecture, the build path, and the build date and time.

#### List of technologies providing reproducible build environments

The following list is an exemplary set of methodologies used to provide reproducible build environments:

- **Containerization:** A widely used approach is the use of so called containers, and the most popular software in this category is Docker[4]. Docker can be used to achieve

[4] https://docs.docker.com/

reproducibility by defining the exact steps to build an *image*, which is a standardized package including all the files, binaries, libraries, and configurations needed to run the build process in an isolated environment. Users, however, need to make sure to use Docker in such a way that it does not introduce non-reproducibility.

- **Virtualization:** Another approach is to provide a virtual machine in which the build process can be executed. Tools like Vagrant[5] help manage such virtual machines and allow you to define their exact configuration, including the operating system, installed software, and the settings. This is, however, again to be used in such a way that no sources of non-reproducibility are introduced, as the tool is not designed to provide reproducible build environments out of the box.
- **Functional Package Management:** Package managers like Nix[6] and GNU Guix[7] (which builds upon the concepts of Nix) make use of functional builds and isolated environments. Both store software in input-addressed directories to separate the software on a file system level and to prevent interference of individual software versions and variants. For more information about reproducible builds using Nix, see Section 2.15.

## 2.12 Hermetic builds

Many reproducible build environments build on the concept of hermetic builds to ensure the reproducibility of the build process. Hermetic builds support the guarantee that passing the same inputs (source code and configuration) to a hermetic build system will result in the same outputs (e.g., binary software artifacts) by isolating the build process from changes external to the build system.

To ensure the isolation of the build process, hermetic build environments are insensitive to the libraries and other software installed on the host machine. They manage the build tools (e.g., compilers) and dependencies (e.g., libraries) themselves by requiring all tools and dependencies to be explicitly declared and fetched in a controlled manner. This might require the developer to specify an exact version number or even the hash value of the software artifact to unmistakably identify it. Since hermetic build systems do not rely on system-wide installations to perform the build process, they are self-contained, and the tools and other dependencies are stored inside a managed file tree, which is independent of the host system.

The build process is then executed in sandboxed environments to achieve runtime isolation, strengthening the consistency of the build environment and it being unaffected by external factors. To ensure reproducibility, hermetic builds also disable network access during the build process. This prevents the build from fetching dependencies or data dynamically, which could lead to non-deterministic results.

Section 2.15.3 highlights the mechanisms used by Nix to ensure that the build process is hermetically isolated.

## 2.13 Bootstrappable Builds

Bootstrappable builds [8] are a concept that is closely related to the concept of reproducible builds, as they share some of the guarantees they want to provide. Both try to ensure that only the expected code and logic are present in the final output of build processes. Reproducible builds do so by requiring the developer to specify all the dependencies and tools that are used to build the final software artifact. This, however,

[5] https://developer.hashicorp.com/vagrant
[6] https://nix.dev/manual/nix/2.30/
[7] https://guix.gnu.org/

requires consumers of reproducible builds to still trust the included dependencies and tools, except if the reproducible build concept is applied recursively to all the involved software artifacts.

At the very end of this recursive approach, there is a single compiler that was built from another compiler, which creates a chicken-and-egg problem and in general requires users and developers to trust that the old compiler does not contain any hidden malicious logic as theorized by Thompson [79]. This is where bootstrappable builds come in, which try to reduce the amount of trust needed to put into the initial binary standing at the very start of this recursive build tree. Bootstrappable builds try to do so by reducing the size of the initial binary by providing tools that are byte-wise smaller compared to the initial compiler and then building the compiler from those smaller tools.

The concept of bootstrappable builds, together with reproducible builds, provides auditability of the whole source code involved in creating a certain binary artifact.

## 2.14 Build Attestations

Build attestations contain information about the build process of a build output. They describe how the output was built and what it contains. Build attestations allow it to inspect a build output, its origin, the entity that created it, how it was built, and which inputs were used to build it. This information enables developers to estimate the impact a software artifact has on the supply chain integrity of their software. It also allows including or excluding dependencies based on policy rules. [39]

The in-toto framework (see Section 3.1.4 for more information) provides a curated list of concrete attestation schemes that can be used to provide build attestations[8].

Recent advancements, such as Attestable Builds (A-Bs) proposed by Hugenroth et al. [36], enhance the concept of build attestations by leveraging Trusted Execution Environments (TEEs) to ensure secure and verifiable build processes. A-Bs integrate attestations with transparency logs, allowing recipients to verify the origin, integrity, and build process of an artifact. This approach mitigates risks such as tampering with source code or build environments and provides strong guarantees of source-to-binary correspondence.

[8] https://github.com/in-toto/attestation/tree/main/spec/predicates#vetted-predicates

## 2.15 Reproducible Builds using Nix

This section introduces the concepts of the Nix programming language, the Nix package manager, and related terms (e.g., Nix expressions, Nix Flakes). First an overview of the terms used in the following sections is given, then more information on the concepts relevant to the thesis is given.

### 2.15.1 The Nix deployment system

In his thesis [22], Eelco Dolstra in 2006 proposed *Nix*, a purely functional software deployment model. With this approach to software deployment, he intended to solve a number of flaws and problems of other, widespread software deployment systems. Among these problems are issues concerning the environment of the deployed software: presence, compatibility, and finding of the software's dependencies; and issues concerning the management of the deployed software: updating, removing, and rolling back an update of the software and its dependencies can create side effects, which interfere with the correct functionality of other software deployed on the system.

The software deployment model proposed by Dolstra should incorporate all the required features, which are also provided by other models, and solve as many of the remaining problems as possible. A complete explanation of the flaws and shortcomings of other deployment models and the advantages of Nix can be found in the thesis [22].

One of the core aspects of the Nix deployment system is that it deploys software in individual parts, which are installed in individual locations on the file system. These parts are the basic units of deployment, so called *components*. Components are in general files or directories with arbitrary nesting levels, and the concept of components in Nix corresponds to the concept of packages in package management [22] and therefore Nix is often referred to as a package manager. The individual installation locations of components allow for the coexistence of software (parts) in multiple variants or versions on the same system without interference. Every component is stored in the *Nix store*, which is a designated directory in the file system (usually at `/nix/store`). Each component in the Nix store has a unique path through which the file system-level isolation is established (more information in Section 2.15.4). The path of components is identical if and only if the component is derived from the same inputs. In addition to the isolation of components, Nix also ensures that deployed components are immutable.

Due to the isolation of the components and the absence of side effects of software deployments:

- Nix supports atomic upgrades that ensure that the system cannot end up with invalid installations if the upgrade process is interrupted.
- Nix allows for constant time rollbacks of a list of applications installed via Nix to a previous list.
- Nix supports automatic garbage collection of unused components (more information in Section 2.15.4)

Nix is at its core a source based deployment system but allows, transparently, to deploy software in binary form instead of building it locally (more information in Section 2.15.4). In order to build the components from source, Nix provides core functionality to sandbox the build environment. This can be used to ensure reproducible environments across different systems, which will result in reproducible builds provided that the build process itself is sufficiently deterministic and that it does not fail if it cannot access information outside the sandboxed build environment (more information in Section 2.15.3).

### 2.15.2 Nix expression language

Nix uses the Nix expression language to describe how to build individual components and how to compose them [22, 23]. These descriptions are called Nix expressions [22]. The Nix expression language is a simple, purely functional language and allows Nix to express build environments in a self-contained and reproducible way. Variability of components can be expressed by describing the components as functions of the desired variability. The language is lazily evaluated, which means that it only evaluates values when they are needed. This allows to define many components in a single file or expression without having to build or even evaluate them all.

```
result = pkgs.stdenv.mkDerivation {
    name = "derivation name";
    version = "X.Y.Z";
    src = pkgs.fetchGitHub {
        # reference to the source code
    };
    buildPhase = ''
        # running the build of the application
    '';

    installPhase = ''
        # persisting the output in /nix/store
    '';
}
```

Listing 2.1: An examplary Nix-code snippet showcasing the invocation of the mkDerivation function.

Listing 2.1 shows exemplary code that calls the *mkDerivation* function—a wrapper function for the *derivation* primitive. A dictionary, which in Nix is called an *attribute set*, is provided as the only argument of the function.

### 2.15.3 Building a component

In order to be reproducible, build processes of components should be deterministic and depend solely on their inputs. The inputs involved in building are typically the software source code, the build scripts, arguments, or environment variables, and the build time dependencies[9]. Before components can be built, however, their (high-level) Nix expressions need to be translated into *derivations*. These derivations are single, specific, and constant build actions, which, in comparison to Nix expressions, cannot express variability. They are stored in the Nix store, which allows for unique identification of these objects of source deployment, encoding the inputs from which the components are derived.

The command `nix-instantiate` evaluates Nix expressions into derivations. These resulting derivations can then be built (i.e., realized) using the command `nix-store -realize`. The high-level package management command `nix-build` combines translation and building of the components.

#### Sandboxed Nix builds

Nix is designed to ensure that all the inputs that influence the creation of components are only inside the Nix store, or, in other words, that component builders are not influenced by external factors. This is a fundamental design principle that ensures that the build environment is identical across different systems. A build process that is not influenced

[9] Runtime dependencies are also provided as inputs alongside with the build time dependencies

by outside factors is referred to as *pure*, and any breach of this isolation is called an *impurity*.

It should be noted that this isolation is different from the isolation that is provided by the naming convention of store objects, which prevents access to components in the store if they are not explicitly mentioned but does not give any isolation mechanisms that prevent access to inputs from outside the Nix store.

To ensure purity, Nix builds components in a sandboxed environment (see Section 2.12 for general information on hermetic builds), isolated from the normal file system hierarchy. When a derivation output is built, it will only see its dependencies in the Nix store, the temporary build directory, and private versions of */proc* and */dev* [18]. On Linux, builds run in private namespaces to isolate them from other processes on the system and from the internet [57]. Nix has built-in functionality to clear the environment variables [22]. Build processes only see the key-value pairs they are provided in the derivation as environment variables, alongside special environment variables, like *$out*, which contains the file path corresponding to the outpath of the derivation. [18, 57]

#### Fixed-output derivations

In addition to building components using derivations, Nix allows to create Nix store entries, for example, from files that are fetched from the web using the concept of fixed-output derivations. These are derivations of which the output, or rather the cryptographic hash of the output, is known in advance. This allows to verify the integrity of files fetched through functions that provide fixed-output derivations. An example of a function producing fixed-output derivations is the *fetchurl* function, which allows to download a file from a given URL. An important aspect for the usability of this function is that the cryptographic hash for the store path of the output does not depend on the URL of the fixed-output derivation, but rather on the content that is being fetched. This allows for updating the URL of the download location, given that the file is still identical, without subsequently needing to update all the derivations that depend on the output.

### 2.15.4 Nix store

The Nix store is the core of the Nix software deployment model and is usually located at `/nix/store`. In contrast to the conventions of the File Hierarchy Standard (FHS), Nix stores all components (e.g., outputs of the build processes of derivations) in subdirectories of the Nix store. Any direct child of the Nix store directory is either a component or an auxiliary file for Nix. These components are also called *"store objects"* [22]; the full path of a store object is referred to as the *store path* (e.g. `/nix/store/<component-name>`).

#### Naming in the Nix store

The name of a component is composed of a representation of a cryptographic hash and a symbolic name. The cryptographic hash is computed over all inputs involved in the building of the component. These inputs typically consist of the source code of the component, the script performing the build, any arguments or environment variables passed to the build script, and all the dependencies, which most often include build time dependencies, like, for example, a compiler, a linker, libraries, and other standard Unix tools, alongside the runtime dependencies [22].

Using a cryptographic hash ensures with very high probability that no two components result in the same hash value if their inputs differ, which also provides guarantees that it is in general infeasible to find a second set of inputs, different from the first set of inputs, that generates the same hash value but a different component (see Section 2.4 for more information on cryptographic hashes).

Any two components that do have the same inputs will evaluate to the same Nix derivation and therefore also have the same output path of the derivation. Provided that the build process is deterministic, two components with the same inputs will result in identical binaries.

These aspects allow Nix to assume that a build that has succeeded once will result in the same component if it is rebuilt, and it therefore comes with built-in functionality to avoid unnecessary rebuilds. It does so by registering the output path of a derivation as *valid* upon successful completion of the build of the derivation. This way a derivation is built only once (as long as the garbage collector was not run meanwhile), and if the realization of the derivation is triggered again, the build will not be executed, because the component exists already. The building of the derivations is also the step that is replaced by the binary deployment, which is further explained in Section 2.15.4.

The usage of cryptographic hashes to derive the store path of components allows the coexistence of component versions and variants even if they have the same name. This is possible, because their differences in the inputs imply that their output paths differ and that the result of the build process will be stored in different directories in the Nix store. Conflicting dependencies, such as different versions of a compiler, no longer cause an issue, as they are stored in isolation from each other. For instance, if package A depends on dependency D of version 1.0.0 and package B depends on dependency D of version 1.2.0, both versions can be installed alongside each other, as the storage in different directories will prevent any faulty dependency resolutions.

#### Garbage collection

Runtime dependencies of software deployed by Nix are specified as store paths within binaries that point to the exact component the software needs. Nix keeps track of these runtime dependencies by scanning through the binaries of deployed software and registering found store paths as dependencies of this component. If no package depends on a specific component, the component will be removed from the file system the next time the Nix garbage collector is run. The same is true for software that was originally installed by a user, but was uninstalled again. The software is not immediately removed from the file system, but rather deleted by the garbage collector, when it is executed, assuming that no other software depends on it.

#### Binary Deployment and Substitutes

Nix is a source based software deployment model but allows for binary deployment as well. The transparent source/binary model of Nix allows combining the flexibility of source deployment systems with the efficiency of binary deployment systems. The binary deployment is enabled by a central repository, which can be queried for output paths for which a pre-built component is available. The repository provides a so called *substitute*, which in its most general form is any file or program that creates a store object through some means apart from the normal build process and thereby substitutes that process. Most often the substitute is used to obtain the output of a derivation by downloading a pre-built file system object (FSO) from a server and placing the FSO into the Nix store.

It should be noted, though, that Nix does not enforce a policy on how substitutes produce store objects and that there is no guarantee that a substitute produces correct output or that the integrity of the resulting component was not compromised. This is a problem for secure deployment and implies a need to trust the repository from which the substitute is obtained.

### 2.15.5 The Nix Packages collection

The Nix Packages collection (Nixpkgs) is a collection of software packages that can be installed using the Nix software deployment model. It contains Nix expressions that describe the components and their composition for more than 120,000 packages. [13]

### 2.15.6 The standard environment

The *standard environment* (stdenv) is a component described by Nixpkgs with special significance, as it provides the basic tools needed for almost all other components. It also includes some convenience functions, for example *mkDerivation* a function to simplify the creation of derivations. [22]

### 2.15.7 Nix Flakes

Nix Flakes are an experimental feature of the Nix package manager. Flakes were introduced in 2021 with Nix version 2.4 and provide a standard way to write Nix expressions (and thereby also Nix packages).

Their concept is closely related to the concepts of VCS and repositories. The term *flake* refers to a file-system tree that contains a *flake.nix* file in the root directory of the tree. The *flake.nix* files are similar to Nix expressions but have special restrictions and a specific structure. [58]

They improve reproducibility of software packaged with Nix outside nixpkgs, among others by pinning versions of dependencies in a lock file. This lock file can be updated programmatically by invoking the command *nix flake update*.

### 2.15.8 Relevance for the thesis

Nix is used in this thesis because it provides tools to ensure the reproducibility of build environments across systems. It guarantees that the same environment variables, tools, and build instructions are available when building the same Nix expression on different systems. It should be noted, however, that Nix itself does not yet guarantee reproducible builds (i.e., bitwise identical outputs), as this requires not only the build environment to be reproducible but also the build instructions to ensure deterministic outputs. If the build process contains instructions that introduce non-determinism (e.g., a random number generator or key generator), then the build output becomes non-reproducible. These mechanisms improving the reproducibility of builds are also provided by many other tools that provide reproducible build environments, but Nix offers additional features that are needed in this thesis.

Nix provides mechanisms to isolate build processes from the host system and the network, which forces the writer of the Nix expression to provide all the necessary tools, libraries, and dependencies within the Nix environment, which ensures transparency of the inputs that influence the build output. Nix, using fixed-output derivations, allows to ensure that artifacts, like, for example, tools and dependencies, that are fetched from the web, are precisely what the developer of the Nix expression intended—it thereby verifies the integrity of this artifact. The isolation mechanisms in combination with the integrity verification of external inputs provide strong fundamentals to harden the build process against manipulations by malicious actors from outside or within the Nix build environment.

Nix also allows for the concurrent existence of tools in multiple versions and variants on a single system, which is needed to build different applications that need the tools in different versions.

## 2.16 Verifiable Logs

The following sections introduce the concept of *verifiable logs* [34, 48, 49], a data structure that allows logging data entries in an append-only fashion, such that entries—once added—cannot be deleted or modified anymore. Verifiable logs provide auditability that also allows consumers of the transparency to detect if the log provider were to secretly add a log entry. These properties and their verifiability allow consumers to rely on a data structure that logs data entries in a transparent, append-only fashion, without needing to trust the provider of the verifiable log.

### 2.16.1 Merkle tree

A popular example of designing a verifiable log is to use a Merkle tree [55] as the underlying data structure. The log starts off empty and is mutated by adding entries. These entries can be arbitrary data that is added as leaf nodes of the tree. The interface that allows the data to be arbitrary and application specific is called *personality* in the context of Google Trillian [33]. A personality defines the data model and validates data entries against it. It should be noted that the tree does not have to be balanced and therefore allows storing an arbitrary amount of data.

### 2.16.2 Root node

Nodes above the leaf nodes hold the cryptographic hash value of their immediate children. This way the value of the root node of the tree depends on the value of its direct children, their value depends on their direct children, and so on, which has the effect that the root node depends on the values of all the nodes in the tree and that if any changes are made in any of the leaf nodes (e.g., by changing a data point), then the root node value will change as well.

### 2.16.3 Consistency proof

One general way to verify that the old log entries are still present in the new log would be to download the whole log and to search it for all the old values. This, however, is very storage- and bandwidth demanding and scales badly.

The fact that the root node of a Merkle tree "summarizes" the state of the verifiable log allows interested parties (so called *monitors*) to efficiently verify the append-only property of the log. A monitor may store previous values of the root node and in order to verify that the mutation of the tree is valid, request the values that have been added since the last root node value has been stored. It then computes the expected new state of the Merkle tree and compares the computed root node value to the one provided by the verifiable log. If the two values match, the new state of the verifiable log is valid, and the new root node value can be saved for a future consistency proof.

### 2.16.4 Inclusion proof

Other parties might be interested in verifying that the tree includes a certain set of entries. In order to prove to the parties (so called *auditors*) that these entries are present, the verifiable log provides a list of the values of the nodes along the shortest path from the respective leaf representing a log entry to the root node of the tree. The auditor can then compute the expected value of the root node based on the provided list of hash values. If the expected hash value of the root node corresponds to the value that the verifiable log publishes, then the entry is present in the Merkle tree.

