JOHANNES KEPLER
UNIVERSITY LINZ

Author
DI **Mario Lins**, BSc
01030360

Submission
Institute of
Networks and Security

First Supervisor
Univ.-Prof. Dr.
René Mayrhofer

Second Supervisor
Prof. **Alastair R. Beresford**

Assistant Thesis
Supervisor
Dr. **Michael Roland**

January 2026

Software Supply Chain Transparency

Doctoral Thesis

to confer the academic degree of

Doktor der technischen Wissenschaften

in the Doctoral Program

Technische Wissenschaften

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

Abstract

Software supply chains have become a central pillar of modern software development practices. While the underlying concept offers substantial advantages—meeting today's complex requirements for efficiency, consistency, and automation—it also increases the attack surface and introduces an evolving threat landscape, making supply chain attacks increasingly attractive to adversaries. This thesis presents new approaches to mitigate still existing threats targeting the software supply chain by introducing and enhancing existing protocols, leveraging existing security controls in novel ways, and, most importantly, increasing transparency to make malicious attempts to compromise software supply chains easier to detect.

We begin by examining threats to the software distribution stage—the point at which artifacts (e.g., mobile apps) are delivered to end users—with an emphasis on mobile ecosystems. A major security objective is to ensure authenticity and integrity of the artifact, which is commonly achieved by using digital signatures. However, singing keys can be lost, stolen, or abused to sign potentially malicious updates of an artifact. To address these threats, we introduce a new protocol that makes unauthorized distribution attempts verifiable and transparent to both developers and end users.

Ultimately, the secure operation of these applications depends on a trustworthy foundation—the operating system. Accordingly, we next focus on the operating system on mobile devices, with an emphasis on mitigating bootloader-targeted attacks. We demonstrate how to compensate lost security guarantees on devices with an unlocked bootloader. This allows users to flash custom operating systems on devices that no longer receive security updates from the original manufacturer without compromising security.

We then move to the source code stage. To better understand supply chain attacks at the source code level, we analyzed a recent attempt to implant a backdoor in an open-source project. This analysis allowed us to identify key takeaways of the adversaries' critical attack path and based on these learnings to propose new crawling methods that identify potential indicators of compromise in open-source projects.

Modern software development practices, however, extend beyond source code; nowadays, development teams often rely on external infrastructure provided by potentially untrusted cloud server providers. To mitigate the associated threats (e.g., insider attacks on the external infrastructure), we introduce a new architecture to ensure strong source-to-binary correspondence by leveraging the security guarantees of Confidential Computing technology.

Finally, we present *The Supply Chain Game*, an organizational security approach that enhances standard risk-management methods. We demonstrate how game-theoretic techniques, combined with common risk management practices, can derive new criteria to better support decision makers.

Kurzfassung

Software Supply Chains sind zu einem zentralen Bestandteil moderner Softwareentwicklungspraktiken geworden. Während das zugrunde liegende Konzept erhebliche Vorteile in Bezug auf die komplexen Anforderungen heutiger Software an Effizienz, Konsistenz und Automatisierung bietet, erhöht es gleichzeitig die Angriffsfläche. Dadurch wird die Bedrohungslandschaft erweitert und Angriffe auf Software Supply Chains werden für Angreifer zunehmend attraktiver. Diese Arbeit präsentiert neue Ansätze zur Minimierung bestehender Bedrohungen, die auf die Software Supply Chain abzielen. Es werden Ansätze vorgestellt, die durch die Einführung und Verbesserung bestehender Protokolle, die alternative Nutzung etablierter Sicherheitsmaßnahmen sowie durch eine erhöhte Transparenz Angriffe auf die Software Supply Chain leichter erkennbar machen.

Wir beginnen bei der Verteilung von Software—dem Punkt, an dem Software-Artefakte (z.B. mobile Apps) an Endbenutzer ausgeliefert werden—mit einem Schwerpunkt auf mobile Ökosysteme. Ein wesentliches Sicherheitsziel ist die Gewährleistung der Authentizität und Integrität des Artefakts. Dies wird üblicherweise durch die Verwendung digitaler Signaturen erreicht. Allerdings können die dafür relevanten Schlüssel verloren gehen, gestohlen, oder missbraucht werden und somit einem Angreifer die Möglichkeit bieten, schädliche Aktualisierungen zu signieren und zu verteilen. Um dieser Bedrohung entgegenzuwirken, stellen wir ein neues Protokoll vor, mit dem es möglich ist, unbefugte Verteilungsversuche sowohl für Entwickler als auch für Endbenutzer überprüfbar und transparent zu machen.

Letztendlich hängt die Sicherheit der Anwendungsschicht von einem vertrauenswürdigem Fundament ab—dem Betriebssystem. Dementsprechend konzentrieren wir uns als Nächstes auf das Betriebssystem von mobilen Geräten mit Schwerpunkt auf der Minimierung von Angriffen auf den Bootloader. Wir demonstrieren, wie verlorene Sicherheitsgarantien auf Geräten mit einem entsperrten Bootloader kompensiert werden können. Dadurch wird es Benutzern ermöglicht, auf Geräten, die beispielsweise keine Sicherheitsupdates des Herstellers mehr erhalten, alternative Betriebssysteme zu installieren, ohne die Security zu beeinträchtigen.

Anschließend wenden wir uns dem Quellcode (“Source Code”) zu. Zum besseren Verständnis von Supply-Chain Angriffen auf Source-Code-Ebene wird ein aktueller Versuch analysiert, eine Backdoor in ein Open-Source Projekt einzuschleusen. Diese Analyse ermöglicht es uns, wichtige Erkenntnisse aus dem kritischen Angriffspfad der Angreifer zu gewinnen und auf dieser Grundlage neue Methoden zur frühzeitigen Erkennung von derartigen Supply-Chain Angriffen, vorzuschlagen. Diese Methoden sollen frühzeitig potenzielle Anzeichen einer Kompromittierung in Open-Source Projekten erkennen.

Moderne Softwareentwicklungspraktiken gehen jedoch über den Source Code hinaus; heutzutage verlassen sich Entwicklungsteams oft auf externe Infra-

strukturen, die häufig von potenziell nicht vertrauenswürdigen Cloud-Server-Anbietern bereitgestellt werden. Um die damit verbundenen Bedrohungen (z.B. Insider Angriffe) zu minimieren, stellen wir eine neue Architektur vor. Dabei wird eine neue Source-to-Binary Korrespondenz unter Nutzung der Sicherheitsgarantien von Confidential Computing sichergestellt.

Abschließend präsentieren wir *The Supply Chain Game*—ein neuer Ansatz im Bereich der organisatorischen Security, der Standard-Risikomanagementmethoden erweitert. Wir demonstrieren, wie spieltheoretische Techniken in Kombination mit gängigen Risikomanagementpraktiken neue Kriterien ableiten können, um Entscheidungsträger besser zu unterstützen.

Funding

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

Acknowledgements

First and foremost, I want to thank my supervisor, René. I often think back to our first meeting at the institute, as I was beginning to search for new research opportunities in security. What began as a scheduled thirty-minute conversation quickly expanded into a memorable discussion about security challenges, new opportunities, and revealed a shared passion for this field—a pattern that continued throughout my PhD journey. Thank you, René, for our insightful meetings, for the countless hours invested in exploring new threats and solutions, and for inspiring me to push the boundaries of security research.

I would also like to express my deepest gratitude to my parents. I can never thank them enough—without their support, I would definitely not be where I am today. From my earliest moments to my decision to study computer science in Graz, they have been a constant source of encouragement. They consistently supported my pursuits and were my source of strength in challenging times. Thank you for everything; your belief in me has made this journey possible.

To Victoria—my partner, my best friend, and my constant pillar in my life—thank you. Your support, your patience in discussing complex security topics, and your tireless encouragement during even the most demanding deadlines have been crucial to my success, my motivation, and my happiness in life. I am deeply grateful for you being in my life.

I also want to thank my colleagues at the institute for valuable discussions and Stefan Rass for introducing me to the world of game theory. In particular, I want to thank Michael R. for his continuous support—not only as a LaTeX expert who ensured my papers were polished to perfection, but also for his insightful feedback and for opening my eyes to new areas in security beyond supply chains. I am also grateful to Tobias for being a stimulating discussion partner, always prompting me with exactly the right questions.

During my PhD, I had the opportunity to conduct a research stay in Cambridge. Thank you, Alastair and René for making this possible. This experience truly helped me grow in ways I never imagined and profoundly impacted both my academic and personal life. I am thankful to SN17, Alex, Daniel, Laurie, Luis, and Michael for their warm welcome, which made this experience so memorable.

Daniel, our meeting in Cambridge led to an incredibly productive brainstorming session on the whiteboard just days after arriving. Some months later we published a paper together at CCS and had an amazing time in Taiwan. Our shared objective to make security accessible to everyone inspired us to co-found Light Squares, and I am excited to start this new venture with you. Thank you for the invaluable collaboration, our productive meetings, and the amazing time we have when travelling to conferences and talks around the world.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In just a few decades, digitalization has profoundly transformed both our professional and personal lives, with software becoming a cornerstone of modern society. From revolutionizing business processes and financial services to fortifying critical infrastructure, software solutions are now indispensable to nations, organizations, and individuals alike. The rise of digital-only banks, digital identities, and other innovations underlines the rapid transformation of this technological evolution.

To keep up with the increasing demand of digitalization, software development has become more complex, requiring continuous adaptation, and accelerated development strategies. This evolution not only impacts the functionality of software but also reshapes how it is developed, assembled, and distributed to a growing number of users. A modern, but also essential development strategy to address these challenges is to leverage already existing development components, such as third-party libraries, development frameworks, or cloud-based infrastructure to ensure consistency and efficiency in the development process. For example, developers can leverage a cryptographic library (e.g., `libsodium`) for its implementations, frequently commit code changes to a repository where automated build processes are triggered, and, in some cases, have the final artifact directly published to their users. The paradigm around developing, building, and distributing software, as illustrated in Figure 1.1 is commonly referred to as *Software Supply Chain*[1].

Although the concept of utilizing software supply chains offers significant advantages for modern and efficient development processes, it has also expanded the attack surface, making it one of the most attractive and valuable targets for cyber attacks. As organizations worldwide already started to invest substantial resources to secure their infrastructure and software products, direct attacks on such fortified systems have become increasingly challenging. In response to that fact, adversaries are often shifting their focus on the software supply chain used by these organizations and try to exploit vulnerabilities in third-party libraries or other components in the supply chain to bypass specific security controls. This type of attack, where adversaries target the weakest link in the chain to compromise the actual target, is part of an emerging threat landscape and is commonly referred to as *supply chain attack*.

This evolved threat landscape, with its increasing sophistication and the potential for widespread attacks, is underscored by several notable supply chain

[1]Throughout this thesis, the term *Software Supply Chain* is used interchangeably with *Supply Chain*.

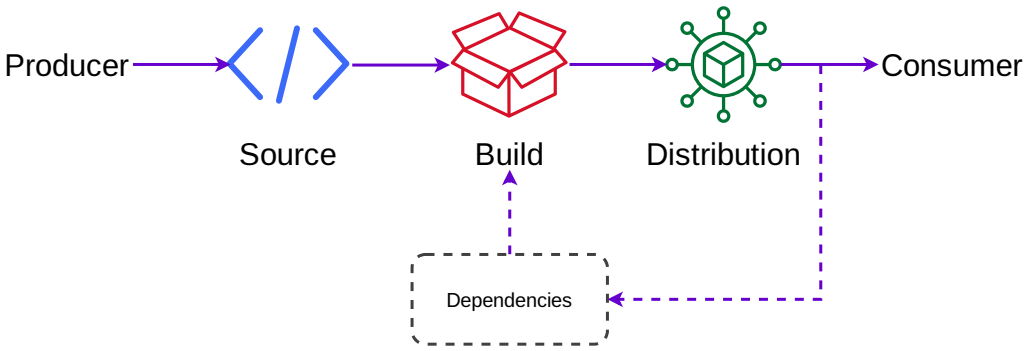


Figure 1.1: Software Supply Chain by SLSA collaboration [42]

attacks in recent years. For example, in the 2020 SolarWinds incident [163], adversaries successfully implanted malicious code into updates for their network management system software, which operates with elevated privileges in the customer's infrastructure. Thousands of customers, trusting the legitimacy of this update, unknowingly installed a malicious artifact, resulting in widespread malware infections. Similarly, in 2023, another roller coaster incident occurred when adversaries trojanized the VoIP solution provided by 3CX [66] (CVE-2023-29059). Apparently, adversaries were able to exploit an existing vulnerability (CVE-2013-3900) enabling them to implant malicious code in already signed artifacts without invalidating their digital signature. This attack had a significant impact, as 3CX's VoIP solution was used by more than 600,000 companies world-wide, leading to widespread trojan infections. Most recently, in 2024, a backdoor was discovered in XZ Utils (CVE-2024-3094), a widely used data compression library. This attack leverages highly sophisticated attack techniques to finally implant the backdoor that enables the adversaries to run commands remotely on vulnerable servers utilizing SSH [97]. Moreover, it perfectly underscores the potential of a supply chain attack where only one single vulnerable component in today's highly interconnected digital world is sufficient to cause far-reaching consequences. The SolarWinds, the 3CX, and the XZ incident perfectly illustrate the risks and potential for severe impacts of such supply chain attacks, emphasizing the need for new mitigation strategies to make it as difficult as possible for adversaries to be successful.

1.2 Objectives and Approach

A software supply chain can be a complex and sophisticated system, involving numerous components, entities, processes, but also potential vulnerabilities allowing adversaries to attack the supply chain at any stage. This field has already gained significant research attention, as the software supply chain become a critical component in a modern development process that encompasses everything from writing the source code, building and distributing the artifact, and finally enabling the end user to verify its authenticity and integrity. However, given the broad scope of software supply chains, it is impractical to develop a single, *all-encompassing* solution to address every relevant threat. Therefore, the objective of this thesis is to focus on specific parts of the soft-

ware supply chain on their own considering already existing solutions to provide new ways of addressing real-world threats still threatening modern software supply chains. A central approach of the suggested solutions is to enhance transparency in critical areas of the software supply chain, including mitigating risks resulting from untrustworthy trust anchors, such as compromised digital signatures due to lost private keys, ensuring the authenticity and integrity of build- and distribution processes, and enabling detection of unauthorized and potentially malicious activities throughout the software supply chain.

Below is a list of the main objectives of this thesis:

1. Introduce an end-to-end threat model for the software supply chain that guides this thesis.
2. Design and introduce new mechanisms to strengthen security at critical stages.
3. Formally specify and verify relevant protocols to ensure correctness and security.
4. Demonstrate effectiveness and practicability by conducting various performance, scalability, and security evaluations.

1.3 Publications

A significant part of this thesis has been published as scientific papers (5 peer-reviewed):

1. **M. Lins**, R. Mayrhofer, M. Roland, and A. R. Beresford: **"Mobile App Distribution Transparency (MADT): Design and Evaluation of a System to Mitigate Necessary Trust in Mobile App Distribution Systems"**, in Secure IT Systems. 28th Nordic Conference, NordSec 2023, Oslo, Norway, LNCS, vol. 14324/2024, Springer, pp. 185–203, 2023.

   The *MADT* paper is the basis of chapter 4. As the first author, I was mainly responsible for the core parts of this work—including threat modeling, protocol design, and evaluation—strengthened through ongoing collaboration and guidance from my supervisors, René and Alastair.

2. **M. Lins**, R. Mayrhofer, M. Roland, D. Hofer, and M. Schwaighofer: **"On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ"**, Preprint, Computing Research Repository (CoRR), arXiv:2404.08987 [cs.CR], 2024.

   This paper was among the first scientific publications on arXiv to summarize the XZ backdoor incident, including some potential mitigation suggestions. As the first author, I was mainly responsible for the core parts of this work—backdoor analysis, and extracting the critical attack path. The other co-authors contributed in listing potential mitigations.

3. **M. Lins**, R. Mayrhofer, and M. Roland: **"Unveiling the critical attack path for implanting backdoors in supply chains: Practical experience from XZ"**, in Cryptology and Network Security, 24th International Conference, CANS 2025, Osaka, Japan, LNCS, vol. 16351, Springer, pp. 521-541,2025.

This paper is the basis of chapter 6. As the first author, I was mainly responsible for the core parts of this work—backdoor analysis, extracting the critical attack path, implementing *SketchyCrawler*, and the evaluation part—strengthened through ongoing collaboration and guidance from my supervisor.

4. **M. Lins**, R. Mayrhofer, and D. Zeuthen: **"Firmware Transparency: Strengthening Security Guarantees Against Bootloader-Targeting Attacks"**, accepted at 24th International Conference on Applied Cryptography and Network Security (ACNS), New York, 2026.

   This paper is the basis of chapter 5. As the first author, I led the design of the protocol, created the formal model including the verification, conducted the threat model, implemented the prototype, and evaluated the security aspects—strengthened through ongoing collaboration and guidance from my supervisor, René.

5. D. Hugenroth[1], **M. Lins**[1], R. Mayrhofer, and A. R. Beresford: **"Attestable builds: compiling verifiable binaries on untrusted systems using trusted execution environments"**, in Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, ACM CCS 2025, Taipei, Taiwan, Association for Computing Machinery, pp. 4514–4528, 2025

   This paper is the basis of chapter 7. Daniel and I are both first authors of this paper, and were mainly responsible for the core parts of this work—threat modeling, architecture, evaluation—strengthened through ongoing collaboration and guidance from our supervisors.

6. **M. Lins**, S. Rass, and R. Mayrhofer: **"Software Supply Chain Security: Can we beat the kill-chain? A Case Study on the XZ backdoor"**, in Secure IT Systems. 30th Nordic Conference, NordSec 2025, Tartu, Estonia, LNCS, Springer Nature Switzerland AG, 2025

   This paper is the basis of chapter 8. As the first author, I was mainly responsible for the core parts of this work—defining attacker and defender capabilities, including assessments. The formal definitions used in this chapter are based on those presented by Rass in our paper [101], specifically those describing the game theoretic aspects.

As part of my research, I contributed as a co-author to the following publications:

1. R. Mayrhofer, M. Roland, T. Höller, P. Hofer, **M. Lins**: **"An Architecture for Distributed Digital Identities in the Physical World"**, Preprint, Computing Research Repository (CoRR), arXiv:2508.10185 [cs.CR], 2025.
2. K. Machetanz, **M. Lins**, C. Roder, G. Naros, M. Tatagiba, H. Hurth: **"Innovative mobile app solution for facial nerve rehabilitation: a usability analysis"**, in Front. Digit. Health, 21 October 2024, Volume 6, 2024

1.4 Outline

The structure of this thesis is designed to align with specific stages of the software supply chain, providing new ways to mitigate still existing threats. We

[1]These authors contributed equally to this work.

begin at the point of use—the end user receiving an artifact via a distribution channel—because trust must ultimately be verifiable where the software component is actually used. We then continue with an abstract view of the Source and Build stages to trace backdoors hidden in open-source projects. Next, we dive deeper into the build infrastructure to establish strong source-to-binary correspondence. Finally, we broaden to organizational security, presenting a holistic approach to risk management and mitigation for supply chain threats.

Software Distribution: Signatures, Transparency, and Provenance. In this chapter we analyze the distribution stage of the software supply chain, with an emphasis on mobile ecosystems. We introduce a new mechanism to increase the level of security in current mobile app distribution approaches. We then develop end-user verifiability techniques that enable independent validation of app provenance and integrity.

Bootstrapping Trust: Strengthening Security Guarantees Against Bootloader-Targeting Attacks. Since the security of the application layer still depends on the trustworthiness of the underlying (operating) system layer, we explore threats at this deeper layer of mobile devices as well and design a security protocol that preserves device security even when the bootloader is unlocked—supporting freedom-of-choice and sustainability when device manufacturers discontinue security update cycles. This part concludes in an end-to-end verification framework spanning both the app and (operating) system layers.

On the Critical Attack Path for Implanting Backdoors in Supply Chains. The third part of this thesis primarily investigates the Source and Build stages of the software supply chain. In this part, we focus on a supply chain attack detected in 2024 targeting XZ Utils as a central case study. We introduce what we call the *Critical Attack Path* that highlights the most relevant stages—from long-term maintainer infiltration to hiding a backdoor in an open-source project, even one that is publicly verifiable. To secure real-world operational aspects, we introduce *SketchyCrawler*, a tool that identifies sketchy signs in open-source repositories (e.g., sudden change of maintainer(s), opaque binary artifacts, etc.) helping to reveal supply chain compromises before they propagate, based on our takeaways from our backdoor analysis.

In Cloud We Trust? Verifiable Strong Source-to-Binary Artifacts. The fourth part of this thesis focuses specifically on build infrastructures where we do not want to or where we cannot trust the underlying (potentially opaque) build pipeline. Particularly important here is the link between verifiable source code and final artifacts that are difficult to audit. This challenge is also related to the "Trusting Trust" [157] problem, popularized by Ken Thompson's Turing Award lecture. A notable example is the SolarWinds compromise, where adversaries infiltrated the build infrastructure: code audits appeared legitimate, and only specific forensic analysis revealed that the malicious code was injected during the build process. To counter this, we leverage modern trusted execution environments (TEEs) and sandboxing to provide strong and verifiable source-to-binary correspondence. We refer to this new approach as *Attestable*

Builds. In contrast to the previous chapter, this part of the thesis focuses more on the build stage rather than the source code itself. However, from a holistic view on supply chain security, both are relevant and have the possibility to complement each other.

The Supply Chain Game - Can We Beat The Kill-chain? Finally, the last part of this thesis, focuses on organizational security, especially on risk management—an essential early-stage practice for preventing supply chain attacks and selecting effective mitigations. Conventional risk management frameworks quantify risk for specific threats, but they rarely provide actionable reasoning on which specific mitigation is better to implement in order to have an advantage over an adversary. Therefore, the fifth part of this thesis demonstrates a new organizational approach that combines standard risk management processes with game-theoretic modeling, helping decision makers prioritize or allocating resources to increase their chances to ultimately win the supply chain game.

Conclusion and Future Work. Ultimately, we conclude on how the proposed mechanisms strengthen the overall security posture of software supply chains, highlighting major improvements in resilience, verifiability, and transparency. We further provide an outlook of future work in this area.

Chapter 2

Background

This chapter establishes the technical foundations for this thesis by introducing relevant components of the software supply chain, existing security controls, data structures that we use, and additional fundamentals of our scientific contribution to counter supply chain attacks.

2.1 Software Supply Chain

Modern software development strategies, typically involve multiple components, such as source code, build instructions, build- and distribution infrastructure. This thesis primarily follows the Software Supply Chain framework (see Figure 1.1) provided by the SLSA team [42]. To improve readability, we use the terms "user" instead of "consumer" and "developer" instead of "producer"; these terms are used interchangeably throughout this thesis. The following paragraphs briefly introduce the stakeholders we consider, as well as the individual stages, to set the constraints for this thesis.

2.1.1 Stakeholders

In this thesis, we primarily focus on four stakeholders of the software supply chain: the developer, the user, the distributor, and a set of (external) service providers. A stakeholder may also have more than one role—for example, a developer who also operates a distribution system (e.g., self-hosted F-Droid instance).

Developer. The developer role—also referred to as producer—does not only cover the developer that writes the source code; it also includes the maintainer, the team, or the organization that owns and is responsible for the code. Typically, this role decides where the source code is stored and how the software is built and distributed to users. A key security objective for the developer is to prevent any unauthorized party from distributing artifacts—such as malicious updates—that appear to originate from the developer. In this thesis, we do not consider confidentiality of the source code as an objective of the developer; this is an orthogonal security investigation.

User. The user—also referred to as consumer—is the party that installs, references, or operates the artifact. This may be a system administrator, another software manufacturer incorporating an external library, or an end user running the software on a PC. The user's primary security objective is to ensure that the artifact is authentic and has not been compromised. Accordingly, a central goal of this thesis is to strengthen the verifiability by increasing transparency.

Distributor. The distributor aims to distribute artifacts (e.g., mobile apps, libraries, etc.) to users via secure channels. Accordingly, the distributor utilizes additional security controls—such as digital signatures (section 2.2.2) to prevent spoofing attacks and the distribution of malicious updates[1].

Service Provider(s). Throughout this thesis, we also consider various (external) service providers. A Repository Hosting Provider (RHP)—for example, GitHub [50] or on self-hosted platforms like GitLab [74]—stores and serves source code. A Build Service Provider (BSP) typically offers a software-as-a-service tooling that builds final artifacts from source. As scalability and flexibility is an important aspect nowadays, both RHPs and BSPs often rely on external cloud infrastructure provided by Cloud Service Providers (CSPs), such as Amazon Web Service (AWS), Microsoft Azure, or Google Cloud Platform. Even alternative solutions, such as GitLab, that could be self-hosted are likewise often deployed on CSPs, like Hetzner Cloud. There is no restriction that the RHP, the BSP, and the CSP is the same entity. In this thesis we do not assume these (external) service providers to be trustworthy.

2.1.2 Stages

Source. Source code defines the software's functionality and behavior and is typically human-readable (unlike binary artifacts). In practice, a software project includes not only code but also other important assets—such as build scripts, metadata, documentation, or configuration files. Modern Software Engineering (SWE) is typically collaborative and tool-mediated: development teams utilize Version Control Systems (VCSs) such as Git [132], along with other tooling to automate and standardize specific development processes. The source code itself is often managed in so-called repositories hosted by RHPs.

Build. The build stage in a modern software supply chain is responsible for transforming source code into the final artifact (e.g., a binary, a package, or a container image). This stage serves as a critical trust anchor: if an adversary compromises the toolchain, they could implant malicious code into the final artifact without modifying the actual source code. Modern SWE often involves external service providers (e.g., BSP, CSP) to build the final artifact and do not use their own development stack for that.

[1] Payment and IP protection mechanisms are already addressed in existing systems, especially in mobile ecosystems and considered out of scope of this thesis.

Distribution. The distribution stage propagates the built artifact to users. Distribution channels include web downloads, package registries, and large app marketplaces such as the Google Play Store.

2.2 Build Processes

Because build processes are critical in the software supply chain, we examine them in depth—starting with introducing modern software engineering practices and finally highlighting existing security techniques (e.g., artifact signing or reproducible builds).

2.2.1 Modern Software Engineering & CI/CD

To meet today's demands for continuity and speed requirements, projects typically leverage Continuous Integration (CI) pipelines that trigger on specific events in the repository such as committing new code to it; these pipelines build, test, lint, and even verify changes before they are merged into the main branch. Many teams also adopt Continuous Deployment (CD) pipelines to build and release the final artifacts to staging or production environments. The instruction how the pipelines are processed are typically defined declaratively in provider-specific configuration files, for example GitHub Actions [48], Jenkins [129], or GitLab CI [49] configurations. The task itself is executed by a BSP and its runners.

2.2.2 Artifact Signing

Another essential step in modern CI/CD pipelines often performed by the BSP is artifact signing. Digital signatures allow end users to verify an artifact's authenticity—i.e., it was truthfully produced by the intended developer or built by a trusted party—and its integrity, ensuring it has not been tampered.

For example, the signing process used by Google Play Store provides an integrated feature, called Play App Signing [60], that manages and protects the private key used for signing the APK file[2]. This approach requires that the private key is managed by Google's Key Management Service, and thus it needs to be stored on Google's infrastructure. For Android apps published before August 2021, the Play App Signing approach is optional and developers can still manage app signing keys themselves. However, for newly published apps, the Play App Signing approach is mandatory. This particular policy adaption by Google results in a centralized trust anchor that has to be trusted by both the user and the developer (more details in chapter 3).

F-Droid [36] is an alternative distribution system for free and open-source Android apps. If a developer wants to sign an APK file, F-Droid provides two possible procedures for that: one approach is to publish two APK versions where one is signed by the developer and the other one is signed by the F-Droid repository

[2] As developer identities are not directly verified by most Android app distribution systems, authenticity of signing keys is typically only guaranteed in the Trust-on-First-Use (TOFU) model.

(with a key held by F-Droid, comparable to Google Play Signing in this case). This is especially useful for distributing updates for apps that have been installed via different distribution channels (e.g., Play Store) and for apps available through F-Droid. This approach requires including the signature of the developer into the corresponding metadata description of the particular app and that the app is still reproducible by F-Droid. The other approach requires the developer to provide a reference to the signed APK file. If F-Droid is able to reproduce the APK file in a way that it matches the referenced one, F-Droid publishes the signed APK of the developer directly without signing it again with the F-Droid repository key [36].

The underlying principles of the signing process apply equally to software artifacts beyond mobile apps, including *.deb* and *.rpm* packages for Linux package managers, for example.

2.2.3 Reproducible Builds

Modern CI/CD pipelines bring substantial benefits, but they also introduce new trust relationships with external service providers (e.g., RHPs, CSPs). From a threat-model perspective, these systems often operate as black boxes: it is difficult to exactly know what they do or whether they interfere with the build process, internally. Because source code is comparatively easy to inspect and binary artifacts are hard to audit and verify, we have to accept the considerable trust we put in this intermediate black box when using modern CI/CD pipelines.

A key security objective, therefore, is to verify source-to-binary correspondence. Reproducible Builds (R-Bs) address this challenge by making the build process fully deterministic—building the same source multiple times yields a bit-for-bit identical artifact. If a build server is malicious and attempts to implant a backdoor in the final artifact, this can be detected by independently rebuilding the same project on a different machine: any divergence in outputs signals that the artifact is either not reproducible or that something potentially malicious is occurring.

Despite their security guarantees, R-Bs are challenging [15, 43, 144] to achieve and are not always feasible. Determinism must hold across all build steps and outputs, which are often influenced by non-deterministic factors such as timestamps, metadata, race conditions, file ordering, path variables, or even uninitialized memory [43, 144]. While many of these sources of entropy can be mitigated (with effort), some elements—such as digital signatures on intermediate assets—are difficult or even impossible to avoid. Reproducibility is also transitive when focusing on dependencies: a downstream package can only remain reproducible if all of its dependencies are reproducible as well. Consequently, developers of libraries must ensure that their upstream dependencies adhere to reproducible practices, too. There are real-world success examples: Debian [128], Chromium [131], and Tor [124] have realized reproducibility of their packages in practice. However, the effort required to achieve this is substantial—not only during initial development but also for ongoing maintenance [43, 90]. According to Debian, it took them twelve years to produce a fully reproducible Debian image [46, 103]. One challenge they highlight is motivating upstream developers to deliver reproducible packages; one strategy has been to offer prioritized inclusion of their packages as a kind of bounty

system [46] if they provide a reproducible package. Their dashboard [35] illustrates steady progress—for example, the amount of unreproducible packages decreased from 6.1% in the Stretch release (2017) to 2.0% in the Bookworm release (2023). We assume that the remaining packages are particularly difficult to make reproducible.

2.2.4 Build Systems for Linux Packages

Linux distributions, such as Debian, contain thousands of packages that work together to ensure seamless functionality. These packages are often developed by multiple open-source contributors and distributed either as already pre-existing components of the operating system through package managers, like Advanced Package Tool (APT), or Dandified YUM (DNF). To include an open-source package in their distribution system, it is a common practice to provide the related source code files. According to e.g., the "UpstreamGuide" of Debian [30], it is considered best practice to provide only source code files that make a rebuild possible. However, they also acknowledge that the rule "rebuild everything" is applied inconsistently, particularly for packages utilizing `autoconf` and `automake`. Thus, the current toolchain of Debian requires source code files to be submitted within a tarball. As a result, it is quite common to find a source tarball on the release pages of various open-source projects available on RHP, like GitHub. Some open-source projects also include pre-built binaries on these release pages. However, these are typically not used by package distribution systems, which build the final artifact from source instead. Another particularly relevant aspect of such package distribution systems is that the upstream sources are also expected to include test suites as part of the source tarball, which will be executed as part of the CI pipeline to verify the intended behavior.

2.2.5 GNU M4

GNU `m4` [52] is used as an implementation of a macro processor standardized by POSIX including some additional built-in functions, such as running shell commands. Additionally, it is used by packages built with GNU `autoconf` for generating the `configure` scripts that customize the Makefile for compilation.

2.2.6 Google OSS-Fuzz

Google's OSS-Fuzz [59] is a continuous fuzzing tool for open-source software to detect programming errors, like buffer overflows. According to the documentation [53] only projects with a significant user base and/or critical projects are included in the fuzzing process. To include a new project, it is necessary to open a pull request with information about the project (e.g., repository URL, primary contact).

2.3 Confidential Computing

A key challenge in using external, potentially untrusted service providers is executing code in an environment that neither the developer nor the user fully trust. For example, a company processing sensitive health or personally identifiable information (PII) in the cloud may not want to rely on the cloud provider's statement that no insider or nation-state actor can access the data. This problem is orthogonal to familiar data-security mechanisms such as securing data in transit and at rest; here, the objective is to protect data in use (i.e., during computation).

Trusted Execution Environments (TEEs)—often mentioned in context of "confidential computing"—may provide these security guarantees. A TEE allows code to run inside a specific enclave such that execution state and the memory are isolated from the operating system, the hypervisor, or adversaries with physical access. The memory regions utilized by the enclave are encrypted, mitigating even insider attacks that intercept the communication channel between the CPU and RAM. The interaction between the hypervisor and the enclave is restricted; only specific interfaces are available—such as *vsock* or shared memory [83].

In an earlier iteration this technology, such as Intel SGX [25], executed an enclave on a process level but was susceptible to side-channel attacks [18, 112, 146, 159]. Although side-channel attacks still remain possible, the technology significantly improved since then and hardware like Intel TDX [23], AWS Nitro, or AMD SEV-SNP [73] became available. This modern technology is even capable of booting entire virtual machines (VMs) as an enclave.

To verify whether the code is running inside such a protected environment, TEEs support attestation. The verifier sends a challenge to the software that claims to run inside a TEE. Next, the software requests the attestation from the TEE including this challenge and receives specific measurements of the boot process and the system configuration.

2.4 Android

In this thesis, we often use Android as our reference platform—owing to its open-source stack, already existing security implementations, and wide deployment. This section introduces the Android components most relevant to this thesis. Although some of the components are Android-specific, many of the underlying techniques (e.g., Verified/Secure Boot, hardware-backed key management, etc.) can also be leveraged in other domains such as embedded systems and IoT security.

2.4.1 Android Bootloader

The boot process of Android involves multiple components [134, 154]. The first component getting active when pressing the power button, is called the *Primary Boot Loader (PBL)*. The PBL is responsible for loading the *eXtensible Boot Loader (XBL)* into memory and typically implemented on a tamper-resistant chip by

the chipset manufacturer. Thus, we assume this component to be trustworthy [105].

Compared to the PBL, the XBL (formerly Secondary Boot Loader (SBL)) can be customized by the specific device manufacturer and is responsible to load the next bootloader stage, called *Android Boot Loader (ABOOT)*.

The main purpose of ABOOT is to load the kernel that initializes the operating system. Furthermore, it implements an interface to run custom commands directly in the bootloader stage of a device, for example, to flash a new image to a specific partition or to perform recovery activities. This bootloader part and especially the implemented interface is different depending on the manufacturer. As an example: Google calls this interface *fastboot* [139], while on Samsung devices this interface is called *ODIN* [158].

2.4.2 Android Security Controls

Google provides several security controls to mitigate tampering attempts, as outlined by "The Android Platform Security Model" by Mayrhofer et al. [105]. These controls address runtime-security considerations, including *Mandatory Access Control* or the TEE, but also deal with booting up the operating system securely through methods such as *Android Verified Boot* (AVB).

Android Verified Boot (AVB)

AVB is used to protect the system against malicious tampering attempts by verifying executable code and relevant partitions (e.g., boot, system, vendor, OEM partitions) used to boot up the operating system (see Figure 2.1). This security control verifies the chain of trust starting from the hardware-protected PBL stage to the XBL and ABOOT stage ending up with the verification of pertinent partitions. As stated in section 2.4.1, we consider the chain of trust starting at the PBL up to (but not including) the ABOOT stage to be trustworthy. The earliest stage at which the chain of trust may be customized is the ABOOT stage by setting a custom root of trust. This custom root of trust is used to sign the expected hash value of the relevant partitions, that are either stored on the specific partition itself or on a dedicated one called *vbmeta*.

The vbmeta struct contains data, like hashes or hash trees in case of larger file systems, that are used by AVB for verification purposes. For example, it contains the hash of the *boot* partition and hash trees representing the *system* and *vendor* partitions. Furthermore, the vbmeta partition also includes an index used to prevent rollback attacks (e.g., flashing an older and potentially vulnerable system image on a device). The main partition used for storing the vbmeta struct is called *vbmeta*. However, additional *vbmeta* partitions (e.g., *vbmeta_vendor*) used to store vendor specific information are also possible. The term *vbmeta digest* refers to a digest including all vbmeta structs, including the root and vendor specific vbmeta partitions. It can be used to verify the authenticity and integrity of the vbmeta structs and additionally, it is also embedded in the hardware-based attestation data so that it can be queried for verification even if the operating system is not trustworthy.

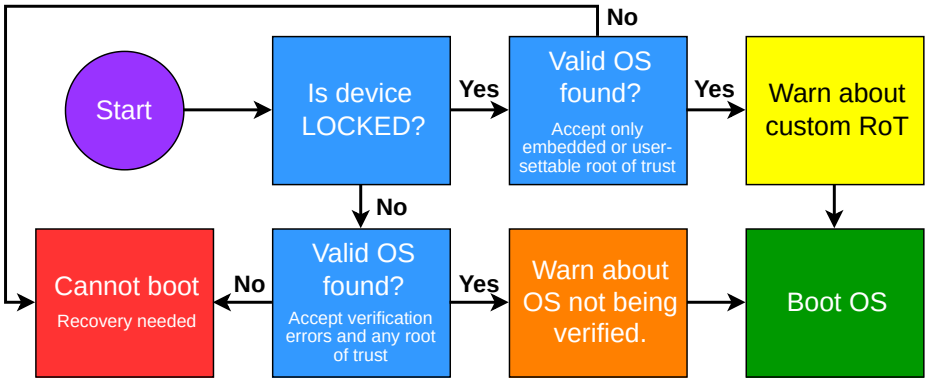


Figure 2.1: Simplified Android verified boot flow based on [55]. Taken from our paper [98].

AVB defines four possible device states – *GREEN*, *YELLOW*, *ORANGE*, and *RED*. Device states are used to indicate if a user can flash a custom image and whether the verification controls are enforced by AVB.

- GREEN: Device is locked without custom root of trust.
- YELLOW: Device is locked with custom root of trust.
- ORANGE: Device is unlocked.
- RED: Device is either in I/O error mode[3] or no valid operating system (OS) is found.

The *GREEN* state indicates that the device is locked without a custom root of trust. It includes all available verification steps and requires the user to not set a custom root of trust, which means that the verification is performed by using original root of trust set by the manufacturer. The *YELLOW* state indicates a locked device with a custom root of trust. Availability of this stage depends on whether the manufacturer permits setting a custom root of trust. Currently, Google is the only (major) manufacturer that allows users to set a custom root of trust on Pixel devices. The verification part of the *YELLOW* state is equal to the one used in *GREEN* state as a root of trust is present and thus the integrity of the partitions can be verified. The most relevant state for the contribution in chapter 5 is *ORANGE* where the bootloader is *unlocked*. An unlocked bootloader allows the user to flash arbitrary images on the partitions and furthermore it disables verification steps, such as signature verification. This state does not require the operating system to be signed by any root of trust. The *RED* state indicates errors that prevent booting the operating system.

At the time of writing this thesis, there was just one mobile device family allowing to set a custom root of trust (*YELLOW* state), namely Google Pixel. Other manufacturers like Motorola, Nokia, Samsung, or Fairphone [37] do not allow setting a custom root of trust, but do also not fully prevent users to flash a custom image on the device (*ORANGE* state). Hence, if a user wants to flash a custom image on a device other than Google Pixel, they must unlock the bootloader, and once unlocked, they cannot re-lock it again with the custom

[3]This mode arises from dm-verity verification errors, like invalid hash of boot partition or verification issues on the system partition.

image due to manufacturer restrictions, which do not allow setting a custom root of trust. This is a significant disadvantage for this particular user group as evidenced by a security- and privacy-focused alternative Android fork called GrapheneOS [64], which does not even support other devices than Google Pixel due to exactly these reasons. We believe that nowadays, users should have full ownership and, consequently, control over their devices whenever feasible with respect to security, and the freedom of choice to determine which operating system they want to run on their hardware. Thus, in chapter 5, we focus on compensating security controls to overcome lost security guarantees of a device that is in the *ORANGE* state. However, our security concept can also be used in other states to include an additional layer of security in terms of transparency and verifiability from a third-party point of view.

Trusty

Trusty is a specific operating system kernel that is isolated through hardware assisted means from the main operating system kernel (in the case of Android, the Linux kernel) and provides a TEE. Software that is running in a TEE, like cryptographic key storage (Android KeyMint), password authentication (Android Weaver), etc. can be executed in an isolated enclave to remain secure in the event of an operating system compromise.

StrongBox

Android StrongBox is a dedicated hardware security module (HSM) with its own processor and memory, either implemented through custom hardware or applets on a generic secure element (SE). We consider StrongBox, to be secure and functioning as expected, as it leverages the security guarantees provided by the hardware layer. Furthermore, we assume equal security guarantees when comparing it to a Trusted Platform Module (TPM) that is considered a standard component in server and desktop systems. Thus, we believe that our concept is also applicable to other systems that use a standard TPM.