### 2.16.5 Availability and synchronization

In order to increase the availability of verifiable logs, they should be hosted on multiple servers across the globe. This, however, introduces new problems of consistency between the different instances of the log and requires a synchronization strategy [47].

### 2.16.6 Split-view attacks

Verifiable logs can be susceptible to the so called split-view attack [54, 59]. Malicious log providers can present different log representations to different clients while still maintaining the append-only property and therefore seeming valid to parties that perform inclusion and consistency proofs on the log. One way of counteracting such attacks is to have globally distributed independent parties that regularly query the root node values of multiple logs. These parties are then called *witnesses* [54, 76] and a consumer of the verifiable log can ensure the absence of a split-view attack by verifying that (most of) the witnesses agree that the current root node value corresponds to the one presented to the client.

### 2.16.7 "Bad" entries in a verifiable log

The presence of an entry in the verifiable log does not say anything about the "correctness" of the content of the entry. Even in the absence of a split-view attack the verifiable log can contain a bad, malicious, wrong, or invalid entry. Such an entry must then be marked as "bad" in some way (e.g., by revoking it in the case of certificates [47]).

### 2.16.8 Relevance for the thesis

Verifiable logs allow us to provide a global infrastructure to distribute data that is needed in trust-establishing processes without needing to trust the provider of the log itself. This can be leveraged by parties (e.g., the developers of E2EE messenger applications) to provide reproducible builds and to publish the R-B process and result in an auditable, append only log. This log will then also include the hash value of the software that can be built from the R-B and thereby allow consumers of the software to verify the integrity of their application by comparing its hash value against the one published in the log.

In order to ensure that the R-B processes and their results are valid, they need to be verified by independent parties. These parties must then be able to mark the log entries as invalid in case the published results are not reproducible, and consumers must be able to know which entries have been marked as invalid. If this cannot be provided by the verifiable log infrastructure, the consumers cannot trust the entries of the log and must therefore verify the R-B themselves and the verifiable log does not add any value.

# Chapter 3

# Related work

This section will explore other projects and tools aiming to improve supply chain integrity of general software (Section 3.1) and of Android applications in specific (Section 3.1.6.)

## 3.1 Projects and tools to improve Supply Chain Integrity

Since the topic of Supply Chain security has gained importance in recent years, many projects aim to provide methods and structures to improve it. These projects allow, for example, verifying dependencies of a build process, creating isolated build environments, or independently verifying the correspondence of source code and the binary representation of an application.

### 3.1.1 Gradle dependency verification

Including external dependencies and plugins in a build process opens up new attack vectors (see Section 2.9.2) and thereby creates a need to verify the integrity of those third-party artifacts. Tools like the *Gradle Build Tool* [40] allow for automatic verifications of dependencies. It natively supports verifying the integrity and provenance of dependencies and plugins used in the build process and detecting compromised artifacts. In order to verify the dependencies, the developer of the build scripts first needs to create the `verification-metadata.xml` file, which will contain so-called *components* for each dependency, describing its name, group, version, checksum, and optionally the signature of its supplier.

This file can be generated by the Gradle Build Tool when running a build with the corresponding parameters [40], but then requires the developer to still verify created components before being able to verify the dependencies in future builds. Once the `verification-metadata.xml` was created, Gradle automatically uses it in subsequent builds to verify the dependencies.

#### Cargo Vet

In the realm of Rust programming, *Cargo Vet* can be used to improve Supply Chain Integrity. The tool ensures that developers audit and verify the source code of third-party dependencies before they can be used in a Rust project—a concept that could be applied in many other development ecosystems to strengthen the trust in the Supply Chain. The `cargo-vet` policy file then lists the dependencies that are trusted and which require further review.

### 3.1.2 Gitian

Gitian [66] is a secure source-control oriented software distribution method that addresses the problem of verifying that a binary is derived from some source code by making the

build process of software packages reproducible. It does so by executing the build step inside a virtual environment, the isolation of which protects the host from malicious build processes trying to compromise the host system and protects the build process from influences by the host system. The virtualization of the build environment, however, in practice also results in an increased number of tools that need to be trusted.

#### Build environment and information recording

Gitian ensures reproducibility of the build process by building the artifacts within virtualized build environments, the contents of which are measured and recorded in the `.assert` file. After the build, the `.assert` file contains hashes of all inputs of the build process, as well as the hash of the produced output. The file can then be signed using public key cryptography to ensure its authenticity before it is published.

#### Verification process

Gitian allows to verify the build process by letting independent parties execute it and confirm that the same output is obtained. The first step in the verification process is to retrieve the source code of the specified Git commit, identified by its hash value. Then the build process is executed through Gitian, which will generate the build output and the `.assert` file, which should describe the same initial state and outcome as the original `.assert` file. If the generated `.assert` file differs from the one provided by the supplier, even though the inputs of the build process were the same (i.e., the hashes of the inputs in the `.assert` file match), then this means that the build was not reproducible.

#### Limitations of Gitian

While Gitian provides a great concept to ensure reproducibility of the build process, the guarantees of a Reproducible Build using Gitian are limited to being able to audit the build path of a single artifact and do not include the build process of any dependencies or tools included in building the artifact. Inputs to the build process consist of the hashes of the installed Debian packages; this ensures the reproducibility of the build process but does not provide any trust relations, as long as the inputs are not individually trusted or audited. Effectively, the solution provided by Gitian moves the problem of ensuring reproducibility upstream by one step, as the verifiability makes it no longer necessary to trust the binary output of the build process but still requires trusting every binary that is part of the build environment.

### 3.1.3 Guix

GNU Guix [17, 18] is a software deployment and distribution tool that supports provenance tracking, reproducible builds, and reproducible software environments. In comparison to Gitian, it allows modeling reproducible builds for more than just the application in question and allows extending the reproducibility aspect to dependencies of the build process. It is a fork of the Nix package manager and therefore inherits some of its attributes, like, for example, being a purely functional package manager, supporting transactional upgrades, rollbacks, unprivileged package management, per-user profiles, and garbage collection. [18].

Guix distributes packages exclusively by providing source code and package definitions describing how to build the applications from source. The source code is distributed using Git, VCS and updating applications consists of updating the local copy of Guix source code. Guix provides an authenticated way to retrieve new source code revisions from Git in order to combat source code threats. [17]

### 3.1.4 in-toto Framework

in-toto [80] is a supply chain security framework, which aims to provide mechanisms to protect the software supply chain across multiple of its phases. It does so by verifying that each step of those phases was executed as intended by the authorized party and that the result of the step was not tampered with in transit.

The in-toto framework works by requiring the project owner to create a software supply chain layout, which is a machine-readable description of the individual building steps of the software. Steps along the supply chain are carried out by specifically assigned parties, the so-called *functionaries*. These functionaries carry out their entrusted step while recording information about the used commands and related files. This information is stored in a metadata file, which, at the end of the execution, is cryptographically signed by the functionary to ensure its authenticity. This metadata file is used to link inputs and outputs of the build step to the functionary. Recording of build step execution is done by either invoking the commands `in-toto-record start` and `in-toto-record stop` before and after the build step, respectively, or by executing the build step through `in-toto-run`.

in-toto, in contrast to Gitian, puts fewer constraints on the execution environment of the build steps and thereby allows for greater flexibility to model the build steps the way they are executed in real-world environments.

#### in-toto Attestation Framework

The *in-toto Attestation Framework* is an extension of the basic framework that allows adding additional arbitrary authenticated metadata to the description of the individual build steps. This enables a more complete description of the build environment, which in turn allows to verify build steps against policies without the need to make the underlying supply chain layout more rigid.

#### Verification process

The verification in the in-toto framework ensures that each build step of the layout was executed by the specified functionary, and a layout is considered verified if:

1. All inputs and outputs in the cryptographically signed record put together form a chain of build steps without any gaps.
2. The records match up with the layout provided by the project owner.
3. The records are signed with authorized keys of the functionaries.

### 3.1.5 Extending Cloud Build Systems to Eliminate Transitive Trust

In [69] Schwaighofer et al. propose extensions to existing cloud build systems to eliminate the shortcomings of many other build systems when it comes to transitively trusting dependencies of the build process. They designed their approach to allow users to verify transitive dependencies by attaching verifiable metadata to build outputs. This metadata includes build information, details about the build environment, and a remote attestation to enable users to independently verify the trustworthiness of build steps and dependencies.

The approach distinguishes itself from others like Gitian [66] and in-toto [80] by a few key aspects:

- **Flexibility in Execution Environments:** Unlike Gitian, which runs build steps within a constrained virtual machine environment, the approach does not impose

strict constraints on execution environments, allowing greater flexibility in modeling build steps closer to how they are executed in real-world environments. It does however build on Nix and thereby imposes other restrictions that origin from the unique properties of the functional package manager.

- **Elimination of Transitive Trust:** The proposed approach enables independent verifications of transitive dependencies, while Gitian and in-toto rely on trusting the providers of dependencies of the build process to verify or trust their dependencies.
- **Decentralized Trust Models:** It allows users to define their trust models to verify the trustworthiness of the build outputs. This contrasts with other concepts like the one of the in-toto framework, in which the supplier of the software defines the trust model and distributes responsibilities belonging to the Supply Chain of the software.

### 3.1.6 Tools that are tailored to improve Android Supply Chain Integrity

Some sources of non-reproducibility specific to the way Android applications are built make applying supply chain integrity methods harder for a typical Android application build processes. This section will highlight a few tools that have been created to mitigate these problems for Android application supply chains. For further tools designed to help improve the reproducibility of build processes, we refer to the comprehensive list by the *Reproducible Builds project* [65].

### 3.1.7 apksigcopier

The tool *apksigcopier* [72] was written by FC (Fay) Stegerman and helps to create bitwise identical builds of signed Android packages by allowing valid signatures to be used as a build input for the reproducible build process. These signatures are obtained by copying them from a signed reference APK and can then be patched into unsigned APKs.

The tool *apksigcopier* is packaged for the *Nix package manager* (see Section 2.15.5) and can therefore be used in Nix scripts by providing it as a build input. The tool, in addition to the signature copying functionality, also provides a verification method, which allows to verify whether the binaries of a signed and an unsigned APK are bitwise identical. For this, it will copy the signature as described above and subsequently check the validity of the signature for the previously unsigned application—if the signature is valid, it means that the builds are identical (see Section 2.5).

### 3.1.8 reproducible-apk-tools

*reproducible-apk-tools* [73] is another tool by FC (Fay) Stegerman to help make Android applications more reproducible. It is a collection of scripts to help eliminate sources of non-reproducibility (e.g., changing line endings from LF to CRLF).

The tool was not used in this thesis because none of the potential sources of non-reproducibility, which this tool could counteract, appeared in the differences that occurred when reproducing the build process of the Android applications. It could, however, be needed for future extensions of this work, as other build processes might produce differences that can be eliminated using the *reproducible-apk-tools*.

## 3.2 Reproducible Builds on F-Droid

F-Droid supports reproducible builds of the apps that are distributed via their app store. Originally apps were published by being uploaded by the developer and signed by F-Droid before publishing it to the repository. Since 2022, however, F-Droid has encouraged

developers to build their applications reproducibly. They provide guidelines and tools to help developers to achieve this. With this push of F-Droid towards reproducibility, aside from providing general benefits of reproducible builds (see Section 2.11.1), it ensures that applications use only free and open-source software to build. [12]

# Chapter 4

# Threat model

This section defines the threat model, which consists of the threats that this thesis aims to mitigate. The threats are derived from scenarios of two fictional people who were already mentioned in Section 1. The first person is a common user, the second an investigative journalist.

First, Section 4.1 gives a more detailed description of the two people, and the threats that are relevant for the fictional people are mentioned respectively. Subsequently, the threat model based on these descriptions is defined in Section 4.2. The threat model is greatly based on the model from the SLSA, but some threats are considered out-of-scope for this thesis. The threats that are mentioned in the SLSA, but which are not included in this thesis are discussed in Section 9.1.

## 4.1 Detailed introduction of the fictional people

This section provides background information on the two fictional individuals upon whom the threat model of this thesis is based. It begins by introducing each individual and exploring their motivations for using E2EE messaging applications. The discussion then shifts to identifying key threat actors, their potential objectives, and the attack vectors they might use.

For each fictional individual, a brief scenario is outlined to offer a concrete example of how such threats could impact them. The section concludes with an analysis of their awareness of potential threats to the integrity of the applications and the confidentiality of their communications, as well as an assessment of the resources and motivations they possess to ensure the application's integrity. These descriptions serve to illustrate the feasibility of the proposed approach for these individuals.

### 4.1.1 A common user

The first fictional individual represents a typical user of E2EE messaging applications. Their primary goal is to communicate privately and securely with friends, partners, and co-workers. They want to be able to send messages without anyone but the intended recipient being able to receive and read the messages. No government or other third party should be able to read, filter, or otherwise manipulate their potentially sensitive communication.

#### Possible threats and adversaries

A generic user may be vulnerable to untargeted attacks on their privacy. Entities that have an interest in undermining the user's privacy include law enforcement agencies seeking to filter private conversations for illegal content, totalitarian governments aiming to surveil their population, and vendors that want to collect user data for purposes such as targeted advertising.

Other adversaries that may target the E2EE messaging application used by a general user group include hackers. Their objectives could range from stealing sensitive information (e.g., company secrets) to committing identity theft or other forms of fraud.

The aforementioned adversaries may achieve their goals, depending on their available resources, through the following methods:

- Introducing a bugdoor (see Section 2.8) or other vulnerabilities into the application's source code to compromise its confidentiality.
- Tampering with dependencies that are used during runtime or build time of the application, such that the resulting app contains malicious code.
- Compelling, bribing, or compromising vendors or distributors to release a malicious version of the application.
- Abusing the leakage of the vendor's signing key, which is used to sign valid APKs, to distribute malicious applications signed with the compromised key.

#### The scenario

In this scenario, the user of E2EE messaging applications lives in a country that intends to compel vendors of such messaging applications to include certain software logic capable of scanning the content of conversations. The user wants to be able to verify that no such backdoor or scanning logic has been silently included by the vendor or any other entity along the software supply chain.

#### Awareness and available resources

The user is aware that an application might contain unwanted or malicious logic and is willing to make a additional effort to verify that the application has not been tampered with. Their typical method of obtaining applications involves searching for them on their preferred app store and installing them from there. They would like to be able to download applications from a potentially untrusted source and to be able to verify the integrity and trustworthiness of the application before installation.

### 4.1.2 An investigative journalist

The second fictional individual is an investigative journalist working on sensitive topics. The specific nature of their investigation is not particularly relevant in this context, as there are numerous reasons why journalists must be able to rely on the security and integrity of the software they use [74]. These reasons include the need to protect themselves, their sources, or the sensitive information or stories they uncovered. Ensuring privacy is critical for protecting both the journalist and their sources. An application, for example, must not leak any information, such as identities, locations, schedules, or address book contents. Address books can be particularly sensitive, as their leakage can expose multiple sources at once. [74].

#### Possible threats and adversaries

For journalists, adversaries are often the subjects of their investigations, such as totalitarian governments, corporations engaged in illegal or unethical activities, or other powerful entities that want to conceal the information uncovered by the journalist. Additional adversaries may include rival news organizations or a government intelligence agencies interested in their work [74].

Therefore, working as a journalist can involve specific risks and needs respective security measures. These measures begin with basic computer security practices, such as using strong passwords and undergoing awareness training to mitigate phishing attacks. Once these foundational measures are in place, protections against more advanced attacks should be added. Such attacks include compromising the applications or the operating system of the devices used by a journalist. A trojanized application or vulnerability in the software could grant adversaries access to sensitive messages and other private information.

Similar to the threats faced by general users, journalists are vulnerable to attack vectors such as:

- Backdoors or "bugdoors" introduced into the software to compromise its confidentiality.
- Tampering with runtime or build dependencies of the application, resulting in malicious logic as part of the application.
- Unauthorized modification of the application by a malicious distributor.
- Abuse of a leaked signing key to distribute malicious applications with a valid signature.

Journalists are also at risk of more targeted attacks by powerful adversaries. Adversaries might, for example, exploit a vulnerability in the source code of the messaging application or tamper with their network connection when they download the APK of the E2EE messaging application.

As a high value target, journalists may also face non-software-related attacks with the goal to obtain information from them or to prevent them from continuing their work. These attacks can include legal, social, physical, and other technical methods [74]. While these attack vectors are relevant, they fall outside the scope of this software-security-focused thesis and are not further addressed.

#### The scenario

In the scenario considered in this thesis, the journalist operates in countries with limited press freedom and a certain degree of censorship. They need to be able to securely share sensitive information and findings with their colleagues. Given the dangerous nature of their work and the sensitivity of their communications, the journalist must have complete confidence in the confidentiality and integrity claims of the E2EE messaging applications they use. They also need a way to verify the integrity of a downloaded APK, to ensure that the application has not been tampered with.

#### Awareness and available resources

The journalist has access to some resources and trusted collaborators who can help establish a secure method to obtain a trustworthy messaging application. However, they themselves have only limited time available to invest in the verification process and in general, do not have the resources to locally build an application from the source code. Their typical approach to obtaining an application involves visiting the vendor's official website, downloading the application directly, and following the instructions to verify the integrity of the downloaded binary.

## 4.2 Definition of the threat model

For both fictional characters, the described threats and attack vectors occur along the links of the software supply chain of the E2EE messenger applications they use. The

considered threat model therefore includes only threats in that scope. It does not include threats outside the software supply chain, like, for example, Social Engineering attacks [44], attacks on the operating system [4] or potential backdoors existing in the hardware of their devices [3].

The considered threat model includes the following threats:

- **Source threats:** Changes to the source code that either do not reflect the intent of the software developer or that had to be implemented because the software developer was compelled to do so. The changes now put the security or privacy of the user of the application at risk.
- **Build threats:** Changes to the build process or the inputs of the build process that lead to an artifact in the output that contains unintended logic or logic that is unwanted by the user.
- **Dependency threats:** These are indirect threats, as they can introduce unwanted behavior in the application by compromising an artifact that is part of the application's build time dependencies[1].
- **Availability threats:** These are the potential threats denying a user access to the source code or other information that is needed to audit and verify the software. The eventuality of the unavailability of the software itself is included as well, as that prevents users from securely communicating through the application, too.
- **Distribution threats:** This class of threats includes the possibility of a malicious or compromised distributor providing modified software packages that include unwanted logic.
- **Verification threats:** These are threats that are concerned with the verification process of the software and include the possibility of unauthorized changes to the records containing the information that enables the verification of the artifacts or exploitations of cryptographic principles underlying the verification process.

The following sections expand on the short descriptions of the threats listed above and give concrete examples of how the application's integrity could be undermined by those threats. The underlying base model is taken from the software supply chain threat model of the SLSA framework [16].

### 4.2.1 Source threats

The first kind of threats we will look at in more detail are attacks on the integrity of the source code. They include, for example, the introduction of vulnerabilities or backdoors in the code. Some potential scenarios of this category include

- **Insider threats:** A developer within the vendor company becomes malicious and includes harmful logic in the source code of the application.
- **Legal obligation:** The company providing the messenger application is compelled to include some malicious or unwanted logic and changes the source code accordingly.
- **Malicious contribution:** An external developer contributes to the development of the software and submits some changes that introduce unwanted logic—possibly in the form of a bugdoor, which makes the vulnerability harder to detect during the review process.
- **Compromisation of underlying infrastructure:** An adversary gains control over the system hosting the source code, and through an administrator interface or manual changes on the system, they manage to modify the source code.

[1] Runtime dependencies are not considered in the Dependency threats of the threat model, as they are considered to be individual artifacts, that have to be individually protected against Supply Chain Attacks. This is in line with the model of SLSA [16].

### 4.2.2 Build threats

These threats include any threats that introduce unwanted logic during build time—without changing the source code. Potential scenarios with that result are

- **Modifying build scripts**: An adversary gains modifying access to the build scripts and can introduce unwanted modifications of the software logic by changing the build scripts of the applications. These changes might, for example, then build a version that was not intended for production releases or build the application excluding important security features.
- **Changing build parameters:** An adversary might gain access to the interfaces to change the build parameters that dictate the behavior of the build process and alter the resulting application—again, for example, by excluding certain security features.

These subtle changes of the build process could occur, for example, because of an insider threat, a legal obligation, or an adversary that compromised the underlying infrastructure.

### 4.2.3 Dependency threats

Another threat category includes attacks in which a dependency of the artifact is tampered with. The unwanted changes in the logic of the dependency introduce subsequent changes to the behavior of the application.

- **Artifact replacement:** The adversary tampers with network availability during dependency resolution of the build process and lets the user download a modified artifact instead of the intended one.
- **Hostile takeover:** The adversary undermines the integrity of an artifact on which the target application depends (e.g., by offering to take over its maintenance and subsequently introducing malicious updates).
- **Dependency source threat:** Similar threats as for the source code of the application also apply to the dependency of the application. An adversary might submit changes to the source code of the dependency through similar ways as described above.
- **Compromised building tools:** If the adversary manages to compromise a building tool that is used when building the application, then the compromised building tool could introduce changes in the software logic without changing the source code.
- **Compromise included files:** If the adversary can introduce vulnerabilities in files, like assets or prebuilt binaries, that are included in the build process, then these vulnerabilities could also be passed on to the resulting application.

### 4.2.4 Availability threats

This category of threats includes any attack on the availability of resources to verify the application or the availability of the application itself.

- **Missing source code:** The vendor of the application might make the source code unavailable, which renders the verification impossible.
- **Missing source code meta-information:** The vendor might squash commits or remove other meta-information and thereby reduce the auditability of the source code or accountability for changes in the source code.
- **Missing dependency:** A dependency of the application might become (temporarily) unavailable, possibly due to an attack by an adversary. Without this dependency, the user cannot verify the integrity of the application anymore.

### 4.2.5 Distribution threats

This category of threats includes all the risks that occur in the distribution phase of software.

- **Malicious or compromised distributor**: The main threat in this category is the possibility that a software distributor turns malicious, is legally compelled, bribed, or compromised. Independent of the exact reason for the threat, the main problem would be that the distributor stops providing the original APK and offers an APK that has been tampered with to the public or to specific individuals. Under certain circumstances this attack might be very subtle, for example, when the developer is required to give the distributor the ability to sign releases [20].

### 4.2.6 Verification threats

Verification threats include any risks that might make it impossible for the user to verify the integrity of the application.

- **Leaked signing key:** If the signing key was leaked, an adversary can publish arbitrary applications with valid signatures by the vendor.
- **Modified verification configurations:** If the adversary gains access to the configuration and recorded values that are used to verify the application and if they can modify them, any verification process is compromised and cannot be trusted anymore.
- **Exploiting cryptographic weaknesses:** If an adversary finds weaknesses in the cryptographic structures used in the verification process, like, for example, the hashing algorithm used or the signature protocol, then they can forge any verification procedures or push arbitrary applications with the vendor's signature.

## 4.3 Threats in scope

Not all the threats of the described thread model can be mitigated by the proposed approach. Section 7 evaluates the concrete limitations of the approach and which threats are mitigated by it.

# Chapter 5

# Software Supply Chain of an Android application

This section looks at the general steps that are typically encompassed in the Software Supply Chain of an Android application. For this, the supply chain of an application, from providing the source code until the distribution of the APK, is considered. First, Section 5.1 gives a brief overview of the single steps of the Supply Chain and visualizes them in Figure 5.1. Sections 5.2.1 to 5.2.10 look at the steps in more detail, highlight tools that are involved in those steps, and identify which threats could emerge in the respective phases of the software cycle.

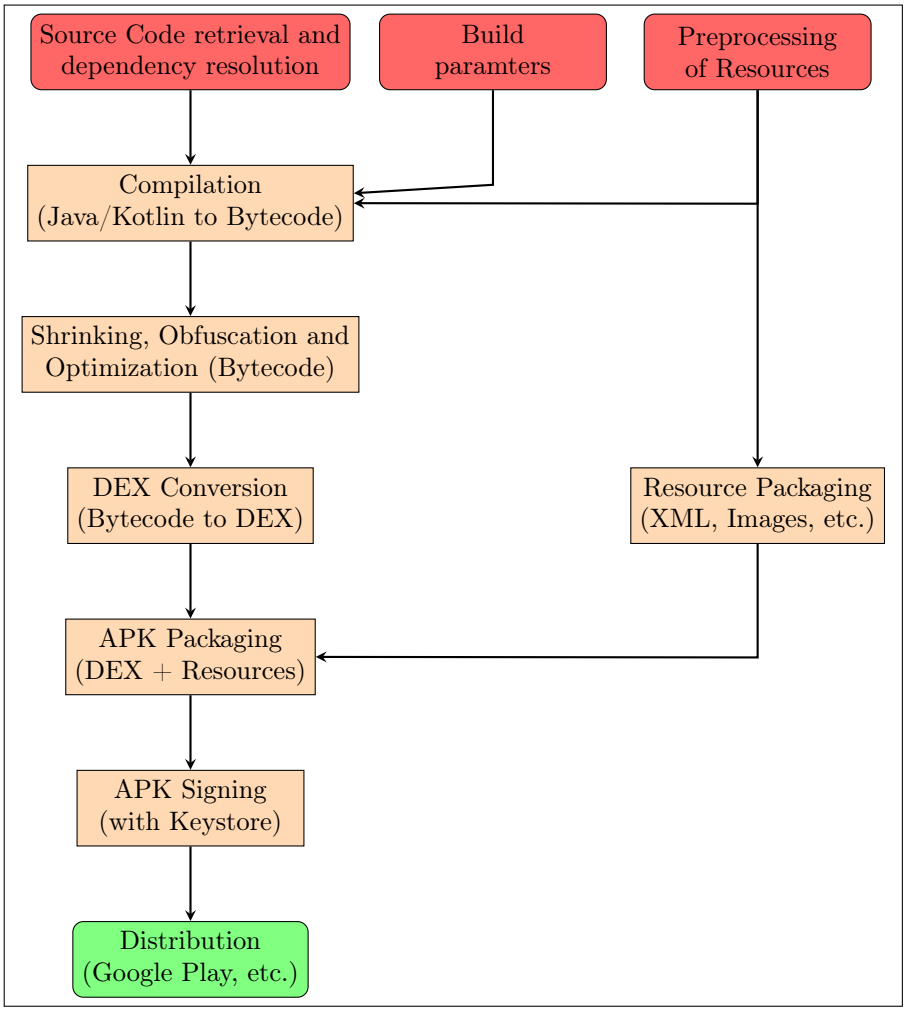


Figure 5.1: The Software Supply Chain of a generic Android application

## 5.1 Overview of the Software Supply Chain

Before listing the single steps of the Supply Chain the limitations of this model are briefly noted:

- **Output artifact:** The Android ecosystem supports different kinds of artifacts that can be the result of the build process. However, in order to keep the model simple, only build processes resulting in an APK of the application are considered. Similar risks and principles apply to the Supply Chain Security, independent of whether the output is an APK, an AAR, or an AAB (see Section 2.1 for more information).
- **Source code:** In the software supply chain model in this thesis the source code is assumed to be managed by a VCS. Any access or modification of the source code occurs through the VCS.
- **Compilation in cloud:** The compilation process of the source code is assumed to be executed in a CI/CD pipeline. This does mean that the model is not meant to include compilation processes executed within a development environment on a local machine. The thesis in general focuses on Gradle build processes to limit the scope for the implementation of the proposed approach (see Section 2.7 for more information).

A graphical overview of the steps of Software Supply Chain of a generic Android application can be seen in Figure 5.1. The single steps are:

1. **Source Code 5.2.1**: The starting point is a specific version of the application's source code.
2. **Build parameters 5.2.2**: The parameters for the build process are passed to the build tool via the build environment.
3. **Resource preprocessing 5.2.3**: Resources are compiled into binary format.
4. **Compilation 5.2.4**: The source code is compiled into Java bytecode.
5. **Optimization and Obfuscation 5.2.5**: In this optional step, the resulting Java bytecode is shrunk, optimized, or obfuscated. The exact actions in this step depend on the build parameters.
6. **DEX Conversion 5.2.6**: The bytecode is subsequently converted into Dalvik Executable (DEX) format.
7. **Resource Packaging 5.2.7**: Resources such as XML files, images, and other assets are packaged separately.
8. **APK Packaging 5.2.8**: The DEX files and resources are combined into an APK.
9. **APK Signing 5.2.9**: The APK is signed using a keystore to ensure its authenticity and integrity.
10. **Distribution 5.2.10**: The signed APK is distributed via platforms like Google Play or other channels.
11. **Verification 5.2.11**: This optional step verifies the authenticity and integrity of the APK to be installed.

## 5.2 Detailed steps of the Supply Chain Model

In the following sections we will expand on the short descriptions of the single steps of the Software Supply Chain.

### 5.2.1 Source code

The process starts with the source code written in Java or Kotlin. The creation of an application naturally involves many steps before that, like, for example, the usability and security design of the application, but in this thesis a specific version of the source code is considered as the starting point for the software supply chain.

Developers will have written Java or Kotlin code within their preferred development environment (like, for example, Android Studio) and have committed the changes to a central VCS. In this thesis we consider the applications to be open source and the VCS of the software project to be hosted on a publicly accessible server or platform (e.g., on a GitHub[1], GitLab[2] or Gitea[3] instance). Many open source projects allow external developers to suggest changes and will integrate them if they align with the goals and guidelines of the project.

The source code also specifies the dependencies of the application, which are typically managed via a dependency management system (e.g., Gradle's dependency resolution mechanism). These dependencies include third-party libraries, internal libraries (e.g., shared modules within the organization), and plugins.

#### Source Threats

In this stage, a malicious insider could commit changes to the VCS that will include malicious logic into the application. If external developers can submit changes as well and they are not carefully checked, this is another threat.

#### Dependency Threats

While the actual consequence of including malicious dependencies only takes effect in later steps of the Android build process, it should be noted that by applying measures, untrusted dependencies can be detected and excluded in this step already and thereby protect the following steps from including malicious logic.

### 5.2.2 Build parameters

The process (e.g., a CI/CD pipeline) that starts the Android build will call a tool to handle the build itself and provide it the necessary build parameters. In most Android builds, the tool that orchestrates the overall process is Gradle [51]. It manages dependencies and build configurations and is controlled by providing it the correct parameters. In the most basic form, this includes the requested build task (e.g., calling `gradle assembleWebsiteProdRelease`).

#### Build Threats

Changing the parameters that are passed to Gradle will result in a differently configured build process and can potentially lead to vulnerabilities, for example, due to skipped build tasks. In general, unauthorized modifications to the process that starts the build itself are to be considered a threat to the Software Supply Chain as this provides the enclosing environment in which the application is built.

[1] https://github.com/
[2] https://about.gitlab.com/
[3] https://about.gitea.com/

### 5.2.3 Resource Preprocessing

One of the first steps in the general Android build process is the compilation of the application's resources. The tool Android Asset Packaging Tool (AAPT) is used to compile all resources in the *res* directory into a binary format. Outputs of this step are the binary versions of the files and the *R.java* file that contains all the resource IDs. [41, 71] Resources may also come from included libraries and asset packs. These sets of external resources are merged with the set of the application's resources, and potential conflicts have to be resolved.

#### Source Threats

In this step the main threats come from existing vulnerabilities hidden within the assets that have to be compiled. These vulnerabilities might propagate through the compilation process and compromise the resulting binaries, which then can result in a compromised application once the compiled assets have been merged into the APK.

#### Dependency Threats

In addition to that, compromised tools, libraries, and asset packs are a threat. If, for example, the used version of AAPT contains malicious logic, attacks similar to the one described by Ken Thompson [79] could be possible.

### 5.2.4 Compilation

The compilation step is one of the most central steps in the Software Supply Chain of an Android application. It combines the source code with external libraries (e.g., `.jar` or `.aar` files) to produce Java bytecode. Most Android application projects use Gradle as the tool to orchestrate the build process; alternatives are Apache ANT, Apache Maven, and Eclipse ADT. [51]

Many more tools and plugins are involved in the compilation of the Java or Kotlin source code of an Android application. These include, for example, the Java Compiler `javac`, the Kotlin Compiler `kotlinc`, which are used to convert Java or Kotlin source code into Java bytecode (`.class` files), respectively. [71]

Android Gradle Plugin (AGP) is one of the plugins that is present in Android builds using Gradle to extend Gradle's capabilities to manage the full build process [71]

Another tool that might be invoked in this step is an annotation processor that generates code for the annotations in the source code [41] and thereby introduces logic into the software that is dependent on the behavior of the annotation processor.

#### Dependency Threats

If one of the libraries, tools, or plugins used in this phase contains malicious logic, it can alter the outcome of the build process without any changes in provided source code. These unauthorized changes, therefore, will remain undetected if one does not analyze the resulting binaries. This threat is again very similar to the attack described by Ken Thompson [79].

### 5.2.5 Optimization and Obfuscation

This is an optional step in the Software Supply Chain, but is rarely omitted because all modern applications are usually shrunk and optimized to ensure a minimal need for storage and CPU power to install and run the application. Many use cases also require some form of obfuscation of the software, which is also done in this phase. Whether the tasks for shrinking and obfuscating the application are executed depends on the configuration and build parameters provided. Common tools used for this step are ProGuard[4] and R8[5], which has better Kotlin support and more size reduction.

#### Dependency Threats

If one of the tools or plugins used in this phase contains malicious logic, it can alter the resulting artifacts of this step, similar to the threats described in Section 5.2.4. The rules configuring the tools used in this step are often provided by external libraries, and therefore, if these dependencies have been compromised, they can alter the functionality and structure of the resulting bytecode.

### 5.2.6 DEX Conversion

In this step, the Java bytecode (`.class` files) is converted into DEX format. This is necessary, because DEX is optimized for Android Runtime (ART). The DEX format consolidates multiple `.class` files into a single `.dex` file, which leads to improved runtime performance due to the thereby reduced overhead of the binary code. In general, the tool used to convert the Java bytecode to DEX is `d8`[6]. `d8` also enables the usage of Java 8 features by converting newer features to Java 7 bytecode in order to ensure the bytecode is supported by ART. [41, 71]

#### Dependency Threats

This step, similar to the optimization and compilation step before, is susceptible to changes in the binary code.

### 5.2.7 Resource Packaging

In this phase of the Supply Chain the already compiled resources are packaged to be included in the APK in the next step. The packaging requires AAPT (e.g. `aapt` or the newer version `aapt2`[7]) and the result of this phase are the resources, merged into a single set to avoid conflicts [41].

#### Dependency Threats

This step could also introduce vulnerabilities or bug doors into the assets of the application and thereby introduce the vulnerabilities or a bug door in the application itself. Due to its rather linear transformation and combination of the assets, it would be rather suspicious if additional logic was added in this step, but nonetheless this step should be protected.

[4] https://www.guardsquare.com/proguard
[5] https://developer.android.com/topic/performance/app-optimization/enable-app-optimization
[6] https://developer.android.com/tools/d8
[7] https://developer.android.com/tools/aapt2

### 5.2.8 APK Packaging

In the final step of the build process of the Android application, the DEX files, the compiled resources, and other assets, like, for example, native libraries, are combined to a single APK file. The tools executing this step are Gradle and AAPT. The resulting artifact can in general be either the APK or the applications AAB[8]. In this thesis we assume, however, the resulting artifact to be an APK, to simplify the supply chain model. [71]

#### Dependency Threats

This step could also introduce unwanted logic into the application if the used tools have been tampered with. However, similar to the resource packing step (see Section 5.2.7), it would be rather odd if this step introduced changes in the binaries of the application.

### 5.2.9 APK Signing

After the APK was built, it needs to be signed in order to verify to the user that this application was actually built by the vendor. Therefore, the developer will sign the APK with a private key stored in a keystore that is passed to tools, like, for example, `apksigner`[9] (recommended) or `jarsigner`[10]. The signature of the APK will then ensure the authenticity and integrity of the APK to the user and will allow them to install it on their device. [63]

#### Dependency Threats

Furthermore, this step could introduce malicious changes to the final application. By modifying the application, this would, however, be very suspicious, as the signing process in general does not modify the binaries of the APK anymore and rather just appends the signature and the information to verify the signature to the APK.

### 5.2.10 Distribution

In the distribution step, the APK is uploaded to the distribution platforms, like, for example, F-Droid or Google Play or other platforms. Some of these platforms require the developers to outsource the signing of the application to them and thereby requires the developer to upload their key to the distribution platform.

#### Distribution Threats

If the platform turns malicious or is compromised, it could distribute applications that have been tampered with but will have a valid signature.

### 5.2.11 Verification

In the verification phase, the authenticity and integrity of the APK to be installed is verified. The basic methodology to ensure the authenticity of the application is to verify the signature of the application. More targeted attacks might require the downloaded

[8] https://developer.android.com/guide/app-bundle
[9] https://developer.android.com/tools/apksigner
[10] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jarsigner.html

APK to be checked by verifying the hash value corresponds to the hash value of the reference APK, to verify that no unwanted modifications have been introduced at any point of the supply chain. In order to compare the hash value of an APK to the expected hash value, a user needs to be able to obtain such a reference value securely.

### Verification Threats

If the infrastructure providing the verification mechanism is compromised or the user's system is compromised, the verification might either be impossible to complete or might be successful even for applications that have been tampered with. Another verification threat arises if the user is unable to obtain a reference value to check the APK against.

# Chapter 6

# Contribution of this thesis

In Section 4 the threat model of the scenarios considered in this thesis was defined. Section 5 explored a typical software supply chain of Android applications and applied the threat model to the individual steps of the supply chain.

Section 6 explores the contribution of this thesis: an approach to verify the integrity of Android applications, that allows users to learn the exact digital resources that were involved in the creation of the application. Section 6.1 provides a short summary of the proposed approach.Section 6.2 reasons about the selection of messaging applications for which the approach has been implemented. Section 6.3 subsequently looks at the concrete Software Supply Chain of the selected applications.

Section 6.4 covers the implementation of the approach.