Android Attestation

Android attestation is a foundational trust anchor for AVB, as it produces integrity-protected evidence about a device's state. In particular, the resulting attestation certificate includes, for example, the boot state, the verified boot key, and the verified boot hash—details that are especially relevant to our proposal. When attestation is enforced by the device's TEE or StrongBox, it is hardware-backed (as mentioned in the Android documentation [58]): relevant information is gathered or generated by code running in the secure environment that is isolated from the Android platform, making the results non-forgeable by an adversary as long as the secure hardware and the boot chain up to ABOOT remains trustworthy—an assumption in our threat model. Consequently, our system requires TEE- or StrongBox-backed attestation; software-only attestation is excluded because it relies on a locked bootloader, which we do not assume. Further details on how we leverage Android attestation are provided in section 5.3.1.

2.5 XZ Utils

As several upcoming chapters refer to the security incident that occurred in the XZ Utils project, this section provides background information on the project. XZ Utils is a widely distributed set of open-source packages used for lossless data compression on Unix-like operating systems, including Linux. Beside supporting compression with the *xz file format*, XZ Utils also supports the legacy *lzma* format [156]. Because of its high compression ratio, in the last decade it has been included in many basic system components such as archival/compression tools, package installers like `dpkg` and `rpm`, the squashfs filesystem,[4] and even the Linux kernel itself for decompression of the kernel image and initramfs.[5]

2.5.1 GNU Indirect Function Support (IFUNC)

A critical component of the XZ backdoor incident was Indirect Function Support (IFUNC) [121], a feature of the GNU toolchain allowing developers to define multiple function implementations and to choose one of them at runtime. For selecting the proper implementation, the developer uses a resolver function. The dynamic loader calls the resolver function during startup and loads the corresponding implementation.

2.6 Transparency Logs

One of the most important data structure used in this thesis are transparency logs. A transparency log [34, 91, 92]—also referred to as *verifiable log*—is a data structure that is based on an append-only ledger that is cryptographically secure. The Merkle tree is a popular example where it is not possible to retroactively insert, delete, or modify any record. One of the main advantages of this data structure is that these properties are auditable, either publicly or at least by its users (e.g., when hosted in an internal network). The data stored in a transparency log is application-specific, but is not defined or restricted by the log itself. A transparency log is stored on one or preferably multiple servers that are accessible by users, which may not necessarily be trusted. Users do not have to trust the log server as the data structure allows integrity verification.

Merkle Trees

We incorporate transparency logs backed by a Merkle tree [108] to allow efficient auditing; combined with the design of hash proofs (section 2.6) and witnesses (section 2.6.3), this provides tamper protection due to the resulting append-only property. The Merkle tree consists of leafs and nodes, with the top node called *root node*. The leaves represent data that are managed by the tree. Values are attached to internal nodes and are calculated as a cryptographic hash function (e.g., SHA-256) of their children, recursively, until a value of the root

[4] https://docs.kernel.org/filesystems/squashfs.html
[5] https://docs.kernel.org/staging/xz.html

node is reached. Trees do not need to be balanced and therefore can store an arbitrary amount of data.

Inclusion Proofs

An *inclusion proof* allows one party to prove to another that a particular leaf exists in a Merkle tree. This proof can be constructed efficiently, as it only requires the so-called Merkle Audit Path [57, 91, 92], which represents the shortest path from the respective leaf to the root node hash of the tree. The remaining leaves and nodes are not needed for this calculation. This approach means that the calculated root node hash is compared to the expected root node hash. If these hashes are equal, we have proven the particular leaf is part of the tree[6]. The left tree in Figure 2.2 highlights the path that is required for such an inclusion proof for *Record 2*. The required components for the calculation are marked in red. Below is a step-by-step description of validating an inclusion proof for the example given in Figure 2.2.

Given an ordered list of node hashes `A` and `C`, the inclusion proof can be verified as followed:

1. Calculate the leaf hash of *Record 2*: `B` = SHA-256(`0x00` || *Record 2*), where `0x00` is used as a prefix for leaf hashes and `0x01` for nodes to provide second pre-image resistance [92].
2. Calculate the node hash `E` = SHA-256(`0x01` || `A` || `B`).
3. Calculate the root node hash `root` = SHA-256(`0x01` || `E` || `C`).
4. Compare the calculated `root` hash with the claimed `root` node hash.
5. The inclusion proof is valid if the hashes are equal.

Consistency Proofs

A consistency proof [92] can be used to verify if the append-only property of the Merkle tree is valid. The append-only property ensures that it is not possible to insert, modify, or delete a leaf or node in the tree retroactively without detection. Therefore, the consistency proof validates if a previously generated version of the tree is part of the current tree that may have been extended by new entries.

Assuming two Merkle trees, `Tree_Old` and `Tree_New` as shown in Figure 2.2, where `Tree_Old` is a previous version of `Tree_New`, a consistency proof provides an ordered list of node hashes in order to perform a verification whether the entries of `Tree_Old` are still equal to the corresponding entries in `Tree_New` or not. Given the root node hash of `Tree_Old`, a root node hash of `Tree_New`, the corresponding consistency proof [`E`,`C`,`D`,`I`], and the structure of both trees, the verification of that proof can be calculated as followed:

1. Calculate the resulting node hash: `X` = SHA-256(`0x01` || `E` || `C`).

[6] We consider a particular tree to be fully represented by its root hash, which can in turn be contained within an updated or larger tree with a different root hash. Within the scope of inclusion proofs we thus use the terms ‘tree’ and ‘root hash’ interchangeably wrt. the provided security guarantee.

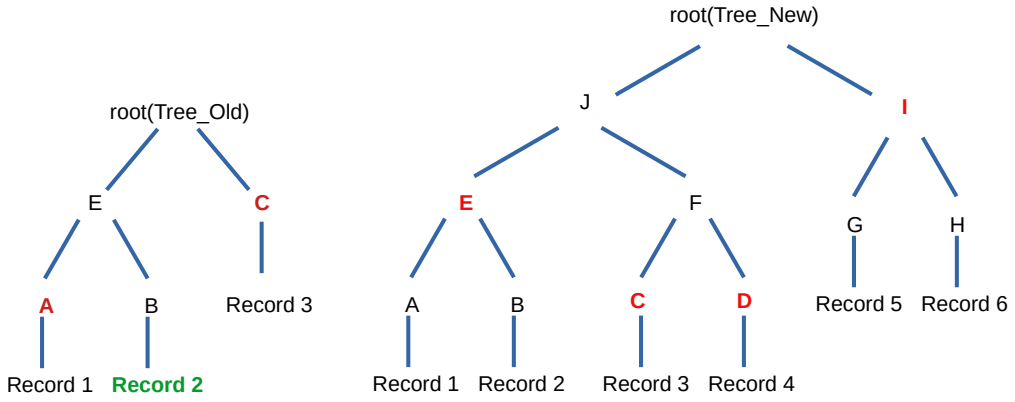


Figure 2.2: Merkle Tree proofs. Taken from our paper [98].

2. Verify that `X` is equal to root hash of `Tree_Old`.
3. Calculate `F` = SHA-256(`0x01` || `C` || `D`).
4. Calculate `J` = SHA-256(`0x01` || `E` || `F`).
5. Calculate root node hash of `Tree_New`: `Y` = SHA-256(`0x01` || `J` || `I`).
6. Compare the calculated root node hash `Y` with the claimed root node hash.
7. The consistency proof is valid if the hashes are equal.

2.6.1 Split-View Attack

A relevant attack on such transparency logs is called split-view attack [106, 120]. A log subject to such an attack would be able to present different log representations to its users while still maintaining the append-only property given by the Merkle tree. This means that all operations performed by a user on a specific log (e.g., inclusion and consistency proofs) seem valid, yet receiving different data than seen by other users. However, once a log carries out a split-view attack, it must consistently maintain different views for each subgroup of users since doing otherwise is detectable. A security evaluation and suggestions to counter this kind of attack are given in section 4.2.1.

2.6.2 Personality

The term *personality* is used by Google Trillian [27] and describes the application-specific interface to access a log server. The main responsibilities of a personality are defining and validating the application-specific data model, optionally providing access control, and in case the personality and the log is maintained by different parties, providing auditable information for external verifiers.

2.6.3 Monitor, Auditor and Witness

One of the main advantages of a transparency log is that it enables interested parties to detect misconduct up to even malicious behavior regarding certain

log entries. It is possible to set up monitors, auditors or witnesses that periodically verify the behavior of the log or notify subscribers in case of suspicious behavior. A monitor may store previous copies of the transparency log in order to verify the consistency between a new and previous versions. An auditor typically verifies the consistency of only a subset of the log by performing inclusion proofs. A witness [106, 149] on the other hand is an independent entity that observes one or more log systems to prevent split-view attacks. Log auditors can thus have more confidence that a log system is truly and globally consistent if multiple independent witnesses have a consensus about the specific state (checkpoint) of the log. The witness cosigns a checkpoint after verifying that an evolution of a previously signed checkpoint is consistent with it. In case the log or the witness are new, the witness uses the trust-on-first-use approach.

2.7 Tamarin

We formally model our security protocols using `Tamarin` [9]. Our security protocols include several parties that somehow need to interact with each other. Every party has a specific state depending on the flow of the protocol represented as *fact* in `Tamarin`. A *fact* can be represented as $F(t_1..t_n)$, where F is the name of the fact and t_i a value. A fact in `Tamarin` can either be linear – so it can be consumed only once – or persistent so that it can be consumed arbitrary times. We also utilize some built-in facts of `Tamarin`, like $Fr(x)$, $In(..)$, and $Out(..)$. $Fr(x)$ can be used to generate a fresh random value and the $In(..)$ and $Out(..)$ facts can be used to communicate via the adversary network. `Tamarin` also provides some built-in functions for hashing [20] and asymmetric encryption that we use within some facts to properly describe our models.

The individual protocol steps indicating a state transition are modeled by using multiset rewriting rules (MSR) with three parts. The first part on the left-hand side describes the required fact to apply the MSR. The middle part, also called *action fact* is used to label the transition so that we can later validate our security properties. The right-hand side represents the result of the state transition. Consider the following MSR example:

```
rule distribution:
  [ Receive(id,image) ]
    -- [ Deploy(id, image) ]->
  [ Out(image) ]
```

This MSR rule gets only part of the trace if the left-hand side applies. Thus, the system state must receive the *Receive(id,image)* from another MSR before. The middle part `-- [ Deploy(id, image) ]->` can be used to reference this state transition when verifying the security properties. Finally, the left-hand side state transitions to an output of the image to the adversary network by calling the $Out(image)$ fact.

To verify the actual protocol behavior including the security objectives, `Tamarin` uses *lemmas* and *action facts*. `Tamarin` tries to find a possible trace using first-order logic that contradicts the defined security property. In case

Tamarin does not find any trace we can assume that the defined security properties are satisfied. It is also possible to express time-dependent relations by considering specific timepoints. Based on action facts and respective timepoints it is possible to derive the traces using first-order logic. The verification performed by Tamarin tries to find a possible trace that contradicts the defined security property. In case Tamarin does not find any trace we can assume that the defined security properties are satisfied. The security property to be verified is described by using a *lemma* that consists of a name and a first-order formula. The following example verifies whether there is a trace where the value x of the action facts $F_A(x)$ and $F_B(x)$ is equivalent where $\#i = \#j$ at the same time.

```
lemma equivalent:
  exists-trace
    "Ex x #i #j. F_A(x)@i &
     F_B(x)@j & #i=#j"
```

2.8 Game Theory: Equilibria

The Nash equilibrium [84, 85, 114] is a well-known concept in game theory for analyzing outcomes in non-cooperative games where each interaction depends on the previous one. In an attacker-defender scenario, the Nash equilibrium can be used to analyze interactions in which both sides make simultaneous moves—either defend or attack. By contrast, the Stackelberg equilibrium designates one player as the leader, who seeks to anticipate the other player's move and choose an optimal tactic. We assume both sides can adapt their strategies based on the other player's actions. However, only the adversary is informed about the defender's security moves, simulating an adversary who can inspect most available security measures in advance. Because the defender does not learn the adversary's moves beforehand, the Stackelberg equilibrium is not suitable for the scope of this thesis (chapter 8).

Game setup. The game consists of two players (adversary and defender), $N = \{1, 2\}$. Each player has a set of AS_1, AS_2, which is common knowledge. The defense and attack strategies of both players are finite, pure strategies PS_1, PS_2, on which we assume, w.l.o.g., that the sets AS_i are infinite and contain all probability distributions. The sets AS_1, AS_2 also include randomized choice rules, i.e. strategies that vary over time, reflecting that in security there is no one-that-fixes-everything (e.g., changing a password, etc.) solution. Let $\ell_i : AS_i \times AS_{-i} \to \mathbb{R}$ for all $i \in N$ be the loss function of each side, where AS_{-i} represents the (joint) actions of both sides, except i (i.e., i's opponents). We model the game itself as a triple $\Gamma = (N, \{AS_i\}_{i\in N}, \{\ell_i\}_{i\in N})$. A *Nash equilibrium* in that particular game Γ is a joint strategy $(\mathbf{x}^*, \mathbf{y}^*) \in AS_1 \times AS_2$ such that $\ell_1(\mathbf{x}^*, \mathbf{y}^*) \leqslant \ell_1(\mathbf{x}, \mathbf{y}^*)$ for all $\mathbf{x} \in AS_1$, and $\ell_2(\mathbf{x}^*, \mathbf{y}^*) \leqslant \ell_2(\mathbf{x}^*, \mathbf{y})$ for all $\mathbf{y} \in AS_2$. The Nash equilibrium is reached when no side can improve its payoffs by changing its behavior, provided the other side does not change its behavior.

2.9 Information Security Governance and Frameworks: CVSS

The Common Vulnerability Scoring System (CVSS) [40], is a well-known framework to assess the severity of vulnerabilities. We utilize CVSS version 3.1 in this thesis to demonstrate the advantages when combining common scoring systems and game theory approaches. The CVSS version 3.1 consists of four metric groups—Base, Threat, Environmental, and Supplemental. In this thesis we focus on the Base group as it can be utilized to highlight the characteristics of a vulnerability uniquely and independently of the user environment. This metric is calculated by assessing the *Exploitability* and the *Impact* of a specific vulnerability. The exploitability of a vulnerability reflects how likely an adversary can exploit it and is calculated by considering sub metrics, such as the Attack Vector (AV), the Attack Complexity (AC), Privileges Required (PR), and whether it requires User Interaction (UI). The AV reflects the context of the exploitation, for example *Network* includes adversaries that can exploit the vulnerability remotely via the network stack. The AC can be set to either *Low* or *High* and takes into account whether the adversary needs to circumvent existing security controls (e.g., firewall). Depending on the required privileges, the PR metric can be set to *None*, *Low*, or *High*. Completing the Base metric group, the UI value can be set to *None* or *Required*. The second part of the calculation considers the impact (e.g., loss of confidentiality, integrity or availability).

Chapter 3

Threat Model

Our main security objective is to strengthen the end-to-end security level of software supply chains with a focus on the integrity and verifiability of each stage. We do not consider confidentiality or availability as security objectives in this thesis, assuming that the source code is not inherently confidential and that ensuring availability of relevant components in the supply chain (e.g., servers, communication channels, etc.) is the responsibility of the infrastructure provider.

3.1 Assumptions

We assume that any hardware-backed security component (e.g., TEE, StrongBox, TPM, etc.) is trustworthy. Below (section 3.1.1) we describe the trust assumption in Confidential Computing technology in more detail. Additionally, we assume any transparency log utilized by our systems is trustworthy and that there are sufficient independent witnesses deployed to prevent split-view attacks (section 2.6.1). From an adversary point of view, we consider well-resourced adversaries that are also capable of issuing a valid TLS certificate, for example.

3.1.1 Attacks Against Confidential Computing

In 2025, De Meulemeester, Wilke, et al. published the BadRAM attack which undermines integrity guarantees of AMD SEV-SNP [28]. Using the BadRAM attack, an adversary manipulates the Serial Presence Detect (SPD) chip which is used to communicate memory properties to the BIOS. In particular, an adversary can manipulate the ciphertext and the reverse map table data structure of AMD SEV-SNP hardware. Normally, this requires physical access.[1] However, the researchers identified certain DRAM vendors that did not lock the SPD chip and thus allowed adversaries to perform the attack remotely through the host system. In addition, BadRAM enables an adversary to present the attestation digest of one enclave image while executing another. However, it does not break the most important part of the attestation: the firmware measurements. As such, recipients of the attestation document can distrust any builds produced

[1] Such vulnerabilities relying on on-line DRAM integrity would have been outside AMD's scope: "These attacks are very complex and require a significant level of local access and resources to perform" [1]

by systems running firmware versions with known vulnerabilities, thus mitigating the impact of this attack on the concepts proposed within this thesis. The BadRAM vulnerability was disclosed to AMD (CVE-2024-21944) and was mitigated through a firmware update [2].

Two additional attacks compromising integrity are `wesee` [142] and `heckler` [143]. In `wesee` (CVE-2024-25742), Schlüter et al. break AMD SEV-SNP by interrupt injections allowing an adversary to compromise the confidentiality and integrity of a VM. The `heckler` (CVE-2024-25744, CVE-2024-25743) attack, similarly, uses adversary-controlled non-timer interrupts to break both the confidentiality and integrity guarantees of both AMD SEV-SNP and Intel TDX. Both attacks have been mitigated by kernel security patches [29, 125, 145].

Overall, the number of CVEs related to AMD SEV(-ES/SNP) and Intel TDX are low despite active research in this area. Misono et al. found that there are 17 known firmware bugs, 3 hardware vulnerabilities, and 4 design issues from 2019 to June 2024 for the Host-to-Guest attack type on AMD SEV-SNP [109]. All of them have been mitigated, e.g., through firmware updates. We therefore argue that, while not perfect, enclave technologies like AMD SEV-SNP match our basic security assumptions—if used with up-to-date firmware.

3.2 Adversary Model

The following list outlines relevant adversary models in scope of this thesis:

A1 **Physical adversary.** Adversary with physical access to the hardware, including storage, the infrastructure, or the user's device and capable of connecting the device to development machines (e.g., via ADB for Android devices). We assume that a physical adversary could also be an insider (A3), as our threat model does not distinguish between attacks that require physical access, regardless of whether the adversary is external or internal.

A2 **On-path adversary (OPA).** Adversary with access to the network infrastructure (e.g., via machine-in-the-middle [MitM] attack) and capability to modify code, attestation data, or other artifacts distributed to users (e.g., via HTTPS download).

A3 **Insider adversary.** An insider adversary with privileged access to the infrastructure, such as the hypervisor of the CSP or the hosting environment of the BSP (section 2.1.1), and potentially also to the signing key(s) of artifacts.

A4 **Remote adversary.** A remote adversary does not (yet) have privileged access to the victim, physical proximity to the device, or the communication channels. This type of adversary is mostly seen as a remote attacker from the Internet that is not part of the organization and mostly operates outside the infrastructure.

3.3 Threats

This section outlines threats addressed in this thesis. They are organized by supply chain stage: User, Distribution, Build, Source Code, and Developer.

Some chapters also introduce chapter-specific threats, which are described in their respective sections.

User. User-specific threats primarily target the integrity and authenticity of software artifacts (e.g., apps, system images, binaries, etc.) on the user's device.

T-U1 **Evil maid attack:** Adversaries with physical access (A1) can exploit physical access to a user's device to manipulate security-sensitive data. On mobile devices, for example, an adversary could flash arbitrary system images—given our assumption of an intentionally unlocked bootloader in chapter 5—without alerting the user about this malicious modification. This is possible as subsequent modifications to the image do not get verified once the bootloader is unlocked. This also involves tampering attempts where the bootloader is vulnerable similar to those experienced by Amnesty International journalists [7].

T-U2 **Different version for targeted attack:** A malicious distributor (A3) can selectively distribute tampered artifacts to only a subset of users. This threat also covers distributors who are forced to publish a modified version for specific users, pressured by nation state actors. Examples include Iran's Internet censorship regime [8] and a study revealing geographical differences in 596 mobile apps [89]. A related attack in this category involves an adversary (A3) performing a split-view attack (section 2.6.1) on the transparency log to provide different log results to a specific set of users. However, given our assumption of sufficiently independent witnesses (section 3.1), we do not further address this specific attack scenario in this thesis.

T-U3 **Undermine verification results:** An adversary (A1,A2,A3) can undermine verification data (e.g., attestation measurements) that users rely on to confirm an artifact's authenticity and integrity. This manipulation may happen at a particular point in the infrastructure (e.g., on the build server) or during transmission to the user.

Distribution. On the distribution side, threats include a malicious intent of the distributor, leakage of signing keys for distribution artifacts, and compromise of the distributor's infrastructure or distribution channel.

T-D1 **Signing key is leaked and used by an unauthorized party:** An unauthorized party (A3,A4) can use a leaked signing key to distribute malicious updates of an artifact (e.g., an Android app). If the holder of the signing key is unaware of the leak, they may not recognize that the key is being misused for potentially malicious purposes. This threat persists even in case the holder of the signing key monitors certain distribution channels, as these could also be untrustworthy or even malicious.

T-D2 **Unauthorized usage of the signing key due to compulsory outsourcing:** For example, the current Google Play policies enforce the developer to store the signing key on Google's infrastructure so that the developer is no longer in control of the signing key (section 2.2.2), and comparable app distributors have similar policies. These policy demands full trust in the external key

storage and requires that no unauthorized entity can access these security-relevant signing key(s). In that particular case, neither the developer nor the user can verify if the signing key has been compromised.

T-D3 **Deliberate use of the signing key:** The keyholder (e.g., an external cloud service provider (CSP)), may be forced to sign the corresponding artifact update by the respective judiciary or due to economic interests. The developer may not have the possibility to detect that the outsourced key has been misused. Reports of government interventions in the mobile world underscore this threat; in 2021, the New York Times [118] reported the removal of tens of thousands of apps from Chinese app stores.

T-D4 **Compromise of the distribution channel:** An OPA (A2) with access to the distribution channel can tamper with an artifact that is transferred to the user. If the artifact is digitally signed and the user (or the application of the user) properly verifies the signature, unauthorized modifications are detectable. However, not all artifacts are digitally signed, and even when they are, verification may be skipped—for example, when flashing an image on a mobile device with an unlocked bootloader (chapter 5).

Build. The build stage is particularly critical because it often involves many untrusted external service providers (e.g., CSP, BSP).

T-B1 **Compromise of the build server:** An adversary (A1, A3) may compromise the build server to modify the final artifact. This threat includes unauthorized modifications within the build scope, such as compromising the source code during the build, build assets (e.g., Makefiles), or even components of the infrastructure itself (e.g., the hypervisor or operating system).

T-B2 **Cross-tenant threats:** Any adversary (A2,A4) operating on shared infrastructure may compromise the underlying host (e.g., the hypervisor), thereby enabling compromise of subsequent or concurrent builds. A related threat is that the build server often executes developer-supplied code; since developers are typically permitted to run arbitrary build steps, this code may be malicious.

T-B3 **Compromise of build assets during transmission:** An adversary (A2) with access to the build infrastructure's communication channels may compromise build assets—such as the underlying source code, external dependencies, tools, or configurations—used by the build process, resulting in malicious modifications to the final artifact. This threat also covers indirect communication channels, such as side-loading libraries to the build infrastructure.

T-B4 **Compromise of the hardware layer:** An adversary (A1) with physical access may perform classical hardware attacks against the build infrastructure, including intercepting RAM access, executing malicious code on CPU cores, or conducting side-channel attacks. However, within the scope of this thesis, we do not assume the adversary can successfully attack hardware-backed security mechanisms such as TEEs (section 3.1). This threat aligns with the threat model and challenges commonly addressed by *Confidential Computing* technologies.

Source. The source stage encompasses everything associated with the source code and the repository in which it is managed.

T-S1 **Implant a backdoor:** An adversary (A3) may implant a backdoor in the repository—through a bugdoor[2] or within other repository assets (e.g., binary files). For such an attack to succeed, the adversary typically requires elevated privileges to make the necessary changes to the repository. To receive these permissions, the adversary may perform a social engineering attack. A real-world example of such a case is the attack on XZ Utils [99], which we discuss in chapter 6. This threat is an orthogonal supply chain concern compared with directly compromising the build infrastructure (T-B4) or the build process itself (T-B1).

T-S2 **Spoofing the repository:** An adversary (A4) can clone an open-source project, introduce malicious changes, and attempt to trick victims into believing it is the original repository, as demonstrated in recent attacks [68]. A related tactic is typo-squatting, used, for example, to spoof dependencies in package managers [116, 153].

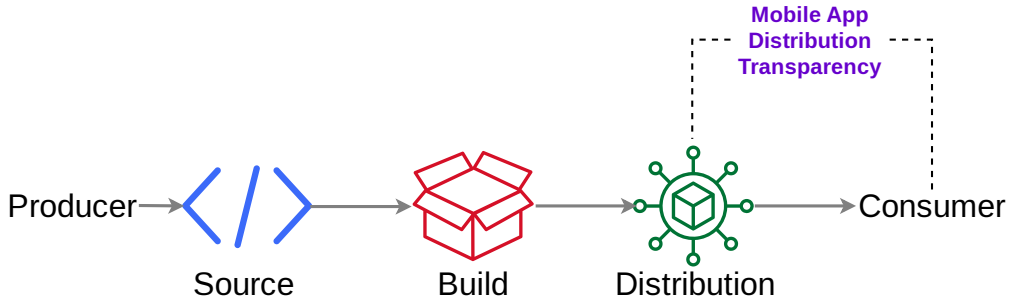
Development. In the development stage, we focus on developers—and, for device manufacturers, OEMs—who may be compromised or act maliciously.

T-P1 **Insider compromise:** An adversary (A3) with access to the developer's infrastructure and potentially also with access to the signing key could either manipulate the source code and sign the resulting artifact or just replace the respective artifact and sign it directly. Compared to distribution threats, the user cannot detect any unauthorized modification of an image because it is properly signed with the official signing key of the developer. This threat also applies when verifying images on devices with a locked bootloader. Since the tampered image would be signed properly, the current verification controls would not detect unauthorized modification even if the bootloader gets locked again and even without using a custom root of trust.

[2] The term "bugdoor" describes a backdoor—typically implanted in source code—that could plausibly be mistaken for a genuine bug.

Chapter 4

Software Distribution: Signatures, Transparency, and Provenance



Paper Reference. This chapter is based on the paper "Mobile App Distribution Transparency (MADT): Design and Evaluation of a System to Mitigate Necessary Trust in Mobile App Distribution Systems" [98].

In particular, this chapter addresses threats at the distribution stage, including attacks that replace legitimate artifacts with modified, potentially malicious variants (T-U2,T-D4). Adversaries commonly compromise this stage by targeting critical trust anchors—such as digital signatures [70]—which are essential to the distribution process and its security objectives. Both users and developers are often required to trust them implicitly, even though there is usually no way to verify whether tampering has occurred.

We primarily focus on mobile app distribution ecosystems (e.g., Google Play and F-Droid), which rely on such digital signatures as critical trust anchor; however, the underlying principles also apply to other distribution systems that utilize digital signatures to establish trust primitives. Although digital signatures are leveraged to ensure authenticity and integrity, current (mobile app) distribution systems are still vulnerable to specific threats (see threat model chapter 3), including leaked signing keys, malicious distributors, insider attacks, and the selective distribution of different app versions to only a specific subset of users. Recent app store policy changes like Google Play Signing (and other similar OEM and free app stores like F-Droid) are a practically relevant case of intentional key sharing: such distribution systems take over key handling and create app signatures themselves, breaking up the previous end-to-end verifiable trust from developer to end-user device. To address these threats, this chapter introduces a transparency-log-based approach that enhances verifiability and enables end users and app developers

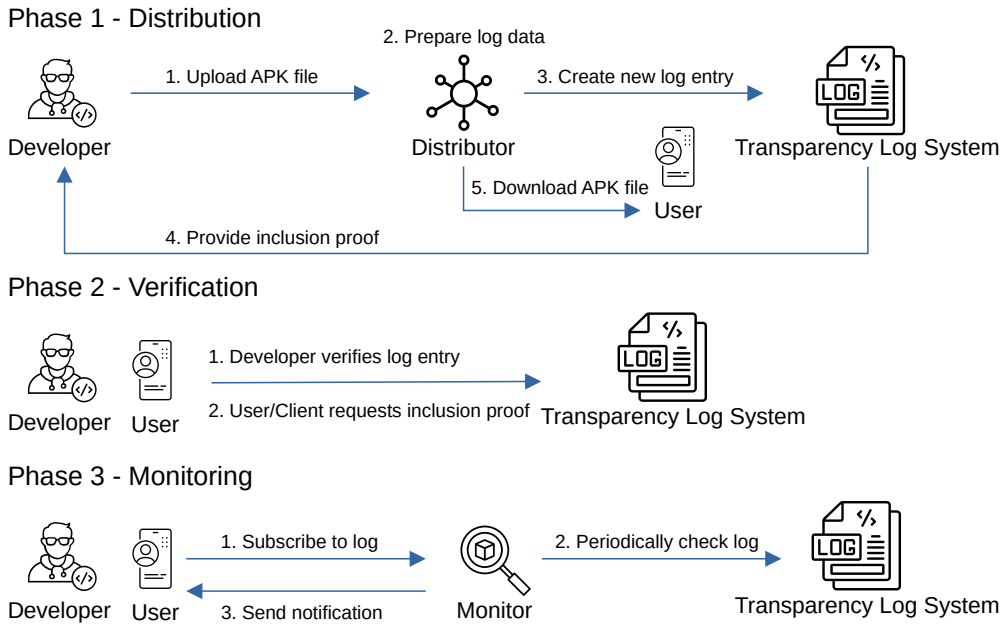


Figure 4.1: Individual protocol flows in Mobile App Distribution Systems. Taken from our paper [98].

to detect signature-related attacks. We analyze the relevant security considerations with regard to our threat model (chapter 3) as well as the security implications in the case where an adversary is able to compromise our proposed system. In particular, this chapter addresses the following threats: T-D1,T-D2,T-D3,T-D4,T-U2. We evaluate our concept to demonstrate its practicability, feasibility, and performance capabilities by developing an open-source prototype.

4.1 Architecture

This section introduces the components used to design our underlying concept. To evaluate and verify the viability of the proposed system, we implement a proof-of-concept prototype. Several parts of the implementation utilize (adapted) open-source components, including F-Droid for distributing Android apps and Google Trillian for the transparency log backend. The first subsection details the individual phases with respect to the previously defined stakeholders (section 2.1.1). The second subsection takes a more software-centric view and lists the system components.

4.1.1 Phases

The proposed system comprises three main phases, defined by the stakeholder's intended use: the distribution, verification, and monitoring phase. Figure 4.1 provides an overview of these phases, the relevant stakeholders, and their tasks.

Distribution phase. The distribution phase starts once the developer has completed the implementation of the app. At this point, the *developer* intents to dis-

tribute the app to *users* via the *distributor's* infrastructure. First, the *developer* either decides to sign the app directly or to delegate signing to the *distributor*. Then, the *developer* uploads the binary or its source code to the store provided by the *distributor*.

At this point, our system extends the workflow by extracting relevant app metadata to be published and creating a corresponding log entry in the dedicated transparency log system. This step is performed by the *distributor*. Once the log entry has been successfully appended, the *distributor* can release the app to its *users*.

Verification phase. Once the app is available through the distributor's channel, client-side verification can be conducted. Verification may be performed by multiple entities: the *developer*, the *user*, and potentially third-party *witnesses*. The developer may verify that the app was properly logged by requesting an inclusion proof from the transparency log. If the inclusion proof verifies, the developer can be confident that the distributor correctly logged the uploaded APK.

On the user side, verification is performed automatically by the distribution client (e.g., F-Droid). The client downloads the requested app, computes the expected logging information, and requests the corresponding inclusion proof from the log by approaching the personality. If the expected and logged information match, no warning is shown to the user. If they do not match, the app can still be installed, but the user is warned. Because our transparency log is publicly accessible, users can also verify the log entry manually, without trusting the distributor's client.

Monitoring phase. The monitoring phase begins after the app has been published and verified by the developer. The developer can subscribe to notifications from a monitoring instance that watches the transparency log for new entries matching the application ID and the version that the developer is interested in (e.g., `com.example.sampleapp:v1.0`). When the monitor detects a new log entry with that namespace, it notifies all subscribers. If the developer has not published a new update, this indicates that someone else is attempting to do so.

4.1.2 System Components

This section describes the system components from a technical perspective, including the prototype implementation. Figure 4.2 illustrates the three main components and their interactions.

Build Server

The build server part includes the components required to build, sign, and deploy the app. The individual tasks of such a build server are to fetch and build the source code, to verify its reproducibility and to sign and publish the APK to the distribution server. Our prototype implementation is built around F-Droid. The F-Droid build server provides the build environment, a dedicated F-Droid repository, and a signing server as illustrated in Figure 4.3. No changes were

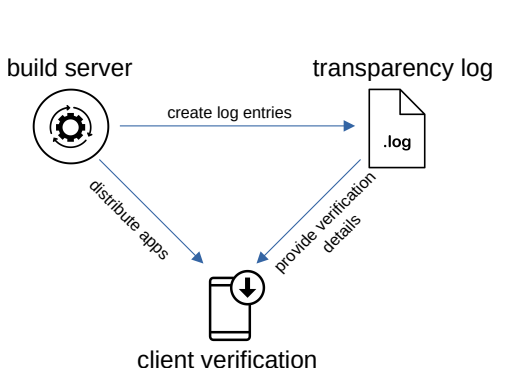


Figure 4.2: High level illustration of MADT components. Taken from our paper [98].

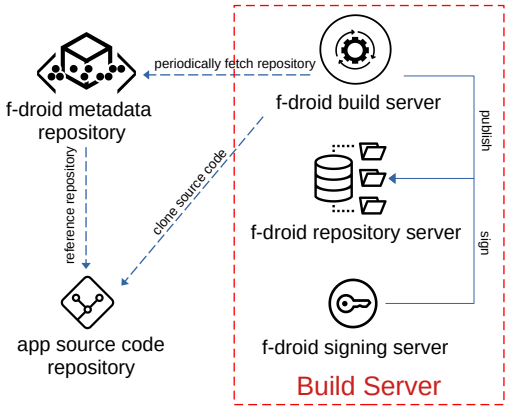


Figure 4.3: Architecture of the Build Server. Taken from our paper [98].

required to the F-Droid repository or signing component. However, for production use, we recommend requesting an inclusion proof before making the APK available to users.

Prototype implementation. The most significant changes in our prototype implementation relate to the build server component itself. During publishing, the build server signs the APK and publishes it to the repository. At this point, our prototype implementation adds additional steps to the workflow before the signed APK is finally deployed to the remote web server, where it becomes available to all users:

1. Extract relevant APK metadata.
2. Select the proper tree ID for the specific repository.
3. Create a JSON object compliant to the personality data.
4. Request an authentication token from the personality.
5. Send tree ID and JSON object to the personality to create new log entry.

Transparency Log

The transparency log implementation of our prototype consists of three components as described below. More background information on transparency logs are given in section 2.6.

Log server. The incorporation of a transparency log mitigates threat T-U2 by forcing the distributor to create a log entry and enable the auditor to verify the corresponding inclusion proof. The log server implementation is based on Google Trillian and required no modifications, as it is designed to be application-agnostic.

Database. The database is used to persist the Merkle tree. Our implementation uses a MySQL database.

Personality. The *personality* is the application-specific interface in front of the transparency log (section 2.6.2). It defines and validates the data structure used to store leaf content in the transparency log and exposes an interface for clients or users to interact with the log. In our prototype, creating new log entries is restricted; accordingly, certain endpoints of the personality are accessible only with proper authentication and authorization.

Prototype implementation. Our prototype includes a dedicated personality, implemented as a REST service using the .NET core framework. Using our build pipeline, we build a Docker image that contains a configurable personality instance. The Google Trillian implementation of the log-server implementation provides the required **.proto* files to interact with the log via gRPC. The personality defines the data structure stored in the Merkle tree and performs any necessary data validation. It also serializes the C# objects into byte array that is ultimately stored in the transparency log. An essential implementation detail is the use of the correct hashing algorithm and element-type prefixes (e.g., `0x00` for leaf elements and `0x01` for node elements), cf. RFC 9162 [92]. To prevent unauthorized write access to the log, we introduced two roles and a token-based authentication scheme. We define an admin role responsible for managing trees (e.g., creating or deleting trees) and a build-server role authorized to create new log entries. Our docker image is parameterized to include predefined credentials for both roles. For example, when the F-Droid build server wants to create new log entries, it must first present valid credentials to the personality. If the credentials are valid, the personality issues an authentication token that can be used to create new entries in the log.

Data Structure

The data structure of the records stored in the transparency log includes the following fields, which uniquely identify the package and enable integrity verification:

- *applicationId:* The unique package name of the APK.
- *version:* The version number of the app release.
- *apkHash:* A cryptographic hash of the APK to verify its integrity.

Client Verification

This component is responsible for verifying that apps are downloaded from a distributor were properly logged. This involves several steps as listed below:

1. The client downloads the APK, but does not start the installation.
2. The verification library gets relevant metadata (application ID, version, and hash value) of the APK file.
3. An inclusion proof is requested by sending a specifically crafted data object including the metadata to the personality.
4. In case there is an inclusion proof available, the verification library calculates the expected root hash locally.

5. The locally calculated root hash is compared with the claimed root hash of the log server.
6. If the root hashes match—thereby validating the inclusion proof—the client proceeds to install the app without further notice.
7. If the inclusion proof is invalid, the user is notified but can still proceed with the installation.

Prototype implementation. We implemented a dedicated Android library to perform the end-to-end verification of distributed APKs, confirming whether they are properly logged. Its core functionality currently validates inclusion proofs provided by the personality. Beyond verification, the prototype implementation handles all other interactions with the personality (e.g., requesting available tree IDs). The library is written in Java and is publicly available (section 4.4). By providing this as a standalone Android library, interested parties can easily use it within separate apps or integrate it into official clients (e.g., the F-Droid client).

4.2 Evaluation

4.2.1 Security Evaluation

This section evaluates our system design to determine whether the related threats, listed in chapter 3, are effectively mitigated. It also analyzes the security implications if an adversary compromises one or more of the newly introduced components.

Threat Mitigation

Our system design makes any distribution attempt through (or by) a distributor transparent and thus verifiable. In particular, the client verifies the log entry before installing an app, and therefore it is not possible to distribute an app without creating a new entry in the append-only, tamper-proofed logging system. As this entry includes the package string, the version, and the hash of the APK, any interested party can verify if this aligns with the corresponding log entry and that it is the same APK version distributed to everyone else (T-U2). A developer or an app distributor monitoring the log will be notified as soon as a relevant entry is created, enabling the detection of unauthorized distribution attempts (T-D1,T-D2.T-D3). Our system design also supports independent, verifiable monitoring instances and therefore does not rely on the trustworthiness of a distributor. This helps prevent falsified or missing information (e.g., suppression of publication attempts) compared to monitoring specific distribution channels, directly (T-D1).

Security Implications

An adversary who compromises one or more of the newly introduced components could significantly impact the authenticity and integrity of the mobile

distribution system. Consequently, we evaluate the security of our approach with these components in scope [1]. The following paragraphs present our evaluation results.

Malicious distributor bypasses the logging system. If a compromised distributor attempts to bypass or manipulate the log entry—e.g., by avoiding the creation of a new log entry to distribute a tampered APK—the client will detect the absence of a corresponding log entry for that APK. It will then notify the user of the security incident, who can decide how to proceed.

Malicious distributor creates a manipulated log entry. Because clients detect missing log entries, a malicious distributor might instead create a manipulated log entry. To make it appear valid, the distributor would compute the hash of the tampered APK and write the that hash to the log. A client verifying the APK would compute its hash and check the corresponding log entry; verification would succeed because the entry exists and the hashes match. However, such manipulation can be detected by monitors. For example, a developer may have registered the APK's namespace with a monitor observing the log. When a new entry appears, the developer (as subscriber) is notified and can compare the known-good hash of the original release with the hash in the log entry, revealing the manipulation attempt. Further response steps are out of scope for this thesis.

Malicious client bypasses the log verification. An adversary might trick a victim into installing a tampered distribution client that bypasses the logging verification, enabling the distribution of malicious APKs through the channel. Beyond the fact that an adversary who is able to trick a victim into installing a malicious client could also install other malicious APKs the same way, two countermeasures apply: (1) log the client app itself in the transparency log, so users can manually verify the corresponding inclusion proof, and (2) provide a dedicated app that performs the required calculations and communication with the personality.

Malicious log server. A log operator might attempt to tamper with a log entry while keeping the same root hash—i.e., modify a single leaf but preserve the published Merkle root hash so that proofs for other entries remain valid. This can be mitigated by performing full audits of the Merkle tree. In a full audit, the root hash is recomputed from all available leafs and compared to the claimed root hash. If the recomputed root hash does not match, the inconsistency reveals tampering. From a component perspective, such full audits can be carried out by independent monitors, for example.

[1] Note that global passive adversaries may learn which apps are installed by clients by monitoring transmitted inclusion proofs, leaf log entries, and/or the embedded APK metadata. However, as there are many other ways to learn the same information under our threat model, we consider this as out of scope and not a reason for keeping such data confidential.