Sections 6.7 through 6.13 explore parts of the implementation in more detail, and Section 6.14 concludes the contributions with the description of the verification process.

## 6.1 Summary of the proposed approach

The approach proposed in this thesis aims at ensuring the integrity of the supply chain of E2EE messenger applications for Android. Part of the approach is to provide reproducible builds for the applications, to allow anyone to rebuild bitwise identical binaries. Like other instances of reproducible builds, the implementation part of this thesis allows a user to rebuild the APK and to verify the resources involved in building the Android applications for Signal and Wire. In contrast to many other instances of reproducible builds, this implementation can give guarantees, that no additional resources have been introduced during the build process, due to the hermetic isolation of the build environment.

## 6.2 Selection of end-to-end encrypted (E2EE) applications

This section explains the process that led to the selection of Signal Android and Wire Android as the two applications to be considered in this thesis.

For the selection of potential applications to be considered in this thesis, they had to meet some criteria:

- **Available for Android:** This thesis set out to analyze and mitigate threats of Android applications, and therefore the first criterion was the availability of the application for the Android platform.
- **E2EE messaging apps:** The applications needed to be E2EE messaging apps, to align with the focus of the thesis.
- **Open source:** The last mandatory criterion was, that the application's source code needed to be publicly available, as this is needed in order to be able to reproduce the build process of the application.

| Messenger | Repository Type | URL |
|---|---|---|
| Signal | Github | https://github.com/signalapp/Signal-Android |
| Wire | Github | https://github.com/wireapp/wire-android |
| Threema | Github | https://github.com/threema-ch/threema-android |
| Cwtch | Other server (Gitea) | https://git.openprivacy.ca/cwtch.im/cwtch-ui |
| Briar | Other server (Gitlab) | https://code.briarproject.org/briar/briar |
| Element | Github | https://github.com/element-hq/element-x-android |
| Telegram | Github | https://github.com/DrKLO/Telegram |

Table 6.1: List of repositories hosting the source code of E2EE messenger applications for Android

These criteria were fulfilled by a list of applications, and the list of considered candidates can be seen in Table 6.1 alongside the URLs pointing to their publicly available source code.

The final selection for the applications Signal Android and Wire Android was based on a subjective ranking among the possible candidates derived from qualitative criteria, "popularity" and "usability." The limitation of the final selection to only two applications was due to time constraints of the thesis. This leaves the possibility to extend this list of applications in future projects.

## 6.3 Supply Chains of selected Android applications

This section takes a brief look at the existing software supply chains for the selected messenger applications: Signal Android and Wire Android. The implementations of the secured supply chains in Section 6.4 are based on the existing supply chains.

### 6.3.1 Existing Supply Chain: Signal Android

The source code for Signal Android [28] is hosted on GitHub, as can be read on their blog [12] about the reproducibility of the application. On every major version release, Signal provides a release on GitHub, which includes a universal APK, that is suitable for most Android devices and the source code from which the released APK is built. The published source code corresponds to the state of the code of the commit that has been tagged with this respective version.

Signal provides instructions [27] on how to build their Android application reproducibly, from which can be taken how to build the universalAPK that is published on GitHub, and the versions of the required tools can be taken from one of the build scripts. They also provide the `verification-metadata.xml` file, which can be used to verify the dependencies used by Gradle during the build process. This file is, however, incomplete, the implications of which will be discussed in Section 8.2.

The signing step of Signal Android is not available in their source code, but would naturally also not be replicable by a third party, as they do not have access to the signing key.

After the building and signing of the APK, the app is distributed. It is published to GitHub as an APK and uploaded to Google Play as an AAB, which allows users to download the applications that are optimized for their device.

In order to verify the application, Signal provides the Python script `apkdiff.py`, which can be used to compare the APK to be installed to a reference APK.

### 6.3.2 Existing Supply Chain: Wire Android

The source code of Wire Android [32] is hosted on GitHub as well, which can be learned from their support website[31]. Similar to Signal, Wire Android provides artifacts for every major release on GitHub, which includes the source code that was used to build the application and a build of the Android application. Alongside these artifacts, Wire Android also publishes the file `version.txt`, which includes version codes and other information, that was generated and used during the build process of the published APK.

The command that builds Wire Android can be taken from one of the GitHub workflows. When building from the source code, one has to take care to also check out the submodule *kalium*, as this is required to build the application. Wire Android does not provide a `verification-metadata.xml` file to allow for verification of the dependencies. A user, therefore, in general cannot ensure to use the same dependencies as were used in the compilation of the APK published by the vendor.

The signing of the application is done as part of the build process that is executed by the GitHub workflow, but is again not replicable, due to the necessary confidentiality of the used signing key.

Once the application is built and signed, it is published at a few different places, including on Google Play, on F-Droid, and on their website.

Wire Android does not provide instructions of how the integrity of the downloaded APK can be verified.

## 6.4 Implementation of this thesis

This section provides an introduction to the implementation of the proposed approach to ensure the integrity of the software supply chain of the two Android applications. The subsequent sections (Section 6.7 until 6.12) give more detailed information on the individual parts.

The implementation part of this thesis consists of multiple components:

- **A simple GitLab pipeline:** This CI/CD pipeline is configured to execute automatically according to the predefined schedule. It will start a Nix-shell, which, as part of the `shellHook`, runs the script `update.py`, and if the update script has found a new release, the pipeline will continue by trying to build the Nix derivation of the new release. The build might fail in the beginning, because it is still missing dependencies. To add those dependencies, the script `extend_verification_metadata_xml.py` is called and the build restarted until all dependencies have been added and the build succeeds. The outputs of the build process are saved as pipeline artifacts on the GitLab instance.
- **Update.py:** This Python script queries the repositories of the Signal Android and Wire Android for the newest release. If a new release is found, the Nix files and other helper files will be updated to build the newest version of the applications.
- **Extend_verification_metadata_xml.py**: This script is used to extract the missing dependencies from the error log of the build process and will programmatically add the necessary dependencies to the `verification-metadata.xml` file of the respective application.
- **Flake.nix:** This file contains the expressions to build the derivations, which run the build process to produce the Android applications and compare the APKs to the respective reference applications. Some logic is outsourced into other Nix scripts to increase the readability.

### 6.4.1 Implementation of the supply chain of the Android applications

Figure 6.1 should give an abstract overview of the steps of the process executed by the derivations declared in the Nix Flake. The included steps are:

1. **Fetch Source Code:** The first step of the Supply Chain is to fetch the source code from the repositories of the applications.
2. **Include `verification-metadata.xml`:** If the source code of the application does not include the `verification-metadata.xml` already, or if the provided file is incomplete, the build process has to include a file that was generated in a previous step (e.g., during the update step).
3. **Fetch Dependencies:** In the next step, using the tool `gradle-dot-nix` [43] all the dependencies and plugins that are needed during the build process are downloaded and cached in a locally provided Maven repository.
4. **Build in a hermetically isolated environment:** The build process of the Android application is executed within the hermetically isolated build environment of the Nix derivation, to isolate the build process from the internet and the influences of the host systems.
5. **Signing the APK:** In the next step, the APK resulting from the build process in the previous step gets signed. One should note, that because the signing key is inaccessible, the signature is added to the APK by copying the signature from the reference APK and thereby is only valid if the binaries of the self-built APK are bitwise identical to the reference binaries.
6. **Comparison to reference APK using `Diffoscope`:** In this step, theAPK, published by the vendor, is downloaded to provide a reference APK. This reference APK is compared to the self-built APK using the tool `Diffoscope` [10]. The result of this step is an *html*-file that contains the differences between the two applications, if there are any.
7. **Publish: APK, Hash value, `Diffoscope` result:** In the last step of my Supply Chain model, the hash value is computed by calling `sha256sum` on the self-built APK and subsequently published as an artifact of the CI/CD pipeline, alongside the application itself and the *html*-file from the comparison in the previous step.

## 6.5 The CI/CD pipeline

The CI/CD pipeline is executed at scheduled times, and for every application it executes an update stage and a build stage. Listing 6.1 shows exemplary code for the stages for the Signal Android application.

In the update stage, the pipeline first executes a script, which updates some meta-information that is saved in the application specific directory. These files contain, for example, the current version number of the app, the commit hash of the current version, and other information that is provided as input for the build process of the application. Next, the pipeline executes a script to update all version numbers and hashes of the fixed-output derivations within the Nix expressions that are needed to build the application. The script uses the tool `nix-update` by Jörg Thalheim [78] to perform this step. The last step in the update stage ensures that the build process will be able to access all needed dependencies and therefore updates the `verification-metadata.xml` file for the respective application (see Section 6.13.2 for more information).

In the build stage of the application, the pipeline essentially just calls the `nix build` command for the respective derivation of `flake.nix` (see Section 6.6) that builds the

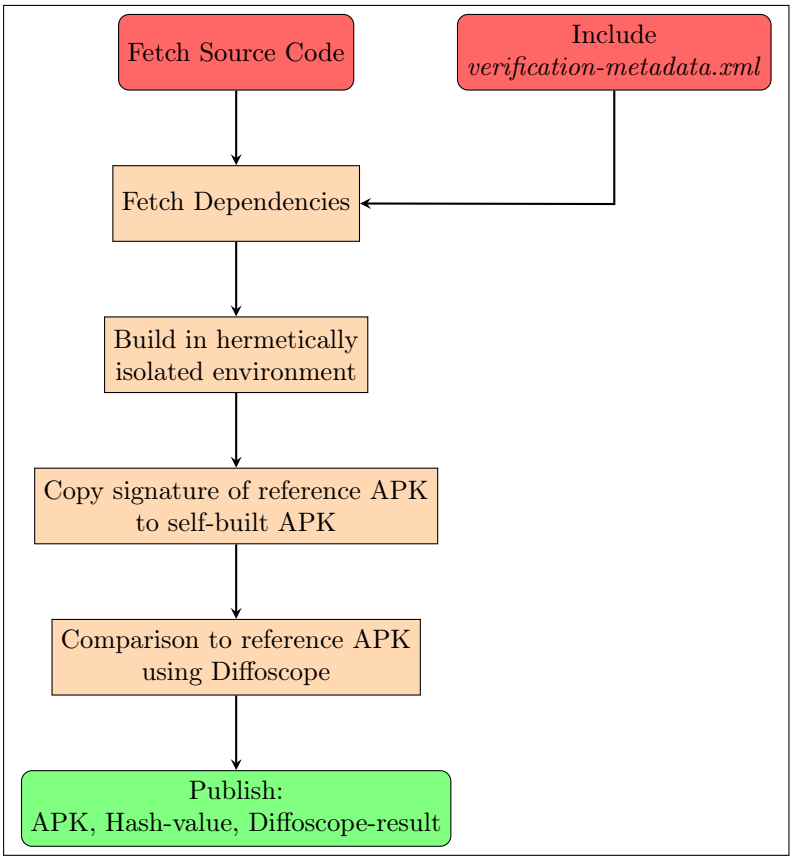


Figure 6.1: Abstract overview of the steps of the build process implementation.

application. The derivations ending in `...-diffoscope` also compare the resulting binary with a reference APK that was downloaded from the official repository of the application vendor. The parameter `-L` prints build logs, and the parameter `-o path` is used to provide a path to a directory in which the symlink to the build results will be created. This stage depends on the update stage to be executed beforehand and will provide the build outputs of the derivation as pipeline artifacts.

```
default:
  tags:
    - nix-container-shell

variables:
  BUILD_DIR_SIGNAL: build_signal
  # ... more build folders for other apps

stages:
  - update-signal
  - build-signal
  # ... more stages for other apps

update-job-for-signal:
  stage: update-signal
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"  # Only run on scheduled pipelines
    - if: $CI_PIPELINE_SOURCE == "web"       # Or manual triggers
  script:
    - nix-channel --add https://nixos.org/channels/nixpkgs-unstable nixpkgs
    - nix-channel --update
```

```
22      - nix-shell
23
24      - git config --global user.email "ci@git.ins.jku.at"
25      - git config --global user.name "GitLab CI"
26      - git add .
27      - git diff --staged --quiet || git commit -m "Automated script changes [skip ci]"
28      - git push https://oauth2:$CI_PUSH_TOKEN@git.ins.jku.at/proj/android-device-
        security/source-transparency/android-clients.git HEAD:$CI_COMMIT_BRANCH
29
30  build-job-for-signal:
31    stage: build-signal
32    script:
33      - nix build -L -o $BUILD_DIR_SIGNAL .#signal-diffoscope
34    dependencies:
35      - update-job-for-signal
36    artifacts:
37      paths:
38      - $BUILD_DIR_SIGNAL/*
```

Listing 6.1: An exemplary code snipped showcasing the update stage and the build stage of the Signal Android application.

## 6.6 The Flake

The core of the Nix expressions for building the Android applications is the `flake.nix` file in the top-level directory of the repository. A single Nix flake can contain multiple derivation definitions, and in this case it defines the needed derivations for each messenger application that should be built reproducibly (which currently includes Signal Android and Wire Android; see Section 6.2 for more information). The derivations describe how to build a specific Android application from the source code and how to compare the self-built APK to a reference binary and produce the result as an output.

Listing 6.2 shows an exemplary code snippet of the `flake.nix` file, which contains the necessary expressions that the approach uses to build an application (in this case, Signal Android).

At the beginning of the file, a string describing the flake is given, followed by the declaration of its inputs. These inputs describe the dependencies of the flake and in this case, include `nixpkgs`, `android-nixpkgs`, and `gradle-dot-nix`. `nixpkgs` [13] is a very central dependency of many flakes, as it provides access to more than 120,000 packages, which include compilers, libraries, development tools and other software. It also provides fundamental functions that are helpful for building derivations, such as `mkDerivation`, `fetchurl`, and `fetchgit`. `android-nixpkgs` [26] is a package by Tad Fisher that provides all packages from the Android SDK repository as inputs for Nix expressions. It is used to provide the necessary build tools in the required version to the build process of each application. `gradle-dot-nix` [43] is a package by Stefan Kempinger that allows Gradle projects to be build within Nix derivation builds, as it fetches dependencies (that can be declared in a `verification-metadata.xml` file) in a Nix compatible way and populates a local Maven repository within the Nix store with these dependencies. It then provides a Gradle init script that can be used to configure the Gradle project build to look for dependencies in this local Maven repository.

These inputs are then passed to the `outputs` function in line 13 in Listing 6.2. Within the outputs function, a `let` expression introduces local bindings and then proceeds to define an attribute set for each application (for Signal Android, it is called `signalOutputs`). This attribute set is defined using another `let` expression, which defines `signal`, which

in turn is an attribute set returned by the Nix expression that builds the Android application (see Section 6.9 for more information), and `signal-reference-binary`, which is a derivation that provides a reference binary that the self-built binary can later be compared against. The derivation is defined using `pkgs.stdenvNoCC.mkDerivation`. It should be noted that it was a purposeful design choice to use `stdenvNoCC` to make sure that no unnecessary binaries are provided in the build environment. The derivation defines the package name (`pname`) and the version attributes to ensure that the tool `nix-update` can correctly recognize the derivation and update the version and hash of the fixed output derivation that is provided by `pkgs.fetchurl`. The URL to the reference binary is thereby constructed dynamically using the `version` attribute.

The `signalOutput` attribute set contains a development shell (see line 46 in Listing 6.2) and a set of packages for the target system, that is defined beginning from the line 36 in Listing 6.2. For the Nix expression for Signal Android, this set of packages consists of the following:

- `signal-reference-binary`: which, if build, results in the reference binary that is also used by the comparison algorithm.
- `signal-maven-repo`: which can be used to debug the local Maven repository to ensure that it is populated as expected. This proved very useful to ensure that the repository worked correctly when the apps would not build because they were unable to find a required dependency.
- `signal-built`: provides the self-built APK provided in the `signal` attribute set from line 19 in the Listing 6.2.
- `signal-diffoscope` compares the self-built APK to the reference binary using the lightweight `pkgs.runCommand`. The commands used to perform the comparison are described in Section 6.11.

In line 60 of the Listing 6.2 the outputs of the individual applications are merged into a single set of packages to allow for building single output derivations using `nix build .#NAMEOFTHEDERIVATION`.

```
1  {
2    description = "Building the End-to-end-encrypted Android applications using Nix";
3
4    inputs = {
5      nixpkgs.url = "nixpkgs";
6      android.url = "github:tadfisher/android-nixpkgs";
7      android.inputs.nixpkgs.follows = "nixpkgs";
8      gradle-dot-nix.url = "github:CrazyChaoz/gradle-dot-nix";
9      gradle-dot-nix.inputs.nixpkgs.follows = "nixpkgs";
10     # ... other versions of nixpkgs for specific gradle versions
11   };
12
13   outputs = {self, nixpkgs, android, gradle-dot-nix, ...}@inputs :
14   let
15     system = "x86_64-linux";
16     pkgs = import nixpkgs { inherit system; };
17
18     signalOutputs = let
19         signal = import ./nix/signal-build.nix { inherit system pkgs inputs; };
20         signal-reference-binary = pkgs.stdenvNoCC.mkDerivation rec {
21           pname = "signal-reference-binary";
22           version = "7.56.10";
23
24           src = pkgs.fetchurl {
25             url = "https://github.com/signalapp/Signal-Android/releases/download/v${
    version}/Signal-Android-website-prod-universal-release-${version}.apk";
26             sha256 = "sha256-FvIwtF8TLqbUxnHtZJBIjAKLA8+xlgNpawv4dH9g1EI=";
```

```
27            };
28
29            phases = [ "installPhase" ];
30            installPhase = ''
31              mkdir -p $out
32              cp -r $src $out/Signal-Android-website-prod-universal-release-${version}.
      apk
33            '';
34          };
35        in {
36          packages.${system} = {
37            inherit signal-reference-binary;
38            signal-maven-repo = signal.maven-repo;
39            signal-built = signal.built-apk;
40            signal-diffoscope = pkgs.runCommand "diffoscope-comparison" {
41              nativeBuildInputs = [ pkgs.diffoscope pkgs.apksigcopier ];
42            } ''
43              # ... comparison of the self-built APK to the reference binary using
      diffoscope
44              '';
45          };
46          devShells.${system}.signal = signal.devShell;
47        };
48
49      # default to ensure that is executed when `nix build` is run without specifying a
        specific derivation
50      defaultOutput = {
51        packages.${system}.default = signalOutputs.packages.${system}.signal-diffoscope
      ;
52      };
53
54      allOutputs = [
55          signalOutputs
56          # ... outputs of other applications
57          defaultOutput
58        ];
59      in
60        nixpkgs.lib.foldl nixpkgs.lib.recursiveUpdate {} allOutputs;
61  }
```

Listing 6.2: An excerpt of the flake.nix file defining the derivations to build the Signal Android application.

## 6.7 Fetching the Source Code

In this first step, the Nix function `fetchgit` is used to obtain the source code to build the Android applications. This step needs to be done within a derivation in order to retain the structure that is expected by `nix-update` [78]. Listing 6.3 and Listing 6.4 show the Nix expressions that retrieve the source code for Signal Android and Wire Android, respectively. The parameters for fetching the source code consist of the URL pointing to the repository, the revision (in this case the version label) identifying the exact state of the source code, and the hash value that is expected when fetching this exact version of the source code.

For the source code of Wire Android, the parameter `fetchSubmodules` has to be provided in order for the function to also include the submodule *kalium* when obtaining the source code.

```nix
built-apk = pkgs.stdenvNoCC.mkDerivation rec {
    pname = "build-signal";
    version = "7.56.10";

    src = pkgs.fetchgit {
      url = "https://github.com/signalapp/Signal-Android.git";
      rev = "refs/tags/v${version}";
      hash = "sha256-GUEVBDsHZmt0bR2ptiBEB9k5K0ptMD/pQni9FR9752I=";
    };

    # ...

};
```

Listing 6.3: The Nix-code snippet fetching the source code for Signal Android.

```nix
built-apk = pkgs.stdenvNoCC.mkDerivation rec {
    pname = "build-wire";
    version = "4.15.0";

    src = pkgs.fetchgit {
      url = "https://github.com/wireapp/wire-android.git";
      rev = "refs/tags/v${version}";
      hash = "sha256-WIAlNonzjd2vLl5lzaSvaKsqs3NiBHoOetVWEOpOs3k=";
      fetchSubmodules = true;
    };

    # ...

};
```

Listing 6.4: The Nix-code snippet fetching the source code for Wire Android.

### 6.7.1 Mitigations

Retrieving the source code within the Nix Flake, using the function `fetchgit` ensures the integrity of the source code, by comparing the hash value of the retrieved source code against the expected hash value. This does not mitigate any threats concerning vulnerabilities or bug doors that have been added to the VCS, but ensures that no manipulations of the source code happened when retrieving it.

## 6.8 Fetching the dependencies

In this step the dependencies and plugins needed for the build process are fetched.

The needed artifacts are listed in the `verification-metadata.xml` file. If this file is included in the source code of the repository, it can be accessed from there. If the `verification-metadata.xml` file is not provided by the vendor or if it is incomplete, it needs to be manually generated in a separate step and included in this step, as can be seen in line 4 of the Listing 6.5.

Additionally to the list of needed artifacts, the tool `gradle-dot-nix` also needs the list of Maven repositories in which it should look for the artifacts (see Listing 6.5. This list generally includes a few common repositories, but in the case of Signal Android, for example, it also included some specific repositories, that are maintained by Signal.

```
# ...

gradle-dot-nix-params = {
      gradle-verification-metadata-file = ./path/to/verification-metadata.xml;
      public-maven-repos = ''
        [
          "https://jitpack.io/",
          "https://dl.google.com/dl/android/maven2",
          "https://repo.maven.apache.org/maven2",
          "https://plugins.gradle.org/m2",
          "https://maven.google.com/",
          "https://repo1.maven.org/maven2/",
          "... other application specific repositories"
        ]
      '';
  };
  gdn-results = import ./gradle-dot-nix.nix {
    inherit (inputs) gradle-dot-nix;
    inherit pkgs gradle-dot-nix-params;
  };
  maven-repo = gdn-results.maven-repo;
  gradleInitScript = gdn-results.gradleInitScript;

  # ...
```

Listing 6.5: An examplary Nix-code snippet showcasing the usage of gradle-dot-nix to fetch and cache the dependencies.

At the end of the fetching process, `gradle-dot-nix` has created an entry in the local *Nix store* for each fetched artifact and thereby provides a local Maven repository hosting the needed dependencies.

`gradle-dot-nix` also provides a script, that can be used to initialize Gradle, such that it will obtain the dependencies from the *Nix store* only. An example of how this initialization will happen in general can be seen in Listing 6.6

```
# ...

buildPhase = ''
  gradle build -I ${gradleInitScript}
'';

#...
```

Listing 6.6: An example Nix-code snippet, showcasing how Gradle is configured to use the local Maven-repository.

### 6.8.1 Mitigations

By using the `verification-metadata.xml` file or generating it and providing it to the public, anyone can build the application using the very same dependencies and plugins. Additionally, `gradle-dot-nix` uses the hash values provided in the dependency verification file in order to guarantee the integrity of the downloaded artifacts. These propositions guarantee that the application builds with the same dependencies, in the same version, and with the same content and thereby mitigate attacks that involve the manipulation of dependencies while they are being retrieved.

## 6.9 Build within the hermetically isolated build environment

In the following section, the process providing the verifiable builds is described. The builds of the E2EE messaging apps Signal Android and Wire Android are executed in the hermetically isolated environment of the Nix derivation build phase. The build environment is configured to only contain the minimal set of tools needed to build the applications. The environment additionally has access to the local Maven repository in the Nix store containing the verified artifacts that have been fetched in the steps described in Section 6.8.

### 6.9.1 Build process aspects common to both apps

The build process of the Android applications is executed within the build phase of the Nix derivation and therefore is unambiguously described by the functional programming language. For the compilation of each application, a separate derivation was implemented in the Nix script, to ensure, that the two build processes cannot interfere with one another.

The first part of this derivation was already shown in Listing 6.3 and Listing 6.4 and defined the source of the build process. The next parameters of the function `mkDerivation` are the optional declarations of the `nativeBuildInputs`, the `postPatch` phase, and the two phases `buildPhase` and `installPhase`.

The declaration of the build inputs is needed to have the necessary tools available in the build environment. The Gradle build instructions are provided in the `buildPhase` parameter. The install phase is then used to define copying the resulting artifact into the output directory of the derivation.

### 6.9.2 Inclusion of the build tools

In order to be able to build the Android applications, it is necessary to provide the Android SDK, Gradle, the Java Development Kit, and Git to the build process. All the tools, except for the Android SDK, are available as Nix packages. Android SDK needs to be declared as a separate input to the Nix script, and once it is added to the inputs, it can be used to define the exact versions of the tools needed, depending on the application. An example of the Android SDK definition can be seen in Listing 6.7.

```
android-sdk = inputs.android.sdk.${system} (sdkPkgs: with sdkPkgs; [
    build-tools-35-0-0
    cmdline-tools-latest
    platform-tools
    platforms-android-35
    ndk-28-0-13004108
    cmake-3-22-1
  ]);
```

Listing 6.7: Android SDK declaration for Signal Android.

When building Android applications using Gradle, the version of the build tool is included in the resulting APK. In order to make the build bitwise identical to a reference binary, the same version of Gradle needs to be used. This can be achieved for most versions of Gradle by using the respective nix-packages version that includes the needed version. For versions that are not packaged for nix-packages, one has to package the versions themselves in order to include and use them in the build process.

### 6.9.3 Build process specifics: Signal Android

Listing 6.8 shows the helper function `mkDerivation` being used to encapsulate the build process in the hermetically isolated environment provided by Nix. The build of Signal Android is executed with the initializing script that `gradle-dot-nix` provides to ensure that the dependencies are obtained from the local Maven repository, that was created in the dependency fetching step described in Section 6.8. The parameter in line 25 in 6.8 is necessary to build the application successfully, because the `verification-metadata.xml` was not trusted by Gradle. The following two parameters, set the paths of Java and AAPT, because the build process could not locate them without it.

```
1  built-apk = pkgs.stdenvNoCC.mkDerivation rec {
2      pname = "build-signal";
3      version = # ...
4      src = # ...
5
6      nativeBuildInputs = # ...
7
8          commit_hash = builtins.readFile ../signal/commit_hash.txt;
9
10     postPatch = ''
11       export GIT_COMMIT_HASH="${builtins.substring 0 12 commit_hash}"
12       export GIT_COMMIT_TIMESTAMP="${builtins.readFile ../signal/commit_timestamp.txt
       }"
13       export GIT_TAG="${builtins.readFile ../signal/tag.txt}"
14
15 #       Replace assertions in the Gradle script
16       substituteInPlace "app/build.gradle.kts" \
17         --replace-fail 'fun getGitHash()' 'fun getGitHash(): String { return System.
       getenv("GIT_COMMIT_HASH") ?: "unknown"} fun formerGitHash()' \
18         --replace-fail 'fun getLastCommitTimestamp()' 'fun getLastCommitTimestamp():
       String { return System.getenv("GIT_COMMIT_TIMESTAMP") ?: "0" } fun
       formerLastCommitTimestamp()' \
19         --replace-fail 'fun getCurrentGitTag()' 'fun getCurrentGitTag(): String {
       return System.getenv("GIT_TAG") } fun formerCurrentGitTag()'
20     '';
21
22     buildPhase = ''
23       gradle assembleWebsiteProdRelease \
24       -I ${gradleInitScript} \
25       -Dorg.gradle.dependency.verification=lenient \
26       -Dorg.gradle.java.home=${pkgs.jdk17}/lib/openjdk \
27       -Dorg.gradle.project.android.aapt2FromMavenOverride=$ANDROID_HOME/build-tools
       /35.0.0/aapt2
28     '';
29
30     installPhase = ''
31       mkdir -p $out
32       cp ./path/to/apk $out/
33     '';
34   };
```

Listing 6.8: The Nix script running Gradle to build the Signal Android application.

#### .git-folder dependency of Signal's build process

The Gradle build for the Signal Android application requires the `.git` folder to be present, because it includes the information, that can be retrieved from that folder, in the build

process. If the `.git` folder is included in a fix-output derivation, the resulting hash code will, however, change with every new commit on the repository. Therefore, it was necessary to patch the build-script `app/build.gradle.kts` to take the necessary information from previously populated environment variables instead of from the `.git` folder (see lines 11 to 19 in Listing 6.8).

### 6.9.4 Build process specifics: Wire Android

```
1   built-apk = pkgs.stdenvNoCC.mkDerivation rec {
2   pname = "build-wire";
3   version = # ...
4   src = # ...
5
6   nativeBuildInputs = # ...
7
8   postPatch = ''
9     cp ${version-txt-file}/version.txt app/version.txt
10    cat ${version-txt-file}/version.txt | grep "Revision" | \
11      awk '{split($0, array); printf "%s", array[2]}' >> app/src/main/assets/
    version.txt
12      # the revision number needs to be added here (can be found in version.txt)
13  '';
14
15  buildPhase = ''
16    CI=true gradle app:assembleProdCompatrelease \
17      -I ${gradleInitScript} \
18      -x includeGitBuildIdentifier \
19      -x generateVersionFile \
20      -Dkotlin.native.distribution.baseDownloadUrl=file:${maven-repo} \
21      -Pkotlin.native.distribution.downloadFromMaven=true  \
22      -Dorg.gradle.java.home=${pkgs.jdk17}/lib/openjdk \
23      -Dorg.gradle.project.android.aapt2FromMavenOverride=$ANDROID_HOME/build-tools
    /34.0.0/aapt2
24  '';
25
26  installPhase = ''
27    mkdir -p $out
28    cp ./path/to/apk $out/
29  '';
30 };
```

Listing 6.9: The Nix script running Gradle to build the Wire Android application.

#### version.txt dependency of Wire

The build scripts for the Wire Android application generate the current sub-version depending on the build time and save the sub-version and other build metadata in the `version.txt` file. To ensure that the built APK would have the same sub-version as the reference, APK the approach was, to fetch the `version.txt` file that was published on the Wire repository and patch the build script such that it would look for the file and read the sub-version number from it.

#### Dependency fetching at build-time

In order to compile native binaries, Kotlin needs the Kotlin-Native libraries. These libraries are by default fetched at build time, if they are not found in a local cache folder.

There exist parameters that can be set to make Kotlin look in a Maven repository for the dependencies and to specify the location of this Maven repository. It was necessary to set these parameters to point to the local Maven repository that is created by `gradle-dot-nix` and add the Kotlin-Native libraries to the dependency list in the `metadata-verification.xml` file to provide Kotlin with the needed libraries during the hermetic build process.

#### Diverging parameter handling

In the build script of Wire Android, the plugin `CompleteKotlin` [5] is included, which is intended to provide Kotlin-Native libraries independent of the architecture of the current host by fetching specific libraries at build-time. This plugin, however, does not fully support the parameters that are supported by Kotlin-Native, and in order to prevent `CompleteKotlin` from trying to fetch the libraries during the hermetic build process and from breaking the build, if it fails to retrieve the libraries, the environment variable `CI=true` has to be set.

### 6.9.5 Mitigations

Running the build processes for the Android applications in hermetically isolated environments increases the confidence that the build process only included those resources that have been specifically declared. The build phase of the Nix derivations is isolated from the internet and can therefore ensure that the Gradle build does not resolve and fetch dependencies during build time. The only dependencies, the build process can access, are the ones described in the `verification-metadata.xml` file.

### 6.9.6 Extra arguments passed to the build command

In order to ensure that the Gradle build tool could access all the necessary tools, it did not suffice for all the tools, to be provided in the `nativeBuildInputs` argument. For some tools, it was necessary to provide their location to Gradle, by specifying them as arguments of the build command. This included:

- Setting the location of `java.home` (see, for example, line 26 in Listing 6.8)
- Override the path for `aapt2` (see, for example, line 27 in Listing 6.8)

After the build of the Android application finishes, the resulting APK is copied to the output folder.

This step completes the reproducible build of the messengers. No matter on which system, at which time, or how many times one runs this build process, the resulting APK will always be bitwise identical to the APKs of previous runs.

## 6.10 Signing of the APK

In this step of the implementation of the R-Bs of Signal Android and Wire Android, the self-built application is provided with a signature, by copying the signature from the reference binary and patching it into the unsigned APK using the tool `apksigcopier`.

It should be noted that, as mentioned in Section 2.5, this is not the way an application is supposed to be signed, but it achieves the identical result. Given that the self-built unsigned APK is bit-wise identical apart from the signature, adding the signature allows to obtain a signed application that is bit-wise identical to the reference binary obtained

from the vendor. This means that the build process is fully reproducible and the self-built application has a valid signature, allowing anyone to install it on their device.

If the self-built unsigend APK however differs from the reference binary by more than just the missing signature, adding the signature to it will not result in a valid APK and therefore cannot be installed. Adding the signature in this case still has the advantage of reducing the list of differences that will be found by `Diffoscope` in the next step (see Section 6.11 and will make the root-cause analysis easier.

It should be noted that the signature is still valid, and the resulting signed application might not be bitwise identical if the metadata of the APK differs from the reference APK. This is a problem that needs to be addressed by future work (see Section 10.

## 6.11 Comparison to the reference APK

The easiest way to show that two files (here the APKs from two different build runs) are bitwise identical, is to compute the hash of both and compare them. If the two files differ even only in a single bit, the hash will look very different, and only if both files are fully identical, the hash of both files will be identical as well. This is true as long as a secure hash algorithm is used, for which it is not possible to find collisions, as elaborated in Section 2.4.

### 6.11.1 Using `Diffoscope` to compare the APKs

In the best case, comparing the self-built APK with the reference binary by computing the hash will confirm that the two applications are indeed bitwise identical, and we can confidently publish the artifacts. In the case that the binaries differ, however, it is relevant to find out where those differences stem from and if those differences could indicate an unwanted change in the application's logic or just some minor source of non-reproducibility. Some potential differences, that might occur and that do not indicate a potential unauthorized modification, but rather hint at deviating configurations of the build process include

- **Missing signatures:** If the previous step was skipped and the signature not copied onto the self-built APK, then this self-built application does not have a signature, but the distributed official applications do. Analyzing the differences between the two binaries will then show that the self-built APK is missing the signature, which results in two big sections in the `Diffoscope` results. The consequence of this difference is merely the fact that the self-built APK should not be published, as it cannot be installed.
- **Build tool versions:** Some build steps might include the version code of the build tools used in the build output, and if the build tools differ or if a different version of the same build tools is used, then this difference will show up in the results of the APK analysis.
- **Misconfiguration of build process:** given that the wrong build task or the wrong build parameters are provided, the build result will look different and for example, have different features enabled. An example would be that instead of the build command for the generic Android application, the command to build the application optimized for a certain device type is provided—the functionality of the resulting APK would be mostly the same, but the binaries would differ. These differences are therefore not a problem by themselves, but are not easily distinguishable from a build that actually includes malicious code and should therefore be carefully avoided.
- **License statements:** The application of Wire Android includes the licensing statement within the APK itself. These licensing statements are missing, in the application resulting from the isolated build process.

- **Missing VCS information:** Since Android Gradle Plugin version 8.3, the file `META-INF/version-control-info.textproto` is generated during the build process and included in the APK. Within the hermetically isolated environment of the Nix derivation build phase, the tool fails to retrieve the necessary information, and therefore the resulting VCS information file differs.
- **File size, but same content:** Another difference found in the outputs of `Diffoscope`, was a file size difference of a file in a ZIP archive, without a corresponding difference in the analysis of the individual file. This is an indicator, for a file that is identical in its uncompressed state but, due to differences in ZIP compression algorithms or levels used when creating the ZIP, shows in the ZIP archive metadata with a differing file size. [30]

All the differences mentioned above are a result of a lack of reproducibility of the build process and indicate a need for further steps to be taken to get the build closer to reproducing a bitwise identical result to the reference APK or for the upstream developers to adapt the build process to facilitate its reproduction.

### 6.11.2 Implementation

The comparison using `Diffoscope` and the comparison of the APKs using their hash values are done within the Nix build environment. Using `Diffoscope` within the build phase is possible because the tool has been included in the `nixpkgs` and can therefore be included in the build environment of the derivation by providing it as a build input parameter. Calling `sha256sum` to compute the hash values of the APKs is possible due to the fact, that the build phase of Nix derivations includes some core utilities. Given, that comparing the two binaries using `Diffoscope` only needs a minimal build environment, one can use the function `runCommand` instead of calling `mkDerivation` to minimize the necessary overhead to create the Nix derivation, as can be seen in the Listing 6.10.

Line 5 of the Listing 6.10 shows the invocation of the tool `Diffoscope`. `Diffoscope` returns with the value `0` if no differences have been found, returns with `1` if the tool has found differences in the input files, and returns with exit codes that are greater than `1` to indicate errors. Due to the fact, that Nix scripts abort the execution if any of the called commands return a non-zero result, the return value of `Diffoscope` needs to be caught and handled. The `OR` expression and the bash condition following the call of `Diffoscope` ensure that this line returns `0` for as long as the return value of `Diffoscope` was either `0` or `1` and thereby ensure that the Nix script continues its execution and only aborts if an actual error occurred in the execution of `Diffoscope`.

```
1  run-diffoscope = pkgs.runCommand "diffoscope-comparison" {
2          nativeBuildInputs = [ pkgs.diffoscope ];
3        } ''
4          # ...
5          diffoscope --html=$out/diff.html ${self-built-apk} ${reference-apk} || [
   $? -eq 1 ]
6        '';
```

Listing 6.10: An examplary Nix-code snippet showcasing how diffoscope is run to compare the self-built APK to the reference binary.

## 6.12 Publishing of the resulting artifacts

The APK resulting from the R-B, the hash value computed from the APK and an analysis of the differences between the built APK and a reference APK are published as pipeline artifacts on the GitLab instance, that is executing the CI/CD

pipeline. The GitLab instance is hosted by the INS, at the Johannes Kepler University Linz and can be accessed at *https://git.ins.jku.at/proj/android-device-security/source-transparency/android-clients*, which allows users to access the results later and for example, verify their locally installed messaging application by comparing their hash value to the published one (for more information about the verification possibilities, see Section 6.14).