Unauthorized write access to the log server. The transparency log should be publicly readable by design, so any interested party can verify log entries. Write access, however, must be carefully controlled: although permitting arbitrary parties to write does not impede verification, the append-only property prevents removal of unwanted data. To keep the log clean and as small as possible, we therefore recommend restricting write permissions to authenticated distribution systems.

Split-view attack by a malicious log server. A split-view attack (section 2.6.1) can be mitigated by utilizing witnesses. Because the log server is application-agnostic, any compatible witness system can be used. To benefit from witness consensus, however, clients must verify witness statements in addition to inclusion proofs—functionality that is not yet implemented in our prototype.

Orthogonally, split-view attacks can be mitigated by querying the personality and log server via anonymity networks (e.g., Tor [155]) to make the delivery of consistent split views to different clients impractical.

Malicious personality. A personality is not necessarily hosted in the same trust zone or is operated by the same operator as the log server. Consequently, an external auditor may also want to audit the personality to verify its behavior. The personality can substantiate correct behavior by independently monitoring the log server and persisting Signed Tree Heads (STHs). If an STH changes retroactively, the personality can detect the inconsistency.

4.2.2 Performance Evaluation

To evaluate the performance of our transparency log system, we used metadata of all publicly available APKs from the official F-Droid repository and created corresponding log entries. We implemented a Python script that (1) fetches the current F-Droid index file[2], (2) parses it to prepare the proper log format required by our personality, (3) requests an access token for the build server user, and (4) issues POST requests to create the new log entries.

For our performance measurements, we used a computer with an Intel i7-1185G7 @ 3.00 GHz CPU and 32 GiB of RAM to fetch the current F-Droid index, prepare the log entries, and issue the POST requests to our transparency log. Our transparency log backend—including the personality and the log itself—is deployed on a virtual machine with 2 cores and 2 GiB RAM (Host CPU: Intel E5-2620 v3 @ 2.40 GHz). For client side end-to-end verification, we used an Android emulator (API level 31) configured with 1,536 MiB RAM.

The official F-Droid repository contained 9,705 APK at the time of evaluation our system. Creating our log took under 47 minutes—less than 30 ms on average per APK— and the log database consumed 8 MiB of disk space. Inclusion proofs consist of 14 hashes (825 B) for the first leaf and 7 hashes (510 B) for the last leaf. Consistency proofs ranged from 14 hashes (821 B) to 8 hashes (533 B). An end-to-end verification for the first leaf—requiring the maximum number of intermediate node hashes—took 296 ms using our Android library.

[2]https://f-droid.org/repo/index-v2.json (accessed: 2023-02-07)

4.3 Related Work

4.3.1 Certificate Transparency

Certificate Transparency (CT) [56] is a process that is part of the web's public key infrastructure. Its primary goal is to detect unauthorized or even maliciously issued TLS certificates for websites by making issuance events transparent and verifiable. Whenever a certificate authority (CA) issues a new certificate, a new entry gets recorded in one of the approved transparency logs. Independent monitors can watch these public logs for suspicious activity, and any interested party can operate such a monitor. Browsers act as auditors by verifying that presented certificates are included in the corresponding transparency log.

Difference. Both CT and our proposed system are built on Merkle trees. Therefore, we can use the same underlying Google Trillian implementation to handle the tree structure. The leaf data, however, is application-specific: CT stores TLS certificate artifacts whereas our system stores mobile app metadata. The most relevant difference lies in end-to-end verification. In the CT ecosystem, the browser is responsible for verifying if the certificate is properly logged by checking the signed certificate timestamp (SCT), typically embedded as an X.509v3 certificate extension [111]. In contrast, our approach requires no SCT. Instead during end-to-end verification, the client deterministically constructs the expected log entry of the specific APK and directly checks the transparency log for a corresponding inclusion, avoiding reliance on digital signatures at this stage.

4.3.2 Binary Transparency in F-Droid

F-Droid has integrated a module [148] that logs the signed app index metadata files in append-only storage. These files contain information about the available APKs in a given F-Droid repository, so any update or change also requires a change on the related file. To meet the append-only requirement, F-Droid uses a git repository that it claims is tamper-proof. This enables interested parties to verify if a specific binary was published by the expected publishing entity, since only an authorized party is authorized to push to the respective git repository.

Difference. A git repository is a content-addressable filesystem [17] based on a Merkle tree—the same data structure we use in our approach and prototype implementation. When a file is added or updated, git computes the SHA-1 hash over its content (called a *blob* object) and stores this information in an internal object database. Filenames are not part of blobs; instead, they are recorded in *tree* objects, which can contain other tree objects. This approach is analogous to the Unix file system, where a blob object would correspond to the data associated with a file object, while a tree corresponds to the entries found in a directory object.

F-Droid's logging approach stores the app index metadata file in a dedicated git repository. This approach is not scalable as metadata of all the available apps in

the F-Droid ecosystem is stored in one single file. Moreover, the verification-relevant data is not managed directly by the tree; it is embedded in a file, so verification cannot leverage tree-based optimizations like efficient inclusion proofs. As a result, a client must download the entire file to verify whether a specific value is present. The file is also larger than necessary for verification, as it includes information that is not relevant for the verification task at all (e.g., license information). While such an approach may work reasonably well for F-Droid, this approach would not scale to the distribution scale seen in larger markets. By contrast, our approach operates directly on Merkle trees and provides efficient inclusion and consistency proofs across snapshots, minimizing communication and computation.

4.3.3 Blockchain

Blockchains are built around a distributed public ledger that, like our approach, offers similar properties, including an append-only data structure, tamper resistance, and transparent verification. Each block includes a hash of the previous block, a timestamp, and transaction data. The challenges addressed by blockchains are orthogonal with respect to authentication, integrity, and non-repudiation [32] that can be addressed by using digital signatures. A digital signature proves that data has not been tampered with afterward and that it is signed by an entity that possesses the respective signing key. However, attributes such as the time of signing still require trust in the signing party—critical in financial or legal contexts. To reduce this trust dependency, blockchains employ a distributed trust mechanism: multiple parties maintain and replicate the transaction history, enabling independent verification that it has not been tampered with.

Difference. From a security perspective, blockchains also provide tamper protection. However, their verification procedure is not scalable[3]. For a full end-to-end verification, a blockchain-based approach requires clients to download the whole chain whereas the transparency log approach only requires the hashes of the audit path ($\log(n)$) and ideally checkpoints signed by independent witnesses.

4.4 Availability

Our prototype implementation consists of the following component repositories and is publicly available.

- **F-Droid server:** The relevant source code segments have been extracted from the fork of the official F-Droid server code. Source Code: `https://github.com/linsm/fdroidserver_transparencyextension`

[3] In terms of efficiency comparison, we are not even assuming proof-of-work consensus algorithms, but permissioned ledgers comparable to the authentication of submitters performed by the personality.

- **Personality:** The personality project contains the code for the application-specific interface between the client library, the F-Droid server, and the Google Trillian logging infrastructure. Source Code: `https://github.com/linsm/mobiletransparency-personality`
- **Android library:** The Android library project contains the code for the end-to-end verification of APK files. Source Code: `https://github.com/linsm/mobiletransparency-androidlibrary`
- **Evaluation setup:** Contains the test script, configuration file and reference data of our performance evaluation. Source Code: `https://github.com/linsm/mobiletransparency-data`

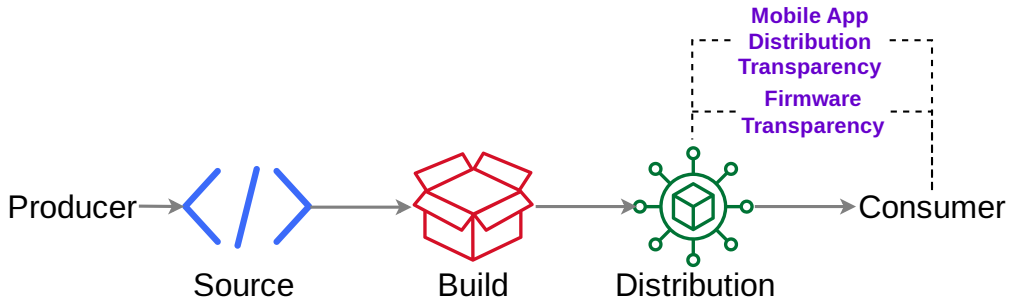
4.5 Summary

Current mobile app distribution systems rely on digital signatures to ensure app integrity and authenticity. However, as listed in our threat model (chapter 3), there are realistic threats which may compromise digital signatures. For example, it is currently impossible to detect unauthorized usage of signing keys. A more general perspective on this kind of problem is how to compliment or enhance the trust placed on digital signatures in mobile app distribution systems.

We proposed a new concept that mitigates these threats by making every distribution attempt transparent and thus verifiable. We also built a prototype to demonstrate practicability and feasibility, and we provided a detailed security analysis of the new components alongside performance measurements. Our evaluation showed that an adversary would need to compromise all involved system components and security controls to successfully distribute a malicious APK file without detection. Although our system targets mobile app distribution systems, the approach generalizes to other domains where digital signatures are used and their trustworthiness may be in question.

Chapter 5

Bootstrapping Trust: Strengthening Security Guarantees Against Bootloader-Targeting Attacks



Paper Reference. This chapter is based on the paper "Firmware Transparency: Strengthening Security Guarantees Against Bootloader-Targeting Attacks" [100].

In the previous chapter, we explore distribution and user verification at the application layer. However, the application layer is only as trustworthy as the underlying (operating) system. If an adversary is able to compromise the system layer, the security mechanisms implemented on the application layer can be circumvented. A critical area where this trust assumption plays a significant role is the booting process of (mobile) devices, which relies on a set of components commonly referred to as the "chain of trust". Most manufacturers implement a range of security controls, such as tamper-resistant hardware or digital signatures, to protect the trust chain and ensure the overall trustworthiness of the device. Digital signature keys used to verify the chain of trust are typically set by the device manufacturer, with most of them prohibiting users to set a custom root of trust. Currently, Google is the only major manufacturer that allows users to set a custom root of trust on Pixel devices. However, there are legitimate use cases—like using an alternative operating system—where the chain of trust would be broken without setting the proper custom root of trust. We advocate for long-term updates due to sustainability aspects and freedom of choice, a fundamental principle we aim to support with our approach. In this chapter, we address compromised security of a broken chain of trust resulting from the necessity of completely disabling key verification on devices that lack support of a custom root of trust. We assume that the hardware layer leverages security guarantees of the "System on a Chip"

(SoC) and thus it can be trusted. Accordingly, the security guarantees provided by the hardware layer are considered out of scope, as this area is explored in other research (see section 3.1.1 for details about attacks against TEEs, for example). To mitigate these lost security guarantees, we propose a new security mechanism, called *Firmware Transparency*. Furthermore, we demonstrate that *Firmware Transparency* enhances verifiability on devices with a locked bootloader. We prove the security properties by formally model and verify our proposed protocol and implement an open-source prototype, demonstrating its practicability and feasibility. Moreover, the proposed concept demonstrates its effectiveness by detecting a malicious modification resulting from the exploitation of a vulnerability that is not detected by current security controls. We choose Android because of its open source basis and because it is used by multiple different manufacturers. Nevertheless, our concept could also be used on other embedded devices.

5.1 Where the Chain of Trust Breaks

Considering the properties of the chain of trust mentioned in section 2.4.2, we assume the ABOOT bootloader to be trustworthy, given that this part of the chain is flashed by the device manufacturer and verified by AVB. We base our assumption additionally on the fact that the verification process of AVB is reliable as long as the root of trust is secure and cannot be tampered. A common mitigation technique used to secure the root of trust is to store the associated private key on a secure element in hardware — and to apply best practices of secure software development to mitigate the risk of security critical bugs in these fundamental components. This makes a tampering attempt unlikely, as an adversary would need to either compromise the HSM or discover a vulnerability on bootloader level (which, while not impossible [69], is rare in practice because of the limited amount of code and slower change rate). Therefore, we put our focus on the last stage of the bootloader process, where the ABOOT part of the bootloader verifies relevant partitions, like the boot partition. Furthermore, this stage signifies the first encounter with potentially user-specific code as it initializes the operating system, which could be an alternative one rather than the stock operating system. A possible strategy used to ensure verifiability of the chain of trust in this stage is to allow the user to set a custom root of trust. AVB uses this key to continue the verification process by loading related hash values from the *vbmeta* struct, verifying the authenticity with the custom root of trust, and comparing the loaded hash values with the actual values of the respective partitions (the *YELLOW* state as described in section 2.4.2).

This is the point where our research addresses a critical challenge: enabling device owners to flash an arbitrary operating system without relying on the manufacturer, who may restrict the ability to set a custom root of trust for verification. As a result, we cannot assume a secure device state. In the following, we introduce additional bootloader-targeting threats beside the threats mentioned in chapter 3,our proposed architecture, including the involved system components, data structures, and attestation capabilities we use to compensate the lost security guarantees by flashing a custom operating system on a device without the possibility to use a custom root of trust and the bootloader being in unlocked state (*ORANGE*).

5.2 Supplemental Stakeholders and Threats

This section introduces new stakeholders that are relevant for this chapter and expands the threat model to cover specific bootloader-targeting threats.

5.2.1 Stakeholders

We introduce two additional roles: the *auditee* and the *auditor*. The primary target group consists of users who try to make use of long term updates due to sustainability reasons, value their freedom of choice, and require new possibilities of end-to-end verification of the custom image on mobile device with an unlocked bootloader. We refer to this specific type of device as *auditee*. On the other hand, we need an independent party that is responsible for verifying the integrity of the booting custom image according to the information provided by the auditee. We refer to this party as *auditor*.

5.2.2 Expanded Threat Model

In this chapter, we focus on devices with an unlocked bootloader, enabling users to flash a customer image without compromising security on their device. Accordingly, the main objective of our approach, called *Firmware Transparency*, is to mitigate lost security guarantees on devices with an unlocked bootloader by making tampering attempts transparent and thus verifiable. We do not consider confidentiality nor availability as security objectives of *Firmware Transparency*, as our focus is primarily on the device's integrity and authenticity.

Primarily, we address threats including (1) evil maid attacks (T-U1), where an adversary exploits physical access to a user's device, (2) on-path compromise (T-B3), in which the image is tampered before it gets flashed to the device, (3) insider attacks (T-P1) targeting OEM infrastructure, and (4) user-targeted attacks (T-U2), where different versions are distributed only to a subset of users.

Beyond these general supply chain threats, we further extend our threat model to cover the following:

T-F1 **Stealing knowledge factor:** An adversary (A1) exploiting physical access, such as an evil maid attack on a device with an unlocked bootloader, might gain access to the knowledge factor provided by the user if no verification possibility is in place beforehand. A practical attack vector involves replacing the current operating system with a malicious one that intercepts the initial authentication process, specifically capturing the knowledge factor. The missing verification controls due to the unlocked bootloader would not detect that an "unknown" operating system is booted.

T-F2 **Relay attack:** An adversary (A1,A4) may relay verification request to a legitimate device and response with these results to the verifier. The verifier requests some form of proof of the user's mobile device, which may already be under the control of the adversary. If the adversary can relay the verification request to a legitimate device to obtain a valid proof, it may appear as an authentic and legitimate audit result.

T-F3 **Replay attack:** An adversary (A1) may replay an already created legitimate verification result to a verified party. This threat addresses the freshness property of a potential verification result. If it is possible to replay older verification results a tampered device can easily provide them to the verifying party.

5.3 Architecture

Our architecture comprises five primary components as illustrated in Figure 5.1: (1) the developer publishing an image, (2) the device of the user with that custom image, (3) the distributor, (4) one of more auditors, and (5) the backend for the transparency log infrastructure. As the auditee could already be malicious, the auditor also has to verify the authenticity and integrity of the information given by the auditee. To prevent relay attacks (T-F2), we incorporate the *Trust on First Use* (TOFU) model. In particular, it requires the auditee to generate a persistent key that is stored in the hardware-backed keystore, enabling the auditor to use a strong device binding verification on subsequent audits. As this persistent key is stored in the hardware-backed keystore, the operating system does not have access to it even if it had been compromised. By utilizing a transparency log we mitigate threat T-U2 as the available version names including their hashes can be publicly verified. We assume that the transparency log is protected against split-view attacks by integrating a sufficient number of independent witnesses (section 3.1).

5.3.1 Design and Implementation Details

As the bootloader is unlocked, and we do not have a custom root of trust (ORANGE state), AVB will have several verification errors, but continues to boot the custom image because the device is unlocked. Once the device is booted, the user can verify the authenticity and integrity of the operating system using our auditor app. This is possible even on top of lock screen and before first unlock, ensuring protection of the user's knowledge factor and the user data partition. Our prototype implementation includes an application-specific API, called *personality* that is mainly responsible for interacting with the Google Trillian backend, and the auditor app that can be started either in *auditee* or *auditor* mode.

Auditor. The auditor app runs on both the auditee and the auditor side. On the auditee side, the auditor app is responsible for gathering relevant information and making it available to the auditor for the verification task. The main information that is required is the vbmeta digest. Android includes the vbmeta digest in the hardware key attestation and thus, the auditee side of the app can use Android key attestation to retrieve a signed copy of the vbmeta digest. The communication between the auditee and the auditor is established by scanning a QR code. The auditee generates a QR code including the attestation details necessary for the verification. The auditor scans the QR code and requests an inclusion proof to verify whether the attestation information of the image has been properly logged. If the verification is successful so that it is proven that the calculated root node hash based on the inclusion proof is equal to the expected

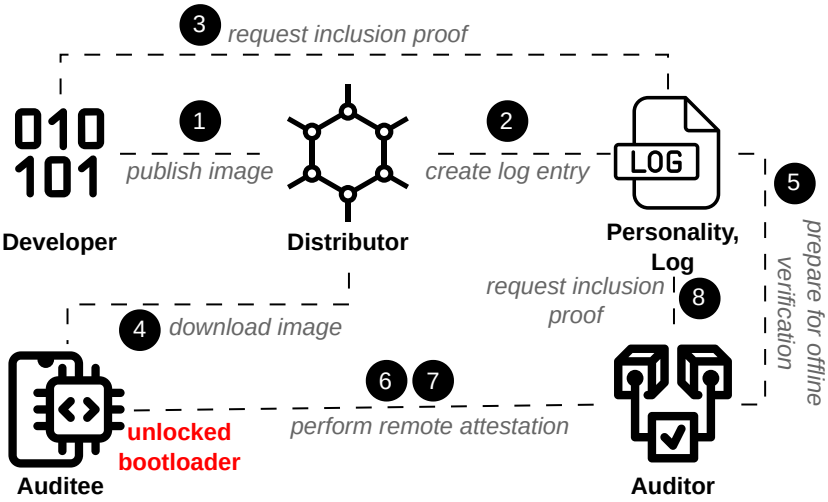


Figure 5.1: Architecture of Firmware Transparency. **(1)** First, the developer uploads the image to the distributor's infrastructure. **(2)** The next step requires the distributor to create the respective log entry before finally distributing the image to its users. **(3)** To ensure that the distributor is not malicious, the developer verifies whether the log entry is available and that it matches the expected log entry in alignment with the image. **(4)** At this point the user can download the image and flash it on the device ending up with an unlocked bootloader and no custom root of trust to verify the newly flashed partitions. **(5)** (optional) There are cases (e.g., no Internet connection) where the audit itself cannot be performed because the auditor cannot request an inclusion proof from the transparency log. In such cases it would be possible to store only relevant audit paths (rather than the entire Merkle tree) of common images (e.g., LineageOS) on the auditor device to enable offline verification. **(6)** The first audit is crucial as we leverage the TOFU model and store a strong device binding property on the auditor device. This security model is required to prevent T-F2. **(7)** Subsequent audits can be performed on top of strong device binding verification. **(8)** Verification of the provided data is done by involving our tamper-proofed transparency log, which contains the legitimate information of the current image version. Taken from our paper [100]

root node hash, the auditor shows a green checkmark to the user – mitigating T-U1, T-D4, T-P1. To prevent relay attacks (see threat T-F2), we utilize the TOFU model (similar to its use in SSH) where a tamper-resistant device binding including the auditee is stored on the auditor device. Subsequent audits must include the TOFU credential.

Personality. The *personality* is the application-specific interface in front of the transparency log. The personality for our prototype is written in Rust as it provides additional security features (e.g., typestate pattern [11]). The Google Trillian implementation provides **.proto* files that we use to enable interaction with the log server via gRPC.

Transparency Log. The incorporation of a transparency log mitigates threat T-U2 by forcing the distributor to create a log entry and enables the auditor to verify the corresponding inclusion proof. More details about transparency logs can be found in section 2.6.

Data Structure

The data structure we use to store log entries contains relevant information needed for full end-to-end verification of the image.

Version. The Android build number is a unique identifier to represent the current software version running on the device. The build number is especially interesting for our concept as it is together with the vbmeta hash unique. Google Pixel, for example, also encodes other potentially interesting information, like the build date, into the build number. For example, the build number *AP1A.240305.019.A1* indicates that the build was created on the 5th of March 2024. Additionally, it also shows if the device is running the latest version and installed security patches on the device.

Hash. This field stores the digest of the custom image. The digest is based on all hashes and hash trees of relevant partitions, like *boot*, and is also part of the vbmeta struct.

Remote Attestation

As already shown in [133], a software-only based attestation for Android is not sufficient to rely on. Therefore, we focus our research on hardware-backed attestation [165]. Android already provides functions to receive a hardware-backed key and the corresponding attestation certificate [61]. To perform key attestation on an Android device, we specify a custom alias of a key pair in order to retrieve the attestation certification chain. To prevent replay attacks (T-F3), we leverage a cryptographic nonce to ensure that every verification session is unique. This nonce must be included in the attestation results provided by the auditee. As we are focusing on the hardware-backed key attestation feature, the root certificate of that chain is signed using the *attestation root key* stored in the hardware-backed keystore. All relevant information required to verify the authenticity and integrity of the operating system is stored in a specific extension called *RootOfTrust* according to an ASN.1 schema. The *RootOfTrust* property contains information about the verified boot key, the lock state of the device,

the verified boot state, and most importantly the verified boot hash. The verified boot hash integrates the digest of the vbmeta struct used by AVB to verify the custom partitions. This approach is recommended by Android to perform remote attestation on Android devices in locked state. Since our research focuses on devices in unlocked state, it is important whether security guarantees such as the integrity and authenticity of the attested vbmeta digest are valid or might have been compromised due to skipped security checks. As existing public documentation is not sufficiently detailed on this (so far, not an officially supported edge case), we experimentally verify if there are any differences between locked and unlocked devices with respect to remote attestation. We use the auditee app in debug mode to gather relevant data for our experiment. We start the experiment with querying relevant information for the attestation request on a Google Pixel 6 with a locked bootloader. Afterward, we query the same information, but with an unlocked bootloader. The experiment shows that the vbmeta digest stays the same when switching the lock state of a device. Thus, it becomes evident that nothing on the respective partitions has been compromised. An interesting observation is that the *verified boot key* is deleted and set to *0* when unlocking the bootloader. One reason for this is to prevent evil maid attacks, as addressed in section 5.7. An essential result partly forming the basis of our trust assumptions is the existence of the root public key[1]. The associated private key of the root public key is stored in the HSM and thus we (have to) assume that an adversary is not able to use this key for signing any attestation details. It does also not change when switching from unlocked to locked state or vice versa. Therefore, we can rely on this trust anchor to verify the authenticity of the provided attestation data. More details about the experiment can be found in section A.1.

Audit. As we cannot trust the device generating this certificate chain, we have to send the signed certificate to an independent party, namely the auditor. We use QR codes as a communication channel to exchange information between the auditee and the auditor. The auditor verifies basic properties of the certificate, like the revocation list, and parses the ASN.1 part of the certificate extension to extract the attestation information. Afterward, the auditor generates the expected log entry using the version and the verified boot hash of the *RootOfTrust* property. The final step is to request the inclusion proof of the generated log entry and verify if the log entry is part of the log itself. Additionally, the auditor can store the current root hash of the log tree to perform consistency proofs in future to verify the append-only property of the transparency log.

Securing User Data

Although a user can verify the authenticity of the booted image by using our proposed system and thus can detect a tampered image, it still would not prevent unauthorized access to the encrypted user data. A common mitigation technique to protect the user data from unauthorized access, is to delete the decryption key every time when the bootloader gets unlocked. This key gets generated by a key derivation function that also considers the user knowledge factor, like the PIN code. To prevent brute force attacks, current Android systems leverage a security function provided by the TEE. However, our design proposal

[1]The full root public key published by Google can be found at https://developer.android.com/privacy-and-security/security-key-attestation#root_certificate.

focuses on devices that have an unlocked bootloader already so we have to assume that the key has not been deleted if an evil maid attack (T-U1) allows an adversary to replace the image because the unlock operation has already been performed. Although the user can verify whether the booted image is authentic, the user has to complete the booting process first and additionally would have to provide the lock screen knowledge factor to start a verification app. Thus, an adversary would also have access to the knowledge factor provided by the user (T-F1). We mitigate this threat by hooking the *boot-complete* signal within the app to start the verification process before the user has to provide the knowledge factor to unlock the user partition. Our prototype implementation demonstrates that the audit can be performed on top of lock screen as well as before first unlock of the device. This enables the user to verify the image before unlocking the user data partition with the knowledge factor. Figure 5.2 present some impressions of the user interface and design of the *Firmware Transparency Auditor App*.

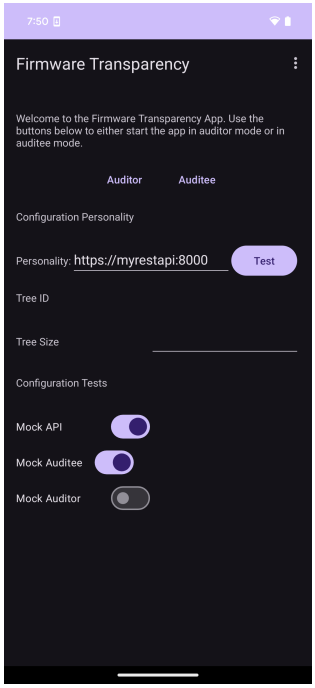
5.3.2 Security Guarantees on Locked Devices

We primarily focus our research on mobile device with an unlocked bootloader. However, we believe that *Firmware Transparency* further enhances verifiability on devices (e.g., Pixel) that already allow users to set a custom root of trust and thus to re-lock the device (YELLOW state). On those devices, AVB performs the verification while booting up the device and would detect any manipulation by an adversary. However, in cases where the adversary has sufficient time to replace the image (e.g., a journalist is forced to provide the device while being interviewed), it would also be possible to replace the custom root of trust used for the verification. In that scenario, AVB does not detect any manipulation attempt as the custom root of trust correlates with the replaced image. However, *Firmware Transparency* does not solely rely on the trustworthiness of that particular key, because the verification allows the user to verify the operating system regardless of the flashed key.

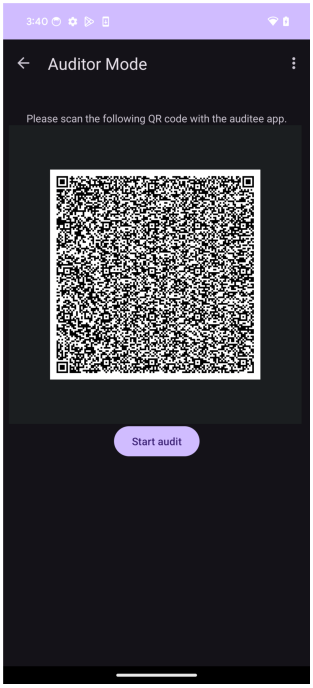
5.4 Verification

We verified *Firmware Transparency* using a FairPhone 3 with LineageOS 22 (Android 15) and Magisk v28.1. Magisk is used to emulate a malicious modification on the *boot.img* of the device system by generating a modified *boot.img* that we flashed on an unlocked device. The first test evaluates the audit of an unlocked device to compare an original LineageOS image with a customized version. We booted the device with the unmodified version, performed an audit and received a positive audit result. Afterward, we flashed the modified version on the device, rebooted it, performed the verification and received a negative audit result. Thus, such tampering attempts get detected by *Firmware Transparency* and displayed to the user.

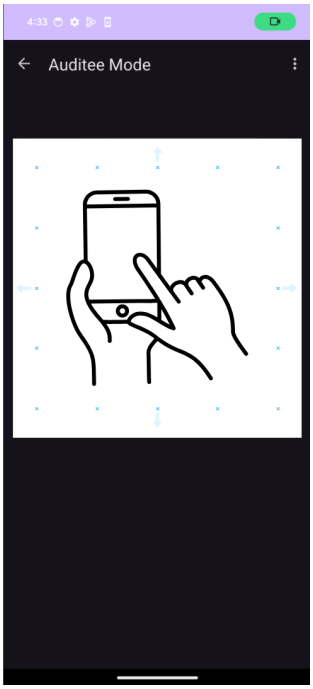
Our second test involved a device in locked state where we were able to use a vulnerability [94] to flash a custom image and still get a *GREEN* boot state. We performed the same steps as we have done before with the unlocked device. Finally, *Firmware Transparency* detected the malicious modification while AVB



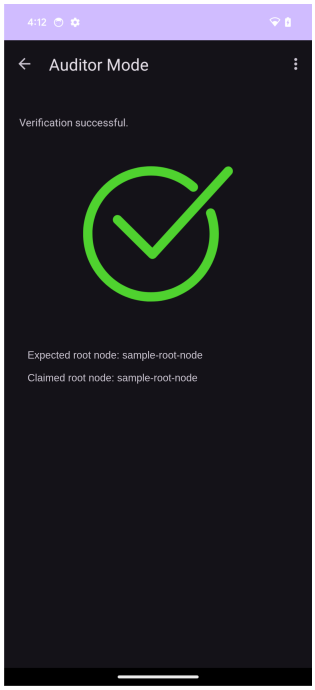
(a) Starting our auditor app.



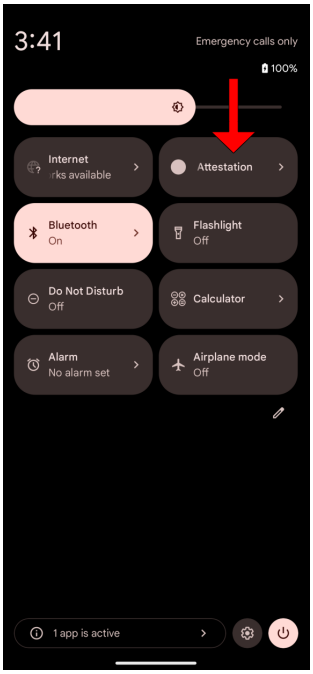
(b) App in auditor mode providing the challenge to the auditee.



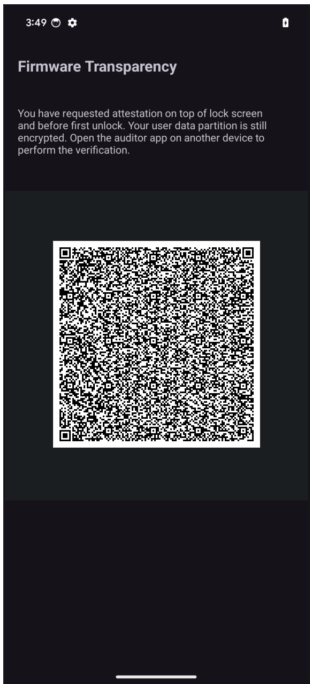
(c) App in auditee mode scanning the challenge provided by the auditor.



(d) Verification confirms that the expected root node hash matches the claimed root node hash.



(e) Tile used to start verification on top of lock screen.



(f) Attestation data provided to the auditor on top of lock screen.

Figure 5.2: Screenshots showcasing some examples of our Auditor app. Adapted from our paper [100].

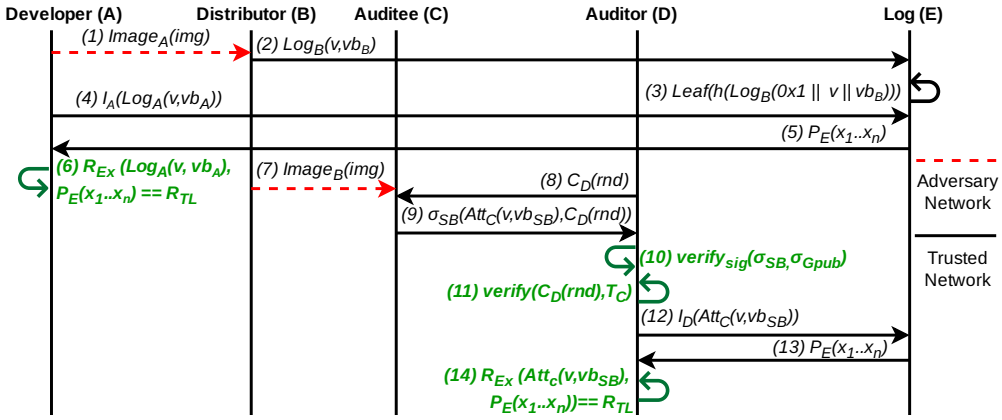


Figure 5.3: Protocol flow of Firmware Transparency. This flow is relevant for the formal model of the underlying protocol. Taken from our paper [100].

failed. It took us approximately 30 seconds to replace a prepared modified image and re-lock the bootloader, demonstrating a practical threat.

5.5 Formal Verification

We formally model and verify the underlying protocol of *Firmware Transparency* using `Tamarin` (see section 2.7 for more information about `Tamarin`), a security protocol verification tool. We use the Dolev-Yao [33] adversary model where the adversary is able to eavesdrop, tamper, or replay messages sent within the network. First, we model our protocol by defining the specific state transitions that represent the behavior of the defined protocol participants. Afterward, we define the security properties automatically verified by `Tamarin` by searching for a trace where an adversary would be able to compromise the specific property.

5.5.1 Protocol Flow

In this section, we describe the protocol that we propose to compensate lost security guarantees of an unlocked bootloader including the relevant security controls and data structures. Figure 5.3 provides an abstract overview of our protocol including relevant stakeholders, relations, and related information or tasks.

Generate image (1). The very first step of our protocol starts with developing (A) the image and sending it to the distributor $\mathrm{Image}_{\mathrm{A}}(\mathrm{img})$.

Create log entry (2). The distributor (B) then creates the transparency log entry $\mathrm{Log}_{\mathrm{B}}(v, \mathrm{vb}_{\mathrm{B}}(\mathrm{img}))$ of the image that is going to be shipped to its users. The data structure of the log entries stored in the transparency log system includes information about the image version v and the hash value of the image file $\mathrm{vb}_{\mathrm{B}}(\mathrm{img})$.

Calculate leaf value (3). Based on the log entry given by the distributor, the personality verifies the input data and calculates the leaf value $Leaf(h(Log_B(0x1||v||vb_B)))$ (more details can be found in section 2.6.2) stored in the Merkle tree.

Verify inclusion proof (4-6). An essential step taken by the developer before the image is distributed is verifying the inclusion proof, such that $R_{Ex}(Log_A(v, vb_A), P_E(x_1..x_n)) == R_{TL}$,, where R_{Ex} is the expected root node hash calculated locally on the developer machine. The calculated root node hash uses x_i which represents the nodes that are part of the audit path. The inclusion proof is valid if the expected root node hash R_{Ex} is equal to the root node hash R_{TL} given by the transparency log. This is important before the distribution as the verification performed by an end-user may fail due to an invalid or missing transparency log entry.

Deploy image (7). Once the log entry is added to the transparency log, the distributor can make the image $Image_B(img)$ available to its users. First, the user has to unlock the bootloader to flash a custom image (e.g., GrapheneOS). To protect the device from unauthorized modification of the operating system, the bootloader has to be locked again after flashing the custom image. However, our proposal ensures sufficient security guarantees even with an unlocked bootloader. Thus, the user neither has to lock the bootloader again nor is able to because there is no way to use a custom root of trust, so we can continue by booting up the custom operating system.

Initialize audit (8). In this stage the end-user wants to verify if the currently booted image is authentic. For that, the user can start our auditor app (in auditee mode) and can scan the QR code generated by the auditor app (running in auditor mode) on a second, independent device. The pairing requires a QR code consisting of a challenge $C_D(rnd)$ used to prevent replay attacks.

Remote attestation (9-10). Next, the auditee requests the attestation and generates a QR code including relevant data and a challenge: $\sigma_{SB}(Att_C(v, vb_{SB}), C_D(rnd))$, where σ_{SB} is the signature function provided by the HSM, Att_C is the attestation function, vb_{SB} is the provided vbmeta digest by the HSM, and $C_D(rnd)$ is the challenge including the random value provided by the auditor. Once the auditor received the data, the authenticity of the provided information can be verified by checking if the information is signed by a trusted key [2]: $verify_{sig}(\sigma_{SB}, \sigma_{Gpub})$

Verify challenge and TOFU basis (11). Next, the auditor verifies the TOFU basis T_C to prevent relay attacks (T-F2) and the challenge $C_D(rnd)$ to prevent replay attacks (T-F3). In case this is the first audit, the auditor stores the TOFU basis for the specific auditee device C.

Request inclusion proof (12). To verify whether the corresponding log entry has been properly logged, the auditor requests the inclusion proof $I_D(Att_C(v, vb_{SB}))$, where Att_C represents the attestation details.

Providing inclusion proof (13). In case the requested log entry can be found in the Merkle tree, the personality responds with the audit path $P_E(x_1..x_n)$ that includes the necessary tree nodes to calculate the expected root hash.

[2] In case of Google Pixel, the public key used by StrongBox can be found on https://developer.android.com/privacy-and-security/security-key-attestation (last accessed on 23.05.2025)

Verify inclusion proof (14). The final step is to calculate the expected root hash with the information available from the inclusion proof combined with the generated expected log entry: $R_{Ex}(Att_C(v, vb_{SB}), P_E(x_1..x_n)) == R_{TL}$ If the locally calculated root hash (also referred to as expected root hash) matches the claimed root hash of the transparency log the auditor confirms that the verification has been successful and that the image is part of the transparency log.

5.5.2 Assumptions

We assume the transparency log to be tamper resistant so that an adversary is not able to retrospectively insert, delete, or modify any record once it is stored in the log. Additionally, we assume that the authentication and authorization mechanism incorporated in our personality prevents the adversary from any action except reading log entries. Therefore, we do not expect communication between the distributor and the personality via the adversary network. Our model also assumes that the attestation information generated from the auditee and sent to the auditor is trustworthy [5]. This assumption is based on the fact that we assume to have an HSM available that performs the attestation and signs the generated data with a verifiable key. This includes the information transfer between the auditee and the auditor as the signature can still be verified by the auditor. We do not consider the verification of the signature itself within our model, because we assume signatures to be secure. To maintain simplicity our model focuses on the protocol behavior itself by abstracting standard defense mechanisms. We do also consider the usage of a transparency log as already proven defense mechanism against threat T-U2.

5.5.3 Security Properties

This section outlines the security properties used to verify the integrity of the image based on some particular threats of our threat model in chapter 3 and specific threats targeting the bootloader (section 5.2.2). Our model defines two types of security properties: one verifies whether there is a trace where an adversary could successfully perform the specific attack without applying our mitigation concept, and the other ensures that there is no trace of success when our verification system is applied. Each security property is associated with an adversary model *A* (section 3.2) and a threat *T* (section 3.3). We model the security properties as formulas in a first-order logic using *lemmas* within `Tamarin`. To express the execution of an *action fact* in `Tamarin`, we use the notation $F(x_1..x_n)@i$ where F denotes the name, x_i the variable, and $@i$ the time variable for the execution.

Evil maid attack and on-path compromise (A1,A2,T-U1,T-D4,T-F1). This security properties are used to verify whether an adversary with physical access to the device or with access to the distribution network is able to compromise the operating system. The first proof we perform using `Tamarin` is to verify whether there is a trace where an adversary can successfully manipulate an image f in case the auditor does not verify the attestation at and the inclusion proof ip. Specifically, this lemma shows that $\exists\, f, at, ip : f = at \wedge \neg(h(at) = ip)$. Thus, we can ensure that an attack would be possible with respect to our formal model in case *Firmware Transparency* is not used. Conversely, we prove that `Tamarin`

does not find any trace where an adversary can compromise the image without detection. Thus, our second lemma verifies $\forall f, at, ip : h(f) = ip \wedge f = at$.

Insider compromise (A3,T-P1). This property verifies if an insider adversary with access to the signing key can compromise the image f without detection. The first lemma determines whether this attack remains undetected if the developer does not verify the inclusion proof ip and the attestation at. Specifically, this lemma shows that $\exists f, ip : \neg(h(f) = ip)$. Conversely, we prove that `Tamarin` does not find any trace if our protocol is used. The related lemma verifies $\forall at, ip : ip = h(at)$.

Relay and replay attack (A1,T-F2,T-F3). These properties address both relay- and replay attacks, where an adversary either relays a valid attestation result (at_r based on at) to the auditor or replays an existing, but still valid attestation. The first relay- and replay lemmas determines whether these attacks are possible without auditor verification. Specifically, the first lemma for relay attacks verifies $\exists at, at_r, t, t_s : at = at_r \wedge \neg(t = t_s)$ and for replay attacks $\exists f, at, at_a, c, c_t : \neg(f = at) \wedge at = at_a \wedge \neg(c = c_t)$. Conversely, we prove that `Tamarin` does not find any trace for such an attack if our protocol is used. The related lemma for relay attacks verifies $\forall at, at_r, t, t_s : at = at_r \wedge t = t_s$ and the lemma for replay attacks $\forall cs, ct : cs = ct$.