## 6.13 Update the Reproducible Build process

In a first attempt, the goal was to build the E2EE Android messengers for Signal and Wire for one certain version only, but as newer versions of the applications are being published by the vendors, users want to update their applications, and therefore also the R-B of the messenger must be brought up to date. In order to allow users to verify and subsequently install newer versions of the applications, the Reproducible Build process must provide this newer version, which requires an updating process for the Nix scripts and other version-dependent files, which are included in the reproducible build process of the application. The list of inputs and references that need to be updated for every new release includes

- **References to the source code and reference binaries:** The Nix scripts describing the build process contain function calls that retrieve the source code and reference binaries from the repositories, which need to be updated to point to the newer versions.
- **New (versions of) dependencies:** After updating the previously mentioned references and inputs of the build process, the build might fail, because the newer version of the Android application needs newer versions of dependencies or depends on other artifacts, it did not depend on before.
- **Build tools:** The build tools used to compile the application need to be updated as well, in order to still obtain bitwise identical artifacts from the build process.
- **Helper files for the R-B:** In order to run the Gradle build process of the applications in such a way that it would generate APKs, identical to the reference binaries, a few helper files had to be created (see Sections 6.9.3 and 6.9.4). These files contain mostly version dependent information and therefore need to be updated as well.

If the new version of the application has other changes in the build process or the requirements of the build process, the update process might not be sufficient, and the build of the applications might fail. In order to recover from this, a maintainer of the R-B process needs to manually identify the source of the failure.

### 6.13.1 Update references

The Nix expressions building the APKs use the function `fetchgit` to retrieve the source code of the application. This function requires arguments to specify the URL and revision to access the right version of the source code, as well as the expected hash value that the obtained code should evaluate to. Similarly, the reference binaries to compare against the self-built APKs are retrieved using the function `fetchurl`, which requires a version dependent URL to point to the artifact to be obtained and its expected hash value.

These arguments, that are provided to the fetching functions can be updated to reference the values for the newest release by calling the tool `nix-update`. This will update all URLs to point to the newest release, and it will also change the expected hash values to the ones corresponding to the new artifacts.

### 6.13.2 Update dependencies

Newer versions of the applications often depend on newer versions of their dependencies or new dependencies, that they did not need before. In order to build the new version of the application, the reproducible build environment also needs to provide the artifacts for the newer build dependencies. This means that the new dependencies need to be added to the `verification-metadata.xml` file, in order to be fetched as described in Section 6.8. The missing dependencies can be added by running the script `extend_verification_metadata_xml.py`. This bash script will look for the descriptors of the missing dependencies in the error log of the build process and search for them in a list of known repositories. If it cannot find a specific dependency in any of the repositories, the build pipeline fails and the update must be finished manually. If, however, all the dependencies could be found in at least one of the repositories, the script will generate the new entries for the `verification-metadata.xml` file, such that the tool `gradle-dot-nix` can later retrieve the actual artifacts and populate the local Maven repository with them, as explained in Section 6.8.

### 6.13.3 Update build tools

If the new releases are built using newer versions of build tools, or if they require new tools, that were not necessary in previous versions, the Nix scripts have to be updated to include these new tools as well. The current implementation does not include an automatic way of updating the environment to include new (versions of the) build tools.

### 6.13.4 Update process trigger

The update script is executed by the GitLab pipeline, which is running based on the defined schedule. Every time a new version of one of the two Android applications is published, the update script will update the necessary files and invoke a build process to verify the reproducibility of the new version.

## 6.14 Verification process: Concept

The verification process is as follows: a user can download an application from any source on the internet and compare its hash value with the hash values that have been published for this app on the GitLab repository of the INS. If the hash value matches either the APK that is built by the GitLab pipeline or the APK that was retrieved from the official vendor's repository (the value would match either both or neither, if the self-built APK is bitwise identical to the reference binary), then the user can look at the reproducible build process to analyze the source code, tools, build parameters, and other artifacts, that are involved in building the application to determine whether that build process will produce a trustworthy APK. If the hash value of the application that the user retrieved, matches only the official binary, but not the APK built by the pipeline, then they have to additionally look at the report provided by Diffoscope to determine if the differences are insignificant enough to be ignored.

This first part of the verification process is possible to perform in the current implementation of the contribution. Figure 6.2 depicts the steps that a user would undergo to ensure the trustworthiness of an application. This scenario assumes that the users trusts the INS to be honest about the relation of R-Bs and their results.

The second part of the verification process requires a trusted independent third party that verifies the correspondence of the described reproducible build process with the results published on the GitLab repository of the INS. This third party should also perform

an analysis of the inputs of the build process to ensure that no malicious software artifact or configuration is present. Once the correctness of the R-B and the benevolence of the inputs were verified, this third party could provide a reference to the description of the R-B (i.e. the exact version of the repository that provides the R-B for the requested version of the application) and a reference to the results of the R-B (i.e. the pipeline artifacts hosted on the GitLab repository that resulted from the execution of the referenced R-B).

These references could be published as entries of a verifiable log (see Section 2.16) to strengthen the guarantees on the trustworthiness of the referenced R-B and the referenced results. This approach requires the user to trust the party that provides the entries of the verifiable log or to have some way of differentiating entries of the verifiable log that are not trustworthy (see Section 9.3 for more information).

Figure 6.3 depicts the extended concept of the verification process that involves the R-Bs and their results being published to a verifiable log. It provides a few extra security guarantees and a better user experience (as the user does not have to verify the R-B or the build inputs themselves), but it also requires a more sophisticated infrastructure (as the verifiable log also needs monitors, witnesses, and auditors to be present).

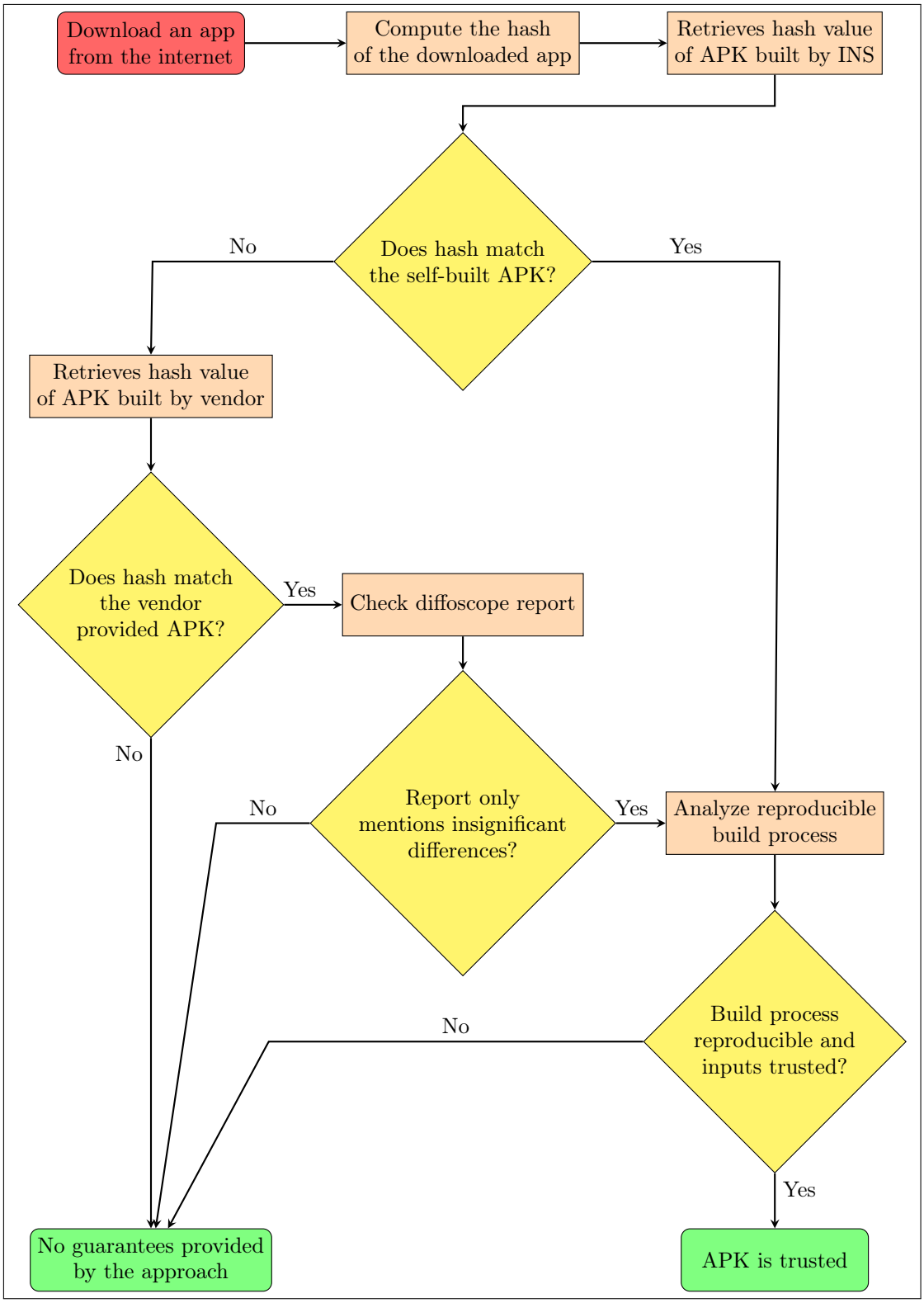


Figure 6.2: Verification process part 1. This part of the process can be performed with the current implementation, but it requires the user to trust the provider of the R-B to be honest, and it requires the user to analyze the build inputs themselves to determine whether the APK is trustworthy.

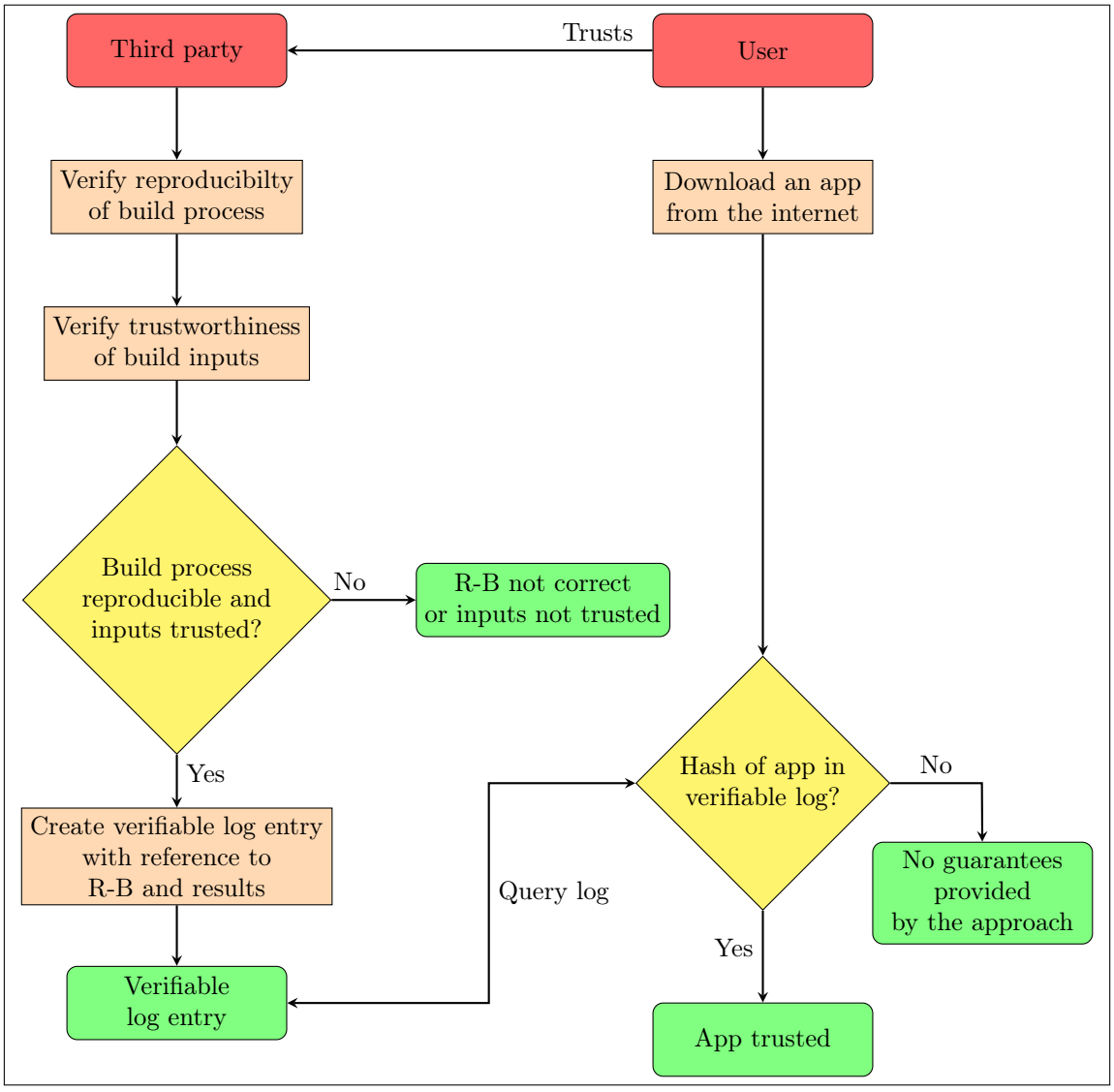


Figure 6.3: Abstract overview of the implementation.

# Chapter 7

# Evaluation

This section gives a critical analysis of the contribution described in Section 6 and the actual mitigations it can provide against the threats described in the threat model in Section 4. After a short summary of the contribution is provided in Section 7.1 and an overview of the limitations of the contribution in Section 7.2, the section reflects which threats are mitigated by the approach and which remain open.

## 7.1 In short

The contribution described in Section 6 provides a method to recognize when the build process of binary applications has been tampered with. It provides reproducible builds for the two Android applications of Signal and Wire and thereby gives a general user of the applications the opportunity to verify the build process of the APK they want to install and use. They can analyze exactly which dependencies and tools have been used to compile the application and can review the exact source code from which the application was built. The integrity of the Supply Chain is ensured at multiple steps throughout the build process by verifying the hash values of inputs and outputs of those steps. The proposed approach, however, does not comply with the requirements of SLSA build level 3 (see Section 2.10 for more information). It provides the required isolation for the build execution, but does not provide build provenance. This problem and the necessary future work are further addressed in Section 10.2.1.

## 7.2 Limitations and Scope

The following list describes aspects that are outside the scope of this thesis:

- **Authenticate source code and reference binaries:** The application vendor must provide source code and reference binaries. The reproducible build process must guarantee to access only valid URLs to ensure the source code and binaries are authentic and original. This has been done manually and based on the information provided on the official website of the vendor.
- **Availability threats:** In general, the proposed approach does not provide mitigations for availability threats, as no measures to ensure continuous availability of the source code or dependencies are included. Furthermore, the build provenance could become unavailable and thereby render the build unreproducible. Such problems could be tackled by distributors of applications if they archive all the needed resources for the R-B or through legal obligations.
- **Trust in source code**: The contribution does not provide any mechanism to detect or prevent the existence of vulnerabilities or backdoors in the vendor's source code. The provided solutions only ensure the integrity of the source code as it was at the time of creation of the R-B. The source code must be secured by other means, including the use of software development tools and distributed audits of members of the open source community.

- **Trust in dependencies and tools:** A user needs to trust or verify the dependencies and tools used of the build process. The contribution does not provide methods to mitigate any dependency threats, aside from guaranteeing the integrity of the declared dependencies and tools. Existent malicious logic, bugs, and other vulnerabilities in those binaries need to be addressed by other means.
- **Trust in the build environment:** Additional trust needs to be put into the binary tools that are used to provide the environment of the R-B. Executables, like the software binaries of Nix, which execute the Nix expressions, need to be trusted or verified. These binaries could have been tampered with and might thereby undermine all guarantees and mitigations, that the approach otherwise provides.

## 7.3 Source threats

The proposed approach pinpoints an exact version of the source code by referencing a specific contribution of the VCS by the label that was applied. The contribution also provides methods to ensure the integrity of this version of the source code. It guarantees that the same source code was used to build the reference binary as well as any other binary that was created using the R-B, by matching the hash value computed over the obtained source code to the hash value of the source code as it was obtained at the creation of the R-B.

An additional benefit that is achieved through the use of VCS is accountability. As contributions are tracked by the VCS, any changes in the source code can be unambiguously assigned to the single contributors of it. This does not prevent the introduction of malicious logic, but provides transparency and the ability to pinpoint introductions of malicious logic to the compromised developer.

## 7.4 Build threats

The contribution protects the integrity of the build scripts of the Android applications, as they are part of the source code and are thereby protected from source threats in the same way as the rest of the Android application source code. The integrity of the build parameters is subject to source threats targeting the description of the R-B. They are protected by the mechanisms of the R-B - any changes to the build parameters, that would lead to changes in the outputs of the build process, will be detected at the stage that compares the build process results with the reference binary and at the verification step, when the hash value of the compromised application does not match with the expected value.

## 7.5 Dependency threats

The proposed approach provides transparency about which dependencies, libraries, plugins, and tools are used during the build process and verification of the integrity of those dependencies and tools. The approach ensures that the very same binaries are used in the build process independent of the executing user, underlying host system, or execution time. The approach mitigates the threat of dependency confusion, because the fetched artifacts are matched against a cryptographic hash value and therefore have to be the exact binary that is specified in the `verification-metadata.xml` file. The R-B of the Android application moves the border of trust by one step, as the build process of the application itself is verifiable, but the reproducibility of the dependencies and tools is not provided by the approach.

## 7.6 Availability threats

The contribution provides the R-Bs using the Nix package manager, which in turn supports caching of build inputs and build artifacts using the Nix store. This feature is leveraged to provide a caching mechanism for dependencies and tools after the first build execution and thereby mitigates the problem of temporarily unavailable dependencies or tools.

## 7.7 Distribution threats

The implementation of the approach allows users to verify that the distributor they are obtaining the APK from, provides an application that was not tampered with. They can verify the absence of unauthorized modifications by comparing the hash value of the obtained APK with the expected hash value provided by provenance data.

## 7.8 Verification threats

Threats like a leaked signing key of the application supplier are mitigated by the contribution, as the user can verify any obtained APK by comparing its hash value to the reference value. The exploitation of cryptographic primitives used in the approach is mitigated by using strong algorithms for the hashing and signing steps included in the Software Supply Chain.

# Chapter 8

# Findings

This section highlights findings of this work. It will first list the versions that were successfully built with the approach of this thesis and will then continue highlighting the encountered difficulties and surprising elements found while implementing the approach.

## 8.1 Versions that were successfully built

Table 8.1 lists the versions of Signal Android and Wire Android that were successfully built at the submission time of the thesis. It should be noted that none of the builds was fully identical to the reference binaries, and therefore, to this point, no version could be built fully reproducibly. The main reasons why the builds are not reproducible are mentioned as future work in Section 10.1.2.