Sanity check. We also introduce some lemmas to verify if there is a trace available according to the expected behavior of our protocol. These lemmas do not represent any security relevant property, but are used in particular to ensure that some specific traces can be reached at all.

5.6 Related Work

This section discusses related and similar solutions concerning secure booting procedures and existing solutions incorporating transparency logs related to mobile devices.

GrapheneOS. GrapheneOS [64] is an alternative mobile operating system focused on security and privacy. From a security perspective it contains various layers of defense such as disabling certain functionalities, hardened SELinux policies for improved sandboxing, and enhancing verified boot. According to their FAQ section [65], GrapheneOS only supports Google Pixel devices due to security and update aspects. The underlying implementation enforces verified boot during startup by using a custom root of trust for the verification process. This is one reason why they only support Google Pixel devices because Google allows flashing a custom root of trust on Pixel devices. One of the listed requirement which is also relevant for this thesis is having access to full hardware security functionalities. Additionally, GrapheneOS provides a hardware-based attestation app for Android devices [63] used to verify the authenticity and integrity of the operating system.

However, GrapheneOS does not provide a secure solution for devices where it is not possible to use a custom root of trust. The question with respect to GrapheneOS is not how it differs from our proposal, but rather how our proposal can complement GrapheneOS, enabling secure booting of devices without flashing a custom root of trust. The auditor app provided by GrapheneOS can be used

to ensure that the mobile device is running an authentic version of the operating system. However, this app is only supported on devices with a locked bootloader – *YELLOW* state. Hence, it is not suitable for the use case addressed by this thesis. Furthermore, the auditor of GrapheneOS necessitates that users provide their knowledge factor (e.g., after rebooting the device) before they can perform any form of verification.

Pixel Binary Transparency. Google introduced the Pixel Binary Transparency [54] to verify the authenticity and integrity of Pixel factory images. An example method of verification involves downloading a stock image from the official website, and requesting an inclusion proof. Although this method provides a full verification approach, it is only applicable on a downloaded image before flashing it on the device. It is also possible to verify the image currently deployed on a device by requesting relevant metadata via Android Debug Bridge (adb). However, using adb does not provide high security confidence as the currently deployed image could already be malicious and thus could tamper the requested values (e.g., the vbmeta digest). At the time of writing this thesis, the Android Binary Transparency website [54] only illustrates the adb approach, but with an additional note that they will also provide an example using Key Attestation for better security confidence. Therefore, we conclude that Pixel Binary Transparency may not be suitable for security-sensitive individuals (e.g., journalists) who wish to verify the device's current state before submitting sensitive information on that device.

Both our proposed system and Android Binary Transparency are based on Merkle trees which are used to store information about images. The similarities of the data structure enable us to use the same underlying Google Trillian implementation for managing the Merkle tree. However, there are significant differences in scope, integration, and operability. Our proposal is not limited to only store information about Pixel images as we focus on mobile devices which do not allow using a custom root of trust for locking the bootloader. The use case of the transparency log within our system affects the integration of the log significantly. While the Android Binary Transparency log must be queried by the user with metadata derived from the device, we make use of a separate auditor app which gets relevant information through remote attestation. This approach ensures a full end-to-end verification of an already flashed image which is at the time of writing this thesis not available for Android Binary Transparency.

Sigstore. The Sigstore ecosystem [117] is an open-source project based on available open-source tools aiming to address security considerations in software supply chains. The primary focus of Sigstore is trust in signatures on any kind of artifacts. First, the Sigstore client requests a certificate from the Sigstore CA using OpenID Connect to authenticate the developer. During this request, the certificate, together with the artifact hash and the signature, is added to the transparency log. The provided certificate can then be used to sign the artifact that is published to its users (e.g., via web download). Finally, the end user is able to verify the signature and if it has been properly logged in the transparency log.

The difference compared to the Sigstore project is that it uses a transparency log to store metadata about an artifact. However, this project focuses on signatures while our proposed system bases the verification on metadata of the image itself. Therefore, Sigstore does not address our threat model as we do

not focus on the downloaded image itself, but on the flashed partitions. Thus, Sigstore cannot be used to mitigate lost security guarantees due to an unlocked bootloader, but would be useful for verifying the downloaded images before flashing it on the device.

5.7 Deployment Consideration

Rethinking trust on first use. Our proposal still relies on the TOFU model to prevent threats like T-F2. Thus, an adversary who is able to compromise the device before the first audit could manipulate the device binding parameter to perform relay attacks on future audits. To strengthen the security guarantees provided by our proposed solution, we are already working on a follow-up research project aimed at exploring even stronger guarantees without relying on the TOFU model.

Incorporate verification in early boot stage. To strengthen our mitigation against evil maid attacks we suggest incorporating our proposed system in the early bootloader stage. Currently, our solution is designed for sensitive users (e.g., journalists) who want to use a custom image and want to have a possibility to verify the authenticity of the image before providing their knowledge factor and unlocking the user data partition. However, a compromised image could just deactivate the verification or try to trick the user to believe a wrong verification result. Although a sensitive user might detect such attempts and explicitly wants to verify the authenticity of the operating system, the overall security could be strengthened by OEMs incorporating our verification control in an early bootloader stage. We are aware that the usability of *Firmware Transparency* in its current state is not practical for the majority of Android users, but strongly believe that it is certainly a mitigation, useful to a selected group of users based on their threat model (e.g., journalists). This matches with the fact that most Android users do not use an alternative operating system. Our proposal needs to be seen in the context of the sub-group of users who are ready to install an alternative operating system.

Convince other vendors. First and foremost, we should convince other vendors to support custom images by allowing users to make use of AVB with a custom root of trust. However, we believe that our proposal still provides an additional layer of security especially to prevent attacks on the image distributor.

Relying on root of trust. Although our current approach can be used to verify the authenticity of an image flashed on a device, it cannot be used on devices without a trusted hardware security module as our design requires a root of trust. To our best knowledge, there is currently no solution that sufficiently mitigates the lost security guarantees due to an unlocked bootloader on devices without that root of trust. This is an orthogonal research question that remains open and requires further exploration in the area of replacing hardware root of trust in general.

5.8 Availability

We provide our prototype implementation though three repositories, all publicly available: **(1)** The personality contains the code for the application-specific interface between the auditor app, the firmware distributor, and the logging backend. Source Code: `https://github.com/linsm/firmwaretransparency-personality`

(2) The auditor project contains the code for the end-to-end verification of the image. Source Code: `https://github.com/linsm/firmwaretransparency-auditor`

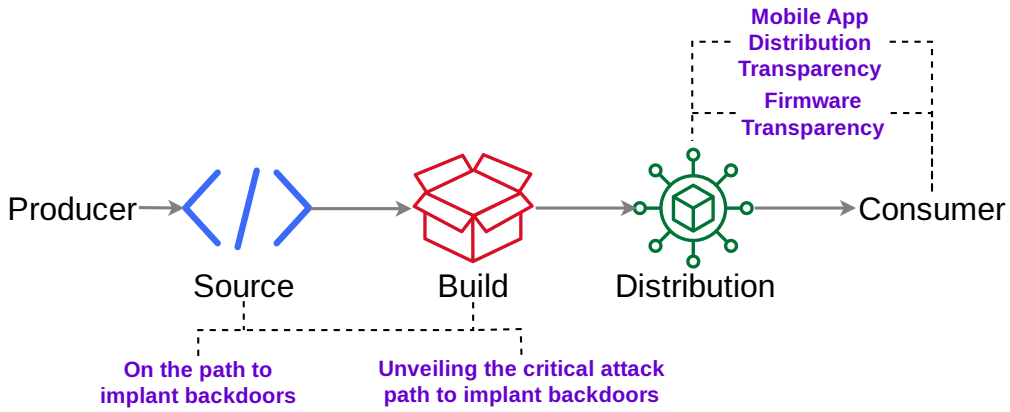
(3) The formal verification project contains the `Tamarin` model and proof of our protocol. Source Code: `https://github.com/linsm/firmwaretransparency-formalverification`

5.9 Summary

We presented a new security mechanism, built on transparency logs, to compensate lost security guarantees of devices with an unlocked bootloader and to enhance verifiability on locked devices that maintain a functional chain-of-trust. One of the primary goals of our approach is to provide new opportunities for applying updates to mobile devices that are no longer maintained by the manufacturer, while preserving important security primitives. We also acknowledge that most mobile device manufacturers prioritize their own software stack over freedom-of-choice, limiting support for using a custom root of trust and thereby restricting usage of important security features. Thus, we aim to support long-term updates due to sustainability aspects and freedom-of-choice without sacrificing security. Additionally, we argue that *Firmware Transparency* further enhances verifiability on devices that already allow users to set a custom root of trust. Our concept successfully mitigates specific threats that we have elaborated within our threat model. To demonstrate the practical feasibility of our concept, we designed and implemented a prototype. The verification of our prototype shows that malicious modifications can be successfully detected, even if a vulnerability enables evasion of current security mechanisms. Finally, we have proven that relevant security properties are satisfied by formally verifying the underlying protocol design based on the considered threat model.

Chapter 6

On the Critical Attack Path for Implanting Backdoors in Supply Chains



Paper Reference. This chapter is based on the paper "Unveiling the critical attack path for implanting backdoors in supply chains: Practical experience from XZ" [97] and "On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ" [99].

In the previous chapter, we explored distribution and user verification at the application and (operating) system layer. In this chapter, we focus primarily on the source code stage, with some attention to the build stage.

We were extraordinarily lucky this time. On Friday, March 29th, 2024, Andres Freund ignited newspapers, social media channels, and security experts around the world when he revealed [44] a critical supply chain attack (CVE-2024-3094) that introduces a backdoor in the widely used XZ Utils (section 2.5). He began investigating the XZ Utils package after noticing a slight 500 ms delay, an increase in CPU consumption, and some *valgrind* errors during test cases utilizing a remote SSH server. The affected package, XZ Utils, is a set of open source components, including `xz` and `lzma`, used for data compression. An attacker, using this backdoor, is able to execute arbitrary code remotely on vulnerable machines utilizing SSH without prior authentication. The seriousness of this security incident is underscored by the widespread usage of XZ Utils on a significant number of Linux and other Unix-based systems, and the resulting CVSS rating of 10.0. Both the level of organizational attack complexity and the high code sophistication that were applied to compromise a widely distributed open-source project with the aim of attacking a specific supply chain (and various other aspects) make this security incident particularly worthy of in-depth

study. It highlights the need for careful consideration, both in addressing similar attacks in the future and in detecting potential active backdoors for which we were not yet as lucky to have become aware of. In this chapter, we address relevant practical experiences with a focus on supply chain attacks by making the following theoretical and practical contributions:

- We analyze the malicious version of the XZ Utils package to identify and emphasize properties that could be relevant to other potential targets of interest for performing such a supply chain attack.
- We identify the essential stages of the critical attack path for implanting and activating a backdoor in an open-source project.
- We generalize from this particular case to identify key takeaways as categories of important signals—both technical and social—to consider within the open-source community in order to prevent similar attacks in the future.
- We implement *SketchyCrawler* as a first prototype to assess GitHub repositories for sketchy signs based on our evaluated takeaways.

6.1 How to Select the Ideal Target for Such an Attack?

First and foremost, XZ Utils is considered a well-known and widely distributed package used by various other packages and is shipped as an integral component of many operating systems, such as Debian. Due to its continuous development and long-standing presence, it has been highly trusted within the open-source community for many years. We also believe that the attacker(s) may have chosen the XZ Utils project because they were looking for an open source project with a single maintainer who could become overwhelmed when increasing the workload. Another key observation relates to evasion, emphasizing that the actual target—servers utilizing OpenSSH—do not have a direct dependency on XZ Utils. Instead, some distributions utilize `systemd` in combination with `libsystemd` to manage services like OpenSSH. Notably, `libsystemd` depends on the library `liblzma`, which is part of the XZ Utils project. Considering further evasion and obfuscation signs with respect to the XZ project, the test files may play a crucial role as well. These files, being binary blobs, are not easily readable yet appear legitimate within the context of a data compression utility.

We outline the (supposed) key reasons for choosing an open-source project like XZ Utils to strategically plan such a supply chain attack:

- Well-known package with sufficiently long history
- Trusted within the open-source community
- Widely distributed package to increase attack surface
- Single maintainer, easily overwhelmed
- Actual target does not directly depend on infected package in upstream source, making detection harder
- Using binary blobs that appear legitimate, but can be used to implant malicious code

Another important consideration when selecting the ideal target for such an attack is to determine how widely distributed the infected XZ Utils package actually is. To address this, we conduct a recursive reverse dependency analysis of the affected `liblzma` and a set of other libraries for comparison. We chose that set as an exemplary representation of libraries (a) providing similar functionality (i.e., other compression libraries), (b) providing essential functionality that we expected to be widely used in general (i.e., GNU standard C library) and particularly in applications with network communication (e.g., OpenSSL's `libssl3`), (c) a randomly-picked image processing library (as an example for (in-)famous bugs leading to exploitable RCE on both consumer devices and server-side applications), and (d) libraries related to the exploitation target chosen in the XZ Utils incident. Table 6.1 summarizes the results of our analysis. As shown, `liblzma` is used by nearly 30,000 other packages. When compared to another popular, and potentially even more well-known tool, such as `curl`, `liblzma`, while supposedly more obscure to both users and administrators, shows a usage rate that is 6 times higher. These facts underscore the significant severity in having malicious code (e.g., a backdoor) hidden in a library with such widespread distribution.

6.2 Attack Analysis

This section describes relevant stages, including both human/social and technical aspects, that lead to the compromise of the XZ project by incorporating a backdoor. We start with an analysis of the currently known facts [3, 13, 21, 22, 26, 44] around the security incident and derive a critical attack path as shown in Figure 6.1 by categorizing the available information (including the temporal sequence of the events) into individual stages.

Table 6.1: Results from a reverse dependency analysis. The dependency analysis*, conducted in January 2026, focuses on a set of selected libraries.

	reverse-depends count on	
Library	Ubuntu:26.04	debian:13-slim
libc6	69,452	64,762
libzstd1	39,229	36,837
liblzma5	**21,311**	**27,343**
libssl3(t64)	30,795	29,499
libbz2-1.0	31,241	29,608
liblz4-1	3,650	3,544
libsystemd0	23,174	21,912
libjpeg(62)-turbo(8)	14,115	14,279
libssh-4	1,157	1,154
libcurl4(t64)	4,743	5,076

*Using: `apt-rdepends -r <package> | grep "Reverse Depends:" | sort -u | wc -l`

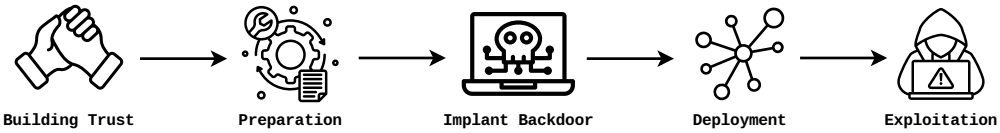


Figure 6.1: Critical Attack Path. Taken from our paper [97].

6.2.1 Stage 1: Building Trust

The XZ security incident did not begin with fancy hacking tools or with sophisticated code; it all started with a hacking technique known as social engineering, which prays on humans' emotions, trust, and pressure. In case of XZ, especially pressure plays a crucial role in the social engineering attack targeting the open-source community. This section summarizes insights into the social engineering tactics employed by the attacker(s) to gain necessary privileges on the repository.

In 2005 [160], a small group including Lasse Collin, started work on a project called LZMA Utils, which was later renamed XZ Utils. In 2021, "Jia Tan" (presumably a pseudonym) started to support the XZ Utils project by regularly contributing code improvements via the XZ developers mailing list. At the time of writing this thesis, we are not aware of any malicious content in the first contributions of Jia Tan. Apparently, Jia Tan seems to be a friendly, interested, and supportive contributor of the open-source project. However, some years later it became clear that Jia Tan had different intentions with the XZ Utils project. It appears that the initial intention of Jia Tan was to build up trust within the open-source project, which plays a significant role with respect to the upcoming attack path. The first suspicious commit [79] involving Jia Tan under the username "JiaT75" dates back to November 2021. The commit was done within another project called "libarchive" and replaced the `safe_fprintf` function with `fprintf`, the less secure variant of this functionality. This commit has been approved and merged to the main branch of that project without noticing that this might be a suspicious change. At the time of writing this thesis, it remains uncertain whether this commit is relevant to the XZ backdoor case.

A few months later, in April 2022, user JiaT75 tried to submit another patch via the XZ developers mailing list [150]. According to the conversation log, there was suddenly another pseudonym, called "Jigar Kumar" involved, who describes this patch as "*quality of life feature*" [88] and complains about the slow release schedule of the project maintainer. As one of the initial responses to this complaint, Jia Tan took on the role of a supportive, friendly, and loyal defender of the original maintainer, Lasse Collin, pointing out that the contributors are hobbyists and that can't dedicate 40+ hours a week. The conversation continued with putting more and more pressure on the original maintainer Lasse Collin. Meanwhile, in a separate project focused on XZ for Java, a new participant, Dennis Ens, enters the scene. This pseudonym also began similarly, adding extra pressure on the original maintainer by, for example, inquiring whether the project was still maintained. In this particular conversation, Lasse Collin mentioned that the project is an unpaid hobby and that he is already in contact with Jia Tan who "will have a bigger role in the future". We detected the first commit [80] solely merged by Jia Tan from December 2022.

Figure 6.2: Highlights of the social engineering attack conversation targeting the XZ Utils project. **(1)** Jia Tan wants to provide a patch via mailing list. **(2)** First signs of pressure by Jigar Kumar. **(3)** Support from Jia Tan. **(4)** Increasing pressure from Jigar Kumar. **(5)** Pressure from another pseudonym, Dennis Ens, on a different project. **(6)** First signs of being overwhelmed by Lasse Collin. **(7)** Attacker(s) want to become co-maintainer. **(8)** Again pressure to pass maintainership to attacker(s). **(9)** Finally, the attacker(s) were successful. Taken from our paper [97].

Thus, we infer that Jia Tan obtained the required permissions for the repository and may have been promoted to co-maintainer of the project.

Based on these conversations and the similarities between these different types of individuals, we assume that the attacker(s) have set up these pseudonyms, commonly referred to as "sock puppet" accounts. More insights on the conversations can be found in Figure 6.2. We explicitly note that attribution, i.e., trying to estimate the real source of these (supposedly coordinated) pseudonyms and tracing back to an attacking organization and/or geopolitical region is outside the scope of this thesis.

6.2.2 Stage 2: Preparation

In the second stage of the critical attack path we outline the various actions taken by the attacker(s) to prepare for the actual supply chain attack on the source code. A key aspect of this stage is that all their actions and steps remain deniable. Thus, if any malicious activity had been detected at this stage, the attacker would still have a valid reason to deny any malicious intent.

In March 2023, the primary contact mail for a component in the testing infrastructure, Google OSS-Fuzz (section 2.2.6), was changed from `lasse.collin@tukaani.org` to `jiat0218@gmail.com` by opening a merge request on OSS-Fuzz. That change of contact was manually approved by Lasse Collin as the original maintainer, confirming that Jia Tan is an official co-maintainer of the XZ project [81]. After some months, a new contributor called "Hans Jansen" provided patches to the XZ Utils project that use the GNU indirect function

(IFUNC) feature (section 2.5.1) [76, 77]. After some reworks and discussions, Jia Tan merged that functionality to the repository. Only later, it turned out that these patches are essential for the attack, because IFUNC is the function hooking mechanism eventually used by the backdoor code. However, the use of IFUNC cannot be fuzzed by OSS-Fuzz, due to compatibility issues. We assume that the adversary expected errors resulting from fuzzing with OSS-Fuzz after enabling IFUNC. Consequently, Jia Tan opened a pull request [82] within the OSS-Fuzz project, to disable IFUNC for the fuzzed XZ Utils build. In this request, Jia Tan mentioned that IFUNC is not compatible with "`-fsanitize=address`" that is also known as *AddressSanitizer* (ASan) and used to detect memory errors. The OSS-Fuzz repository incorporates a GitHub action to automatically label pull requests as "ready to merge" if they originate from the maintainer of a fuzzed project by verifying that the committer has a contribution history for that project and is one of the responsible project contacts. As both applied to Jia Tan, the pull request was labeled as *ready to merge* by the bot, and then quickly approved and merged into OSS-Fuzz.

6.2.3 Stage 3: Implanting Backdoor

This stage consolidates the steps required for implanting the backdoor into an open-source project, ensuring it stays hidden even in a public repository. Compared to the previous stage, the actions taken by the attacker(s) in this stage are no longer deniable.

On Friday, February 23rd, 2024, Jia Tan added various lzma test files [151, 152] to the project. Later, it turned out that the uploaded test files, called `bad-3-corrupt_lzma2.xz` and `good-large_compressed.lzma` contained the binary backdoor and some other obfuscated code necessary to inject the backdoor code. At this stage, the hidden backdoor is already part of the project, but is not yet deployed. Another essential preparation step was to add the file `build-to-host.m4` to the `.gitignore` file of the source code repository so that its source code does not get pushed to the repository. However, a modified version of the `build-to-host.m4` file was still part of the source tarball that gets fetched by the build system to build the final artifact. Comparing a previous version of the `build-to-host.m4` file with that one stored in the source tarball clearly yields some differences. We focus on one of these to show how the attacker(s) were able to obfuscate the malicious parts finally executed on the build machine. Figure 6.3 shows the obfuscated command hidden in the `build-to-host.m4` file.

6.2.4 Stage 4: Deployment

The following day (Saturday, February 24th, 2024), Jia Tan released version 5.6.0 of XZ Utils via GitHub. In addition to the auto-generated source archives in `.zip` and `.tar.gz` form, XZ Utils releases include four additional, manually uploaded source archives with corresponding signature files. The manually uploaded source archives, which were uploaded to GitHub for this release, also include the modified `build-to-host.m4` file, which is not part of the repository itself to evade potential discovery through the repository change history. The

```
1 gl_[$1]_config='sed \"r\n\" $gl_am_configmake |
  eval $gl_path_map |
  $gl_[$1]_prefix -d 2>/dev/null'

2 gl_am_configmake=`grep - aErls "#{4}[[:alnum:]]{5}#{4}$"
  $srcdir/ 2>/dev/null`

3 gl_path_map='tr "\t \-_" " \t_\-"'

4 gl_[$1]_prefix=`echo $gl_am_configmake | sed "s/.*\.//g"`

5 cat ./tests/files/bad-3-corrupt_lzma2.xz |
  tr "\t \-_" " \t_\-" | xz -d
```

Figure 6.3: Obfuscated command hidden in `build-to-host.m4` file. **(1)** Obfuscated command including three variables: `gl_am_configmake`, `gl_path_map`, `gl_[$1]_prefix`. The `sed` command is actually a hidden `cat` command. **(2)** Extracts the relative path of the `bad-3-corrupt_lzma2.xz` test file. **(3)** `tr` command to uncorrupt the content of the test file. **(4)** Echoes the relative path of the test file and removes characters except the file ending: xz. **(5)** Unveiled command that fetches content of test file, pipes it into `tr` command to uncorrupt it, and afterward extracts the hidden shell script by decompressing it with `xz`. Taken from our paper [97].

malicious m4 file extracts the backdoor from the test files when building the deb or rpm packages.

The next step was to convince Linux distributions to include the latest, but malicious, release of XZ Utils in their package repositories. For that, the pseudonym Hans Jansen started to contact and pressure some Linux distributions (e.g., Debian [78]). Once the package maintainer upgraded to the new version by fetching the source code tarball, the malicious `build-to-host.m4` file gets integrated into the package build process. The malicious build script then initiates the steps required to extract the backdoor hidden in the test files. After several obfuscated modification, decompression, and execution steps on the malicious test files, as illustrated in Figure 6.4, an object file named `liblzma_la-crc64_fast.o`, which contains the backdoor code, is linked into the `liblzma` library via setting the respective pointer which gets resolved by the IFUNC functionality. Additional information can also be found on a blog post, written by Russ Cox [26].

6.2.5 Stage 5: Exploitation

The final stage is the exploitation stage (Figure 6.5) in which the backdoor gets triggered on the infected system. Current investigations show that a pointer of a function used by `RSA_public_decrypt` for OpenSSH (more specifically for any process of the executable located at `/usr/bin/sshd`) is changed to point on a function owned by the backdoor. The malicious function extracts a command from the certificate provided by the SSH client. This is only initiated for a specific private key which makes the backdoor only available to attackers who possess the corresponding key. Afterward, the command is passed to the `system()` function, which executes it. Thus, an attacker who controls the private

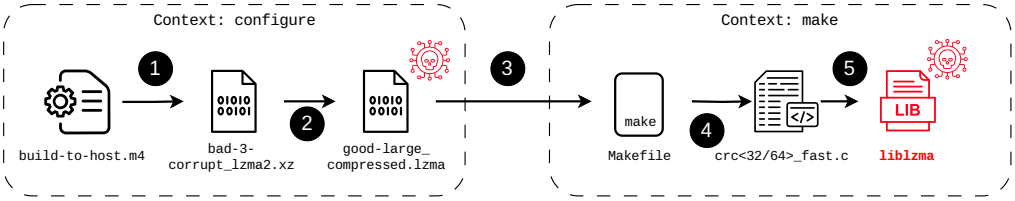


Figure 6.4: Remaining steps to implant the backdoor in the final artifact. The dashed lines are used to visualize the different build contexts. **(1)** The `build-to-host.m4` file initiates the implanting process by "repairing" the `bad-3-corrupt_lzma2.xz` file and executing a script hidden in there. **(2)** This script runs additional commands hidden in another test file called `good-large_compressed.lzma`. This test file is properly compressed with XZ, but contains junk data that gets removed by various operations, such as using the `head` command to get specific bytes. The final result is again piped into shell and gets executed. **(3)** Besides some checks on the environment, the actual binary backdoor also hidden in the `good-large_compressed.lzma` test file, but with another offset, gets extracted into `liblzma_la-crc64-fast.o`. Additionally, the `Makefile` is patched to incorporate the malicious backdoor in the compiled library. **(4)** After some preparation, verification, and cleanup steps, the patched `Makefile` modifies the code of the files `crc64_fast.c` and `crc32_fast.c`. This step is important for the exploitation stage, as the contained function `crc64_resolve` provides the actual IFUNC resolver, which is used to call the `_get_cpuid` function that is provided by the backdoor. **(5)** Finally, the backdoor hidden in the `liblzma_la-crc64-fast.o` file including the IFUNC resolver gets compiled in the liblzma library. Taken from our paper [97].

SSH key can send arbitrary commands to affected SSH servers which are executed remotely without ever completing the authentication to the SSH server.

6.3 Takeaways

We now try to generalize from the specific XZ incident and derive more abstract takeaways that we argue are applicable to different open-source projects under the category of aspects described in section 6.1.

6.3.1 Signs of Social Engineering

Lively discussions within an open-source project are generally seen as positive and encouraged. However, if there are suddenly more and more messages of offensive or otherwise problematic nature such as increasing pressure on specific contributors, it might be a sign that should be taken seriously. Especially time pressure seems to be effective in this context and could be a sign for social manipulation as part of an attack. Other signs of social engineering are extending or transfer of maintainer status for a project or components, as has

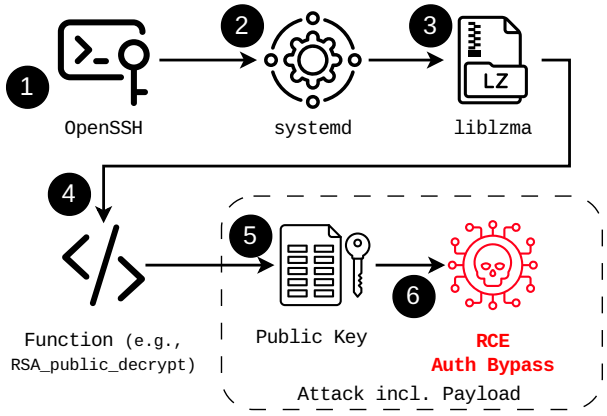


Figure 6.5: Actual exploitation of the backdoor hidden in XZ Utils. **(1)** The actual targets are vulnerable servers utilizing OpenSSH. The attack gets initiated by providing a specific OpenSSH certificate to these servers. **(2)** OpenSSH does not directly depend on the malicious liblzma library, but is managed on some systems via systemd. **(3)** systemd is used to load the malicious library liblzma. **(4)** The malicious library providing the ifunc resolver gets called before the GOT and PLT table are read-only. This allows the backdoor to rewrite the tables to hijack function calls, such as `RSA_public_decrypt`, which is used by SSH. **(5)** The SSH certificate provided by the attacker contains the payload. **(6)** The payload gets executed, so the attacker is able to run commands remotely without prior authentication. Taken from our paper [97].

happened in this case, but also in many others with malicious intent (e.g., in the various NodeJS and Python cases [12, 122, 123]). The **first takeaway (TA-1)** for mitigating comparable future cases is therefore an increased focus on social engineering of project maintainer control.

6.3.2 Small Projects with Large Reverse Dependencies

The XZ Utils project was, in hindsight, an excellent target due to its relatively small code base and clear focus, but with a surprisingly large reverse dependency tree.

This **second takeaway (TA-2)** is potentially the lowest hanging fruit for future research by prioritizing independent analysis on and adding mitigations to exactly such projects matching this filter criterion—which are far fewer than the total number of packages in most Linux distributions or programming language package/module systems. Conversely, we can derive a **third takeaway (TA-3)** for critical software projects (such as OpenSSH) to particularly focus on their own dependency tree in terms of such small, potentially vulnerable projects they might depend on. This is not easy if the dependency tree is large or includes dynamic indirections,[1] potentially even crossing language/platform

[1] One of the changes after the XZ Utils incident is that `libsystemd` now loads compression libraries through `dlopen`, which makes this indirect dependency even harder to follow (cf. https://github.com/systemd/systemd/issues/32028 and https://github.com/systemd/systemd/pull/31550).

boundaries. Nonetheless, the same filter criterion can be applied to prioritize analysis efforts.

6.3.3 Potential for Bugdoors

The quality of source code is not just about making it easy to maintain or improving the readability; complex or poorly written code can introduce significant vulnerabilities. Adversaries might intentionally introduce such poor source code—what we refer to as "bugdoors"—to cover their attack vectors with plausible deniability. The XZ backdoor incident demonstrated how unnecessary complexity in the codebase and software architecture could have been used to prepare for the incorporation of future bugdoors. For example, the use of *autoconf* and the (presumably) malicious activation of IFUNC functionality (which did not seem technically required) raise concerns. A modern software development lifecycle, including code reviews, should prevent merging of such questionable code modifications into the main branch. However, as demonstrated by the XZ case, adversaries often use sophisticated social engineering techniques to legitimize these potentially malicious code changes. Detecting these intentionally obscured bugdoors is inherently difficult, requiring a thorough understanding of technical details of the software and a dedicated, security-focused code review process. The combination of languages like C (which are prone to memory safety issues) and non-hermetic, non-declarative build systems, coupled with the potential of bugdoors, makes it even harder for development teams and the security community to mitigate resulting threats.

Therefore, our **fourth takeaway (TA-4)** is to avoid unnecessary complexity in code, architecture, and build systems. This includes using strongly-typed memory-safe languages (e.g., Rust) whenever possible to make it harder for adversaries to implant bugdoors. On the build server side, declarative build systems are preferable, and complex build approach like using *autoconf* should be avoided when feasible.

6.3.4 Signs of Obfuscation

The final, **fifth takeaway (TA-5)** is perhaps the clearest one: be alert to signs of obfuscation, including explicit ones (such as excerpted in Figure 6.3) and implicit, unexplained binary blobs—which are much more common in many projects. While binary blobs acting as input for training or used for "error" test cases have their purpose, we should now start to demand reproducibility of all such binary blobs. That is, binary blobs should, e.g., be generated by explainable helper scripts checked into the repository. While there can be no guarantee that such helper scripts are not in themselves bugdoors in the sense of generating malicious, dual-purpose binary blobs, we argue that it lowers the probability of occurrence. Remaining input for (regression) test cases without reproducible generation processes that have, e.g., been submitted by users as part of bug reports, should only be used or executed within tight sandboxes during the build process.

6.4 SketchyCrawler

We implemented *SketchyCrawler*, an open-source tool prototype used to crawl GitHub repositories to detect sketchy signs for backdoor potential in open source projects. The main objective is to support software development teams in auditing and assessing such projects before they incorporate them in their own software project. Although there are established, well-known practices for mitigating certain risks in the supply chain, many challenges in this emerging threat landscape remain unresolved as demonstrated with the XZ incident. Established practices involve vulnerability scans, Software Composition Analysis (SCA), or manual auditing of the respective component and its source code. While auditing third party components before integrating them in a product is valuable, it is often impractical to (thoroughly) audit every component or update.

Our objective is to demonstrate how software development teams can leverage such a tool to prioritize their (ideally regular) audits and assess third party repositories, following the five key takeaways outlined in section 6.3. First, we explain how *SketchyCrawler* addresses these takeaways. Then, we illustrate some of the information gathered by the first prototype. Finally, we discuss relevant deployment considerations including our assessment of the results.

6.4.1 Implementation

SketchyCrawler is a set of Python and Shell scripts used to crawl GitHub repositories. Currently, users of our tool simply need to provide their crawling targets and some additional information in a CSV file to collect relevant assessment information automatically. This includes the repository URL, owner, repository name, the time range for the analysis, currently trusted maintainers, potential sketchy files and types, and the package name. Below we describe the main features of *SketchyCrawler*:

Signs of social engineering (TA-1, cf. section 6.3.1). Trust is essential not only within the open-source community but also from a supply chain perspective when using third party components in software projects. In case of XZ, the primary objective of the adversary was, as outlined in section 6.2.1, to build sufficient trust to gain commit permissions within the open-source project. Based on this, one feature of *SketchyCrawler* is about detection of new maintainers that are not marked as trustworthy yet. Since auditing every update of the incorporated third party components requires significant effort, we believe that focusing on major changes (e.g., change of maintainers) offers a valuable advantage. Using a third party component requires some sort of trust in the maintainer anyway. However, when the previously trusted maintainers change, it may be a sign for a necessary re-audit of this component to ensure valid and conscious trust assumptions.

Reverse dependency analysis (TA-2, cf. section 6.3.2). *SketchyCrawler* also includes a module for analyzing a package's reverse dependencies required at

runtime. The main objective here is to highlight packages with a high number of reverse dependencies as these can increase the attack surface. In case of XZ, the adversaries planted a vulnerability into SSH servers not by compromising OpenSSH directly, but one of its (indirect) dependencies. The reverse dependency analysis is recursive so we also count the dependencies of the dependencies, and so on.

Forward dependency analysis (TA-3, cf. section 6.3.2). The third feature is similar to the previous one, but addresses forward dependencies at runtime and build time. From an adversary point of view, forward dependencies may be interesting for critical software projects (such as OpenSSH) as the attack surface increases for every dependency that could be vulnerable too. We showcase the analysis of build dependencies by leveraging the *cargo tree* command to count the dependencies recursively. However, *SketchyCrawler* currently focuses on forward package dependencies and does not yet detect dependencies that are dynamically loaded during runtime, e.g., through `dlopen`.[2]

Identify bugdoors (TA-4, cf. section 6.3.3). Our fourth takeaway addresses the potential for bugdoors in open-source projects. Thus, *SketchyCrawler* incorporates a module that allows defining a list of potential sketchy files and crawls the repository if those files exist. A good example is the `build.rs` file used in various Rust packages; legitimately, this can, e.g., be used to compile third-party C libraries before building the actual Rust package [141], but potentially for attacking the build process or host itself. Rust projects with a `build.rs` file cause Cargo to execute its content before building the Rust package. A report published by the Phylum Research team [126], dedicated to identifying software supply chain attacks, highlights an attack targeting Rust developers. They observed a case where an adversary initially tried to typosquat a package (e.g., name it postgress instead of postgres) containing almost no code. This was followed by a series of small, seemingly non-suspicious code changes. However, at some point the adversary included a `build.rs` file that was used to establish a communication channel to the adversary. This example illustrates why Rust projects using a `build.rs` file may require a closer look.

Additionally, we check whether sketchy files have been deliberately added to the `.gitignore` file. However, this specific crawling feature only highlights files that are not expected in the `.gitignore` file and adds a more critical sign when combined with the results of TA-5 (e.g., hidden file is part of the source tarball).

Signs of obfuscation (TA-5, cf. section 6.3.4). In this category we try to identify sketchy mime types that may reveal attempts to obfuscate code (e.g., by incorporating binary blobs). A possible example for a mime-type is *application* that is typically used for a binary file rather than a text file. Another indication of obfuscation is the presence of discrepancies between the actual source code

[2] Work is independently underway to improve tooling for these dynamically loaded dependencies to be listed in the ELF header (cf. https://github.com/systemd/systemd/blob/main/docs/ELF_DLOPEN_METADATA.md). Future versions of *SketchyCrawler* should use this data when more broadly available.

repository and the published source tarball. In case of XZ, the adversaries tried to hide the malicious `build-to-host.m4` file by adding it to the `.gitignore` file. However, the source tarball, used to build the actual package, contained the malicious file.

6.4.2 Evaluation

We evaluate *SketchyCrawler* by crawling GitHub repositories of Debian packages to identify sketchy signs for a potential backdoor. To select our test examples, we use Debian PopCon,[3] a list of the most popular Debian packages. Currently, we support open-source projects hosted on GitHub only, but we believe that with some engineering effort, integrating additional version control systems would be straightforward. We select the top-three packages on Debian PopCon with the most installations that are also available on GitHub. This results in `libpam-modules`, `libblkid1`, and `zlib1g`. Although `liblzma5`, with 250,211 installations, ranked 95th out of 200,909 packages only, we included it in our evaluation to discuss the findings in case of XZ. Additionally, we also include two packages written in Rust in our evaluation as this ecosystem focuses on secure software development. We crawl 6,369 packages to find at least two Rust packages—`mdevctl` on rank 4,097 and `ripgrep` on rank 6,369. The final selection of packages can be found in Table 6.2. The crawler results are illustrated in Table 6.3. TA-3 includes the forward package- and build dependencies in the format x/y, where x represents the number of forward package dependencies and y represents the number of build dependencies. TA-5 includes the number of sketchy file types and differences in the format x/y, where x represents the number of detected sketchy file types and y represents the number of differences between the source tarball and the repository for a given release tag. For simplicity, and to provide an initial indication of sketchy signals, we display only the number of differences that occur in either the source tarball or the repository. The specific list of differences can be printed to a log output. A full run, including fetching commits within the given time range, cloning the repository, performing the dependency-, file-, and file type analysis including the differences between the source tarball and the repository took 503.70 seconds on an Intel Core i7-1185G7 @ 3.00GHz CPU with 32GiB of RAM.

6.4.3 Discussion

We believe that tools such as *SketchyCrawler* can be utilized as part of a risk management process to gather metrics about sketchy findings and include them in further risk assessments. For example, a package maintained by many developers is likely more difficult to verify than one with a single maintainer (e.g., `zlib1g` or `mdevctl`), and can therefore be assigned to a higher risk category. In Table 6.3, `liblzma5`—relevant in case of XZ—is listed with 5 commits made by an untrusted maintainer, specifically Jia Tan, the adversary. Additionally, *SketchyCrawler* highlights packages with a high number of reverse dependencies (e.g., `liblzma5` and `zlib1g`), which may be a valuable target for an adversary, as demonstrated in the XZ case. Next, we identify packages with a high number of forward dependencies, which may increase the likelihood of a

[3] https://popcon.debian.org/main/by_inst

Table 6.2: Selected Debian packages. Taken from our paper [97].

Package	# Inst.	Since	Until	Trusted maintainer	Sketchy files	File types
liblzma5	250150	2022-01-01	2023-01-02	Larhzu, web-flow	configure.ac	app.
libpam	251820	2022-01-01	2023-01-02	ldv-alt, thkukuk	configure.ac	app.
libblkid1	251817	2022-01-01	2023-01-02	karelzak	configure.ac	app.
zlib1g	251815	2022-01-01	2023-01-02	madler	configure.ac	app.
mdevctl	14072	2024-01-01	2025-01-02	jonner	build.rs	app.
ripgrep	6167	2024-01-01	2025-01-02	BurntSushi	build.rs	app.

Table 6.3: *SketchyCrawler* results. Taken from our paper [97].

Package	TA-1	TA-2	TA-3	TA-4	TA-5
liblzma	5	33436	4/80	1	99/38
libpam	8	5704	32/74	2	199/28
libblkid1	26	14448	4/85	3	422/45
zlib1g	0	41461	4/75	5	650/60
mdevctl	0	1	53/87	1	758/14
ripgrep	21	1	5/84	3	869/25

vulnerability. Currently, dynamically loaded dependencies during runtime are beyond the scope of the current implementation. However, there are already discussions [4, 127] within the open-source community going on concerning dependencies loaded dynamically by `dlopen()`. One potential risk mitigation—particularly on projects with a high number of dependencies—is that software developers may consider additional security controls before publicly exposing such components. As expected, the number of sketchy files represented in TA-4 is low, since we use only the `configure.ac` and `build.rs` files to showcase the functionality of our crawler. However, in projects like XZ, where sketchy files such as `build-to-host.m4` are included in the `.gitignore` file, we expect to see an increased number here. TA-5 illustrates sketchy types, especially binary files with respect to our sample set. This can certainly be legitimate; however, it is a sign that there is also a lot of potential to obfuscate malicious code such as a backdoor. Finally, TA-5 also highlights the number of differences between the source tarball and the actual repository for a specific tag. While the results show significant differences, most of the files, such as the `.gitignore` and the `.git` directory, are expected not to be part of the source tarball.

6.4.4 Deployment Consideration

Typically, Secure Software Development Life Cycles involve license clearing and component approval before third party libraries are integrated into a software product. As a result, software development teams may audit those third party components before adding them. However, since those components are often updated over time, it can be challenging for developers to audit every new version. We believe that integrating a tool such as *SketchyCrawler* into a CI/CD pipeline (section 2.2.1) to identify sketchy signs allows development teams to focus first on critical parts in their supply chain. Particularly within the Rust ecosystem with its focus on secure software development, future integration with development team processes using the `cargo vet`[4] and/or `cargo geiger`[5] tools might be helpful to better manage audit resources.

6.5 Related Work

The challenge of describing different supply chain attacks to develop effective mitigation techniques has been explored in several academic papers. Naik et al. [113] compare a range of various attacker models on a more abstract level. Several works focus on generalizing security frameworks (e.g., MITRE ATT&CK [24] or the Diamond Model [16]). Additionally, there is also research [67, 122] focusing on analyzing malicious software packages in general. Moreover, specific research efforts have investigated particular attacks, such as the SolarWinds attack presented by Martínez and Durán [104]. The variety of research tracks in the field of software supply chain security highlights the difficulty in mitigating such attacks properly and the XZ incident serves as a clear example that we still face open issues. Specifically concerning the XZ incident,

[4] https://mozilla.github.io/cargo-vet/
[5] https://github.com/geiger-rs/cargo-geiger

there are existing attempts to counter such attacks, such as differential property monitoring [14] or a fuzzing approach to detect backdoor triggers at runtime [87]. However, in this chapter we investigate emerging supply chain attacks such as the XZ incident, aiming to identify new methods in every stage by which adversaries can still successfully implant a backdoor in an open-source project. In comparison with existing research, we outline a new critical attack path including corresponding mitigation techniques to support mitigation for similar attacks in the future. We abstract the critical attack path observed in the XZ incident, and believe this attack strategy could be applied to analogous supply chain attacks. During the course of our work, Hamer et al. [67] also conducted a study providing recommendations for preventing such supply chain attacks. However, their perspective focuses more on existing frameworks—suggesting the use of starter kits, improvements to current frameworks, and continued research into threat intelligence. In contrast, we analyze the critical attack path of a software supply chain attack based on new attacker strategies used in the XZ incident, highlight the main key takeaways, and provide a tool prototype to showcase how sketchy signs in open-source projects can be identified.

6.6 Availability

We provide an open-source repository of *SketchyCrawler* on `https://github.com/linsm/sketchy-crawler`.

6.7 Summary

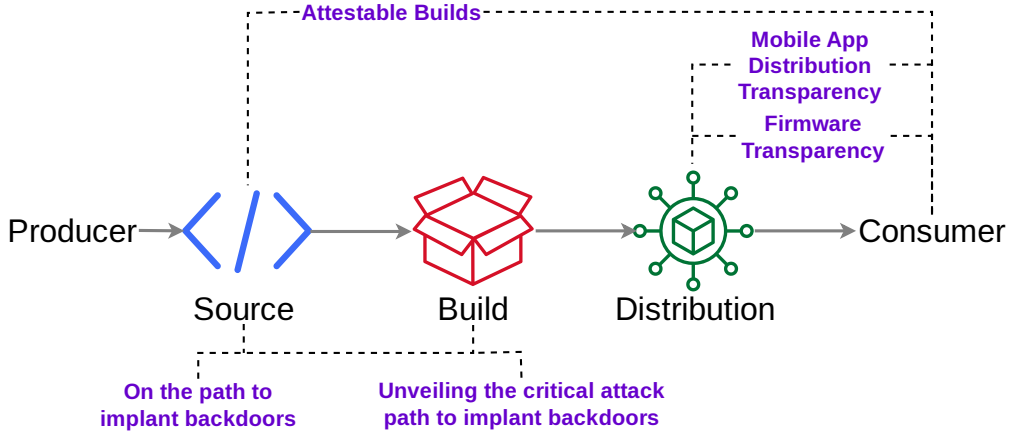
The XZ Utils backdoor is another example highlighting the severity of supply-chain attacks, particularly given their widespread opportunity for adversaries. A single vulnerable component within a supply chain can lead to a critical security incident. Additionally, the XZ case demonstrates the significant resources—especially the time constraints—adversaries are nowadays willing to invest to achieve their objectives. Considering the attack complexity and the very long timeframe—from initial trust-building to backdoor implantation in an open-source project—we speculate that the threat actor(s) involved required stable funding.

In this chapter, we summarized what we define as the *critical attack path* for this specific supply chain attack. We described the critical elements of each stage in this path that enabled the final backdoor implantation, despite its early detection. Based on the learnings, we derived five generic key takeaways that can be used to mitigate similar attacks, especially for open-source projects. A particularly important aspect is the usage of old build toolchains, which provides an extended attack surface and allow adversaries to maintain plausible deniability through the incorporation of "bugdoors" that can later be exploited. Furthermore, we showed how a tool-based approach can support development teams in identifying emerging threat indicators within the software supply chain. Our prototype tool, *SketchyCrawler*, demonstrates how such a solution can be implemented to reveal specific signs that development teams should investigate before integrating the specific third-party library in their codebase.

The XZ case provides an ideal source for educational material and to learn how we can proactively strengthen our supply chains. As a security community, it is our responsibility to detect such compromise attempts, and to continue observing, monitoring, and learning from such attacks to prevent future incidents. We hope that *SketchyCrawler* serves as a prototype to assist others in addressing this difficult challenge.

Chapter 7

In Cloud We Trust? Verifiable Strong Source-to-Binary Artifacts



Paper Reference. This chapter is based on the paper "Attestable Builds: Compiling Verifiable Binaries on Untrusted Systems using Trusted Execution Environments" [71].

This chapter introduces attestable builds (A-Bs), a new approach for establishing a verifiable trust link between source code and the resulting software artifact. In a typical build process, the inputs are source code, build configurations, and potentially additional dependencies, while the output is a binary—a digital and opaque black box. Although source code can often be inspected—whether because it is open source and publicly accessible or, in commercial settings, via internal reviewers or external auditors—the final artifact is rarely verifiable directly. Moreover, verifying source code and rebuilding from scratch requires specialized expertise, computing resources, and time. A-Bs close this gap by providing cryptographic evidence that a specific artifact was truthfully produced from a particular source code snapshot.

In 2020, adversaries successfully compromised the build infrastructure of SolarWinds that allows them to implant malicious code during the build process of their widely deployed network-management software, without modifying the source code directly. Since the adversaries did not make any changes on the repository itself, the malicious change could not be detected by just reviewing the source code, even after the attack became known; only forensic analysis revealed that the tampering occurred within the build environment itself. More

recently, the XZ Utils backdoor demonstrated a different but also related attack vector: in brief, as detailed in chapter 6, adversaries implanted a backdoor into the release tarball—removed from the actual repository—so it got injected during the build process without being obvious in the source code itself. Both cases illustrate the importance of a secure build system and more important the ability for verification whether the input really corresponds to the output.

A modern approach to tackle this challenge is making the build reproducible (i.e. rebuilding the software leads to a bit-per-bit identical binary). Although this might seem to be the ideal solution for this challenge, practice revealed that reproducible builds are often time- and resource intensive to achieve and more importantly costly to sustain. Moreover, the verification of reproducibility is also often not feasible for end users—particularly for closed-source software or when the toolchain and build process are not accessible due to licensing, for example. In practice, these barriers limit broad adoption and prevent consumers from verifying their artifacts. Another aspect is that today's software development teams often use untrusted build services that are running on an untrusted infrastructure of a CSP. This is the point where A-Bs can bridge the gaps by providing strong source-to-binary correspondence even when reproducible builds are not feasible or just unavailable. A-Bs complement reproducible builds by allowing consumers to verify that a specific artifact was produced from a specific source snapshot without the need to rebuild it on their own. We tackle this challenge by utilizing TEEs and sandboxing technologies to generate verifiable build attestations. Although TEEs are typically used to ensure confidentiality, we emphasize their integrity guarantees.

Under the hood, A-Bs starts the build process by initiating a new instance running an open-source image that is booted within a TEE. Another sandbox running inside the TEE continues by downloading the source code and stores a hash of the downloaded files in the isolated environment provided by the TEE that cannot be manipulated during the build process itself before running the potentially malicious build scripts. Once the build is completed, the TEE is used to perform the attestation, covering the underlying image, the hash value of the downloaded source files, and the final build artifact. The resulting attestation certificate is then shared to the consumer and in addition logged in a transparency log (section 2.6). The consumer that gets the artifact and the attestation certificate is now able to verify that the artifact has been truthfully built from a particular source code snapshot in a trustworthy environment. An important key element of this design is the utilization of nested sandboxing because of limitations inside the secure enclave (e.g., AMD SEV-SNP). This is relevant as the Confidential Computing technology provides confidentiality and integrity guarantees to protect the enclave from unauthorized external access between the infrastructure provider (e.g., the hypervisor) and the VM running inside the secure enclave. However, this technology does not provide protection at run time—an essential requirement for A-Bs—as it is not restricted what kind of code (potentially malicious) is executed in the build environment. Therefore, we utilize nested sandboxing that allows us to run the actual potentially malicious build step within the enclave while protecting the integrity of the output artifact. These properties can only be achieved by using both the TEE technology and sandboxing in combination.

Additionally, we also demonstrate how A-Bs and R-Bs can be used together to achieve new security properties by performing the R-B as an A-B running

on different TEE vendors. This distributes the trust anchor (i.e. the TEE hardware) among different vendors and the end consumer only needs to trust one of them—this paradigm is also called any-trust model. Our proposal also contributes to a better understanding of R-B setups as the consumer of an R-B artifact usually cannot verify the underlying environment and thus also cannot ensure whether there are the same vulnerabilities in there. Additionally this chapter also shows the formal model and verification of our protocol using `Tamarin`. The following list is a comprehensive overview of the objectives in this chapter:

1. We introduce a new approach called Attestable Builds (A-Bs) to provide strong source-to-binary correspondence by combining modern TEE and sandboxing technologies.
2. We demonstrate how combining A-Bs and R-Bs strengthens provenance in any-trust models.
3. We demonstrate practicability and feasibility of our architecture by implementing an open-source prototype used to build existing open-source projects, such as Clang, Linux Kernel, and packages that are challenging to reproduce at all.
4. We evaluate A-Bs by running performance tests, which resulted in 42 seconds start-up cost, which is negligible compared to the overall build duration of a software artifact. We show that the performance overhead is just around 14 % and 62 % when using a specific hardened version.
5. We formally model and verify the underlying protocol of A-Bs using `Tamarin`.

7.1 Architecture

The primary scope of A-Bs focuses on cloud-based CI/CD pipelines that utilize the build systems of BSPs who may leverage untrusted external infrastructure form CSPs (see stakeholders in section 2.1.1). A key technology used in our proposal is Confidential Computing (section 2.3); however, our design is technology-agnostic and does not rely on a specific hardware implementation (e.g., AMD SEV-SNP). It is designed to be compliant with various technology providers. Figure 7.1 illustrates the architecture of A-Bs including its individual protocol steps (1)-(11). We detail these steps in the following section.

At a high level, the A-Bs approach begins the build process with a request for a new build (1). This request is submitted to an Instance Manager operated by the host instance, which then initiates a new secure enclave (2) within the TEE of the specific confidential computing hardware (e.g., AWS Nitro, AMD SEV-SNP).

The TEE provides necessary security guarantees, such as confidentiality (not relevant for our threat model) and integrity, by leveraging hardware-backed security technology. This mitigates the risk of even insider attackers (A3)—with access to the hypervisor and direct hardware access—compromising the untrusted external cloud infrastructure, preventing them from reading or modifying code and data running within the TEE. To enable attestation and

subsequently verifiability of the build process and its environment, the secure enclave employs remote attestation to guarantee that is based on a known image and that it is running within a TEE. Implementing this approach mitigates T-B4.

However, this build process remains vulnerable; an adversary controlling the external source code executed within the enclave could still manipulate the `internal state` used for attestation. To mitigate this vulnerability, we introduce the Enclave Client, an integrity-protected observer of the internal state. This observer communicates with a dedicated sandbox running within the secure enclave, isolating the build process form the internal state information.

Upon receiving a build request (1), the secure enclave boots (2) from a publicly known image, after which the Enclave Client is initialized (3). From a security perspective, the Enclave Client executes within the TEE, ensuring its integrity. Subsequently, the Enclave Client establishes communication—facilitated through shared memory—with the Instance Manager running on the host system. To retrieve source code from the external RHP, the instance manager shares a short-lived token with the Enclave Client and receives build process updates.

Following initialization and token authentication, the Enclave Client launches the sandbox (3) within the secure enclave. This sandbox is critical for isolating the build process and preventing potential tampering with the internal state protected by the Enclave Client. This state represents the initial measurement of the source code and build instructions. By protecting and verifying this stage, we mitigate T-B2. Furthermore, the sandbox captures logs of all incoming and outgoing communication, enabling enhanced logging and auditing of the build process.

After the sandbox has fully started, the Enclave Client passes the authentication token to the build running inside the sandbox. The build runner can then fetch the source code (4) and build instructions from the external repository using the authentication token. This communication is tunneled via shared memory, as the enclave itself has no direct Internet access (or any other TCP/IP communication). Based on the downloaded source code and build instructions, the sandbox calculates the corresponding commit hash (CT) and passes it to the Enclave Client. This hash includes additional metadata, such as commit messages, which can also contain developer signatures. This information is critical for verifying the provenance of the source code and mitigates T-B3.

Subsequently, the commit hash CT is secured by the Enclave Client (5) within the TEE. At this stage, A-Bs can safely execute the untrusted and potentially malicious build instructions inside the sandbox without compromising the integrity of the initial secure state. From the moment the build instructions are executed, we consider the internal state within the sandbox as untrustworthy as code could tamper it easily.

Once the build process completes, the sandbox reports the path of the resulting artifact, computes the hash A of the artifact, and provides this information to the Enclave Client (6) for later user verification.

After finishing the build and collecting all relevant information, the Enclave Client initiates attestation from the TEE, requesting an attestation document AT (7) that encompasses the PCR values of the booted image (including the Enclave Client and the sandbox), the initial measurement of the source code and

build instructions CT, and a hash of the final build artifact A. The Enclave Client receives AT form the TEE and shares it with the sandbox (8) and the Instance Manager (8), enabling the sandbox to include it with the artifact distribution (9) and the Instance Manager to publish it to a transparency log (9). This cryptographically verifiable information allows users to verify the provenance of the artifact, mitigating T-U3.

We incorporate a transparency log to support revocation possibilities, such as revoking trust in attestation information from a TEE environment vulnerable to hardware attacks. The user now receives (10) the artifact, the artifact hash, and the attestation document. These components, along with the inclusion proof from the transparency log (11), which provides an up-to-date state of the distributed artifact, allows the verification of the source-to-binary correspondence and thus supports mitigating T-B4 and T-S1. Furthermore, developers can monitor the transparency log for leaked signing keys T-D1. A full end-to-end verification requires a transparency log entry for every build artifact. If an adversary attempts to distribute an artifact signed with a compromised or lost signing key, a developer monitoring the transparency log would detect this, mitigating T-S2.

An important design decision to mitigate T-B2, and T-B4 is to make the build process itself stateless by destroying the enclave after each build. This results in a hermetic build, i.e. the build does not rely on external information, as the input and instructions are clearly defined.

7.1.1 Composing A-Bs and R-Bs

Combining A-Bs and traditional R-Bs can enhance security and trust of various aspects of build processes. One benefit is that artifacts produces by A-Bs can be integrated into R-B setups by committing to a specific hash of those artifacts—a technique analogous to lockfiles used in dependency managers like NPM or Cargo Rust. Conversely, R-B artifacts can also be leveraged within A-B workflows. Furthermore, an R-B project utilizing an A-B build environment may reduce the number of independent R-B builders required. This synergy allows R-B projects to utilize independent A-B builders operating on different Confidential Computing hardware (see Figure 7.2).

This combined approach facilitates an *any-trust* model, enabling users to verify build integrity and authenticity by trusting only one single Confidential Computing manufacturer, regardless of which one.

A critical trust anchor in the A-Bs environment is the build image, which users can verify by comparing its PCR values. We believe that employing R-Bs to build the initial image for an A-B ecosystem improves bootstrapping and verifiability. In a real-world implementation, we recommend that this initial image is reproducible, based on as little code as possible to make it easy to inspect and to limit the attack surface. Projects like Bootstrappable Build [130] provide more information and suggestions in that regard.

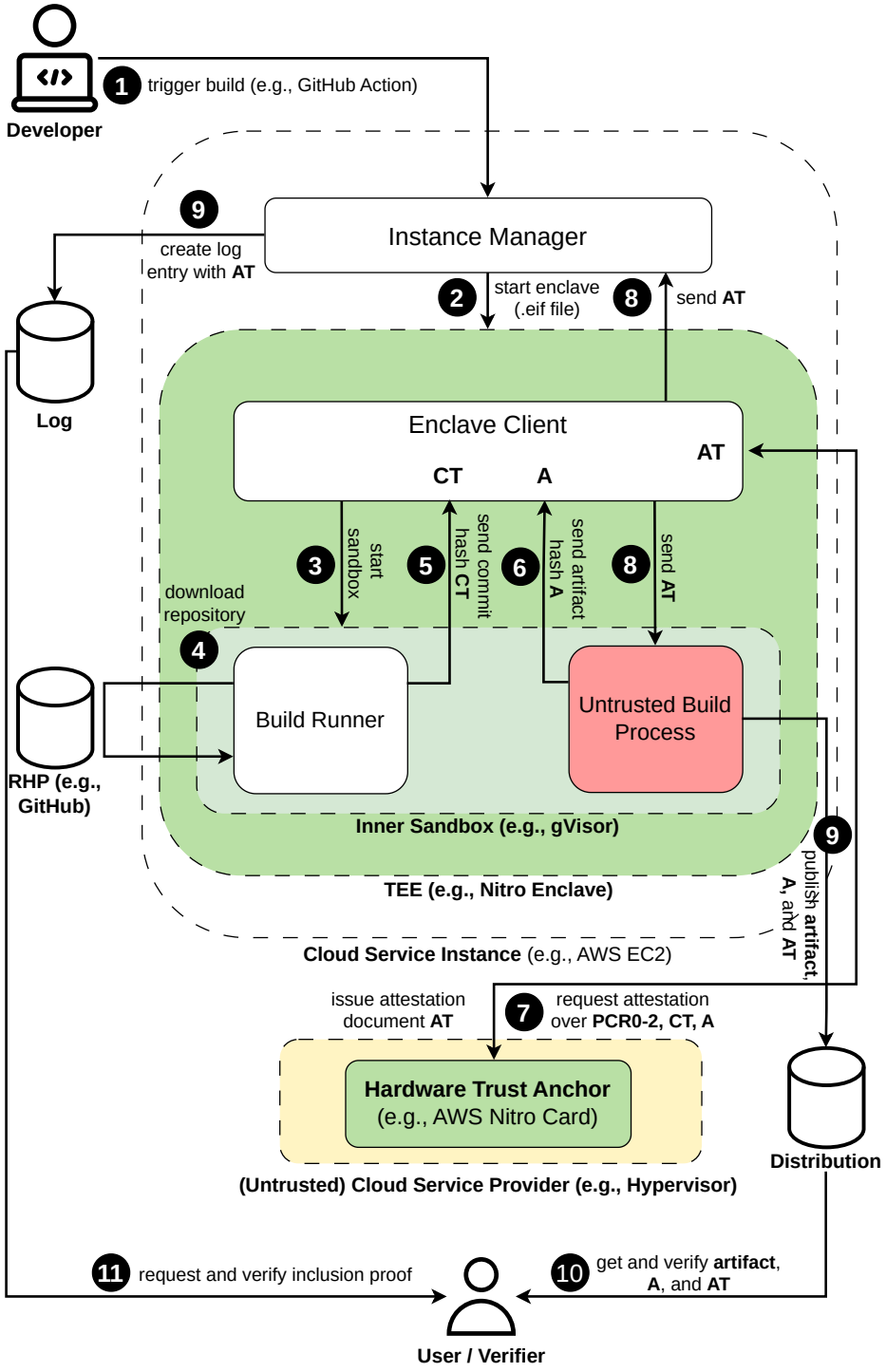


Figure 7.1: Architecture of A-Bs including its individual protocol steps. Adapted from our paper [72].

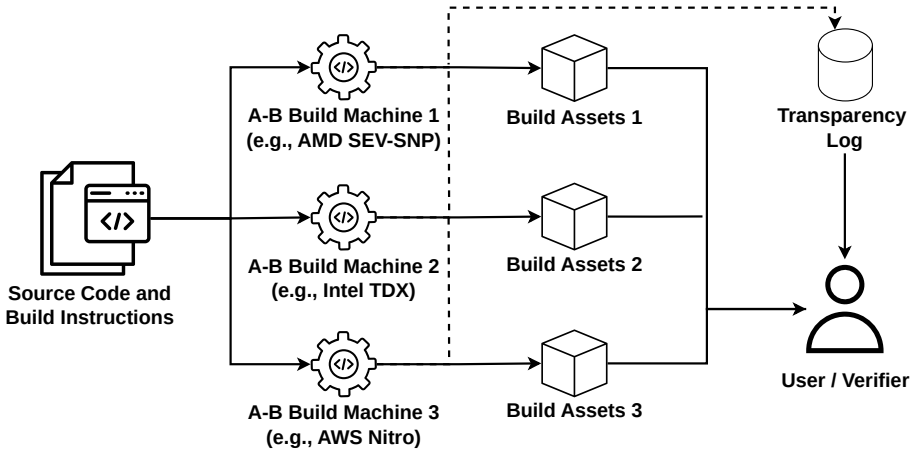


Figure 7.2: A-Bs used with different environments. Leveraging different A-B environments (e.g., AMD SEV-SNP, Intel TDX, AWS Nitro) can be used in R-B setups to build identical artifacts. Users need to trust that only one of the TEEs has not been backdoored. Adapted from our paper [72].

7.1.2 Build Dependencies

An essential aspect of trust in build processes is the underlying Trusted Computing Base (TCB). The TCB describes the environment and components involved in the build process, including build dependencies, toolchains, software libraries, the TEE design, it's underlying firmware, the base image used to build the artifact, and software modules of A-Bs such as the Enclave Client and the Instance Manager.

In our prototype, we incorporated the compiler toolchains (e.g., Rust Cargo) into the base image to enable verification via PCR0 measurement. However, in a production environment, the build image can also be modularized, for example, by moving the compiler toolchain to the build configuration.

A-Bs further support tracking known CVEs associated with TCB components through integrating a Software Bill of Material (SBOM) in the attestation data. This approach can be implemented by extending the in-toto standard [41].

The process of A-Bs itself is not necessarily hermetic, i.e. being executed without Internet access. To mitigate T-B3, it is sufficient to pin dependencies using tools such as *Cargo.lock* in Rust. Nevertheless, hermetic builds (and availability) can be supported by following best practices such as copying the external source code and other relevant assets of the dependency into the own repository. A-Bs is compliant with both scenarios—pinned dependencies and hermetic builds.

7.1.3 Attacks on TEEs and Their Impact on A-Bs

The TEE is the most critical trust anchor in the design of A-Bs. While recent attacks, including breaking of integrity guarantees, have been demonstrated against established Confidential Computing hardware such as AMD SEV-SNP,

manufacturers typically address such vulnerabilities rapidly through firmware patching. A-Bs allows users to verify whether an artifact was built using a vulnerable firmware version, as this information is included in the attestation measurements. Currently, we are not aware of any publicly known attacks targeting the TEE used in our prototype (AWS Nitro), although this may be due to limited researcher access to Nitro cards.

A-Bs primarily rely on integrity guarantees and most importantly the remote attestation capabilities of TEEs; confidentiality is not necessarily a requirement in scope of build processes within this thesis. This is important because integrity attacks are inherently active, which reduces the window of opportunity for an adversary significantly.

An advantage of using A-Bs is that artifacts built inside a vulnerable TEE can be revoked and rebuilt with a patched firmware. However, this still requires that the remote attestation, including the firmware version, remaining unforgeable. Ultimately, if an adversary completely compromises a TEE—for example, exfiltration of the private keys stored within—artifacts built on those compromised build servers can be revoked and rebuilt using a newer TEE generation.

7.2 Practical Evaluation

We demonstrate the practicability and feasibility of our A-Bs design by running performance tests on various real-world open-source projects.

7.2.1 Prototype Implementation

This section provides an overview of the most relevant aspects of our prototype implementation.

TEE. We implemented our prototype using AWS Nitro Enclaves, as AWS provided accessible tooling at the time of writing. Nevertheless, the A-Bs concept is applicable to other Confidential Computing technologies—such as AMD SEV-SNP—and requires engineering effort for adaption. AWS Nitro Enclaves run in dedicated Nitro Cards, providing hardware-backed isolation from the infrastructure provider, the hypervisor, and other tenants. This isolation is achieved through the assignment of dedicated resources, including CPU cores and memory, exclusively to each enclave.

Enclave. The enclave, booted with the TEE, is based on a Docker image converted to an .eif file, which includes PCR0-2 measurements used to verify the booted image. Because direct access to hardware components—such as network interfaces—is unavailable within the enclave, necessary communication is tunneled via *vsock* through shared memory to the hypervisor. This is particularly important for exchanging information (e.g., the attestation document) between the Instance Manager (running on the hypervisor) and the Enclave Client (running within the enclave) and for enabling the build runner to access the Internet to retrieve source code from the RHP (e.g., GitHub).

Sandbox. Within the secure enclave, we employ sandboxing technology to protect the initial state of the build runner from unauthorized modifications by build instructions. We demonstrate the compatibility of A-Bs with various sandboxing technologies, including *containerd* and the security-hardened *gVisor*. Parameters are provided to the sandbox via environment variables (e.g., RHP access token), and Linux network namespaces are used to handle TCP/IP communication.

Build trigger. To demonstrate the integration of A-Bs into real-world environments, we leveraged GitHub Actions through webhooks connected to the Instance Manager. Furthermore, we integrated the official GitHub Action Runner within our sandbox.

Programming language. Aside from some supporting scripts (e.g., Python, shell), we implemented the majority of the A-Bs components in Rust. This enables us to leverage features such as their Typestate Pattern [10] to mitigate logic errors and, more generally, reduce the attack surface due to Rust's memory safety.

7.2.2 Build Targets

To demonstrate the applicability of A-Bs' design to real-world projects, we selected open-source projects with challenging aspects from a building perspective. We began with five Debian packages that remain unreproducible. To identify these packages, we generated a list of all unreproducible packages and then chose those with the fewest dependencies—at least two per package. Subsequently, we sorted the resulting list based on reverse dependencies, as these can significantly impact the overall build graph. Additionally, we included one package with a larger number of dependencies to broaden our evaluation. Ultimately, our final selection comprised *ipxe*, *hello*, *gprolog*, *scheme48*, and *NeoVIM*.

Our second objective was to demonstrate the feasibility of A-Bs for building large software projects, including the *Linux Kernel* and *LLVM Clang*. These targets are well-suited to our evaluation, as they allow us to demonstrate A-Bs' ability to handle complex build configurations and are additionally both relevant for bootstrapping the base image.

Our third selection comprises projects relevant either to the build toolchain or of particular security perspective. In this category, we included *tinyCC*, *libsodium* (a well-known cryptographic library), *XZ-Utils* (see chapter 6), and our own verifier client to demonstrate A-Bs' capability to build its own components.

7.2.3 Measurement Results

This section presents our measurement results of the impact on build durations when utilizing A-Bs within a CI pipeline. For the majority of our builds, we employed the *m5a.2xlarge* EC2 instance, equipped with 8 vCPUs and 32 GiB

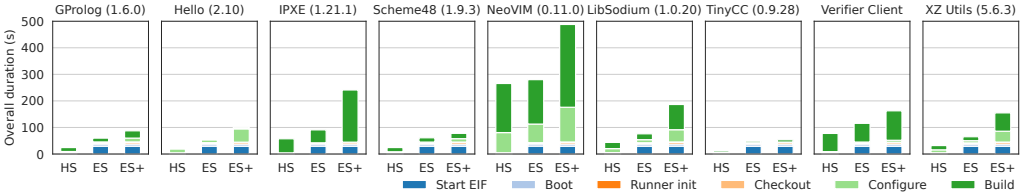


Figure 7.3: Duration of each build stage. The duration was measured for selected open-source projects using three build configurations: a sandbox directly on the host (HS); a sandbox (containerd) running within a secure enclave (ES); and a hardened sandbox (gVisor) running with a secure enclave (ES+). Taken from our paper [71].

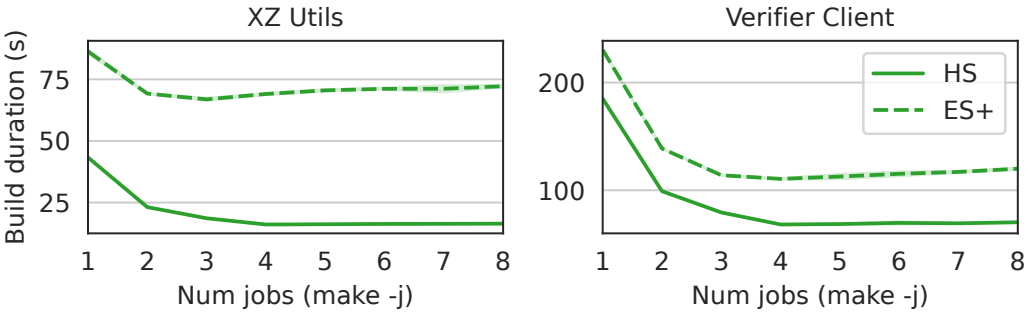


Figure 7.4: Impact of job count on build duration for `make -j` (left) and `cargo build -j` (right) using four CPUs. Taken from our paper [71].

of RAM. However, for the larger projects—the Linux Kernel and LLVM Clang—we utilized a larger machine, the *m5a.8xlarge* EC2 instance with 32 vCPUs and 128 GiB of RAM. To ensure a fair comparison between enclave and non-enclave execution, the enclave was allocated half of the instance's CPU and memory resources. From a cost perspective, the *m5a.2xlarge* EC2 instance cost approximately $0.32 per hour at the time of performing the performance tests for our paper [71]. Additional charges were incurred due to increased storage limits (1000 MiB/s and 10,000 operations/s) to mitigate potential I/O bottlenecks.

The experiments were executed three times with varying configurations. We employed the host-sandbox (HS) configuration, which utilized the GitHub Runner with *containerd*, executing directly on the host machine and primarily representing a self-hosted build server scenario. We then implemented two further configurations: the enclave-sandbox (ES) configuration, using a stan-

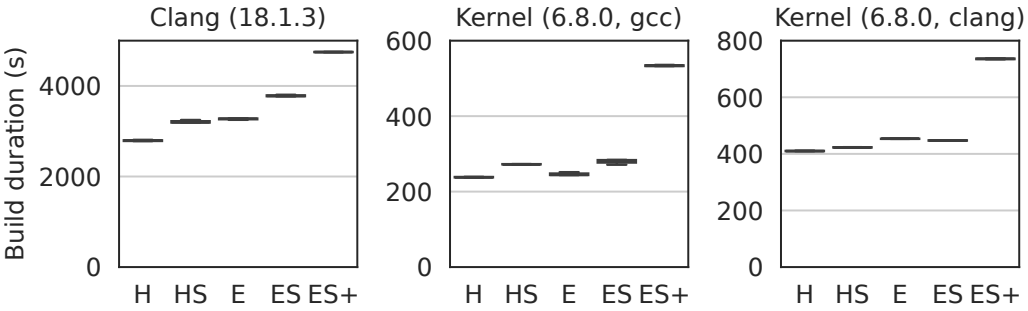


Figure 7.5: Build duration for the complex targets *clang* and *kernel* without sandboxes on the host *H* and enclave *E*. Taken from our paper [71].

dard *containerd* sandbox, and the enclave-sandbox-plus (ES+) configuration, utilizing a hardened variant, *gVisor*, both running inside a secure enclave. For the two larger projects—the *Linux Kernel* and the *LLVM Clang* project—we also included two configurations without a sandbox: building on the host without a sandbox (H), and building within an enclave without a sandbox (E).

To quantify the impact on build duration when utilizing A-Bs within a CI pipeline, we defined several measurement time points: (1) *Start EIF*: initializing the TEE and loading the .eif file of the base image into the enclave's assigned memory; (2) *Boot*: booting the base image; (3) *Runner Init*: connecting to RHP (e.g., GitHub); (4) *Checkout*: downloading source code and build instructions; (5) *Configure*: performing the build file configuration step; (6) *Build*: compiling the artifact. We calculated the average build time across each combination of configuration (e.g., host-sandbox) and build target (e.g., *libsodium*).

In Figure 7.3, we present the overall build durations for the selected targets (section 7.2.2) across all three configurations. We observe that for smaller builds (e.g., GProlog), a significant portion of the overall build time is spent initializing and booting the secure enclave, which typically takes around 37.6 seconds for our 1,474 MiB image file. We determined that enclave startup time could be improved through pre-warming. Further observation indicates that small targets have a reduced build time when executed within an enclave (ES) compared to building them directly on the host (HS). For instance, the *NeoVIM* build duration decreases from 184.9 s to 167.3 s. We hypothesize that this performance gain is due to the enclave's entirely in-memory execution, thereby avoiding I/O limitations. However, utilizing the hardened version with *gVisor* introduces a substantial impact on build time. For example, the *NeoVIM* build time increases by 62 % when using *gVisor* compared to direct host execution (HS).

The time spent initializing the build runner and downloading source code is negligible, averaging approximately 9 seconds overall. Despite TCP/IP traffic being tunneled via *vsock* through shared memory in case of building the targets inside an enclave (ES), we observe no significant performance loss compared to the host-based approach (HS). In some instances, the checkout step for large projects, such as *LLVM Clang*, has a noticeable difference in duration when comparing host-sandbox mode (148.0 s) and enclave-sandbox mode (117.2 s). We hypothesize that I/O limitations in HS mode may be a reason for this performance gain in ES mode. Regarding *gVisor*, we also observe overhead during the checkout process. This overhead is up to 2 seconds for small targets and increases from 117.2 s (ES) to 132.8 s (ES+) for large targets like *LLVM Clang*.

We further investigated potential methods for reducing build durations in *gVisor* (ES+) configurations. We determined that employing parallelized builds (e.g., the *-j* argument for *make*) improves build times. The optimal degree of parallelism closely aligns with the number of available CPUs (see Figure 7.4). However, while increasing the number of parallel builds enhances host-based build times (HS), it negatively impacts the *gVisor* configuration (ES+).

Next, we evaluated build durations for the larger projects, the *Linux Kernel* and *LLVM Clang* (see Figure 7.5). As we allocated half of the available resources to the enclave, these two build instances were executed with 16 vCPUs and 64 GiB of RAM. We observed that increasing memory allocation resulted in an extended *Start* duration for the base image, from 29.5 s to 46.4 s. Additionally, the overall

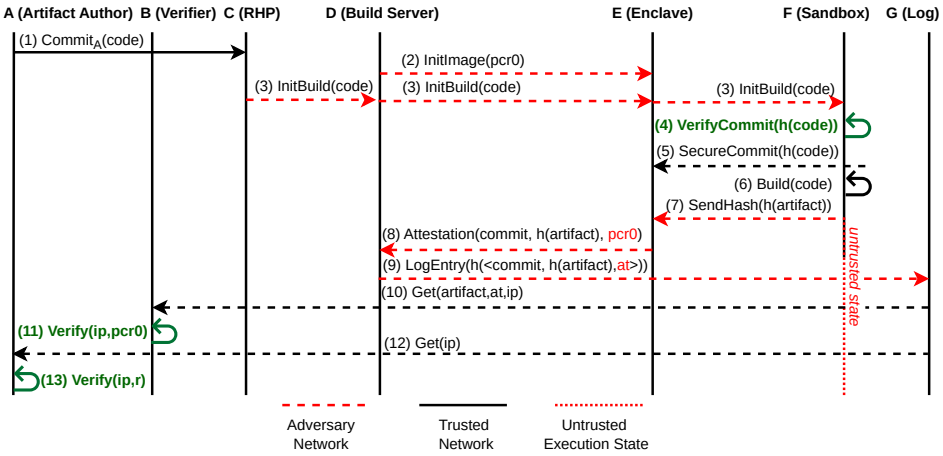


Figure 7.6: Protocol flow relevant for the formal model of A-Bs, including the interactions and data exchanges between system components and adversary channels. Taken from our paper [71].

build duration also increased. Building the *LLVM Clang* project increased from 54 minutes (HS) to 63 minutes (ES) and 79 minutes (ES+). This represents an 18 % overhead when building in ES mode compared to HS mode and a 48 % overhead when comparing ES+ to HS mode.

Ultimately, we measured the overall overhead by initiating build jobs for nine small targets and two large targets. In HS mode, the total build time was 1 hour and 22 minutes; with A-Bs (ES), it was 1 hour and 34 minutes (+14 %), and with A-Bs using *gVisor* (ES+), it was 2 hours and 14 minutes (+62 %). We excluded the average start and boot overhead of 42.1 seconds from these numbers.

7.3 Formal Verification

We formally model and verify the underlying protocol of A-Bs using `Tamarin` (section 2.7).

This section outlines various attack categories on security properties used to verify source-to-binary correspondence, including the authenticity of the repository. These attack categories are based on related threats model described in chapter 3 and we link each category with the respective threat(s) alongside a reference for clarity. The underlying trust assumptions of our threat model (section 3.1) also apply for the formal model. We model the security properties as formulas in a first-order logic using `Tamarin` *lemmas*. To verify both protocol behavior and data integrity we utilize *action facts* in the form $\mathtt{F}(x_1..x_n)\#i$ where $\mathtt{F}$ represents the name of the *action fact*, $x_1..x_n$ the data, and $\#i$ the time variable for the execution. Each subsequent paragraph describes the respective attack category and consists of two proofs: one demonstrating that the specific security property can be successfully compromised when not utilizing A-Bs, and another to ensure that there exists no trace where an adversary would be successful when using A-Bs. We use the function $h(..)$, which represents a hash function and variables c, ct, a, at, ip representing the data: code,

commi**t**hash, **a**rtifact, **at**testation, and **i**nclusion**p**roof. The full lemmas of the security properties described below as well as an example illustration of a detected attack by `Tamarin` are provided in section B.3 for reference.

Code manipulation (T-B2, T-B4). This attack category examines whether an adversary can successfully manipulate code during the build process. Specifically, this includes compromising code on the build server, attacking shared infrastructure, and considering hardware attacks (assuming the TEE to be trustworthy). Our formal verification begins with proving that `Tamarin` can find a trace where an adversary can compromise code c when specific verification controls, used to verify the commit hash ct, are not incorporated. Specifically, this lemma proves that $\exists\, c, ct : \neg(h(c) = ct)$. For the second proof of this attack category, which includes the verification step, `Tamarin` does not find any trace where an adversary is able to manipulate code without detection. This proof verifies that $\forall\, c, ct : h(c) = ct$.

Build asset manipulation (T-B1, T-B2, T-S1, T-B4). The attack category examines whether an adversary can successfully manipulate a build asset (e.g., the artifact) including potentially malicious libraries side-loaded from external sources. `Tamarin` is able to find a trace where an adversary can successfully compromise a build asset a when specific verification controls, used to verify the inclusion proof ip, are not incorporated. Specifically, this lemma shows that $\exists\, c, ct, at, ip : \neg(h(< ct, h(c), h(c) >) = ip)$. In case of incorporating the verification of the inclusion proof, provided by the transparency log, based on code sent via the adversary network and the attestation at provide by the TEE, `Tamarin` does not find a trace where an adversary can manipulate a build asset without detection. Specifically, this lemma shows that $\forall\, c, ct, a, at, ip : h(c) = ct \wedge h(< ct, h(a), at >) = ip$.

Build infrastructure manipulation (T-B1, T-B2, T-B4). This attack category focuses on successful attacks in which an adversary is able to compromise the infrastructure environment, i.e. the enclave image. To model this scenario, we transfer the build image through the adversary network so that the adversary can modify it. This analogously covers physical attacks against the machine running the image in an enclave. Thus, our first lemma in this category verifies whether an adversary can provide an attestation document at based on a compromised build image without using the trusted PCR value p to verify the attestation. Specifically, it proves that $\exists\, c, ct, a, p, at : (\neg(< c, h(a), p >) = at)$. However, if we include the proper verification in our model, `Tamarin` does not find any trace where an adversary can use a manipulated build image without detection. The respective proof shows that $\forall\, c, ct, a, p, at : (< c, h(a), p >) = at$.

Repository Spoofing (T-S2). The last attack category is particularly relevant for spoofing attacks with respect to the repository. An adversary might be able to spoof the repository and to create a valid inclusion proof for a particular commit hash of this repository. In this case, a verifier trying to audit the spoofed repository would get a valid inclusion proof. The first lemma, used to verify whether an adversary can successfully spoof the repository when not verifying

the inclusion proof shows that $\exists\, c, ct, a, at, ip : \neg(h(< h(c), h(a), at >) = ip)$. To prevent such spoofing attacks, the artifact author also needs to verify the corresponding inclusion proof according to the trustworthy reference r. Thus, the second lemma shows that $\forall\, c, ct, a, at, ip, r : h(c) = ct \wedge r = ip$.