## 8.2 Missing dependencies in provided verification-metadata.xml

The `verification-metadata.xml` is a file that can be generated by Gradle to ensure the integrity of all dependencies of the build process. Therefore, it should contain all dependencies and plugins of a Gradle build task it was generated for. The file can then be used to verify the integrity (by the provided hash value) and optionally also the authenticity (using signatures) of those dependencies. Given that the vendor has generated the file and included it in the source code, it allows for improving the reproducibility of the build process and additionally allows the user to trust the vendor to have verified the benevolence of the included dependencies. In the repository of Wire Android, the file was not included and therefore potentially also not used by the vendor. In the repository of Signal Android, a verification file was included.

One finding of this thesis was, however, that the `verification-metadata.xml`, which was included in the repository of Signal Android, did not include all the necessary dependencies to build the application. While it may be the case that the file was not updated recently, further investigations have strongly suggested that a newly generated `verification-metadata.xml` file also does not include all dependencies.

The finding occurred, because the build process failed to find dependencies in the hermetic build environment, even though the tool `gradle-dot-nix` fetched

| | |
|---|---|
| Signal Android | v7.47.2, v7.53.2, v7.53.4, v7.53.5, v7.56.10 |
| Wire Android | v4.14.1, v4.14.2, v4.15.0, v4.15.1, v4.15.2 |

Table 8.1: Versions of the E2EE messenger applications that were successfully built using the approach of this thesis. To this point no version was built fully reproducible.

all the dependencies that are listed in the included `verification-metadata.xml` file and correctly populated the local Maven repository. Consequently, the `verification-metadata.xml` file was generated by running the Gradle build task with the parameter `–write-verification-metadata sha256`, a command that is supposed to list all needed dependencies of the build task at hand.

This added quite a few dependencies to the file of Signal Android, and also, when running for Wire Android, generated a rather long list of dependencies needed to build the application (see Section 8.3), but running the build after this unexpectedly failed again, because there were still dependencies missing.

The solution was to write a small Python script to programmatically filter all the missing dependencies from the log of the failed build and generate the missing dependencies with this script, to add them to the `verification-metadata.xml`.

So the finding was, for both applications, that some dependencies were not included in the proper way for Gradle to find them to generate the verification file.

## 8.3 (Very) long list of dependencies

As mentioned before already, the list of dependencies contained in the `verification-metadata.xml` was very long, with more than 12,000 elements for the Wire Android application and around 5,500 elements for the Signal Android application to be downloaded by `gradle-dot-nix` upon first running the build process. This resulted in an initial run-time of the R-B for Wire Android of more than 3 hours.

## 8.4 Unavailable dependencies

Some versions of Signal included an artifact in the build process, which could not be found in any of the public Maven repositories. The source code of the dependency was available on GitHub, and older versions of the dependency were available in the Maven repositories, but this version was not uploaded yet. This meant that the newer version of Signal could be build by building the dependency from the source code and providing it to the build process. However, because the artifact could not be built reproducibly and presumably differed from the artifact built by Signal, the application resulting from the otherwise reproducible build of Signal Android differed from the reference binary. The dependency was eventually found to be listed in a Maven repository hosted on a GitHub repository of Signal.

## 8.5 Difficulty integrating the Gradle build process in Nix expressions

Nix expressions are designed to provide build environments for R-Bs. It is well documented and has many resources to learn the usage and principles from [75, 77]. Nevertheless, it proved to be quite challenging to start off with the implementation of the Nix expressions for the Gradle build processes, as, for example, it was difficult to attribute whether errors came from mistakes in the Nix expressions or from the Gradle build process, which was unable to perform certain tasks due to the isolated environment it was executed in. Wrapping the Gradle build process was found to be cumbersome in parts, as Gradle handles many of its tasks, for example, the fetching of dependencies, in a way that is incompatible with the principles of Nix expressions.

### 8.5.1 Fetching Gradle dependencies

The dependencies of a Gradle build process are normally fetched from any repository that provides the artifact in question, which means that the artifacts might be obtained from different repositories in different invocations of the build process and therefore might even have different hash-values because of minor differences between instances of the same artifact on different repositories. This makes the integrity verification very difficult.

This process could be simplified thanks to the tool `gradle-dot-nix` [43], which allows the integration of the Gradle dependency fetching in Nix environments by programmatically fetching all the dependencies listed in the `verification-metadata.xml` file and populating a local Maven repository with the artifacts before the isolated build phase.

### 8.5.2 Sandboxing issues: missing tools and other processes failing

Other difficulties that were met during the work of making the build processes reproducible in the hermetic environments included tools not being found by the build processes and differing behavior by some Gradle tasks due to the isolation from the internet:

- **Missing JDK:** The first tool that was needed and not found despite its declaration in the build inputs of the *derivation* function, was Java Development Kit (JDK). It needed to be provided to the Gradle build process by providing the corresponding argument with the path to the location of the JDK.
- **AAPT2 not found:** Despite providing Android SDK to the build environment, Gradle was unable to locate the binaries for the AAPT and needed an argument to be passed to the build invocation, which provided the path to the tool.
- **VCS Information generation:** Despite including the `.git` folder when fetching the source code for the applications and providing *Git* to the build environment, Gradle was unable to generate the same VCS information file, as it did outside the hermetic environment.
- **Changed behavior of *AboutLibraries*:** The tool *AboutLibraries* [62] behaved differently inside the hermetic build environment from when Gradle was executed outside the isolated environment. In contrast to the reference binary, the binary from the R-B process did only include URLs pointing to the licenses of the application rather than including the full text of the license. This hints at the possibility that the tool accesses the internet during the build process to fetch the license content, which it cannot do from within the hermetic build environment.

## 8.6 Limited performance of the build process

Nix expressions allow to improve the performance of their execution by retrieving outputs of *derivations* that have been build before instead of building them again. These mechanisms to cache intermediary results, however, are not integrated in the Gradle build tool yet, which means that Gradle cannot access caches of previously performed build tasks. If any of the attributes of the *derivation* containing the Gradle build process have been changed, the *derivation* will be build again, which means that every single build step of the build process will be executed anew. The result of this missing support of the caching mechanism is a multiple minute long waiting time for every change in the script invoking the Gradle build, which made the development and debugging of the Nix expressions for the Android application build processes very time consuming and difficult.

## 8.7 Debug information in release builds

Executing Signal Android's build task `assembleWebsiteProdRelease` generates a release build of the application. Invoking the task in the hermetic build environment, however, generated APKs that still contained debug information. The reason for this was, that a tool could not be found by the build process, and so it silently failed to strip the debug information from the binary. This may in general become an issue in situations where the build is executed automatically and upon failing to find the tool, publishes the application that still contains debug information.

# Chapter 9

# Discussion

In this section we will discuss the list of threats that were not included in the threat model of this thesis (Section 9.1) and the features that set the proposed approach apart from other approaches (Section 9.2). Following this, the section will introduce the necessity to apply the approach recursively to all dependencies of the build process of the applications to properly ensure the absence of malicious logic in Section 9.4 and in Section 9.5 it will highlight the events and current trends that heighten the relevance to ensure Supply Chain Integrity of E2EE messenger applications in the current times.

## 9.1 List of threats, excluded from the threat model

Other sources , like the SLSA standard, include threats in their model for Software Supply Chains which are not considered in this thesis [16].This is due to the deliberate limitation of the scope and in the case of some threats, because they do not apply to the Supply Chain instance considered by this thesis. Other models of Supply Chains might include more threats, and the following non exhaustive list is meant to be a starting point to explore such threats in other works.

- Submission of changes to the VCS without review
- Evading the commit process and its review processes
- Compromised layer underlying the Supply Chain (e.g., the host operating system or the CI/CD program)
- Improper usage of artifacts

## 9.2 Differences to other approaches

In contrast to other related work (Section 3) the contribution of this thesis does not only try to ensure the reproducibility of a single Android application, but rather provide a case study and reference for an approach to wrap build processes for APKs in such a way that they become reproducible. In addition to the aspect of reproducibility, this work also builds the application in a hermetic environment to ensure that the list of dependencies for the application is complete and no other software artifacts are included during the build process. This allows a user to audit the inputs of the application's build process and verify the absence of unwanted or malicious logic by reviewing the reproducible build process of the application.

## 9.3 Use of verifiable logs in the verification process

A verifiable log could be provided by the entity that hosts a repository containing the R-Bs to allow a user to look up if the application of interest can be built reproducibly and whether the build process does not contain any malicious logic.

A user may fully trust this entity, which then requires no further third party that verifies that the entries in the verifiable log are honest and correct. It should be noted that the other parties involved in the ecosystem around the verifiable log (i.e., verifiers, monitors, and auditors) are still needed.

The verifiable log would then provide a simplification for the user to verify the benevolence of a certain application, as they only have to look up the existence of the log entry for the APK. If the provider of the verifiable log, however, does not analyze the reproducibility of the build process and the benevolence of all dependencies and build steps involved (i.e., the benevolence of the resulting application), then the user must not trust the verifiable log, without another party that independently verifies the entries.

## 9.4 Beyond Supply Chain Integrity of a application

Once the integrity of the Supply Chain of an application has been ensured by applying methods like the ones proposed in this thesis, the line of trust was pushed beyond the build process and distribution of the application. The next step to increase trust in the application is to also ensure the Supply Chain Integrity of the tools and dependencies involved in the Supply Chain of the application. This would require the same protection mechanisms to be applied to those tools and dependencies: protecting and reviewing the source code, ensuring the integrity of the used build tools and dependencies, securely distributing information to allow for verifications of authenticity and integrity of the binary artifacts.

### 9.4.1 Recursive verification of dependencies

In order to ensure the absence of malicious logic in the resulting artifact of a Software Supply Chain it is required to also ensure the absence of malicious logic in any of the dependencies and tools included in the Software Supply Chain. Extending the approach of this thesis, this would mean to recursively provide R-Bs for every single tool and dependency needed in the build process of the application and the build processes of the dependencies themselves. This approach, then, is very similar to the idea of Bootstrappable Builds (see Section 2.13 for more information), with the benefit, that not all the software artifacts have to be rebuilt from scratch, but due to the reproducibility of the build processes, one can trust (or verify) that the outputs cached, e.g., in the *Nix store,* are the ones that would result from the build processes.

### 9.4.2 Source threats for OSS

In the case of open source applications, source threats are still relevant, even aside from threats like hostile takeovers of the repositories. Even though OSS allows for auditability of the source code, this is no guarantee for the security of the application's source code. Bugs and errors are an inherent part of software development and can always be introduced in code accidentally. Their frequency and impact can be minimized by rigorous testing frameworks and software verification tools. A malicious developer or contributor on the internet might, however, purposefully commit source code that includes backdoors hidden inside bugs—so called bugdoors. These kinds of backdoors are more difficult to detect and prevent, and commits containing such code might be accepted if the reviewers did not recognize them. The auditability of OSS is merely a probabilistic mitigation providing increased confidence in the source code with increased review activity of the open-source community. The persistence of source threats in OSS is mostly due to underfunding of developers of open-source software and a lack of contributors or auditors. Ultimately a user needs to trust the source code either because they checked it themselves, because others have audited it, or because they trust that the legal or reputational

incentives for the organization are strong enough to ensure the security and correctness of the source code.

## 9.5 Relevance of this work in the current times

The approach to reproducibly build E2EE messenger applications for Android as proposed in this thesis and other works (see related work in Section 3) allows users of such applications to verify the integrity of the applications and review the source code and dependencies to ensure the absence of malicious or otherwise unwanted logic. Users can increase their confidence, that no malicious actor has compromised the Software Supply Chain of the application and that the vendor of the application cannot secretly modify the source code of the application to include unwanted logic.

# Chapter 10

# Future work

## 10.1 Improve implementation

The following sections will highlight a list of projects to improve the implementation of the proposed approach. The most pressing improvement is the elimination of the differences between the self-built APK and the official APK, which could not be closed by the current implementation and which therefore still need to be resolved (Section 10.1.2). Section 10.1.3 highlights improvements to the updating process of the implemented approach. The approach could be extended with a verification program (Section 10.1.4), improved with the recursive verification using R-Bs for all tools and dependencies (Section 10.1.5) and optimized by conceptualizing a secure cache for the results of the R-Bs.

### 10.1.1 Populating the build environment

As mentioned in Sections 6.9.3 and 8.5.2, the build process for Signal Android could not find the binaries for JDK17 and AAPT. The build process for Wire Android could not find the binaries either, which required the paths to be passed to Gradle as build arguments. This did ensure that the build ran without problems, but it should be noted that overriding the AAPT binary location is an experimental feature. A future version of the implementation could improve this by ensuring that Gradle can find the necessary binaries without needing to specify build parameters.

### 10.1.2 Close remaining sources of non-reproducibility

The implementation of the hermetically isolated build process (see Section 6.4) was adapted in multiple iterations to produce artifacts that are as identical to the reference binaries as possible. However, due to the limited time frame of this work, not all the differences between the self-built APKs and the reference APKs could be resolved. These include the following:

- **Version Control Info:** The binary published by Wire and Signal includes the file `version-control-info.textproto` in the `META-INF` folder, which contains information about the current state of the VCS at the time of compilation. While it would be possible to gather the needed information by ensuring that the hidden `.git` folder is present in the fixed-output derivation, this would mean that the hash resulting from the fetched source code would change with every commit and thereby make it infeasible to use in a fixed-output derivation. A possible solution to this could be to retrieve the necessary information for generating the `version-control-info.textproto` through different means.
- **Metadata of the signature:** The tool `apksigcopier` allows copying the signature of a validly signed APK to an unsigned binary, and given that the APKs are identical apart from the fact that one of them is signed while the other is unsigned, copying the signature from the signed to the unsigned binary should result in bitwise identical APKs. This, however, is not true in practice, as the used tool currently still excludes some of the signature's metadata, which means that the APKs differ in those parts.

### 10.1.3 Improve update process

Future work could improve on the updating process, for example, by ensuring that the right version of the build tools is automatically extracted from the source code of the project and set in the *Nix expressions*, such that the hermetic build process always uses the same versions of the build tools, as the build process of the reference binary does. Currently the versions of build tools (e.g., NDK, Gradle, etc.) are recovered from the source code and build instructions published by the vendor. An automatic process would allow scaling the update process.

### 10.1.4 Implement distributed verification method

A future extension of this work's implementation should include means to allow users to programmatically ensure the integrity of an APK, such that users can download an APK and run a script, which verifies the integrity of the APK by querying the hash value from a list of independent parties, which verified the R-B of the application. This allows the user to define a list of such parties that they trust and check that the hash value of the downloaded APK matches the value provided by the trusted parties.

### 10.1.5 Recursive Integrity Verification of dependencies

As introduced in Section 9.4, in order to fully ensure the absence of malicious logic in the application and to verify the link between the source code of the dependencies and tools used in the build process of the application, the approach would need to be recursively extended to the build process of all the dependencies as well. A future project could create R-B for each dependency and tool in the dependency tree of the application. The resulting artifacts could then be used to verify the dependencies and tools and if all the verifications of the R-Bs for the dependencies, tools and the application itself succeed, this will have established a link from the source code of every artifact involved in the application's Software Supply Chain to the final binary format of the application.

### 10.1.6 Additional trust placed in build environments

Using tools providing build environments, requires trusting those tools and any other binary artifact that is involved in the creation of the reproducible build environments. Nix, for example, will by default include some toolchains and core utilities in every derivation and thereby introduce the need to trust those tools in addition to the binaries of Nix itself. Future work could provide a mechanism to verify these binaries instead of requiring the user to place trust in them.

### 10.1.7 Optimization of the distributed verification

In order to allow users to verify software artifacts without needing to verify all of the R-Bs themselves—which, especially with the recursive dependency verification, would result in a huge number of build processes—independent parties could verify the R-Bs and publish the results to a global cache. Future projects could explore the possibility to publish R-Bs of dependencies used in Android application builds to the NixOS cache[1]. The work would need to analyze the security and trust implications of such an approach.

[1] https://cache.nixos.org/

## 10.2 Extend/improve approach

The following sections highlight further enhancements of the general concept of the proposed approach.

### 10.2.1 Missing build provenance

While executing builds within a Nix build environment ensures that the build is isolated from external influences and that all the necessary information (e.g., source code, environment variables, build inputs, build script, etc.) is well defined, this is not sufficient to comply with the requirements of SLSA build level 3. In order to fulfill the requirements, it is also necessary to provide an authenticated provenance (see Section 2.14), signed by the builder of the artifact. This provenance must be distributed along with the artifact, in a format (e.g., SLSA provenance schema [14]) such that there exists a verifier for this (e.g., slsa-verifier [15]).

Future work should focus on providing the provenance for artifacts that have been built using the approach of this thesis. It should establish a concept to distribute the provenance and to allow for verification of the authenticity of the provenance and its compliance with policies on who to trust.

A first attempt could try to integrate the signing process of Hugenroth et al. [36] to provide a comprehensive and verifiable record of the build process.

# Chapter 11

# Conclusion

This thesis presents an approach to ensure the integrity of E2EE messaging applications for Android through reproducible builds. By leveraging the Nix programming language to create a hermetically isolated build environment, the proposed method guarantees that only the exactly specified resources will be part of the build process and no unauthorized modifications can be introduced during the build of the application. The Nix scripts are being called from a regularly run CI/CD pipeline, which checks for new releases of the Android applications and updates the Nix scripts accordingly. The Nix script then executes all the necessary steps to retrieve the source code and dependencies, to build the application, and to compare it to a vendor-published APK to verify the reproducibility of the build process. If the build process yields a binary that is bitwise identical, the resulting application will be published alongside its hash value, on the same instance of GitLab, that also hosts the R-B. This allows users to review the build process and execute it on their devices, as well as just comparing their locally installed APK to the one built by the Nix script.

The approach of this thesis allows to make statements on the trustworthiness of applications even if the applications resulting from the implemented build process are only *almost* bitwise identical.

The APKs for the E2EE messengers are being built in hermetically isolated build environments in a reproducible fashion. This way the complete set of tools, libraries, and other software artifacts that are involved in the build process are transparently described in the build process descriptions. The built APKs are also compared to the respective reference binaries that are published by the vendor. If the hash values of both APKs are identical, then the binaries are bitwise identical as well, and if they are not bitwise identical, then the self-built binary and the reference binary are being compared with one another using the tool diffoscope. The resulting report provides insight in the differences that exist between those two binaries, which allows users to judge whether the differences should not have any significant influence on the behavior of the application. If the differences are negligible and the users trust the inputs of the build process, then both the self-built and the official binary are trusted, and with that also any other binary that is bitwise identical to either of them.

If the differences could have a significant influence on the behavior of the application, the contribution does not provide any assurances on the trustworthiness of the official APK. If not all the inputs involved in the build process of the application are trusted, the contribution does not provide any assurances on the trustworthiness of either the self-built or the official binary.

While the approach successfully addresses many threats to the software supply chain, it does not mitigate vulnerabilities inherent in the source code or malicious logic embedded in build scripts. These threats need to be mitigated by other means, like, for example, source code analysis.