7.4 Related Work

Several existing works address the challenge of building distributing software artifacts securely. An early example is a 1974 US Air Force report on the Multics system, which introduces the concept of compiler trapdoors [86]. A significant concern is highlighted by Ken Thompson in his 1984 Turing Award lecture on "Trusting Trust": source code analysis alone cannot protect against malicious build processes, such as a compromised compiler that injects vulnerabilities even when self-compiling from a clean codebase [157]. David Wheeler proposed Diverse Double-Comping (DDC), utilizing a trusted compiler to verify the recompilation of the main compiler. However, DDC does not address the fundamental problem of obtaining an initial trusted compiler or establishing a trusted execution environment. Finally, projects like Bootstrappable Builds advocates building software—especially compilers—entirely from source using minimal pre-compiled and auditable set of inputs [130].

A-Bs vs. R-Bs. The difficulty of establishing a trusted compiler for DDC, as proposed by David Wheeler, could be mitigated by employing the R-B approach. The initial trust assumption on the build environment can either be achieved by using a trusted local build environment or by setting of multiple environments providing the same output artifact—an "any-trust" model. The fundamental challenges of establishing a trusted toolchain and execution environment are well-documented in academic literature [31, 90]. Further research explores increasing the determinism of environments and tools [51, 115, 162], focuses on business use cases [15] and includes reports from large-scale commercial ecosystems [144]. Additionally, concerns raised by software developers [43] are also addressed.

However, an important security consideration arises with deterministic and reproducible builds: a binary, which is identical across all execution environments may increase adversary opportunities. Once such a binary is compromised, adversaries may be able to reliably reproduce exploitation more easily. Moreover, the consistency of the binary may facilitate vulnerability discovery through local debugging opportunities. The paradigm of software diversity aims to reduce predictability and is therefore incompatible with R-Bs. This is a key distinction between A-Bs and R-Bs, as A-Bs support diversity while R-Bs do not. Nevertheless, both approaches are compatible with runtime mitigation techniques such as Address Space Layout Randomization (ASLR).

Confidential Computing. Leveraging Confidential Computing technologies to address challenges in domains such as our A-Bs approach for untrustworthy build systems is also evident in other scenarios. For example, Russinovich et al. introduce Confidential Computing Proofs (CCP) as an alternative to Zero

	Requirements	Focus
L4	Attestable build	Attested trust in builder
L4	Reproducible build	Verifiable trust in builder
L3	Hardened build platform	Tampering during the build
L2	Signed provenance from a hosted build platform	Tampering after the build
L1	Provenance showing how the package was built	Mistakes, documentation
L0	n/a	

Table 7.1: The table is adapted from [42] and introduces potential new L4 levels for A-Bs and R-Bs. Taken from our paper [71].

Knowledge Proofs [140], and others explore Confidential Computing as a Service (CCaaS), the latter inspiring our deployment model, as demonstrated by Chen et al. [19] A-Bs can be viewed as a form of CCP, incorporating a persistence layer with a transparency log. Furthermore, TPMs are utilized to minimize software dependency lists, as shown by Meng et al. [107] However, these works lack a comprehensive security model and do not address cloud-based CI/CD pipelines, a key focus of A-Bs.

Confidential Computing technology is the major trust anchor in A-Bs; however, no technology provides absolute security and, in recent years, various researchers have been able to break the security premises of TEEs. Since A-Bs do not require confidentiality, many attacks [18, 45, 62, 93, 95, 96, 110, 161, 164], including classical side-channel attacks, do not affect its security properties. However, A-Bs rely on strong integrity protection provided by the underlying TEE. To the best of our knowledge, there are no successful attacks that compromise the integrity guarantees of AWS Nitro Enclaves. More details about three recent attacks against comparable TEEs are given in section 3.1.1.

Related projects. The *Supply-chain Levels for Software Artifacts* (SLSA) framework outlines valuable security aspects related to the software supply chain, including threat modeling and security guarantees provided by third-party software [42]. We propose that both R-Bs and A-Bs can be seen as a new Level 4 layer within the SLSA framework (see Table 7.1).

The CHAINIAC project demonstrated the distribution of updates using skipchains and verified builds [119].

The Sigstore project [117] focuses primarily on artifact signing and verification, utilizing a transparency log for verification — a component also integrated in A-Bs. However, Sigstore stores authentication certificates alongside artifact hashes and signatures to ensure signature verifiability, whereas in A-Bs we incorporate a transparency log to store metadata about the attested build process.

7.5 Deployment Consideration

This section addresses various opportunities and considerations related to the deployment of A-Bs in real-world environments. We discuss practical aspects of integration and new opportunities when enhancing our current approach.

Going beyond executable binaries. While the primary focus of A-Bs within this thesis is on executable binary artifacts, the underlying design principles of A-Bs can be extended to encompass other outputs of the build process, such as software libraries. Notably, software libraries are particularly relevant within the scope of this thesis addressing supply chain threats. By building every dependency with A-Bs and requiring downstream builders to verify each one, the security guarantees propagate transitively. Furthermore, A-Bs are not limited to binary artifacts; it can also be applied to non-binary outputs, such as vulnerability scans, for example.

Integrating with existing CI/CD systems. Integrating A-Bs into existing CI/CD pipelines requires minimal configuration changes, as shown in section B.2 with respect to GitHub Actions. Furthermore, the operational overhead of A-Bs can be outsourced to third-party providers, thereby the effort for substantial adaptations to existing development workflows is negligible.

Mitigating performance impact. As discussed in the evaluation section of this chapter (section 7.2), the initialization of enclaves introduces a start-up overhead. However, this can be mitigated through the provision of "pre-warmed", i.e. already started, enclaves. From a scalability perspective, the utilized EC2 instances can host multiple enclaves, with load balancers mediating build requests to distribute them effectively.

Extending the log. The current design of A-Bs incorporates transparency logs to manage log entries for verifying source-to-binary correspondence. However, extending the information contained within these logs can provide additional security guarantees, such as signed source code snapshots, which may be marked as *audited* to ensure compliance with relevant norms and standards.

7.6 Availability

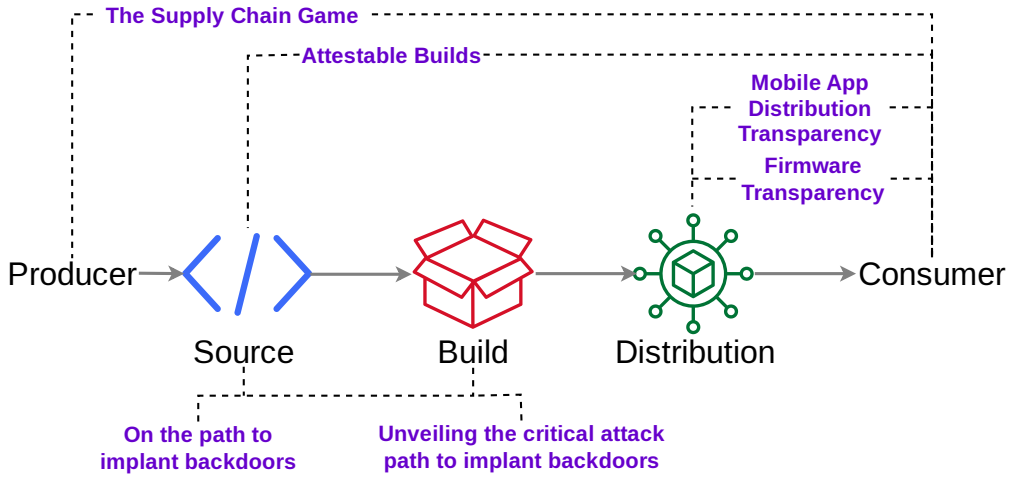
The artifacts presented in our A-Bs paper [71] completed artifact evaluation, and we received all three badges: Artifacts Available, Artifacts Evaluated & Functional, and Results Reproduced. The archived version of our artifacts is available at Zenodo (https://doi.org/10.5281/zenodo.17026892). We also provide an open-source repository of A-Bs on `https://github.com/linsm/attestable-builds`.

7.7 Summary

In this chapter, we introduced Attestable Builds (A-Bs), a novel approach for establishing strong source-to-binary correspondence in software artifacts. A-Bs primarily enables users to verify that a specific binary artifact was built from a given source code snapshot in a secure environment. We demonstrate the practicability and feasibility of A-Bs through a prototype implementation and used it to build a diverse set of open-source projects, including complex code-bases like the Linux Kernel and LLVM Clang, making their builds attestable. Our evaluation shows that the performance overhead of A-Bs is acceptable for use in production environments, regardless of project complexity. Finally, we discussed potential extensions of A-Bs to new use cases involving non-binary artifacts and its combination with Reproducible Builds (R-Bs) to further enhance security guarantees. In scenarios where R-Bs are challenging to implement, A-Bs provides a pragmatic alternative for addressing security concerns, and can also be effectively deployed independently to achieve valuable security objectives—specifically to mitigate threats targeting modern build systems.

Chapter 8

The Supply Chain Game: Can We Beat The Kill-chain?



Paper Reference. This chapter is based on the paper "Software Supply Chain Security: Can we beat the kill-chain? A Case Study on the XZ backdoor" [101]. The formal definitions used in this chapter are based on those presented by Rass in our paper [101], specifically those describing the game theoretic aspects.

In this chapter, we transition from technical mitigation techniques to organizational security measures, the final area of this thesis focusing on modern supply chain threats. We present a new way to enhance existing organizational security techniques, applicable across all stages of the supply chain, and demonstrate its applicability using the XZ Utils incident analyzed in chapter 6 as a case study.

An established approach to systematically addressing security issues is known as *Risk Management*. The risk value is often based on the likelihood and impact of a security issue and is used by decision makers to decide about proper treatment. Beside transferring, avoiding, or even accepting a specific risk, some of them need to be mitigated through the implementation of security controls. However, the decision maker must still choose which specific control should be applied to mitigate the risk appropriately. This is the point where our research addresses an important challenge: although existing risk models provide valuable guidance on how to properly identify, assess and manage a risk, these

models do not tell the decision maker what to decide specifically. Our approach enhances standard risk models by combining existing risk management features and game theory approaches to systematically analyze multi-stage attacks. In this chapter we focus on assessing vulnerabilities based on the critical attack path, also referred to as kill-chain, of the XZ backdoor incident (chapter 6). Based on realistic attacker and defender capabilities we demonstrate how to apply this combination to support selecting the more efficient defense strategies. In this chapter, we make the following contributions:

- We demonstrate how elements of the standard risk model can be combined with game theory techniques to elaborate a potential mitigation strategy for multi-stage attacks.
- We show the practical feasibility of our approach by applying it to the critical attack path of the XZ backdoor incident.
- We analyze the attacker and defender capabilities based on the critical attack path of the XZ backdoor incident.

We use the XZ backdoor incident as an example on how to apply the game theory approach for security concerns. However, the underlying concept can be applied to similar attack paths (or kill-chains) as well.

8.1 Attacker vs. Defender Capabilities

Our analysis is based on the critical attack path (as outlined in section 6.2 and illustrated in Figure 6.1), which shows the most critical steps for implanting a backdoor, using the XZ incident as an example. This section lists potential capabilities of an attacker (denoted as $A_{i.j}$) and a defender (denoted as $D_{i.j}$), where i is the stage and j a consecutive number.

8.1.1 Stage 1: Building Trust

This stage is based on the analysis presented in section 6.2.1, beginning with a summary of key findings relevant to this chapter.

The XZ incident began with a technique commonly referred to as *social engineering*. The main goal of the attacker, "Jia Tan" (presumably a pseudonym) was to exploit trust within the open-source community engaged with the XZ project. At a first glance, Jia Tan seemed to be a supportive contributor to the project. However, the real intention was to hide a backdoor in widely distributed XZ Utils. Over several months, various users attempted to pressure the original maintainer, Lasse Collin, into transferring maintainer permissions to Jia Tan. After numerous social engineering attempts using sock-puppet accounts controlled by the adversary, the attacker ultimately convinced Lasse Collin to grant them access to the XZ repository.

A1.1 **Anonymous acting.** A common way to contribute to open-source projects is by opening a pull request, which can be done anonymously since repository providers like GitHub do not require developers to prove their real identity. One reason for this is prioritizing the verification of source code

rather than the identity of specific contributors. This was also the case with respect to the XZ incident, where the adversary successfully established trust within the open-source community without ever proving the real identity. Thus, one capability of an adversary is the ability to create anonymous account(s) that do not require identity verification.

A1.2 **Instrumenting sock puppets.** The adversary can create an unlimited number of anonymous accounts and strategically use them as sock puppets when needed. Sock puppets can significantly enhance the efficiency of social engineering attacks by enabling the adversary to create the illusion of consensus within the community while keeping the primary attacker account in a supportive role. This was exactly the situation with XZ, where we assume that the adversary strategically deployed multiple sock puppets to pressure the original maintainer into granting maintainer permissions to Jia Tan while this account was simultaneously used to defend the original maintainer. This is one example of how such behavior can be leveraged to strengthen the trust relationship between the adversary and the privileged maintainer, leading to the next capability described below.

A1.3 **Exploiting trust.** Finally, the adversary gains trust of the maintainer and is likely able to convince the maintainer to grant additional privileges. As already mentioned, there are multiple ways for an adversary to increase trust, such as actively contributing, supporting the community, or even defending the people in case of social engineering pressure. Although it is a remarkable aspect of human nature that we can learn to trust others even if we never met them in person, adversaries may exploit that trust for malicious purposes. As we have seen in case of XZ, the Jia Tan account holder successfully exploited that trust to implant a backdoor.

D1.1 **Verify the identity of contributors.** An apparent defender capability is to verify the contributor's identity prior to granting relevant privileges. This approach is commonly used in commercial settings, where a developer typically applies for a position, signs a contract, and undergoes an interview as part of the hiring process. In more sensitive environments, such as critical infrastructures, it is even common to conduct a comprehensive background check. However, we do not consider this mitigation suitable for open-source projects as we assume that a significant number of contributors prefer to stay anonymous or at least under their pseudonym.

D1.2 **Increase support of open-source projects.** Many open-source projects are maintained by a single person, who can become overwhelmed by feature requests and bug fixes, especially when the project is widely distributed. Emotional situations like these can lead to bad decisions, such as granting maintainer permissions to an adversary, for example. The discussions about supporting open-source projects, particularly when they are integrated into commercial products, has intensified following the XZ incident.

D1.3 **Recognize the signs.** One key takeaway from the XZ incident is the importance of recognizing the early signs of overwhelmed open-source maintainers. This can be achieved by closely monitoring discussions directly on the repository, or in case of XZ, by reviewing mailing lists. Early detection may provide an opportunity to step in and prevent adversaries from exploiting human limits.

8.1.2 Stage 2: Preparation

This stage is based on the analysis presented in section 6.2.2, beginning with a summary of key findings relevant to this chapter.

A key aspect of the preparation stage is that everything done within this stage remain deniable. In case of XZ, the most important actions involved changing the primary contact mail, enabling a complex feature (likely unnecessary for this project), and disabling specific test areas. The mail address is used by Google OSS Fuzz (section 2.2.6), a fuzzing tool for detecting programming errors, to verify whether the change request is legitimate. Another preparation step was to enable the GNU indirect function (IFUNC) feature (section 2.5.1) used later to hook the actual backdoor on the victim's system. To prevent suspicious results from the fuzzing tool, the adversary finally disabled the related test step.

A2.1 **Change technical contact addresses.** This capability assumes that the adversary already established a trust relation with the original maintainer. Google OSS Fuzz required an approval of the previous maintainer before they change the primary mail address. In case of XZ, this happened within the corresponding pull request.

A2.2 **Introduce complex development features.** Another key takeaway of the XZ incident is that introducing complex development features, such as IFUNC, should be a trigger for more awareness. Although, the adversary tried to explain why this is necessary, a closer look may have revealed that enabling this feature is not required.

A2.3 **Deactivate testing features.** At this stage, an adversary may also have the capability to deactivate certain test features, such as IFUNC fuzzing, in case of XZ. Again, while this may still be deniable, and an adversary could provide a plausible justification, it remains a sign to examine such particular changes more closely.

D2.1 **Keep It Simple, Stupid.** Complex or even unnecessary code might lead to "mistakes", that may also have security relevant consequences. Especially unnecessary code or even complex architectural design decisions might indicate a preparation step for a future bugdoor. In case of XZ, two aspects are especially notable: the use of *autoconf*, which includes additional complexity for the build system, and the introduction of IFUNC, despite this feature being technically unnecessary. Although detection of such a bugdoor is especially tough as they are designed to stay hidden, we believe that an important defender capability is to identify and avoid unnecessary code or complex architecture designs.

D2.2 **Internal test infrastructure.** The initial step in deactivating certain test features is to determine whether it is necessary. While such deactivation may seem plausible—such as the deactivation of IFUNC fuzzing in case of XZ— the test can still remain activated within a local or independent testing environment. In commercial environments, it is common practice to use an internal test infrastructure to ensure a proper level of security before integrating third-party components into a product, for example.

D2.3 **Cooldown phase.** It might be worth to consider additional criteria that must be met before performing a security sensitive operation. An exam-

ple related to the XZ incident might involve a specific cooldown phase when changing the primary technical contact or granting permissions to a new maintainer. If such a significant organizational change has been made recently, a possible mitigation could be to restrict the new maintainer from making significant changes until a grace period has ended. Although this defender capability may not directly mitigate the corresponding threat, it could provide reviewers with additional time to detect suspicious changes.

8.1.3 Stage 3: Implant Backdoor

This stage is based on the analysis presented in section 6.2.3, beginning with a summary of key findings relevant to this chapter.

The adversary's main objective at this stage is to ensure that the backdoor does not get detected. This is likely even more challenging in an open-source project as it is publicly accessible, allowing anyone to verify the code and any commit made. A common approach of an adversary in such cases is to use obfuscation, making it difficult to spot malicious parts even if the code is available for public review. Another approach, also utilized in case of XZ, is the use of binary blobs. Since binary blobs are typically not human-readable, they do not raise suspicion among potential reviewers of the code. In case of XZ, the adversary implanted the backdoor code into two binary blobs used for testing purposes of the extraction functionality. A notable aspect of these test files is that they were titled with *bad-3-corrupt_lzma2.xz* and *good-large_compressed.lzma* implying indeed a corrupted non-readable binary blob and one that has a large file size. Both aspects were needed to implant the backdoor into the project.

A3.1 **Implant malicious code in public repositories.** Many open-source projects are the result of collaborative efforts, with multiple people working together toward a common objective. As a result, they often collaborate on developing new features by contributing individual code parts to the project. Unfortunately, not all contributions are done in a good faith — sometimes adversaries try to implant malicious code in such a project. Thus, the attacker capability in this particular case is to implant malicious code in a public repository by creating a pull request, for example.

A3.2 **Obfuscation of malicious content.** An essential objective of every adversary trying to distribute malicious software is hiding it. One way to hide malicious code or commands is obfuscation in various forms. Examples are using complex bit shifting operations, intentionally corrupt data to make it even harder for a human to read it, or hide it in binary files. The approach used by the adversary in the XZ incident is especially noteworthy as they were able to make a reasonable story around that. They implanted the backdoor in an obviously corrupt test files that seems legitimate as it is supposed to be used to test extraction of corrupted files.

A3.3 **Implant malicious code only in source tarballs.** One lesson from XZ is that adversaries can hide malicious code parts in source tarballs instead of having them in the source code repository directly—where the history of changes is much easier to review.

D3.1 **Code Review.** Many open-source projects encourage contributions of code snippets or other feature implementations. Typically, a contributor creates a pull request with the proposed code changes or extensions. However, from a security perspective, it is crucial for the project team to review these code changes for malicious content before merging it to the main branch. This control is meant to be a first line of defense to detect malicious code right after the adversary trying to implant it.

D3.2 **Demand reproducibility of binary blobs.** Software that is used to work with binaries, such as XZ, often have test files to verify the proper behavior in certain cases (e.g., corrupt or big files). However, these binaries are often used as they are and do not get verified how the author created them. Enforcing reproducibility makes it harder to hide malicious code in binary files. That would allow either the build system directly to create them if needed or an auditor to check their content.

8.1.4 Stage 4: Deployment

This stage is based on the analysis presented in section 6.2.4.

A4.1 **Compromise the build system.** Hiding malicious code directly in code is more likely detectable through code reviews, for example. However, if an adversary is able to utilize standard build features to implant the malicious code, it becomes less likely that the backdoor will be detected. This also involves compromising the build system itself by controlling security relevant parts as we have seen in case of SolarWinds [39].

A4.2 **Provide different build input to packager.** In the previous stage we describe how an adversary could implant malicious code in a tarball, but not in the repository. This is also relevant for the deployment stage as this allows an adversary to provide a custom build input to the packager, which is different from the actual source code in the repository.

D4.1 **Reproducible Builds.** Reproducible Builds is a technique to ensure that the final artifact originates from a given source code. However, this is often challenging to achieve, as demonstrated by the case of getting a reproducible Debian image that took twelve years [47].

D4.2 **Attestable Builds.** Attestable Builds [72] is a new approach leveraging security guarantees from confidential computing to enable verifiability of a strong source-to-binary correspondence. More details can be found in chapter 7.

8.1.5 Stage 5: Exploitation

This stage is based on the analysis presented in section 6.2.5.

A5.1 **Remote code execution.** The most severe attacker capability is arbitrary remote code execution allowing an adversary to extract sensitive information, or manipulate data.

D5.1 **Monitoring.** Monitoring of system resources is essential to detect malicious activities on a system. The main reason why the XZ backdoor got

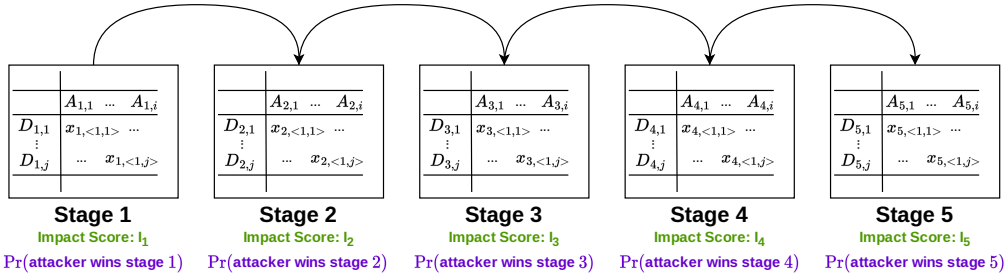


Figure 8.1: Game Theory applied to critical attack path. Taken from our paper [101].

detected was because Andres Freund monitored test behavior and recognized that his test cases suddenly took slightly longer.

D5.2 **System Hardening and compartmentalization.** Proactively implementing hardening measures (e.g., mandatory access controls) provide a significant security advantage in case of an incident and can potentially limit the capabilities of an adversary.

8.2 Assessment

Following the strategies defined in section 8.1, we model the supply chain game as a sequential game (see Figure 8.1). An adversary can only advance to stage $j+1$ if they win stage j. This results in the following pure strategies, consisting of strategy sets $P_{i,j}$ for player i in stage j, and probability densities $AS_{i,j}$ representing the "optimal likelihood" of each player selecting the strategy that maximizes their win chances:

$$PS_{1.1} = \{D1.1, D1.2, D1.3\}, \qquad PS_{2.1} = \{A1.2, A1.2, A1.3\} \tag{8.1}$$
$$PS_{1.2} = \{D2.1, D2.2, D2.3\}, \qquad PS_{2.2} = \{A2.1, A2.2, A2.3\} \tag{8.2}$$
$$PS_{1.3} = \{D3.1, D3.2\}, \qquad PS_{2.3} = \{A3.1, A3.2, A3.3\} \tag{8.3}$$
$$PS_{1.4} = \{D4.1, D4.2\}, \qquad PS_{2.4} = \{A4.1, A4.2\} \tag{8.4}$$
$$PS_{1.5} = \{D5.1, D5.2\}, \qquad PS_{2.5} = \{A5.1\} \tag{8.5}$$

Based on these (finite) strategy sets, we generate game (payoff) matrices for each stage, with columns representing the adversary's moves and rows the defender's moves (see Figure 8.2). The value of the loss function for a given player move combination is contained within the corresponding matrix cell. Specifically, the loss function for the k-th stage is:

$$\ell_k(i,j) := \Pr(\text{attack } Ak.j \text{ bypassed defense } Dk.i)$$

In this thesis, we take the defender's point of view. Therefore, our main objective is to minimize ℓ_k, which represents the adversary's probability of passing stage k, while each player seeks the optimal strategy. We also assume that if the adversary's motivations turn out to be different than we expect, the defender's ℓ_k will improve. This means we do not need to build a detailed model

	A1.1	A1.2	A1.3
D1.1	0.18	0.18	0.42
D1.2	1.00	1.00	0.42
D1.3	0.57	0.57	0.42

(a) Stage 1

	A2.1	A2.2	A2.3
D2.1	0.53	0.18	0.32
D2.2	0.53	0.32	0.00
D2.3	0.53	0.32	0.32

(b) Stage 2

	A3.1	A3.2	A3.3
D3.1	0.51	0.18	0.32
D3.2	0.18	0.18	0.00

(c) Stage 3

	A4.1	A4.2
D4.1	0.00	0.73
D4.2	0.00	0.42

(d) Stage 4

	A5.1
D5.1	0.57
D5.2	0.42

(e) Stage 5

Figure 8.2: Game matrices per stage. Taken from our paper [101]

of the adversary's behavior, which can be very difficult. As a result, we can calculate the Nash equilibrium (section 2.8) $v_k = val(\Gamma_k)$ in the k-th stage game $\Gamma_k := (N, \{AS_1, AS_2\}, \{\ell_k, -\ell_k\})$, with the guarantee that

$$\Pr(\text{adversary passes stage } k) \leqslant v_k, \tag{8.6}$$

However, this does not account for zero-day attacks (i.e., $a \notin AS_2$).

Remark. We acknowledge that assigning probability values is challenging as highlighted by Starmer et al. [147]. Typically, these probability values are either based on objective methods—for example, empirically observing relevant data [138]—or subjective assessments, such as CVSS, as seen in [38]. While statistical methods exist to estimate probabilities from experience [75, 138], data required by these methods may not be accessible. Therefore, in this thesis, we focus on subjective assessments based on CVSS (section 2.9), an established industrial standard for vulnerability assessment that offers reproducibility through its defined parameters.

8.3 Assessment: Loss Definitions

Assessing attacker and defender capabilities shares similarities with threat modeling and risk assessment paradigms. Identified attacker capabilities represent potential threats, while defender capabilities represent potential mitigation techniques. A typical risk management process begins by assessing the risk score of each identified threat, followed by a treatment procedure. While risks can be treated in various ways—accepted, transferred, or avoided—our focus is primarily on mitigation.

Framework. We use CVSS 3.1 (section 2.9) to assess the capabilities, specifically leveraging the *Exploitability* metric. First, we assess the initial exploitability of the attacker capabilities and subsequently the estimated exploitability which considers the defender capabilities. The related vector strings can be found in section C.1.

Assumptions. We assume the primary objective of the adversary is to implant a backdoor, thus the only technical impact is *Loss of Integrity*. We do not consider confidentiality as the project is open source already, and we do not consider availability as it is about implanting a backdoor rather than making the repository or the final artifact unavailable to its users. The attack vector (AV) is always *Network* as open-source projects typically are hosted via external cloud service (e.g., GitHub) to support international cooperation.

Calculation. We use the exploitability value of the defender capabilities applied to the attacker's capabilities. We derive the subjective probability representing the adversary's success by normalizing it against the highest possible score 3.89 (CVSS:3.1/AV:N/AC:L/PR:N/UI:N) equals 100%.

Stage 1: Building Trust

Initial exploitability. The primary objective of the adversary is to gain trust and to escalate privileges. We begin with the exploitability metric AC and classify that as *Low* for $A_{1,1}$ and $A_{1,2}$ as everyone can easily create an anonymous account, or instrument sock puppets. However, we assume that AC for $A_{1,3}$ is *High* as the final convincing step might be more complex depending on the individuals. There are no privileges required at this stage as everyone can perform actions. We do also not expect any required UI, expect for $A_{1,3}$ where the original maintainer needs to manually assign the privileges.

Estimated exploitability. We argue that identity verification ($D_{1,1}$) can be used to mitigate $A_{1,1}$ and $A_{1,2}$ because it significantly limits the possibility to create an anonymous account and sock puppets. However, we still consider some nation-state actors that might be able to easily create fake IDs. Therefore, we assume that $D_{1,1}$ raises AC to *High* and PR to *High* assuming only authorized entities can create IDs. We do not see any required UI for creating fake accounts. We believe that $D_{1,2}$ does not have a direct impact of any adversary capabilities. Detection of sketchy behavior or code changes are crucial especially in code audits. Thus, we assume that $D_{1,3}$ is a proper mitigation technique that raises the AC to *High* for all the adversary capabilities.

Stage 2: Preparation

Initial exploitability. We classify AC as *Low* for $A_{2,1}$, $A_{2,2}$, and $A_{2,3}$ as changing contact details and test configurations does not require specific knowledge or skills. In the previous stage the adversary gains maintainer privileges that are necessary to do the preparation steps — thus we assess the PR metric as *High*. However, for changing the contact address ($A_{2,1}$) we assume that the adversary does not necessarily need maintainer permissions as it could also be done as low privileged user with a simple pull request. Therefore, we classify PR for $A_{2,1}$ as *Low*. UI is only necessary for $A_{2,1}$ as the previous technical contact needs to confirm the change request.

Estimated exploitability. The defender move $D_{2,1}$ is about keep it simple, stupid and thus would increase the AC metric of $A_{2,2}$ to *High* as it requires the adversary to convince other developers to switch to a complex code structure. The defender move $D_{2,2}$ using an internal test infrastructure completely mitigates the attacker move of deactivating test features $A_{2,3}$. The $D_{2,3}$ move is an additional security control but does not directly mitigate the corresponding adversary move, thus we argue that we do not lower the exploitability move directly.

Stage 3: Implant Backdoor

Initial exploitability. We classify AC as *High* for $A_{3,1}$ and $A_{3,2}$ as implanting malicious code in public repositories (T-S1) and doing proper obfuscation requires advanced knowledge. The AC of with respect to $A_{3,3}$ is *Low* as hiding code in the source tarball itself is not complex and can easily be done even for non-malicious cases. In stage 1, the adversary gains maintainer privileges that are necessary to do the preparation steps—thus we assess the PR metric as *High*. There is also no UI necessary for all three attacker capabilities.

Estimated exploitability. The defender move $D_{3,1}$ is about code review that is a common technique to prevent bugs in code, thus we argue that code needs to be approved by an independent party, so UI is set to *Present* for $A_{3,1}$. However, it does not help if the adversary uses obfuscation techniques properly, so the UI is not relevant if the malicious code is properly obfuscated. $D_{3,2}$ does mitigate $A_{3,3}$ as reproducibility requires providing source code used to create the tarballs.

Stage 4: Deployment

Initial exploitability. $A_{4,1}$ is about compromising the build system itself (T-B1,T-B4). Assuming that major distributors, like Debian, have proper defense controls in place, we classify AC as *High*. $A_{4,2}$ is about providing a custom source tarball that does not require special skill so we classify AC here as *Low*. We assume that the distribution process itself always needs some sort of UI.

Estimated exploitability. In an ideal world, the build itself is bit-per-bit reproducible (section 2.2.3) or attestable (see chapter 7) to ensure a strong source-to-binary correspondence. We believe that both $D_{4,1}$ and $D_{4,2}$ would make it less likely for an adversary to implant malicious code only by compromising the build infrastructure. Therefore, we argue that these mitigations can properly mitigate $A_{4,1}$, intentionally assuming that the source code itself has not been tampered. If the source code is not reproducible and the distribution process depends on the source tarball, Attestable Builds can provide guarantees about how the tarball was built (chapter 7). Therefore, we argue that $D_{4,2}$ would increase the AC to *High* for $A_{4,2}$ as it is verifiable.

Stage 5: Exploitation

Initial exploitability. From a security point of view, this type of malware is one of the most severe types because an adversary has full control of the victim. Al-

Stage game	eq. for defender (Di.1, ..., Di.n)	eq. for adversary (Aj.1, ..., Aj.n)	saddle point value v_i
1	(0.7073, 0.2927, 0)	(0, 0, 1)	0.42
2	(0, 0, 1)	(1, 0 ,0)	0.53
3	(0, 1)	(0, 1, 0)	0.18
4	(0, 1)	(0, 1)	0.42
5	(0, 1)	(1)	0.42

Table 8.1: Nash equilibria of the (sub-)game(s). Taken from our paper [101].

though the most complex part is to initially infect the machine, we classify the AC of $A_{5.1}$ as *High* as exploiting the vulnerability still needs specific knowledge. As in case of XZ, in the worst the adversary does not need any prior authentication or other privileges. Furthermore, there is no UI required.

Estimated exploitability. Security techniques to prevent backdoor exploitation are, for example, monitoring $D_{5.1}$ and system hardening $D_{5.2}$. We argue that proper monitoring can reveal sketchy behavior and thus makes such attacks more complex. However, monitoring does not directly prevent an attack and depends on whether the thresholds are set properly to trigger an alarm. Thus, we believe that the AC does not decrease significantly and still classify it as *High*. System hardening, on the other hand, may directly affect the vulnerability by employing the least privilege principle, which necessitates additional privileges.

8.4 Results

Using linear programming and the algorithms described by Rass et al. [136], we computed the Nash Equilibria based on the payoff matrices in Figure 8.2. We identified one Nash equilibrium, and summarized the results in Table 8.1.

The results indicate that there is not necessarily a single optimal defense strategy; for example, in stage 1, we achieve the equilibrium payoff by playing the pure strategy D1.1, alongside the mixed strategy derived from linear programming, demonstrating flexibility in how a defense can be established. A similar observation applies to stage 3, where the adversary could consistently choose strategy A3.1 and still would get the same level of success. To further refine these results, we propose incorporating practical considerations to enhance the equilibria, such as accounting for real-world defense costs or the impact of changing adversary strategies. We discuss these considerations in the future work paragraph (section 8.5), as they primarily involve applying more complex algorithms (e.g., [102, 135]) and do not present new theoretical challenges.

Mixed strategies offer the advantage of increased unpredictability for the defender. From a supply chain perspective, these strategies can be implemented either by dynamically changing defense measures or by allocating resources to relevant players based on the mixed strategy. For instance, Hamilton's method

has been successfully applied to physical object protection, as demonstrated in [6].

Overall success rate. We assume that the stages are stochastically independent, meaning there is no strategic advantage gained by the adversary in subsequent moves given success in the previous stage. It is also crucial that the adversary can only progress to the next stage upon successfully completing the current one. Therefore, assuming stochastic independence across stages, we have:

$$\Pr(\text{attacker traverses the entire kill-chain}) \leqslant \prod_{k=1}^{n} \nu\alpha l(\Gamma_k), \tag{8.7}$$

given that the adversary successfully passes all n stages. For our example, this results in an overall success rate of $\approx 0.71\%$ for the adversary to win the supply chain game.

8.5 Discussion

This section covers some discussion points regarding game models and strategies.

Send back the adversary. If the defender can force the adversary back to a previous stage—for example, by replacing a vulnerable component with one from a different vendor—this alters the game model. However, we have considered this scenario outside the scope of this thesis and refer the reader to other research [137] for further details.

Devoting Resources. In reality, both defenders and adversaries typically operate with a limited budget for protecting or attacking the supply chain and may concentrate their resources on specific stages. Assuming the same strategy set $AS = \{1, 2, \dots, n\}$ for both players implies focusing on exactly one of the n stages at a time. If both players concentrate their efforts on the same stage k, the resulting payoff is the expected impact $I_k \cdot \nu\alpha l(\Gamma_k)$, where Γ_k represents the game of stage k. However, if the defender allocates their resources to stage k while the adversary focuses on a different stage $k' \neq k$, the defender looses this stage definitely and will cause the maximum potential loss $M > \max_k I_k$. This can be formally represented as:

$$\ell_{KC}(i, j) = \begin{cases} I_k \cdot \nu\alpha l(\Gamma_k) & \text{if } i = j = k, \\ M & \text{if } i \neq j. \end{cases}$$

Finding a Nash equilibrium provides a strategy for proportionally allocating resources based on the expected loss associated with each stage.

The Zero-Day Advantage. A zero-day advantage represents an adversary move that is entirely unknown to the defender. Within our current game model, this constitutes the worst-case scenario for the defender, as the adversary can immediately win a stage—effectively setting the k-th overall success rate to 1. Therefore, if an adversary possesses a set of zero-day exploits $Z \subsetneq \{1, 2, \dots, n\}$ their advantage is:

$$\mathbf{Adv}_{0\text{-day}}(Z) := \prod_{\substack{k=1 \\ i \notin Z}}^{n} val(\Gamma_k) - \prod_{k=1}^{n} val(\Gamma_k)$$

Applying this to our example in stage 3, the adversary would gain an advantage of ≈ 3.22 percentage points.

8.6 Summary

This chapter focuses on enhancing existing traditional organizational security techniques through combining them with game theoretic methods to support decision-making in risk management processes. We demonstrated the practicality of this approach using the XZ Utils incident as a case study. We began by elaborating potential attacker and defender capabilities in the XZ backdoor incident, followed by an assessment utilizing the Common Vulnerability Scoring System (CVSS)—a widely adopted industry standard. We then modeled the supply chain game and identified one Nash equilibrium based on our analysis and assessment. Finally, we discuss real-world strategies enhanced by game theory, such as prioritizing resource allocation to areas where the greatest impact is anticipated, leading to the recommendation that supply chains should be hardened first in the identified areas.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this thesis, we address still existing real-world threats targeting different stages of modern software supply chains and present new approaches to support mitigation by primarily utilizing transparency and enabling enhanced verifiability. We began by addressing threats targeting the distribution stage, introducing a new protocol designed to make unauthorized distribution attempts verifiable and transparent to both developers and users. Considering the importance of the underlying (operating) system layer for overall device security, particularly on mobile devices, we then presented *Firmware Transparency*—a new approach to mitigate lost security guarantees on devices with unlocked bootloaders, while supporting freedom-of-choice and sustainability. Moving to the source code level we analyzed the XZ backdoor incident, one of the most significant supply chain attacks in recent years. Based on this analysis, we extracted the critical attack path and implemented *SketchyCrawler*, a tool to identify potential signs of backdoors in open-source projects. The next stage—arguably one of the most critical one—is the build stage, where development teams nowadays often rely on potentially untrusted external cloud infrastructure. We introduced a new approach to ensure strong source-to-binary correspondence by leveraging the security guarantees of Confidential Computing technology. Finally, we presented *The Supply Chain Game*, an organizational security approach that enhances standard risk-management methods. We demonstrated how game-theoretic techniques, combined with common risk management practices, can provide new criteria to better support decision-makers choosing optimal defense strategies.

There is no one-that-secures-them-all solution. This thesis introduces new ways of making it harder for adversaries to compromise our software supply chains. However, no system is ever 100 % secure, including the individual stages of modern software supply chains. The recent attack on the XZ Utils project underscored that adversaries with sufficient resources can find new ways to bypass existing security controls.

While the presented security measures primarily focus on specific stages of the various ways how the software supply chain could look like, they can be combined according to the specific threat model to achieve even more resilient security guarantees. Beyond obvious combinations, such as integrating our Mobile App Distribution Transparency system with Firmware Transparency to secure both the application and underlying (operating) system layer on mobile

devices, we believe that other combinations can provide increased security in certain use cases. For example, our Attestable Builds (A-Bs) approach can be utilized with Firmware Transparency to (a) make the build process of alternative operating systems transparent and protect them from threats targeting the build infrastructure, and (b) protect the securely and attestable build artifacts once deployed to devices from e.g., evil-maid attacks. From an organizational perspective we demonstrate how to use game-theoretic techniques to enhance standard risk management methods, thereby assessing the best strategy for selecting one or more of the concepts presented in this thesis for a specific environment utilizing a software supply chain.

Concluding thoughts. The dynamic nature of our security landscape has always been fascinating to me, demanding continuous learning and adaption to new adversary techniques, technologies, and also shifting user needs. What I also always found exciting was the holistic view required for effective security—including everything from hardware, secure development practices, network security, and even psychological aspects of attacks like social engineering.

Throughout my PhD, I learned to have a more nuanced understanding of security and had to start thinking further beyond existing boundaries, especially in terms of questioning existing security controls and how the combination of them can provide new opportunities to increase our security posture.

My **first key takeaway** of my PhD is that I personally believe that threat modeling remains to be a valuable tool for development teams—clearly defining security requirements, asset scope, and potential incidents is essential to get a better understanding of the relevant security aspects. During our discussions for several papers, threat modeling was always a good opportunity to think about existing controls, the security properties we want to achieve, and especially also about the trust anchors that we are willing to accept. Consequently, I began to criticality evaluate (threat model) the trust anchors in my personal life, with a particular focus on the supply chain integrations I am utilizing. It remains crucial to continue thinking beyond our existing boundaries, as adversaries will persistently seek new ways to compromise and bypass our security controls.