The current implementation of the approach allows to build the most recent versions of Signal and Wire Android, but the resulting binaries differ from the reference APKs and therefore future work should concern itself with closing these gaps to ensure full reproducibility of the applications.

Future work could focus on extending the verification process to include recursive integrity checks for dependencies and improving the automation of the update process for build tools and environments.

This thesis underscores the importance of reproducible builds in enhancing trust in software and provides a reference implementation for securing the supply chains of E2EE applications.

# Bibliography


[1] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. 2021. Solar Winds Hack: In-Depth Analysis and Countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. (July 2021), pp. 1–7. DOI: 10.1109/ICCCNT51525.2021.9579611. Retrieved 05/03/2025 from https://ieeexplore.ieee.org/document/9579611.

[2] Andres Freund. 2024. oss-security - backdoor in upstream xz/liblzma leading to ssh server compromise. (March 2024). Retrieved 07/28/2025 from https://openwall.com/lists/oss-security/2024/03/29/4.

[3] Anatoly Belous and Vitali Saladukha. 2020. Hardware Trojans in Electronic Devices. en. In *Viruses, Hardware and Software Trojans: Attacks and Countermeasures*. Anatoly Belous and Vitali Saladukha, (Eds.) Springer International Publishing, Cham, pp. 209–275. ISBN: 978-3-030-47218-4. DOI: 10.1007/978-3-030-47218-4_3. Retrieved 08/23/2025 from https://doi.org/10.1007/978-3-030-47218-4_3.

[4] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2010. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications* (*HotMobile '10*). Association for Computing Machinery, New York, NY, USA, (February 2010), pp. 49–54. ISBN: 978-1-4503-0005-6. DOI: 10.1145/1734583.1734596. Retrieved 08/23/2025 from https://doi.org/10.1145/1734583.1734596.

[5] Louis CAD. 2025. LouisCAD/CompleteKotlin. original-date: 2021-05-24T12:18:55Z. (May 2025). Retrieved 06/26/2025 from https://github.com/LouisCAD/CompleteKotlin.

[6] Chiara Castro. 2025. The EU still wants to scan all your chats – and the rules could come into force by October 2025. en. (July 2025). Retrieved 08/08/2025 from https://www.techradar.com/computing/cyber-security/the-eu-could-be-scanning-your-chats-by-october-2025-heres-everything-we-know.

[7] European Commission. 2022. Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL laying down rules to prevent and combat child sexual abuse. en. (April 2022). Retrieved 08/08/2025 from https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=COM%3A2022%3A209%3AFIN.

[8] Bootstrappable.org Contributors. 2025. Bootstrappable builds. (August 2025). Retrieved 08/19/2025 from https://www.bootstrappable.org/.

[9] Debian Contributors. 2025. Reproducible Builds / reproducible-notes · GitLab. en. (June 2025). Retrieved 06/29/2025 from https://salsa.debian.org/reproducible-builds/reproducible-notes.

[10] Diffoscope Contributors. 2025. diffoscope: in-depth comparison of files, archives, and directories. (May 2025). Retrieved 05/31/2025 from https://diffoscope.org/.

[11] F-Droid Contributors. 2025. Docs | F-Droid - Free and Open Source Android App Repository. (August 2025). Retrieved 08/11/2025 from https://f-droid.org/en/docs/.

[12] F-Droid Contributors. 2025. Reproducible Builds | F-Droid - Free and Open Source Android App Repository. en. (2025). Retrieved 08/19/2025 from https://f-droid.org/docs/Reproducible_Builds/.

[13] NixOS Contributors. 2025. nixpkgs. original-date: 2012-06-04T02:49:46Z. (August 2025). Retrieved 08/05/2025 from https://github.com/NixOS/nixpkgs.

[14] SLSA Contributors. 2025. Provenance. en. (August 2025). Retrieved 08/14/2025 from https://slsa.dev/spec/v1.1/provenance.

[15] SLSA Contributors. 2025. slsa-framework/slsa-verifier. original-date: 2022-03-25T21:01:47Z. (August 2025). Retrieved 08/14/2025 from https://github.com/slsa-framework/slsa-verifier.

[16] SLSA Contributors. 2025. Supply-chain Levels for Software Artifacts. en. (June 2025). Retrieved 06/18/2025 from https://slsa.dev/.

[17] Ludovic Courtès. 2022. Building a Secure Software Supply Chain with GNU Guix. en. *The Art, Science, and Engineering of Programming*, 7, 1, (June 2022), 1:1–1:26. Publisher: AOSA, Inc. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2023/7/1. Retrieved 05/03/2025 from https://programming-journal.org/2023/7/1/.

[18] Ludovic Courtès. 2013. Functional Package Management with Guix. en. arXiv:1305.4584 [cs]. (May 2013). DOI: 10.48550/arXiv.1305.4584. Retrieved 06/24/2025 from http://arxiv.org/abs/1305.4584.

[19] Google Android Developers. 2025. About Android App Bundles | Other Play guides. en. (August 2025). Retrieved 08/07/2025 from https://developer.android.com/guide/app-bundle.

[20] Google Android Developers. 2025. Build multiple APKs | Android Studio. en. (August 2025). Retrieved 08/07/2025 from https://developer.android.com/build/configure-apk-splits.

[21] Google Android Developers. 2025. Create an Android library | Android Studio. en. (August 2025). Retrieved 08/07/2025 from https://developer.android.com/studio/projects/android-library.

[22] Eelco Dolstra. 2006. *The purely functional software deployment model*. en. OCLC: 71702886. s.n., S.l. ISBN: 978-90-393-4130-8.

[23] Eelco Dolstra and Eelco Visser. 2007. Automated software testing and release with nix build farms. In *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS'07)*, pp. 65–77.

[24] William Easttom. 2022. *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. en. Springer International Publishing, Cham. ISBN: 978-3-031-12303-0 978-3-031-12304-7. DOI: 10.1007/978-3-031-12304-7. Retrieved 08/08/2025 from https://link.springer.com/10.1007/978-3-031-12304-7.

[25] Benjamin Elder. 2021. Remove Bazel by BenTheElder · Pull Request #99561 · kubernetes/kubernetes. en. (February 2021). Retrieved 04/18/2025 from https://github.com/kubernetes/kubernetes/pull/99561.

[26] Tad Fisher. 2025. tadfisher/android-nixpkgs. original-date: 2018-10-21T06:17:39Z. (August 2025). Retrieved 08/20/2025 from https://github.com/tadfisher/android-nixpkgs.

[27] Signal Foundation. 2025. reproducible-builds. en. (June 2025). Retrieved 06/29/2025 from https://github.com/signalapp/Signal-Android/tree/main/reproducible-builds.

[28] Signal Foundation. 2025. Signal-Android. original-date: 2011-12-15T20:01:12Z. (June 2025). Retrieved 06/29/2025 from https://github.com/signalapp/Signal-Android.

[29] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. 2023. It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1527–1544. DOI: 10.1109/SP46215.2023.10179320.

[30] Martin Geisse. 2013. Answer to "Compressing and Decompressing same files produces different size". (November 2013). Retrieved 06/29/2025 from https://stackoverflow.com/a/20073435.

[31] Wire Swiss GmbH. 2025. Technology. en-US. (April 2025). Retrieved 06/29/2025 from https://support.wire.com/hc/en-us/articles/4405932904209-Technology.

[32] Wire Swiss GmbH. 2025. wire-android. original-date: 2020-06-04T09:44:05Z. (June 2025). Retrieved 06/29/2025 from https://github.com/wireapp/wire-android.

[33] Google. 2025. Personalities. en. (August 2025). Retrieved 08/18/2025 from https://github.com/google/trillian/blob/05001d1876f9340e42ba8b839c94e1b79246207b/docs/Personalities.md.

[34] Google. 2015. VerifiableDataStructures. en. (November 2015). Retrieved 08/18/2025 from https://github.com/google/trillian/blob/30160804ab5203cde4412fe26f55a4149112bd92/docs/papers/VerifiableDataStructures.pdf.

[35] Klaus Horn. 2018. Code Signing Android and iOS Applications. en, (November 2018).

[36] Daniel Hugenroth, Mario Lins, René Mayrhofer, and Alastair Beresford. 2025. Attestable builds: compiling verifiable binaries on untrusted systems using trusted execution environments. arXiv:2505.02521 [cs]. (May 2025). DOI: 10.48550/arXiv.2505.02521. Retrieved 09/15/2025 from http://arxiv.org/abs/2505.02521.

[37] Vlad Iftimie. 2020. Some points on Android APK files. en. (March 2020). Retrieved 08/07/2025 from https://medium.com/@vlad.iftimie88/some-points-on-android-apk-files-231a36cbc91c.

[38] Thomas Hunter II. 2018. Compromised npm Package: event-stream. en. (November 2018). Retrieved 06/28/2025 from https://medium.com/intrinsic-blog/compromised-npm-package-event-stream-d47d08605502.

[39] Docker Inc. 2025. Build attestations. (August 2025). Retrieved 08/13/2025 from https://docs.docker.com/build/metadata/attestations/.

[40] Gradle Inc. 2025. Verifying dependencies. (June 2025). Retrieved 06/25/2025 from https://docs.gradle.org/current/userguide/dependency_verification.html.

[41] Aniket Indulkar. 2024. From Code to APK: The Complete Breakdown of Android Build Tasks. en. (November 2024). Retrieved 06/15/2025 from https://medium.com/@aniketindulkar/from-code-to-apk-the-complete-breakdown-of-android-build-tasks-dab1368a4107.

[42] Kashif Iqbal. 2019. APK - What is an APK file? en. (October 2019). Retrieved 08/07/2025 from https://docs.fileformat.com/compression/apk/.

[43] Stefan Kempinger. 2025. CrazyChaoz/gradle-dot-nix. original-date: 2024-03-12T11:37:17Z. (May 2025). Retrieved 06/26/2025 from https://github.com/CrazyChaoz/gradle-dot-nix.

[44] Arif Koyun and Ehssan Al Janabi. 2017. Social Engineering Attacks. en. *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, 4, 6.

[45] Chris Lamb and Ximin Luo. 2017. "SOURCE DATE EPOCH specification". Retrieved 06/29/2025 from https://reproducible-builds.org/specs/source-date-epoch/.

[46] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software*, 39, 2, (March 2022), 62–70. ISSN: 1937-4194. DOI: 10.1109/MS.2021.3073045. Retrieved 04/18/2025 from https://ieeexplore.ieee.org/document/9403390.

[47] Ben Laurie. 2014. Certificate transparency. *Commun. ACM*, 57, 10, (September 2014), 40–46. ISSN: 0001-0782. DOI: 10.1145/2659897. Retrieved 04/22/2025 from https://dl.acm.org/doi/10.1145/2659897.

[48] Ben Laurie, Adam Langley, and Emilia Kasper. 2020. Certificate Transparency. Request for Comments RFC 6962. Num Pages: 27. Internet Engineering Task Force, (January 2020). DOI: 10.17487/RFC6962. Retrieved 08/18/2025 from https://datatracker.ietf.org/doc/rfc6962.

[49] Ben Laurie, Eran Messeri, and Rob Stradling. 2021. Certificate Transparency Version 2.0. Request for Comments RFC 9162. Num Pages: 53. Internet Engineering Task Force, (December 2021). DOI: 10.17487/RFC9162. Retrieved 08/18/2025 from https://datatracker.ietf.org/doc/rfc9162.

[50] Mario Lins, René Mayrhofer, Michael Roland, Daniel Hofer, and Martin Schwaighofer. 2024. On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ. arXiv:2404.08987 [cs]. (April 2024). DOI: 10.48550/arXiv.2404.08987. Retrieved 04/18/2025 from http://arxiv.org/abs/2404.08987.

[51] Pei Liu, Li Li, Kui Liu, Shane McIntosh, and John Grundy. 2023. Understanding the quality and evolution of Android app build systems. en. *Journal of Software: Evolution and Process*, 36, 5, e2602. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2602. ISSN: 2047-7481. DOI: 10.1002/smr.2602. Retrieved 06/15/2025 from https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2602.

[52] Pei Liu, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. 2020. Androzoooopen: Collecting large-scale open source android apps for the research community. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 548–552.

[53] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, Dianne Hackborn, Bram Bonné, Güliz Seray Tuncay, Roger Piqueras Jover, and Michael A. Specter. 2021. The Android Platform Security Model (2023). *ACM Transactions on Privacy and Security*, 24, 3, (August 2021), 1–35. arXiv:1904.05572 [cs]. ISSN: 2471-2566, 2471-2574. DOI: 10.1145/3448609. Retrieved 07/28/2025 from http://arxiv.org/abs/1904.05572.

[54] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. 2020. Think Global, Act Local: Gossip and Client Audits in Verifiable Data Structures. arXiv:2011.04551 [cs]. (November 2020). DOI: 10.48550/arXiv.2011.04551. Retrieved 08/18/2025 from http://arxiv.org/abs/2011.04551.

[55] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. en. In *Advances in Cryptology — CRYPTO '87*. Carl Pomerance, (Ed.) Springer, Berlin, Heidelberg, pp. 369–378. ISBN: 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2_32.

[56] Lorraine Morgan and Patrick Finnegan. 2007. Benefits and Drawbacks of Open Source Software: An Exploratory Study of Secondary Software Firms. en. In *Open Source Development, Adoption and Innovation*. Joseph Feller, Brian Fitzgerald, Walt Scacchi, and Alberto Sillitti, (Eds.) Springer US, Boston, MA, pp. 307–312. ISBN: 978-0-387-72486-7. DOI: 10.1007/978-0-387-72486-7_33.

[57] Nix documentation team. 2025. nix.conf - Nix 2.28.4 Reference Manual. (June 2025). Retrieved 06/24/2025 from https://nix.dev/manual/nix/2.28/command-ref/conf-file.html.

[58] NixOS Wiki. 2025. Flakes. Documentation. (June 2025). Retrieved 06/24/2025 from https://nixos.wiki/wiki/flakes.

[59] Linus Nordberg, Daniel Kahn Gillmor, and Tom Ritter. 2018. Gossiping in CT. Internet Draft draft-ietf-trans-gossip-05. Num Pages: 57. Internet Engineering Task Force, (January 2018). Retrieved 08/18/2025 from https://datatracker.ietf.org/doc/draft-ietf-trans-gossip.

[60] Department of Commerce National Telecommunications and Information Administration. 2021. Software Bill of Materials Elements and Considerations. en. (June 2021). Retrieved 06/22/2025 from https://www.federalregister.gov/documents/2021/06/02/2021-11592/software-bill-of-materials-elements-and-considerations.

[61] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. Version Number: 1. (2020). DOI: 10.48550/ARXIV.2005.09535. Retrieved 04/20/2025 from https://arxiv.org/abs/2005.09535.

[62] Mike Penz. 2025. mikepenz/AboutLibraries. original-date: 2014-04-22T10:01:06Z. (June 2025). Retrieved 06/26/2025 from https://github.com/mikepenz/AboutLibraries.

[63] Google Android Open Source Project. 2025. App signing. en. (June 2025). Retrieved 06/19/2025 from https://source.android.com/docs/security/features/apksigning.

[64] Reproducible Builds Project. 2025. Reproducible Builds — a set of software development practices that create an independently-verifiable path from source to binary code. (May 2025). Retrieved 05/03/2025 from https://reproducible-builds.org/.

[65] Reproducible Builds Project. 2025. Tools — reproducible-builds.org. (June 2025). Retrieved 06/25/2025 from https://reproducible-builds.org/tools/#verifiers.

[66] Dev Random. 2025. devrandom/gitian-builder. original-date: 2011-01-30T21:08:50Z. (June 2025). Retrieved 06/25/2025 from https://github.com/devrandom/gitian-builder.

[67] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software*, 29, 6, (November 2012), 56–61. ISSN: 0740-7459. DOI: 10.1109/MS.2012.24. Retrieved 04/20/2025 from http://ieeexplore.ieee.org/document/6148202/.

[68] Jeremy Scahill and Josh Begley. 2015. The CIA Campaign to Steal Apple's Secrets. en-US. (March 2015). Retrieved 06/28/2025 from https://theintercept.com/2015/03/10/ispy-cia-campaign-steal-apples-secrets/.

[69] Martin Schwaighofer, Michael Roland, and René Mayrhofer. 2024. Extending Cloud Build Systems to Eliminate Transitive Trust. en. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. ACM, Salt Lake City UT USA, (November 2024), pp. 45–55. ISBN: 979-8-4007-1240-1. DOI: 10.1145/3689944.3696169. Retrieved 04/18/2025 from https://dl.acm.org/doi/10.1145/3689944.3696169.

[70] Scientists and researchers from across the globe. 2025. Joint statement of scientists and researchers on EU's proposed Child Sexual Abuse Regulation. (September 2025). Retrieved 09/18/2025 from https://csa-scientist-open-letter.org/Sep2025.

[71] SHISHIR. 2023. Android Build Process Step by Step. en. (December 2023). Retrieved 06/19/2025 from https://shishirthedev.medium.com/build-process-in-android-8c955d6467b8.

[72] FC (Fay) Stegerman. 2025. obfusk/apksigcopier. original-date: 2021-03-25T01:20:33Z. (June 2025). Retrieved 06/25/2025 from https://github.com/obfusk/apksigcopier.

[73] FC (Fay) Stegerman. 2025. obfusk/reproducible-apk-tools. original-date: 2022-11-22T04:03:34Z. (June 2025). Retrieved 06/25/2025 from https://github.com/obfusk/reproducible-apk-tools.

[74] Jonathan Stray. 2014. Security for Journalists, Part Two: Threat Modeling. en. (August 2014). Retrieved 06/18/2025 from https://source.opennews.org/articles/security-journalists-part-two-threat-modeling/.

[75] Determinate Systems. 2025. Zero to Nix. (June 2025). Retrieved 06/26/2025 from https://zero-to-nix.com/.

[76] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. 2016. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. (May 2016), pp. 526–545. DOI: 10.1109/SP.2016.38. Retrieved 08/18/2025 from https://ieeexplore.ieee.org/document/7546521.

[77] Nix Documentation Team. 2025. Welcome to nix.dev — nix.dev documentation. en. (June 2025). Retrieved 06/26/2025 from https://nix.dev/index.html.

[78] Jörg Thalheim. 2025. Mic92/nix-update. original-date: 2020-03-05T11:31:09Z. (June 2025). Retrieved 06/29/2025 from https://github.com/Mic92/nix-update.

[79] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM*, 27, 8, (August 1984), 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. https://doi.org/10.1145/358198.358210.

[80] Santiago Torres-Arias. 2020. *In-toto: Practical Software Supply Chain Security*. phd. New York University Tandon School of Engineering. AAI27963570 ISBN-13: 9798662407565.

[81] Jeff Williams. 2020. Removing a false sense of (open source) security. *Computer Fraud & Security*, 2020, 6, (June 2020), 8–10. ISSN: 1361-3723. DOI: 10.1016/S1361-3723(20)30062-2. Retrieved 08/08/2025 from https://www.sciencedirect.com/science/article/pii/S1361372320300622.