However, reinventing security controls is not always the best strategy—often, the most effective solutions involve strategically combining existing ones to address new security challenges and ensure relevant security guarantees. Therefore, my **second key takeaway** is the importance of being open to new technologies, trends, and paradigms—Confidential Computing being my major example here. Exploring this field significantly broadened my understanding and filled some of my missing puzzle pieces when it comes to mitigate necessary trust anchors and enhancing transparency.

While a green checkmark and lock icons often imply security to end-users, we must try to make security guarantees more accessible, verifiable, and understandable for everyone. Therefore, my **third key takeaway** is the importance of proactively establishing new ways for verifying trust, rather than completely stop trusting in the companies and products we rely on in our everyday life—a "trust-but-verify" principle.

Ultimately, my PhD journey concludes with some thoughts that increasing transparency and establishing verifiable trust are essential to prevent adversaries from winning the *(Supply Chain) Game*.

9.2 Future Work

A software supply chain is a complex system comprising numerous components, entities, and processes, and is still (inherently) vulnerable to attacks at any stage. This multifaceted area remains a highly valuable target for adversaries, thus requires holistic security concepts and robust security measures. I believe that there are still plenty of opportunities for us as a security community to enhance supply chain security by introducing new protocols, and novel ways of using existing data structures, for example, and that we should continue learning from emerging attacks to react properly and efficient.

Specifically, we see potential to reduce trust anchors in mobile devices, such as reliance on external infrastructure, and to further strengthen the (operating) system layer. For example, the verification capabilities introduced in our *Firmware Transparency* work could be incorporated into early boot stages, even before the unlock screen is presented to the user. Even if an adversary is able to compromise the operating system of a mobile device—for example on a device with an unlocked bootloader—the enhanced security controls introduced in this thesis would detect tampering attempts even before the operating system finished booting up.

The core concept behind our Attestable Builds (A-Bs) approach demonstrates broad applicability to various use cases and still open security challenges. Leveraging Confidential Computing technologies offers a promising approach to strengthen areas where current security controls remain weak or introduce unnecessary trust anchors, particularly in untrusted cloud infrastructure. These opportunities extend to other areas, including IoT devices, medical environments, and even business processes requiring data protection not only at rest or in transit, but also while in use. Further examples include privacy-focused data processing and subsequently support for GDPR compliance.

A specific area of our ongoing research focuses on enhancing the security of issuing authorities. The security concepts presented in this thesis also offer opportunities to improve existing processes handling security-sensitive credentials, for example. Our objective is to introduce a fully end-to-end verifiable and transparent issuing authority, ensuring that malicious attempts to compromise security-related assets (e.g., signing key) get detected. Specifically, we will try to leverage a comprehensive suite of security controls, starting from hardware-backed security features, such as TEEs and TPMs, to bootloader-security, application layer protections, runtime analysis, and intrusion detection systems, with a focus on ensuring full end-to-end verifiability throughout the entire system.

Bibliography

[1] Inc. Advanced Micro Devices. 2025. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf, Last accessed July 2025. (2025).

[2] Inc. Advanced Micro Devices. 2024. Undermining Integrity Features of SEV-SNP with Memory Aliasing. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3015.html, Last accessed July 2025. (2024).

[3] Akamai Security Intelligence Group. 2025. XZ Utils Backdoor — Everything You Need to Know, and What You Can Do. https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know. Last accessed April 2025. (April 2025).

[4] Daroc Alden. 2024. Identifying dependencies used via dlopen(). https://lwn.net/Articles/969908/. Last accessed July 2025. (2024).

[5] Abdulla Aldoseri, Tom Chothia, Jose Moreira, and David Oswald. 2023. Symbolic modelling of remote attestation protocols for device and app integrity on Android. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (*ASIA CCS '23*). Association for Computing Machinery, Melbourne, VIC, Australia, pp. 218–231. ISBN: 9798400700989. DOI: 10.1145/3579856.3582812. https://doi.org/10.1145/3579856.3582812.

[6] Ali AlShawish. 2021. *Risk-based Security Management in Critical Infrastructure Organizations*. PhD Thesis. University of Passau.

[7] Amnesty International. 2024. A digital prison. https://www.amnesty.org/en/wp-content/uploads/2024/12/EUR7088132024ENGLISH.pdf. Last accessed April 2025. (2024).

[8] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. 2013. Internet Censorship in Iran: A First Look. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI '13)*. https://www.usenix.org/conference/foci13/workshop-program/presentation/aryan. USENIX Association, Washington, DC, USA, (August 2013).

[9] Basin, David and Cremers, Cas and Dreier, Jannik and Meier, Simon and Sasse, Ralf and Schmidt, Benedikt. 2025. Tamarin Prover. https://tamarin-prover.com/. Last accessed September 2025. (2025).

[10] Cliff L. Biffle. 2024. The Typestate Pattern in Rust. http://cliffle.com/blog/rust-typestate/. Last accessed December 2024. (2024).

[11] Cliff L. Biffle. 2025. The Typestate Pattern in Rust. http://cliffle.com/blog/rust-typestate/. Last accessed May 2025. (2025).

[12] Alex Birsan. 2021. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. Medium. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610. Last accessed December 2024. (February 2021).

[13] Evan Boehs. 2024. Everything I know about the XZ backdoor. https://boehs.org/node/everything-i-know-about-the-xz-backdoor. Last accessed April 2024. (March 2024).

[14] Otto Brechelmacher, Dejan Ničković, Tobias Nießen, Sarah Sallinger, and Georg Weissenbacher. 2024. Differential Property Monitoring for Backdoor Detection. In *Formal Methods and Software Engineering*. Kazuhiro Ogata, Dominique Mery, Meng Sun, and Shaoying Liu, (Eds.) Springer Nature Singapore, Singapore, pp. 216–236. ISBN: 978-981-96-0617-7.

[15] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. 2023. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal*, 31, 3, 687–719.

[16] Sergio Caltagirone, Andrew Pendergast, and Christopher Betz. 2013. The diamond model of intrusion analysis. *Threat Connect*, 298, 0704, 1–61.

[17] Scott Chacon and Ben Straub. 2022. *Pro Git*. (second ed.). Apress, Berkeley, CA, USA. https://git-scm.com/book/en/v2.

[18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPpectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, pp. 142–157.

[19] Hongbo Chen, Haobin Hiroki Chen, Mingshen Sun, Kang Li, Zhaofeng Chen, and XiaoFeng Wang. 2023. A verified confidential computing as a service framework for privacy preservation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 4733–4750.

[20] Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. 2023. Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, (August 2023), pp. 5899–5916. ISBN: 978-1-939133-37-3. https://www.usenix.org/conference/usenixsecurity23/presentation/cheval.

[21] Gynvael Coldwind. 2024. xz/liblzma: Bash-stage Obfuscation Explained. https://gynvael.coldwind.pl/?lang=en&id=782. Last accessed April 2024. (2024).

[22] Lasse Collin. 2024. XZ Utils backdoor. https://tukaani.org/xz-backdoor/. Last accessed April 2024. (2024).

[23] Intel Corporation. 2024. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Last accessed November 2024. (2024).

[24] The MITRE Corporation. 2025. ATT&CK. https://attack.mitre.org/. Last accessed April 2025. (2025).

[25] Victor Costan. 2016. Intel SGX explained. *IACR Cryptol, EPrint Arch*.

[26] Russ Cox. 2024. The xz attack shell script. https://research.swtch.com/xz-script. Last accessed April 2024. (2024).

[27] Al Cutter and David Drysdale. 2022. Trillian Personalities. https://github.com/google/trillian/blob/05001d1876f9340e42ba8b839c94e1b79246207b/docs/Personalities.md. (2022).

[28] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *46th IEEE Symposium on Security and Privacy (S&P)*. (May 2025).

[29] Debian. 24. CVE-2024-25743. https://security-tracker.debian.org/tracker/CVE-2024-25743, Last accessed July 2025. (24).

[30] Debian. 2025. UpstreamGuide. https://wiki.debian.org/UpstreamGuide. Last accessed April 2025. (2025).

[31] Xavier de Carné de Carnavalet and Mohammad Mannan. 2014. Challenges and implications of verifiable builds for security-critical open-source software. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 16–25.

[32] Massimo Di Pierro. 2017. What Is the Blockchain? *Computing in Science & Engineering*, 19, 5, 92–95. DOI: 10.1109/MCSE.2017.3421554.

[33] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29, 2, 198–208. DOI: 10.1109/TIT.1983.1056650.

[34] Adam Eijdenberg, Ben Laurie, and Al Cutter. 2015. Verifiable data structures. https://github.com/google/trillian/blob/30160804ab5203cde4412fe26f55a4149112bd92/docs/papers/VerifiableDataStructures.pdf. Last accessed April 2025. (2015).

[35] Holger Levsen et al. 2025. Overview of various statistics about reproducible builds. https://tests.reproducible-builds.org/debian/reproducible.html. Last accessed April 2025. (2025).

[36] F-Droid. 2025. Docs – F-Droid – Free and Open Source Android App Repository. https://f-droid.org/docs/. Last accessed December 2025. (2025).

[37] Fairphone. 2025. Manage the Bootloader. https://support.fairphone.com/hc/en-us/articles/10492476238865-Manage-the-Bootloader. Last accessed September 2025. (2025).

[38] FIRST — Forum of Incident Response and Security Teams. 2023. Exploit Prediction Scoring System (EPSS). (2023). Retrieved 06/06/2023 from https://www.first.org/epss.

[39] Fortinet, Inc. 2020. Solar Winds Cyber Attack. https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack. Last accessed February 2025. (2020).

[40] Forum of Incident Response and Security Teams. 2025. Common Vulnerability Scoring System SIG. https://www.first.org/cvss/. Last accessed June 2025. (2025).

[41] The Linux Foundation. 2025. in-toto: A framework to secure the integrity of software supply chains. https://in-toto.io/. Last accessed April 2025. (2025).

[42] The Linux Foundation. 2025. Safeguarding artifact integrity across any software supply chain. https://slsa.dev/. Last accessed April 2025. (2025).

[43] 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 (S&P)*. IEEE, pp. 1527–1544.

[44] Andres Freund. 2024. backdoor in upstream xz/liblzma leading to ssh server compromise. Post on mailing list oss-security@openwall. https://openwall.com/lists/oss-security/2024/03/29/4. Last accessed April 2024. (2024).

[45] Stefan Gast, Hannes Weissteiner, {Robin Leander} Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. English. In *Network and Distributed System Security (NDSS) Symposium 2025*. Network and Distributed System Security Symposium 2025 : NDSS 2025, NDSS 2025 ; Conference date: 23-02-2025 Through 28-02-2025. (February 2025). https://www.ndss-symposium.org/ndss2025/.

[46] Paul Gevers. 2023. Bits from the Release Team: Cambridge sprint update. https://lists.debian.org/debian-devel-announce/2023/12/msg00003.html. Last accessed April 2025. (2023).

[47] Paul Gevers. 2023. Bits from the Release Team: Cambridge sprint update. https://lists.debian.org/debian-devel-announce/2023/12/msg00003.html. Last accessed June 2025. (2023).

[48] GitHub. 2025. Automate your workflow from idea to production. https://github.com/features/actions. Last accessed August 2025. (2025).

[49] GitHub. 2025. Get started with GitLab CI/CD. https://docs.gitlab.com/ci/. Last accessed August 2025. (2025).

[50] GitHub. 2025. GitHub. https://github.com/. Last accessed December 2025. (2025).

[51] Maria Glukhova et al. 2017. Tools for ensuring reproducible builds for open-source software.

[52] GNU Project. 2024. GNU M4 1.4.19 macro processor. https://www.gnu.org/software/m4/manual/m4.html. Last accessed December 2024. (2024).

[53] Google. 2025. Accepting New Projects. https://google.github.io/oss-fuzz/getting-started/accepting-new-projects/. Last accessed April 2025. (2025).

[54] Google. 2025. Android Binary Transparency. https://developers.google.com/android/binary_transparency/overview. Last accessed April 2025. (2025).

[55] Google. 2025. Boot flow. https://source.android.com/docs/security/features/verifiedboot/boot-flow. Last accessed May 2025. (2025).

[56] Google. 2025. Certificate Transparency. https://certificate.transparency.dev/. Last accessed September 2025. (2025).

[57] Google. 2023. How Log Proofs Work – Certificate Transparency. https://sites.google.com/site/certificatetransparency/log-proofs-work. Last accessed January 2023. (2023).

[58] Google. 2025. Key and ID attestation. https://source.android.com/docs/security/features/keystore/attestation. Last accessed November 2025. (2025).

[59] Google. 2025. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Last accessed February 2025. (2025).

[60] Google. 2023. Use Play App Signing – Play Console Help. https://support.google.com/googleplay/android-developer/answer/9842756. Last accessed January 2023. (2023).

[61] Google. 2025. Verify hardware-backed key pairs with Key Attestation. https://developer.android.com/privacy-and-security/security-key-attestation. Last accessed September 2025. (2025).

[62] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pp. 1–6.

[63] GrapheneOS. 2025. Device integrity monitoring. https://attestation.app/about. Last accessed September 2025. (2025).

[64] GrapheneOS. 2025. GrapheneOS. https://grapheneos.org/. Last accessed September 2025. (2025).

[65] GrapheneOS. 2025. GrapheneOS - Frequently Asked Questions. https://grapheneos.org/faq. Last accessed September 2025. (2025).

[66] Andy Greenberg. 2023. The Huge 3CX Breach Was Actually 2 Linked Supply Chain Attacks. https://www.wired.com/story/3cx-supply-chain-attack-times-two/. Last accessed February 2025. (2023).

[67] Sivana Hamer, Jacob Bowen, Md Nazmul Haque, Robert Hines, Chris Madden, and Laurie Williams. 2025. Closing the Chain: How to reduce your risk of being SolarWinds, Log4j, or XZ Utils. Computing Research Repository (CoRR), arXiv:2503.12192v1 [cs.SE]. (March 2025). DOI: 10.48550/arXiv.2503.12192.

[68] Jossef Harush. 2025. Large Scale Campaign Created Fake GitHub Projects Clones with Fake Commit Added Malware. https://checkmarx.com/blog/large-scale-campaign-created-fake-github-projects-clones-with-fake-commit-added-malware/. Last accessed January 2025. (2025).

[69] Roee Hay. 2017. fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. https://www.usenix.org/system/files/conference/woot17/woot17-paper-hay.pdf. USENIX Association, Vancouver, BC, (August 2017), pp. 383–398.

[70] Trey Herr, Will Loomis, Stewart Scott, June Lee, and Emma Schroeder. 2021. Breaking Trust – Shades of Crisis Across an Insecure Software Supply Chain. https://www.usenix.org/conference/enigma2021/presentation/herr. Last accessed January 2023. (February 2021).

[71] Daniel Hugenroth, Mario Lins, René Mayrhofer, and Alastair R. Beresford. 2025. Attestable Builds: Compiling Verifiable Binaries on Untrusted Systems using Trusted Execution Environments. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '25*). Association for Computing Machinery, Taipei, Taiwan, pp. 4514–4528. ISBN: 9798400715259. DOI: 10.1145/3719027.3765128. https://doi.org/10.1145/3719027.3765128.

[72] Daniel Hugenroth, Mario Lins, René Mayrhofer, and Alastair R. Beresford. 2025. Attestable Builds: Compiling Verifiable Binaries on Untrusted Systems using Trusted Execution Environments. In *2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. DOI: 10.1145/3719027.3765128.

[73] Advanced Micro Devices Inc. 2024. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html. Last accessed November 2024. (2024).

[74] GitLab Inc. 2024. GitLab Homepage. https://about.gitlab.com/. Last accessed December 2025. (2024).

[75] Jay Jacobs, Sasha Romanosky, Octavian Suciu, Benjamin Edwards, and Armin Sarabi. 2023. Enhancing Vulnerability Prioritization: Data-Driven Exploit Predictions with Community-Driven Insights. arXiv:2302.14172 [cs]. (February 2023). DOI: 10.48550/arXiv.2302.14172. Retrieved 06/06/2023 from.

[76] Hans Jansen. 2023. Add ifunc check to CMakeLists.txt. https://git.tukaani.org/?p=xz.git;a=commitdiff;h=b72d21202402a603db6d512fb9271cfa83249639. Last accessed April 2024. (2023).

[77] Hans Jansen. 2023. Add ifunc check to configure.ac. https://git.tukaani.org/?p=xz.git;a=commitdiff;h=23b5c36fb71904bfbe16bb20f976da38dadf6c3b. Last accessed April 2024. (2023).

[78] Hans Jansen. 2024. xz-utils: New upstream version available. Debian Bug report #1067708. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=1067708. Last accessed April 2024. (2024).

[79] JiaT75. 2021. Added error text to warning when untaring with bsdtar. Pull request #1609. https://github.com/libarchive/libarchive/pull/1609. Last accessed April 2024. (2021).

[80] JiaT75. 2022. CMake: Update .gitignore for CMake artifacts from in source build. https://github.com/tukaani-project/xz/commit/8ace358d65059152d9a1f43f4770170d29d35754. Last accessed April 2024. (2022).

[81] JiaT75. 2023. XZ updates. Pull request #9960. https://github.com/google/oss-fuzz/pull/9960. Last accessed April 2024. (2023).

[82] JiaT75. 2023. xz: Disable ifunc to fix Issue 60259. Pull request #10667. https://github.com/google/oss-fuzz/pull/10667. Last accessed April 2024. (2023).

[83] Jingdong Lu. 2025. How OP-TEE Architecture Evolved on Intel Platforms. https://www.intel.com/content/www/us/en/developer/articles/technical/how-op-tee-architecture-evolved-on-platforms.html. Last accessed December 2025. (2025).

[84] 1996. *Chapter 4: Non-cooperative Games. Essays on Game Theory*. Edward Elgar Publishing, Cheltenham, UK, pp. 22–33. ISBN: 9781781956298. DOI: 10.4337/9781781956298.00009.

[85] John Nash Jr. 1996. *Essays on Game Theory*. Edward Elgar Publishing, Cheltenham, UK. ISBN: 9781781956298. DOI: 10.4337/9781781956298. https://www.elgaronline.com/view/book/9781781956298/9781781956298.xml.

[86] Paul A Karger and Roger R Schell. 2002. Thirty years later: Lessons from the multics security evaluation. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.* IEEE, pp. 119–126.

[87] Dimitri Kokkonis, Michaël Marcozzi, Emilien Decoux, and Stefano Zacchiroli. 2025. ROSA: Finding Backdoors with Fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, pp. 720–720. DOI: 10.1109/ICSE55347.2025.00183.

[88] Jigar Kumar. 2022. Re: [xz-devel] [PATCH] String to filter and filter to string. Reply on mailing list xz-devel. https://www.mail-archive.com/xz-devel@tukaani.org/msg00555.html. Last accessed April 2024. (2022).

[89] Renuka Kumar, Apurva Virkud, Ram Sundara Raman, Atul Prakash, and Roya Ensafi. 2022. A Large-scale Investigation into Geodifferences in Mobile Apps. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. https://www.usenix.org/conference/usenixsecurity22/presentation/kumar. USENIX Association, Boston, MA, USA, (August 2022), pp. 1203–1220.

[90] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 39, 2, 62–70.

[91] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. RFC 6962: Certificate Transparency. (2013). DOI: 10.17487/RFC6962.

[92] Ben Laurie, Eran Messeri, and Rob Stradling. 2021. RFC 9162: Certificate Transparency Version 2.0. (2021). DOI: 10.17487/RFC9162.

[93] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pp. 557–574.

[94] Ernst Leierzopf, Stefan Kempinger, René Mayrhofer, and Michael Roland. 2025. AVBTestKeyInTheWild: Bypassing Android Verified Boot Using A Firmware Supply Chain Vulnerability on Locked Devices. In *SPICES 2025 co-located with EWSN '25: Proceedings of the 2025 International Conference on embedded Wireless Systems and Networks*.

[95] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 337–351. DOI: 10.1109/SP46214.2022.9833768.

[96] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 717–732.

[97] Mario Lins, René Mayrhofer, and Michael Roland. 2025. Unveiling the critical attack path for implanting backdoors in supply chains: Practical experience from XZ. In *Cryptology and Network Security. 24th International Conference, CANS 2025* (*LNCS*, volume 16351). Springer, Osaka, Japan, (November 2025), pp. 521–541. DOI: 10.1007/978-981-95-4434-9_24.

[98] Mario Lins, René Mayrhofer, Michael Roland, and Alastair R. Beresford. 2023. Mobile App Distribution Transparency (MADT): Design and evaluation of a system to mitigate necessary trust in mobile app distribution systems. In *Secure IT Systems. 28th Nordic Conference, NordSec 2023* (*LNCS*, volume 14324/2024). Springer, Oslo, Norway, (November 2023), pp. 1–19. DOI: 10.1007/978-3-031-47748-5_11.

[99] 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 preprint arXiv:2404.08987*.

[100] Mario Lins, René Mayrhofer, and David Zeuthen. 2026. Firmware Transparency: Strengthening Security Guarantees Against Bootloader-Targeting Attacks. In *TBD. 24th International Conference on Applied Cryptography and Network Security, ACNS 2026* (*LNCS*). Springer, New York, United States, (June 2026).

[101] Mario Lins, Stefan Rass, and René Mayrhofer. 2025. Software Supply Chain Security: Can we beat the kill-chain? A Case Study on the XZ backdoor. In *Secure IT Systems. 30th Nordic Conference on Secure IT Systems, NordSec 2025* (*LNCS*). Springer, Tartu, Estonia, (November 2025).

[102] G. Liuzzi, M. Locatelli, V. Piccialli, and S. Rass. 2021. Computing mixed strategies equilibria in presence of switching costs by the solution of nonconvex QP problems. en. *Computational Optimization and Applications*, 79, 3, (July 2021), 561–599. ISSN: 0926-6003, 1573-2894. DOI: 10.1007/s10589-021-00282-7. Retrieved 09/06/2021 from.

[103] LWN.net. 2025. Debian bookworm live images now fully reproducible. https://lwn.net/Articles/1015402/, Last accessed April 2025. (2025).

[104] Jeferson Martínez and Javier M Durán. 2021. Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. *International Journal of Safety and Security Engineering*, 11, 5, 537–545.

[105] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Trans. Priv. Secur.*, 24, 3, Article 19, (April 2021), 35 pages. ISSN: 2471-2566. DOI: 10.1145/3448609. https://doi.org/10.1145/3448609.

[106] 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. *Computing Research Repository (CoRR), arXiv:2011.04551*. DOI: 10.48550/ARXIV.2011.04551.

[107] Ce Meng, Yeping He, and Qian Zhang. 2009. Remote attestation for custom-built software. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*. Volume 2. IEEE, pp. 374–377.

[108] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*. LNCS. Volume 293. Carl Pomerance, (Ed.) Springer, Berlin, Heidelberg, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[109] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2024. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proc. ACM Meas. Anal. Comput. Syst.*, 8, 3, Article 36, (December 2024), 42 pages. DOI: 10.1145/3700418. https://doi.org/10.1145/3700418.

[110] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*, pp. 1–6.

[111] Mozilla. 2023. Certificate Transparency. https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency. Last accessed January 2023. (2023).

[112] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 1466–1482.

[113] Nitin Naik, Paul Jenkins, Paul Grace, and Jingping Song. 2022. Comparing Attack Models for IT Systems: Lockheed Martin's Cyber Kill Chain, MITRE ATT&CK Framework and Diamond Model. In *2022 IEEE International Symposium on Systems Engineering (ISSE)*, pp. 1–7. DOI: 10.1109/ISSE54508.2022.10005490.

[114] John F. Nash. 1950. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36, 1, 48–49. DOI: 10.1073/pnas.36.1.48.

[115] Omar S Navarro Leija, Kelly Shiptoski, Ryan G Scott, Baojun Wang, Nicholas Renner, Ryan R Newton, and Joseph Devietti. 2020. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 167–182.

[116] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond typosquatting: an in-depth look at package confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 3439–3456.

[117] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software Signing for Everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '22*). Association for Computing Machinery, Los Angeles, CA, USA, pp. 2353–2367. ISBN: 9781450394505. DOI: 10.1145/3548606.3560596. https://doi.org/10.1145/3548606.3560596.

[118] Jack Nicas, Raymond Zhong, and Daisuke Wakabayashi. 2021. Censorship, Surveillance and Profits: A Hard Bargain for Apple in China. *The New York Times*, (May 2021). https://www.nytimes.com/2021/05/17/technology/apple-china-censorship-data.html. Last accessed January 2023.

[119] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pp. 1271–1287.

[120] Linus Nordberg, Daniel Kahn Gillmor, and Tom Ritter. 2018. Gossiping in CT. Internet-Draft draft-ietf-trans-gossip-05. Work in Progress. Internet Engineering Task Force, (January 2018). 57 pages. https://datatracker.ietf.org/doc/draft-ietf-trans-gossip/05/.

[121] Carlos O'Donell. 2024. GNU_IFUNC. https://sourceware.org/glibc/wiki/GNU_IFUNC. Last accessed April 2024. (2024).

[122] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (*LNCS*, volume 12223). Springer, Cham, pp. 23–43. DOI: 10.1007/978-3-030-52683-2_2.

[123] Alessandro Parilli and James Maclachlan. 2021. No Unaccompanied Miners: Supply Chain Compromises Through Node.js Packages. Google Cloud Blog. https://cloud.google.com/blog/topics/threat-intelligence/supply-chain-node-js/. Last accessed December 2024. (December 2021).

[124] Mike Perry and The Tor Project. 2024. Deterministic Builds Part One: Cyberwar and Global Compromise. https://blog.torproject.org/deterministic-builds-part-one-cyberwar-and-global-compromise/. Last accessed November 2024. (2024).

[125] Borislav Petkov. 24. x86/sev: Harden #VC instruction emulation somewhat. https://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git/patch/?id=e3ef461af35a8c74f2f4ce6616491ddb355a208f, Last accessed July 2025. (24).

[126] Phylum Research. 2023. Rust Malware Staged on Crates.io. https://blog.phylum.io/rust-malware-staged-on-crates-io/. Last accessed April 2025. (2023).

[127] Lennart Poettering. 2024. Expose dlopen() dependencies in an ELF section, and add spec for it. https://github.com/systemd/systemd/pull/32234. Last accessed July 2025. (2024).

[128] Debian Project. 2024. Reproducible Builds. https://wiki.debian.org/ReproducibleBuilds/About. Last accessed November 2024. (2024).

[129] Jenkins project. 2024. Jenkins Homepage. https://www.jenkins.io/. Last accessed December 2025. (2024).

[130] The Bootstrappable Builds project. 2024. Bootstrappable Builds. https://bootstrappable.org/. Last accessed November 2024. (2024).

[131] The Chromium Project. 2024. Deterministic builds. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/deterministic_builds.md. Last accessed November 2024. (2024).

[132] The Git project. 2025. Git Website. https://git-scm.com/. Last accessed December 2025. (2025).

[133] Bernd Prünster., Gerald Palfinger., and Christian Kollmann. 2019. Fides: Unleashing the Full Potential of Remote Attestation. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - SECRYPT*. INSTICC. SciTePress, pp. 314–321. ISBN: 978-989-758-378-0. DOI: 10.5220/0008121003140321.

[134] Qualcomm Technologies, Inc. 2019. Secure Boot and Image Authentication. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/secure-boot-and-image-authentication-version_final.pdf. Last accessed April 2025. (2019).

[135] S. Rass and B. Rainer. 2014. Numerical Computation of Multi-Goal Security Strategies. In *Decision and Game Theory for Security* (*LNCS 8840*). Radha Poovendran and Walid Saad, (Eds.) Springer, pp. 118–133. DOI: https://doi.org/10.1007/978-3-319-12601-2_7.

[136] Stefan Rass, Sandra König, and Ali Alshawish. 2020. R Package 'HyRiM': Multicriteria Risk Management using Zero-Sum Games with vector-valued payoffs that are probability distributions, version 2.0.0. (2020). https://CRAN.R-project.org/package=HyRiM.

[137] Stefan Rass, Sandra König, and Emmanouil Panaousis. 2019. Cut-The-Rope: A Game of Stealthy Intrusion. In *Decision and Game Theory for Security*. Volume 11836. Tansu Alpcan, Yevgeniy Vorobeychik, John S. Baras, and György Dán, (Eds.) Springer, Cham, pp. 404–416. DOI: 10.1007/978-3-030-32430-8_24. Retrieved 06/19/2020 from http://link.springer.com/10.1007/978-3-030-32430-8_24.

[138] Stefan Rass, Sandra König, and Stefan Schauer. 2021. Semi-automated Parameterization of a Probabilistic Model Using Logistic Regression - A Tutorial. en. In *Game Theory and Machine Learning for Cyber Security*. (1st ed.). Charles A Kamhoua, Christopher D Kiekintveld, Fei Fang, and Quanyan Zhu, (Eds.) Wiley, (September 2021), pp. 438–484. DOI: 10.1002/9781119723950.ch22. Retrieved 11/02/2023 from.

[139] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. BootStomp: On the Security of Bootloaders in Mobile Devices. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, (August 2017), pp. 781–798. ISBN: 978-1-931971-40-9. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini.

[140] Mark Russinovich, Cédric Fournet, Greg Zaverucha, Josh Benaloh, Brandon Murdoch, and Manuel Costa. 2024. Confidential Computing Proofs: An alternative to cryptographic zero-knowledge. *Queue*, 22, 4, 73–100.

[141] Rust. 2025. The Cargo Book. https://doc.rust-lang.org/cargo/reference/build-scripts.html. Last accessed April 2025. (2025).

[142] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4220–4238. DOI: 10.1109/SP54263.2024.00262.

[143] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security*.

[144] Yong Shi, Mingzhi Wen, Filipe R Cogo, Boyuan Chen, and Zhen Ming Jiang. 2021. An experience report on producing verifiable builds for large-scale commercial systems. *IEEE Transactions on Software Engineering*, 48, 9, 3361–3377.

[145] Kirill A. Shutemov. 23. index : kernel/git/torvalds/linux.git. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b82a8dbd3d2f4563156f7150c6f2ecab6e960b30, Last accessed July 2025. (23).

[146] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. 2019. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 318–331.

[147] Chris Starmer. 2000. Developments in Non-Expected Utility Theory: The Hunt for a Descriptive Theory of Choice under Risk. *Journal of Economic Literature*, 38, 2, 332–382. ISSN: 00220515. http://www.jstor.org/stable/2565292.

[148] Hans-Christoph Steiner. 2023. Binary Transparency Log for https://guardianproject.info/fdroid. https : / / github . com / guardianproject/binary_transparency_log. Last accessed January 2023. (2023).

[149] 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)*. IEEE, San Jose, CA, USA, (May 2016), pp. 526–545. DOI: 10.1109/SP.2016.38.

[150] Jia Tan. 2022. [xz-devel] [PATCH] String to filter and filter to string. Post on mailing list xz-devel. https://www.mail-archive.com/xz-devel@tukaani.org/msg00553.html. Last accessed April 2024. (2022).

[151] Jia Tan. 2024. Tests: Add a few test files. https://git.tukaani.org/?p=xz.git;a=commitdiff;h=cf44e4b7f5dfdbf8c78aef377c10f71e274f63c0. Last accessed April 2024. (2024).

[152] Jia Tan. 2024. Tests: Update two test files. https://git.tukaani.org/?p=xz.git;a=commitdiff;h=6e636819e8f070330d835fce46289a3ff72a7b89. Last accessed April 2024. (2024).

[153] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending against package typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*. Springer, pp. 112–131.

[154] The LineageOS Project. 2018. Qualcomm's Chain of Trust. https://lineageos.org/engineering/Qualcomm-Firmware/. Last accessed September 2025. (2018).

[155] The Tor Project. 2023. Tor Project – Anonymity Online. https://www.torproject.org/. Last accessed February 2023. (2023).

[156] The Tukaani Project. 2024. LZMA Utils. https://tukaani.org/lzma/. Last accessed April 2024. (2024).

[157] Ken Thompson. 1984. Reflections on trusting trust. *Communications of the ACM*, 27, 8, 761–763.

[158] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. 2018. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, (August 2018), pp. 273–290. ISBN: 978-1-939133-04-5. https://www.usenix.org/conference/usenixsecurity18/presentation/tian.

[159] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf. Last accessed November 2024. (2020).

[160] Reilly Watson. 2013. History of LZMA Utils and XZ Utils. https://github.com/kobolabs/liblzma/blob/87b7682ce4b1c849504e2b3641cebaad62aaef87/doc/history.txt. Last accessed April 2024. (2013).

[161] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (*CCS '24*). Association for Computing Machinery, Salt Lake City, UT, USA, pp. 79–93. ISBN: 9798400706363. DOI: 10.1145/3658644.3690230. https://doi.org/10.1145/3658644.3690230.

[162] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. 2022. Towards build verifiability for java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 297–306.

[163] Kim Zetter. 2023. The Untold Story of the Boldest Supply-Chain Hack Ever. *Wired*.

[164] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, (August 2024), pp. 1135–1151. ISBN: 978-1-939133-44-1. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi.

[165] Ziyi Zhou, Xuangan Xiao, Tianxiao Hou, Yikun Hu, and Dawu Gu. 2024. On the (In)Security of Manufacturer-Provided Remote Attestation Frameworks in Android. In *Computer Security – ESORICS 2023*. Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, (Eds.) Springer Nature Switzerland, pp. 250–270. ISBN: 978-3-031-51482-1.

Appendix A

Firmware Transparency

A.1 Experiment

The following results were obtained from our experiment, where we requested an attestation of a device in both locked and unlocked state to compare the responses.

```
Locked state:
vbmeta digest:
AEFEE75C50C79.....6EA6C8005D39EE6C40BC
verified boot key:
0F6E75C80183B.....B0819DE1F150BA0FF9D7
root public key:
MIICIjANBgkqh.....i8WEg5UmAGMCAwEAAQ==
```

```
Unlocked state:
vbmeta digest:
AEFEE75C50C79.....6EA6C8005D39EE6C40BC
verified boot key:
0000000000000.....0000000000000000000
root public key:
MIICIjANBgkqh.....i8WEg5UmAGMCAwEAAQ==
```

A.2 Formal Verification

A.2.1 Security Properties

This section provides a more detailed definition of selected security properties as representative examples.

The following security properties are used to verify whether an adversary with physical access to the device or with access to the distribution network is able to compromise the operating system. The first equation verifies if the adversary would be successful without using the auditor to validate the image. The notation of the following lemmas is described in Table A.1.

$$\exists C, D, E, at, ip, \#i, \#k.\ ((GetAtt_D(D, C, at)\ @\#i).$$
$$\wedge(GetIP_D(D, E, ip)\ @\#j)) \wedge (\neg(h(at) = ip))$$

Table A.1: Notations used in `Tamarin` lemmas for Firmware Transparency

Notation	Description
A, B, C, D, E	Participants: A(Developer), B(Distributor), C(Auditee), D(Auditor), E(Personality)
f, at, ip	Data: f(Image), at(Attestation), ip(Inclusion Proof)
#	Time variable
$h()$	Hash function
$Audit_D()$	Auditor function to verify combination of attestation data and inclusion proof
$InitVerify_A()$	Developer check to verify if image is properly logged
$Publish_A()$	Developer publishes new image
$Distribute_B()$	Distributor distributes image or creates inclusion proof
$GetImage_C()$	Device downloads image
$SendIP_E()$	Personality sends inclusion proof
$GetAtt_D()$	Auditor receives attestation data from device
$GetIP_D()$	Auditor receives inclusion proof from personality

The second lemma verifies all possible traces if a tampered image can be detected.

$$\forall A, B, C, D, E, f, at, ip, \#v, \#w, \#x, \#y, \#z.$$
$$(((((InitVerify(A, f, ip)\,@\#v) \wedge (GetImage_C(C, B, f)\,@\#w)) \wedge$$
$$(GetIP_D(D, E, ip)\,@\#x)) \wedge (GetAtt_D(D, C, at)\,@\#y)) \wedge$$
$$(Audit_D(D, ip, at)\,@\#z))$$
$$\Rightarrow ((h(f) = ip) \wedge (f = at))$$

$$\exists A, B, C, E, f, ip, \#i, \#j, \#k. (((Publish_A(A, B, f)\,@\#i) \wedge (Distribute_B(B, E, f)\,@\#j)) \wedge$$
$$(SendIP_E(E, C, ip)\,@\#k)) \wedge (\neg(h(f) = ip))$$

The lemma below shows that an insider attack gets detected when using *Firmware Transparency*.

$$\forall A, D, f, at, ip, \#x, \#y. ((Audit_D(D, ip, at)\,@\#x)$$
$$\wedge(InitVerify_A(A, f, ip)\,@\#y)) \Rightarrow (ip = h(at))$$

A.2.2 Attack Path Examples

Figure A.1 and Figure A.2 are examples of the identified traces by `Tamarin` used to verify physical, on-path, and insider attacks on devices with an unlocked bootloader.

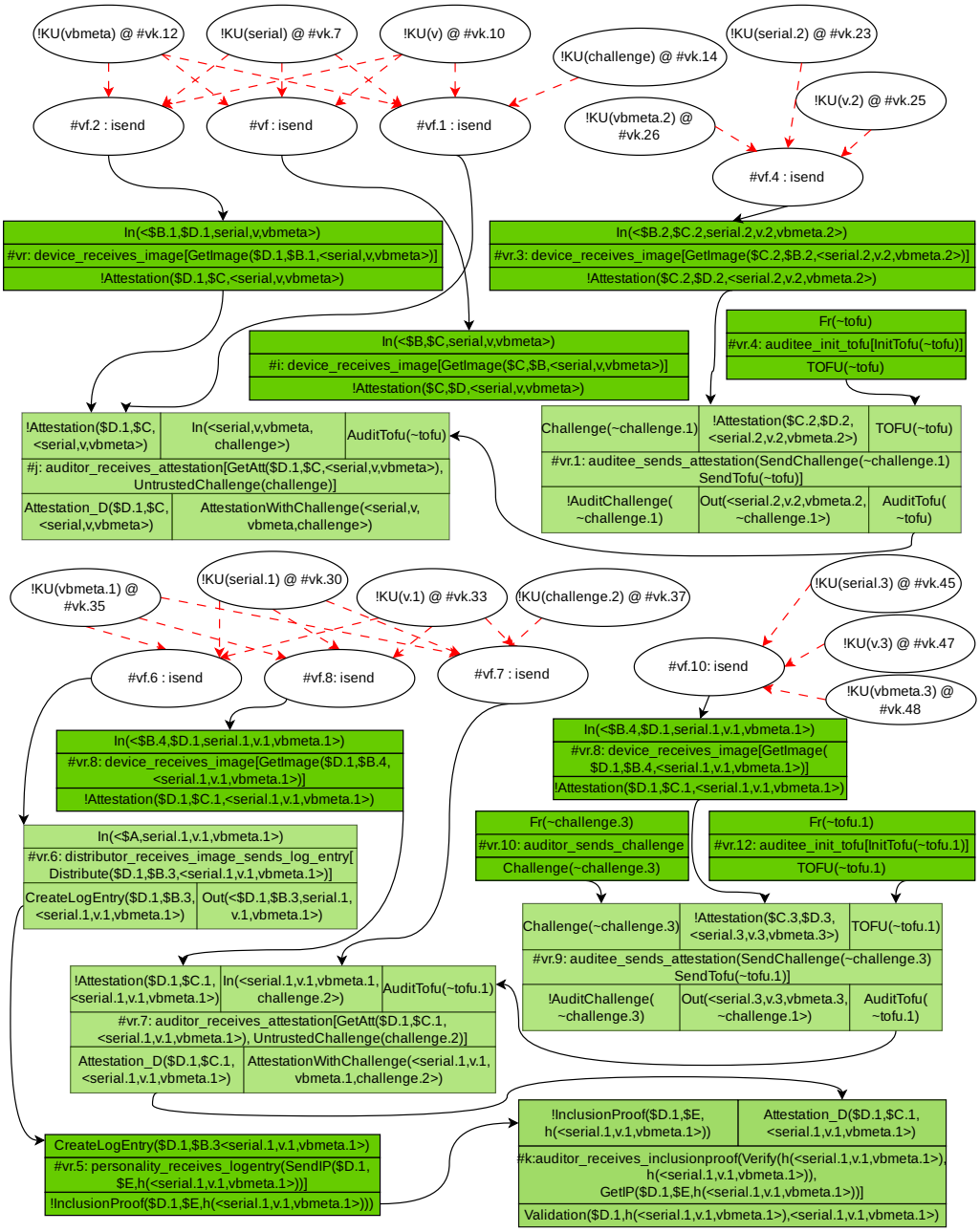


Figure A.1: This figure shows attack traces found by `Tamarin` for physical and on-path attacks. We use them to verify whether the specific attack would be successful if *Firmware Transparency* is not used. Once *Firmware Transparency* is used, `Tamarin` no longer detects any traces. Taken from our paper [100].

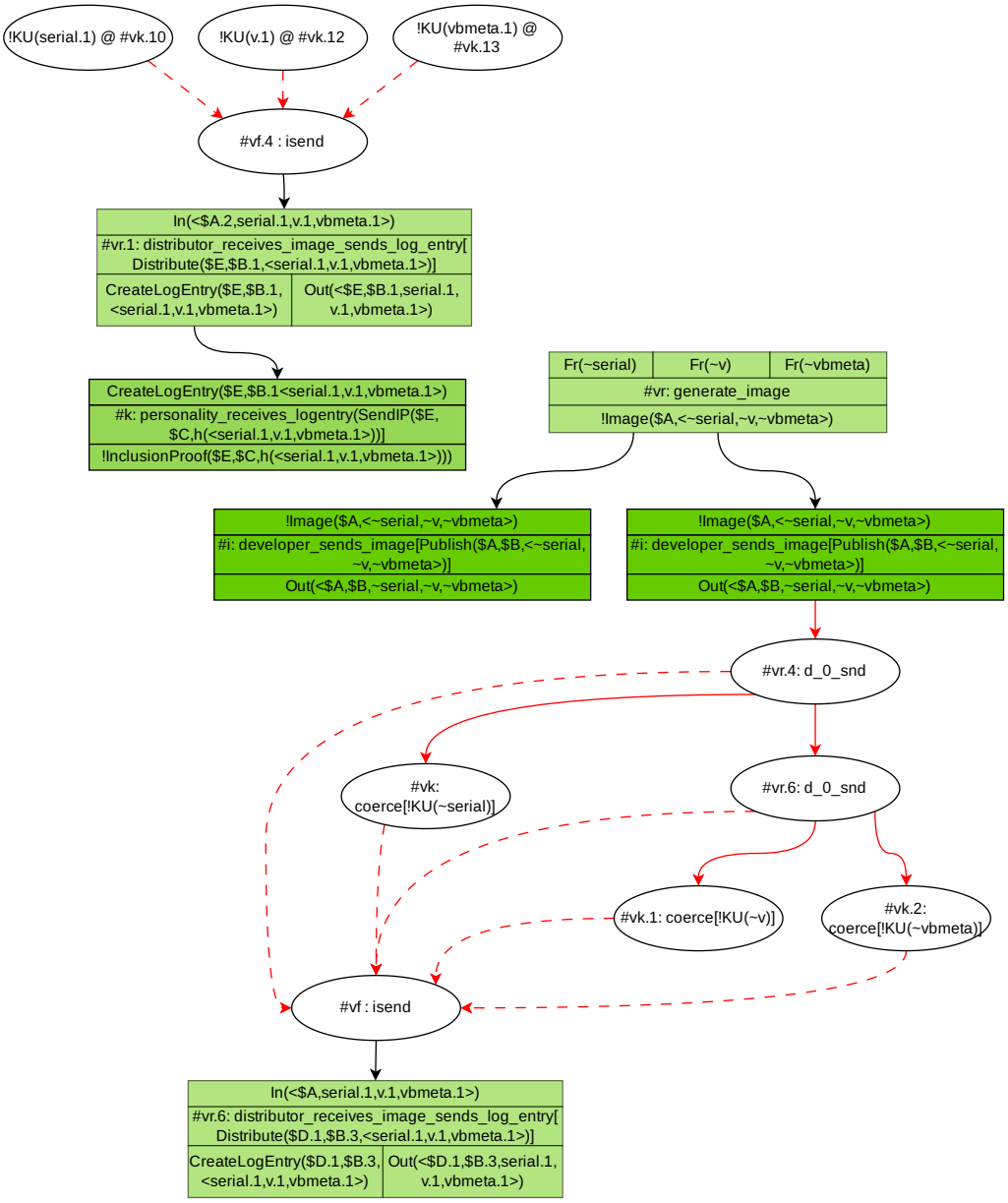


Figure A.2: This figure shows attack traces identified by `Tamarin` for insider attacks. Once *Firmware Transparency* is used, `Tamarin` no longer detects any traces. Taken from our paper [100].

Appendix B

Attestable Builds

B.1 Additional Figures

Figure B.1 shows the GitHub Action CI running the attestation.

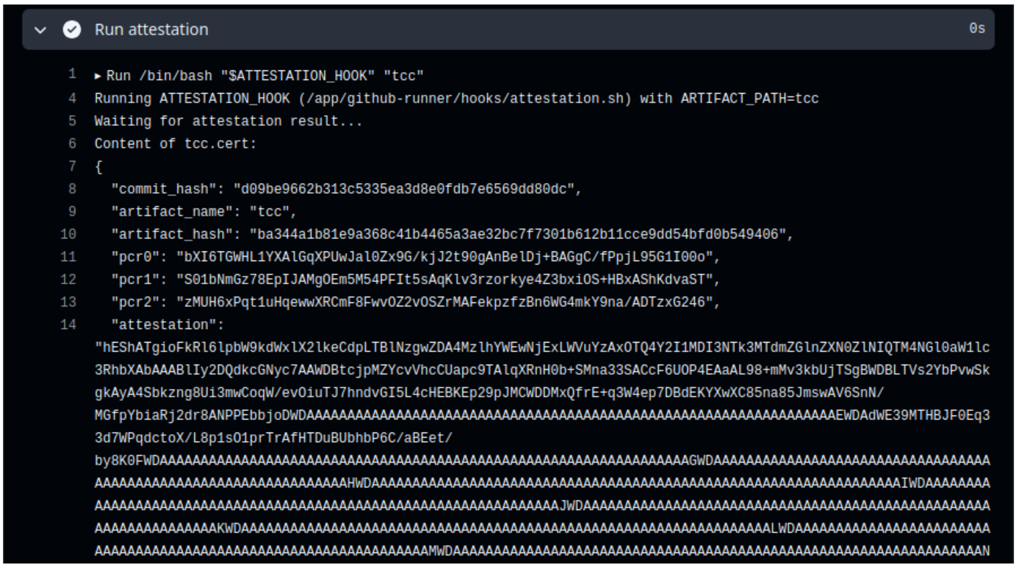


Figure B.1: Screenshot showing the GitHub Action CI running the attestation. Adapted from our paper [71].

B.2 GitHub Action Integration

The following listings, from our paper [71], present an exemplary GitHub Action workflow file, both in its original form (Listing B.1) and after applying the modifications required for our A-B prototype (Listing B.2).

```
1   name: CI for a Rust project
2
3   on:
4   push:
5   branches: [ "main" ]
6   pull_request:
7   branches: [ "main" ]
8
9   jobs:
10  lint:
11  runs-on: ubuntu-24.04
12  name: Lint (clippy & fmt)
13  steps:
14  - name: Check out code
15  uses: actions/checkout@v4.2.0
16  - name: Lint
17  run: cargo clippy --verbose
18  - name: Format
19  run: cargo fmt -- --check
20
21  build-and-test:
22  runs-on: ubuntu-24.04
23  name: Build and Test
24  steps:
25  - name: Check out code
26  uses: actions/checkout@v4.2.0
27  - name: Build
28  run: cargo build --verbose
29  - name: Test
30  run: cargo test --verbose
31
32
33  - name: Upload artifacts
34  uses: actions/upload-artifact@v4.6.0
35  with:
36  name: artifacts
37  path: |
38  target/debug/executable
39  _
```

Listing B.1: GitHub Action workflow file for a small Rust project.

```
name: CI for a Rust project

on:
push:
branches: [ "main" ]
pull_request:
branches: [ "main" ]

jobs:
lint:
runs-on: attested-build-runner
name: Lint (clippy & fmt)
steps:


- name: Lint
run: cargo clippy --verbose
- name: Format
run: cargo fmt -- --check

build-and-test:
runs-on: attested-build-runner
name: Build and Test
steps:


- name: Build
run: cargo build --verbose
- name: Test
run: cargo test --verbose
- name: Attestation
run: ATTESTATION_HOOK target/debug/executable
- name: Upload artifacts
uses: actions/upload-artifact@v4.6.0
with:
name: artifacts and certificate
path: |
target/debug/executable
target/debug/executable.cert
```

Listing B.2: Adapted version of the GitHub Action workflow file using our prototype.

B.3 Security Properties

This section outlines the lemmas used for our formal verification (section 7.3). We use single letter variable names for better readability and to keep lemmas short. See Table B.1 for the used notation.

Code Manipulation The following lemma shows that the adversary is able to successfully compromise the code without detection when the verification controls are not used.

$$\exists\, c,\, ct,\, \#i,\, \#j.\ ((\mathrm{Publish}(c)\ @\#i) \wedge (\mathrm{SecureCommit}(ct)\ @\#j)) \wedge \left(\neg\left(\mathrm{h}(c) = ct\right)\right)$$

Table B.1: Notations used in Tamarin lemmas for A-Bs

Notation	Description
c, ct, a, at, p, ip	c(code), ct(commit), a(artifact), at(attestation), p(pcr0), ip(incl. proof)
$Commit(c)$	Artifact author commit code to RHP.
$Publish(c)$	Publish code to adversary network.
$InitImage(c)$	Initialize image with PCR and publish it via the adversary network.
$InitBuild(c)$	Fetch Code from adversary network.
$Secure\text{-}Commit(ct)$	Store commit hash before entering untrusted execution state.
$Commit\_Verify(c, ct)$	Verify the commit hash (h(c) = ct).
$Artifact(a)$	Provide build artifact.
$Attestation(at)$	Provide attestation document based on PCR from adversary network.
$LogEntry(ip)$	Provide incl. proof of given log entry.
$LogVerify(f, ip)$	Verify inclusion proof for given f where f = <ct,h(a),at>
$ATVerify(f, at)$	Verify attestation for given f where f = <c,h(a),p>.

This lemma shows that the adversary is not able to compromise the code without detection when using the specific verification controls.

$$\forall c,\ ct,\ \#i,\ \#j,\ \#k,\ \#l.\ ((((Commit(c)\ @\#i) \wedge (Publish(c)\ @\#j)) \wedge (SecureCommit(ct)\ @\#k)) \wedge (CommitVerify(c,\ ct)\ @\#l) \Rightarrow (h(c) = ct))$$

Build Asset Manipulation

$$\exists c,\ ct,\ at,\ ip,\ \#i,\ \#j,\ \#k,\ \#l.\ (((((Publish(c)\ @\#i) \wedge (SecureCommit(ct)\ @\#j)) \wedge (Attestation(at)\ @\#k)) \wedge (LogEntry(ip)\ @\#l)) \wedge (\neg(h(c) = ct))) \wedge (\neg(h(< ct, h(c), h(c) >) = ip))$$

The lemma below verifies the inclusion proof using the action fact $LogVerify(< ct, h(a), at >, ip)$, and Tamarin does not detect any trace, where such an attack is possible.

$$\forall c,\ ct,\ a,\ at,\ ip,\ \#i,\ \#j,\ \#k.\ (((Publish(c)\ @\#i)\ (CommitVerify(c,\ ct)\ @\#j)) \wedge (LogVerify(< ct,\ h(a),\ at >,\ ip)\ @\#k)) \Rightarrow ((h(c) = ct) \wedge (h(< ct,\ h(a),\ at >) = ip))$$

Build Infrastructure Manipulation

$$\begin{aligned}
&\exists c, ct, a, p, at, \#i, \#j, \#k, \#l, \#m. \\
&(((((\mathsf{Publish}(c)\ @\#i)(\mathsf{SecureCommit}(ct)\ @\#j)) \wedge \\
&(\mathsf{Artifact}(a)\ @\#k)) \wedge (\mathsf{InitImage}(p)\ @\#l)) \wedge \\
&(\mathsf{Attestation}(at)\ @\#m)) \wedge (\neg(< c, h(a), p >= at))
\end{aligned}$$

The following lemma utilizes the verification control $ATVerify(..)$ provided by Attestable Builds. `Tamarin` cannot find any trace where an adversary is able to successfully compromise the build image without detection.

$$\begin{aligned}
&\forall c, ct, a, p, at, \#i, \#j, \#k, \#l, \#m, \#n. \\
&((((((\mathsf{Publish}(c)\ @\#i)(\mathsf{SecureCommit}(ct)\ @\#j)) \wedge \\
&(\mathsf{Artifact}(a)\ @\#k)) \wedge (\mathsf{InitImage}(p)\ @\#l)) \wedge \\
&(\mathsf{Attestation}(at)\ @\#m)) \wedge \\
&(\mathsf{ATVerify}(< c, h(a), p >, at)\ @\#n)) \\
&\Rightarrow (< c, h(a), p >= at)
\end{aligned}$$

Repository Spoofing The following lemma shows that an adversary would be able to successfully spoof a repository when the corresponding verification control is not involved.

$$\begin{aligned}
&\exists c, ct, a, at, ip, \#i, \#j, \#k, \#l, \#m, \#o. \\
&(((((((\mathsf{Commit}(c)\ @\#i) \wedge (\mathsf{Publish}(c)\ @\#j)) \wedge \\
&(\mathsf{SecureCommit}(ct)\ @\#k)) \wedge \\
&(\mathsf{CommitVerify}(c, ct)\ @\#l)) \wedge (\mathsf{Artifact}(a)\ @\#m)) \wedge \\
&(\mathsf{Attestation}(at)\ @\#n)) \wedge (\mathsf{LogEntry}(ip)\ @\#o)) \\
&(\neg(h(< h(c), h(a), at >) = ip))
\end{aligned}$$

Our model involves a verification step $RepositoryVerify(..)$ performed by the artifact author to mitigate repository spoofing. The following lemma shows that an adversary cannot successfully spoof a repository when using Attestable Builds.

$$\begin{aligned}
&\forall c, ct, a, at, ip, r, \#i, \#j, \#k, \#l, \#m, \#o, \#p. \\
&(((((((\mathsf{Commit}(c)\ @\#i) \wedge (\mathsf{Publish}(c)\ @\#j)) \wedge \\
&(\mathsf{SecureCommit}(ct)\ @\#k)) \wedge \\
&(\mathsf{CommitVerify}(c, ct)\ @\#l)) \wedge (\mathsf{Artifact}(a)\ @\#m)) \wedge \\
&(\mathsf{Attestation}(at)\ @\#n)) \wedge (\mathsf{LogEntry}(ip)\ @\#o)) \\
&(\mathsf{RepositoryVerify}(r, ip)\ @\#p)) \Rightarrow ((h(c) = ct) \wedge (r = ip))
\end{aligned}$$

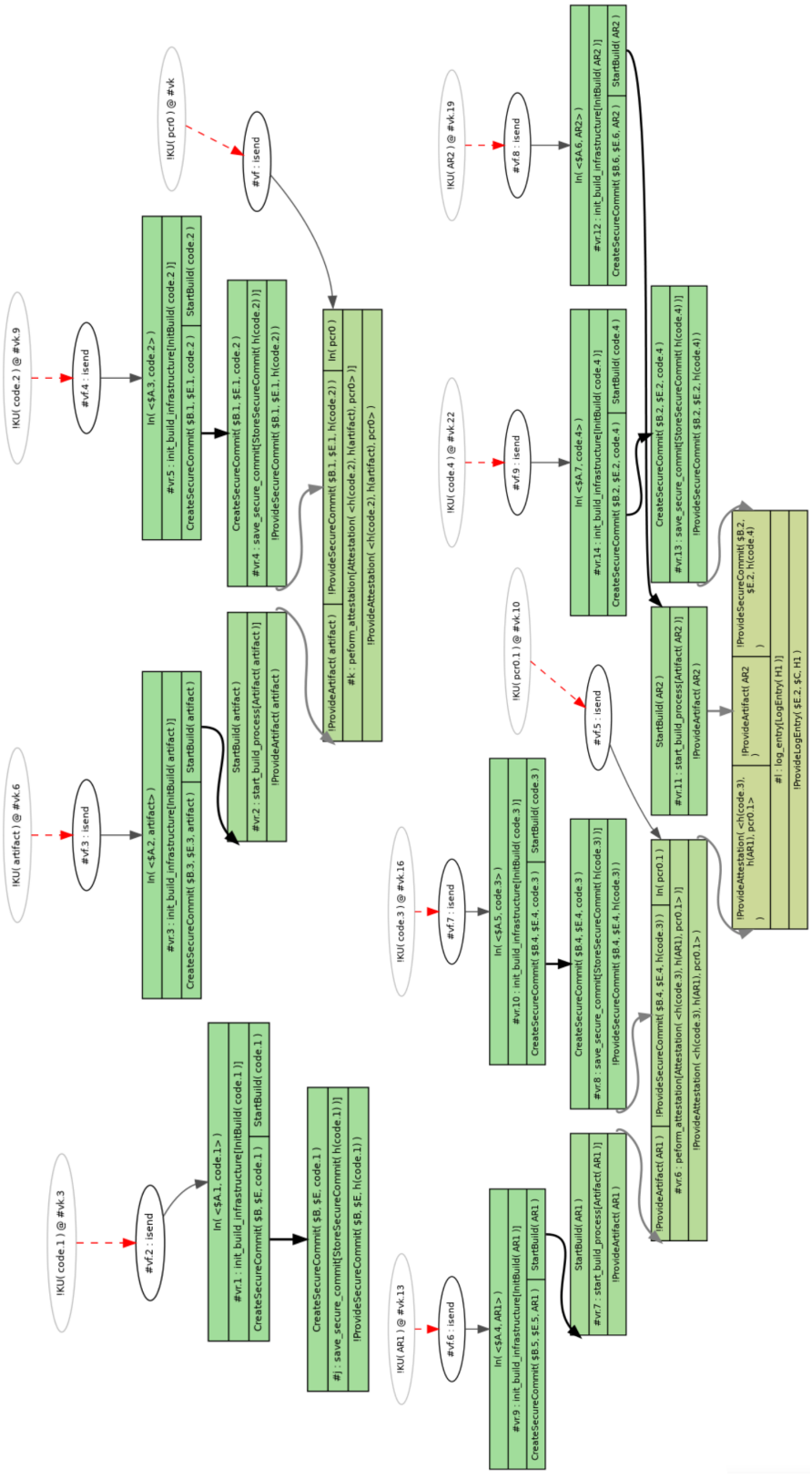


Figure B.2: This plot illustrates attack traces showing that our initial adversary assumptions would affect regular build systems. When enabling the A-B constraints, `Tamarin` fails to find any. Taken from our paper [71].

B.4 Additional Evaluation Material

This appendix includes additional data and plots, from our paper [71], generated by performing the practical evaluation (section 7.2). Table B.2 lists the dependencies and reverse dependencies for the unreproducible Debian packages that we have selected for our evaluation. Table B.3 shows all individual durations from our main evaluation. Tables B.4–B.5 show all individual durations from the job number experiments for XZ Utils and Verifier Client respectively. Figure B.4 is a larger version of Figure 7.3. Figure B.3 shows the stacked bar chart for the complex targets.

Table B.2: Selected unreproducible Debian packages.

Package	Number of Dependencies	Number of Reverse Dependencies
ipxe	3	2
hello	4	3
gprolog	4	2
scheme48	4	2
neovim	16	39

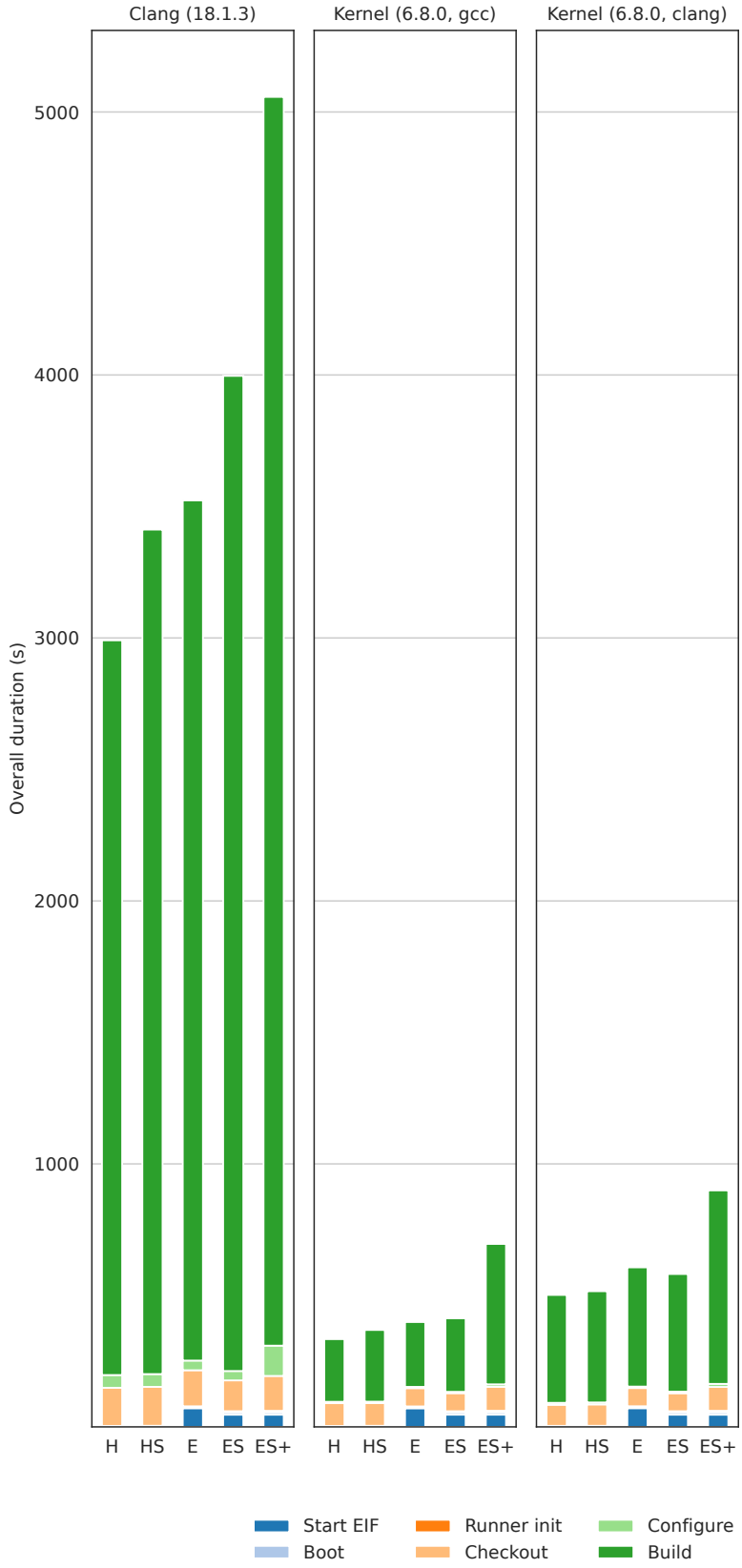


Figure B.3: This figure shows a variant of Figure B.4, where the large targets *LLVM Clang* and the *Linux Kernel* are built without using a sandbox on the host *H* and inside an enclave *E*. Taken from our paper [71].

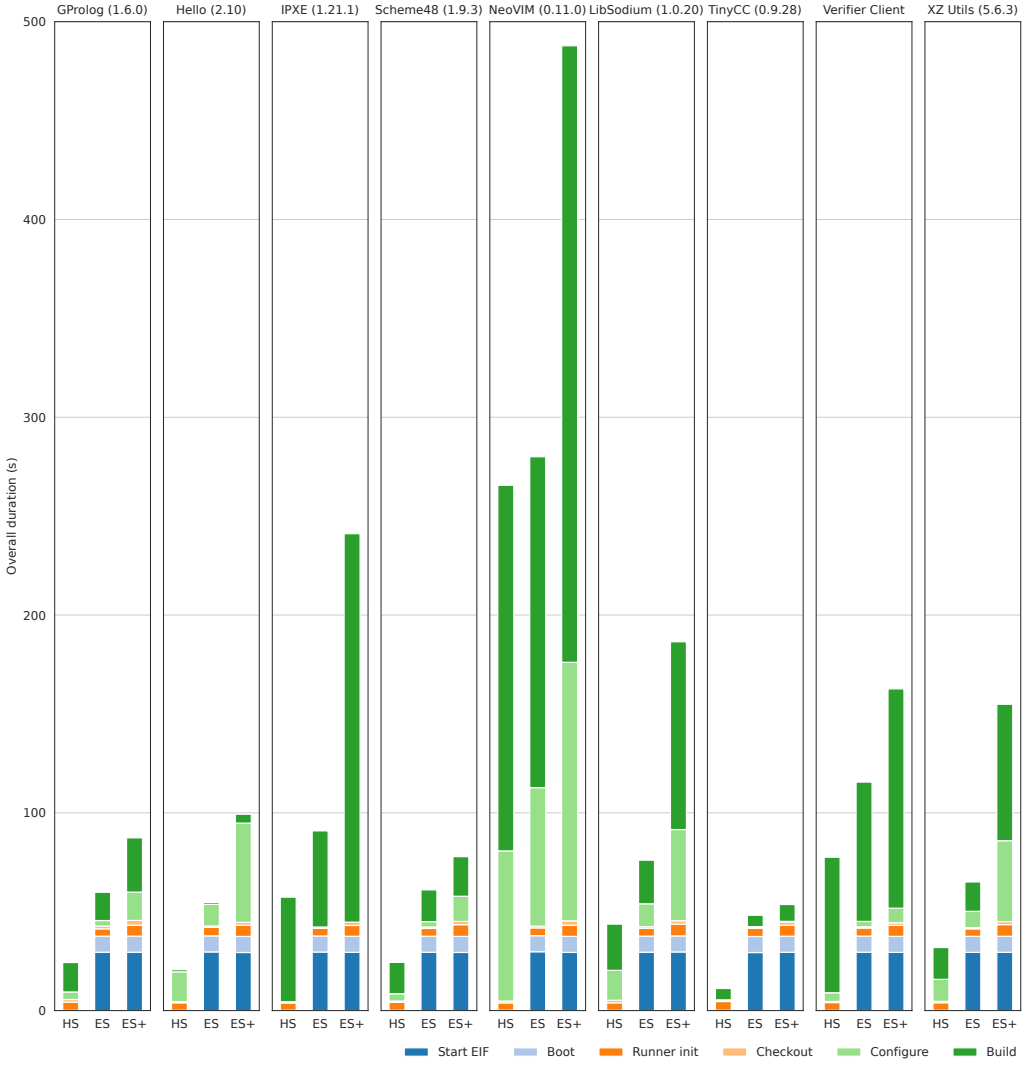


Figure B.4: This figure is a taller version of Figure 7.3 and illustrates the duration of each build stage was measured for selected open-source projects using three build configurations: a sandbox directly on the host (HS); a sandbox (containerd) running within a secure enclave (ES); and a hardened sandbox (gVisor) running with a secure enclave (ES+). Taken from our paper [71].

Table B.3: This table shows the build times for all targets and configurations. Some entries are missing due to unmet dependencies in Amazon Linux 2023. The values presented in this table are given in seconds and represent the average of three iterations. Taken from our paper [71] and reformatted.

Target	**Mode**	Start EIF	Boot	Runner init	Checkout	Configure	Build
Clang (18.1.3)	H	0.0s	0.1s	3.7s	145.0s	48.4s	2793.7s
	HS	0.0s	0.1s	4.2s	148.0s	48.5s	3211.3s
	E	71.0s	3.1s	4.1s	137.2s	36.7s	3270.9s
	ES	46.4s	9.1s	4.1s	117.2s	35.6s	3784.1s
	ES+	46.4s	9.1s	5.5s	132.8s	115.1s	4748.5s
Kernel (6.8.0, gcc)	H	0.0s	0.1s	3.5s	86.5s	5.3s	238.2s
	HS	0.0s	0.1s	4.1s	86.1s	5.6s	272.6s
	E	70.8s	3.1s	3.9s	69.5s	4.8s	246.5s
	ES	46.4s	9.1s	3.9s	68.9s	5.0s	279.4s
	ES+	46.4s	9.1s	5.7s	91.0s	9.6s	533.8s
Kernel (6.8.0, clang)	H	0.0s	0.1s	3.8s	80.1s	7.5s	410.2s
	HS	0.0s	0.1s	4.1s	81.3s	7.4s	423.0s
	E	71.1s	3.1s	4.2s	69.3s	5.5s	453.3s
	ES	46.4s	9.1s	4.0s	68.6s	5.9s	447.3s
	ES+	46.3s	9.1s	5.6s	90.8s	11.6s	735.7s
GProlog (1.6.0)	H	0.0s	0.1s	3.8s	1.3s	3.7s	12.8s
	HS	0.0s	0.1s	4.2s	1.3s	3.8s	14.9s
	E	43.5s	3.1s	4.0s	1.3s	2.9s	13.5s
	ES	29.5s	8.1s	3.7s	1.4s	2.8s	14.3s
	ES+	29.5s	8.1s	5.5s	2.4s	14.3s	27.4s
Hello (2.10)	H	0.0s	0.1s	3.6s	0.6s	14.5s	0.9s
	HS	0.0s	0.1s	3.8s	0.6s	15.0s	1.1s
	E	43.7s	3.1s	3.8s	0.6s	11.5s	0.9s
	ES	29.7s	8.1s	4.3s	0.7s	11.0s	1.0s
	ES+	29.4s	8.1s	5.7s	1.4s	50.2s	4.5s
IPXE (1.21.1)	H	0.0s	0.1s	3.9s	0.6s	0.0s	n/a
	HS	0.0s	0.1s	3.7s	0.6s	0.0s	52.9s
	E	43.5s	3.1s	4.0s	0.7s	0.0s	45.8s
	ES	29.6s	8.1s	3.9s	0.7s	0.0s	48.6s
	ES+	29.5s	8.1s	5.5s	1.6s	0.0s	196.4s

Table B.3: This table shows the build times for all targets and configurations. Some entries are missing due to unmet dependencies in Amazon Linux 2023. The values presented in this table are given in seconds and represent the average of three iterations. Taken from our paper [71] and reformatted. (Continued)

Target	Mode	Start EIF	Boot	Runner init	Checkout	Configure	Build
Scheme48 (1.9.3)	H	0.0s	0.1s	3.7s	0.6s	3.5s	14.8s
	HS	0.0s	0.1s	4.0s	0.7s	3.7s	15.9s
	E	43.4s	3.1s	3.7s	0.7s	2.8s	15.2s
	ES	29.5s	8.1s	3.9s	0.7s	2.7s	16.0s
	ES+	29.4s	8.1s	5.9s	1.6s	12.7s	20.0s
NeoVIM (0.11.0)	H	0.0s	0.1s	3.7s	0.9s	66.1s	255.9s
	HS	0.1s	0.1s	3.7s	0.9s	75.9s	184.9s
	E	43.5s	3.1s	3.8s	0.9s	65.3s	158.9s
	ES	29.8s	8.1s	3.9s	0.9s	70.0s	167.3s
	ES+	29.4s	8.1s	5.7s	2.2s	130.7s	311.7s
LibSodium (1.0.20)	H	0.0s	0.1s	3.9s	1.1s	13.2s	20.2s
	HS	0.0s	0.1s	3.8s	1.3s	15.1s	23.4s
	E	43.8s	3.1s	3.7s	0.8s	10.6s	19.9s
	ES	29.5s	8.1s	4.0s	0.8s	11.5s	22.1s
	ES+	29.6s	8.1s	5.9s	1.7s	46.1s	94.9s
TinyCC (0.9.28)	H	0.0s	0.1s	3.6s	0.7s	0.1s	4.5s
	HS	0.0s	0.1s	4.5s	0.7s	0.1s	5.7s
	E	43.5s	3.1s	3.9s	0.7s	0.1s	5.6s
	ES	29.4s	8.1s	4.2s	0.7s	0.1s	5.8s
	ES+	29.5s	8.1s	5.6s	1.6s	0.4s	8.5s
Verifier Client	H	0.0s	0.1s	3.7s	0.8s	4.4s	69.4s
	HS	0.0s	0.1s	4.0s	0.6s	4.4s	68.5s
	E	43.4s	3.1s	3.9s	0.7s	3.2s	70.5s
	ES	29.5s	8.1s	4.1s	0.6s	2.7s	70.4s
	ES+	29.4s	8.1s	5.6s	1.3s	7.3s	110.9s
XZ Utils (5.6.3)	H	0.0s	0.1s	3.7s	0.7s	9.7s	13.6s
	HS	0.0s	0.1s	3.8s	0.7s	11.1s	16.1s
	E	43.5s	3.1s	3.9s	0.6s	8.1s	14.1s
	ES	29.4s	8.1s	3.8s	0.6s	8.2s	14.8s
	ES+	29.4s	8.1s	5.9s	1.5s	40.9s	69.0s

Table B.4: Build times for the C-based "XZ Utils" across all modes and job number combinations. All values in seconds and averaged over three iterations. Taken from our paper [71] and reformatted.

Target	Mode	Start EIF	Boot	Runner init	Checkout	Configure	Build
XZ Utils (5.6.3) (j1)	H	0.1s	0.1s	3.6s	0.6s	9.7s	36.1s
	HS	0.0s	0.1s	4.0s	0.6s	11.1s	43.3s
	E	43.4s	3.1s	3.8s	0.7s	8.1s	36.3s
	ES	29.5s	8.1s	4.1s	0.7s	8.2s	38.7s
	ES+	29.4s	8.1s	6.0s	1.5s	40.6s	86.3s
XZ Utils (5.6.3) (j2)	H	0.0s	0.1s	3.6s	0.7s	9.7s	19.4s
	HS	0.0s	0.1s	4.0s	0.7s	11.2s	23.1s
	E	43.5s	3.1s	3.6s	0.6s	8.1s	20.4s
	ES	29.4s	8.1s	3.7s	0.6s	8.3s	21.5s
	ES+	29.4s	8.1s	6.0s	1.6s	41.1s	69.2s
XZ Utils (5.6.3) (j3)	H	0.0s	0.1s	3.9s	0.7s	9.7s	15.7s
	HS	0.0s	0.1s	3.9s	0.7s	11.1s	18.6s
	E	43.5s	3.1s	4.0s	0.6s	8.1s	16.3s
	ES	29.4s	8.1s	3.9s	0.6s	8.3s	17.3s
	ES+	29.4s	8.1s	5.5s	1.5s	40.7s	66.9s
XZ Utils (5.6.3) (j4)	H	0.0s	0.1s	3.6s	0.7s	9.7s	13.6s
	HS	0.0s	0.1s	4.1s	0.7s	11.2s	16.1s
	E	43.5s	3.1s	4.2s	0.6s	8.2s	14.2s
	ES	29.4s	8.1s	3.7s	0.7s	8.3s	14.9s
	ES+	29.4s	8.1s	5.5s	1.5s	40.7s	69.0s
XZ Utils (5.6.3) (j5)	H	0.0s	0.1s	3.5s	0.7s	9.7s	13.7s
	HS	0.0s	0.1s	3.8s	0.7s	11.2s	16.2s
	E	43.4s	3.1s	3.8s	0.7s	8.1s	14.3s
	ES	29.4s	8.1s	3.8s	0.7s	8.2s	14.8s
	ES+	29.5s	8.1s	5.7s	1.6s	41.0s	70.5s
XZ Utils (5.6.3) (j6)	H	0.0s	0.1s	3.4s	0.7s	9.8s	13.8s
	HS	0.0s	0.1s	3.9s	0.7s	11.2s	16.3s
	E	43.8s	3.1s	3.9s	0.6s	8.2s	14.4s
	ES	29.5s	8.1s	4.0s	0.6s	8.3s	15.2s
	ES+	29.8s	8.1s	5.9s	1.6s	41.3s	71.2s
XZ Utils (5.6.3) (j7)	H	0.0s	0.1s	3.4s	0.7s	9.7s	13.8s
	HS	0.0s	0.1s	4.1s	0.7s	11.1s	16.4s
	E	43.6s	3.1s	3.9s	0.7s	8.1s	14.5s
	ES	29.4s	8.1s	3.9s	0.7s	8.3s	15.2s
	ES+	29.5s	8.1s	5.6s	1.5s	40.4s	71.2s
XZ Utils (5.6.3) (j8)	H	0.0s	0.1s	3.5s	0.7s	9.7s	13.7s
	HS	0.0s	0.1s	3.9s	0.7s	11.2s	16.4s
	E	43.5s	3.1s	3.7s	0.7s	8.1s	14.5s
	ES	29.4s	8.1s	4.1s	0.6s	8.3s	15.2s
	ES+	29.4s	8.1s	5.6s	1.5s	41.1s	72.2s

Table B.5: Build times for the Rust-based "Verifier Client" across all modes and job number combinations. All values in seconds and averaged over three iterations. Taken from our paper [71] and reformatted.

Target	**Mode**	Start EIF	Boot	Runner init	Checkout	Configure	Build
Verifier Client (j1)	H	0.0s	0.1s	3.6s	0.6s	4.3s	186.5s
	HS	0.0s	0.1s	3.9s	0.6s	4.6s	185.1s
	E	43.4s	3.1s	3.9s	0.6s	3.5s	181.2s
	ES	29.4s	8.1s	3.9s	0.6s	2.7s	181.2s
	ES+	29.5s	8.1s	5.8s	1.4s	7.3s	230.4s
Verifier Client (j2)	H	0.0s	0.1s	3.6s	0.6s	4.4s	100.0s
	HS	0.0s	0.1s	4.1s	0.6s	4.2s	99.1s
	E	43.4s	3.1s	3.9s	0.6s	3.1s	97.8s
	ES	29.4s	8.1s	3.8s	0.6s	2.8s	98.3s
	ES+	29.4s	8.1s	5.9s	1.4s	7.3s	138.8s
Verifier Client (j3)	H	0.0s	0.1s	3.4s	0.6s	4.3s	80.2s
	HS	0.0s	0.1s	3.9s	0.6s	4.2s	79.4s
	E	43.4s	3.1s	7.1s	0.6s	3.2s	80.1s
	ES	29.4s	8.1s	4.0s	0.6s	2.7s	80.6s
	ES+	29.4s	8.1s	5.9s	1.3s	7.3s	114.0s
Verifier Client (j4)	H	0.0s	0.1s	3.7s	0.7s	4.3s	69.2s
	HS	0.0s	0.1s	4.1s	0.6s	4.7s	68.3s
	E	43.4s	3.1s	3.6s	0.6s	3.2s	70.2s
	ES	29.4s	8.1s	3.8s	0.7s	2.7s	70.4s
	ES+	29.4s	8.1s	5.6s	1.4s	7.5s	110.6s
Verifier Client (j5)	H	0.0s	0.1s	3.6s	0.6s	4.2s	69.5s
	HS	0.0s	0.1s	3.8s	0.6s	4.4s	68.7s
	E	43.5s	3.1s	3.8s	0.7s	3.4s	70.7s
	ES	29.5s	8.1s	4.1s	0.6s	2.9s	71.7s
	ES+	29.7s	8.1s	5.6s	1.3s	7.5s	112.8s
Verifier Client (j6)	H	0.0s	0.1s	3.6s	1.0s	4.3s	70.8s
	HS	0.0s	0.1s	3.8s	0.6s	4.4s	69.7s
	E	43.8s	3.1s	3.8s	0.6s	4.2s	72.2s
	ES	29.4s	8.1s	3.9s	0.7s	2.7s	72.1s
	ES+	29.7s	8.1s	5.8s	1.3s	7.3s	115.3s
Verifier Client (j7)	H	0.0s	0.1s	3.7s	0.6s	4.2s	70.4s
	HS	0.0s	0.1s	4.2s	0.6s	4.6s	69.3s
	E	43.4s	3.1s	3.8s	0.7s	3.8s	72.3s
	ES	29.4s	8.1s	3.6s	0.6s	3.2s	72.1s
	ES+	29.5s	8.1s	5.7s	1.3s	7.5s	117.0s
Verifier Client (j8)	H	0.0s	0.1s	3.5s	0.6s	4.2s	71.3s
	HS	0.0s	0.1s	3.9s	0.6s	4.2s	70.3s
	E	43.4s	3.1s	3.8s	0.6s	3.1s	73.1s
	ES	29.4s	8.1s	4.0s	0.6s	3.2s	73.3s
	ES+	29.4s	8.1s	5.9s	1.4s	7.4s	120.0s

Appendix C

The Supply Chain Game

C.1 CVSS Vector Strings

Threat	Expl.	Equation	Vector String (Exploitability)
A1.1	3.89	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.85 \cdot 0.85$	CVSS:3.1/AV:N/AC:L/PR:N/UI:N
A1.2	3.89	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.85 \cdot 0.85$	CVSS:3.1/AV:N/AC:L/PR:N/UI:N
A1.3	1.62	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.62$	CVSS:3.1/AV:N/AC:H/PR:N/UI:R
A2.1	2.07	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.62 \cdot 0.62$	CVSS:3.1/AV:N/AC:L/PR:L/UI:R
A2.2	1.23	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:L/PR:H/UI:N
A2.3	1.23	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:L/PR:H/UI:N
A3.1	0.71	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:H/UI:N
A3.2	0.71	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:H/UI:N
A3.3	1.23	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:L/PR:H/UI:N
A4.1	1,62	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.62$	CVSS:3.1/AV:N/AC:H/PR:N/UI:R
A4.2	2.84	$8.22 \cdot 0.85 \cdot 0.77 \cdot 0.85 \cdot 0.62$	CVSS:3.1/AV:N/AC:L/PR:N/UI:R
A5.1	2.22	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:N/UI:N

Table C.1: CVSS vector strings before mitigation. Taken from our paper [101].

Mitigation	Expl.	Equation	Vector String
$D_{1,1} \cdot A_{1,1}$	**0.71**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:H/UI:N
$D_{1,1} \cdot A_{1,2}$	**0.71**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:H/UI:N
$D_{1,1} \cdot A_{1,3}$	**1.62**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.62$	CVSS:3.1/AV:N/AC:H/PR:N/UI:R
$D_{1,3} \cdot A_{1,1}$	**2.22**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:N/UI:N
$D_{1,3} \cdot A_{1,2}$	**2.22**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:N/UI:N
$D_{1,3} \cdot A_{1,3}$	**1.62**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.62$	CVSS:3.1/AV:N/AC:H/PR:N/UI:R
$D_{2,1} \cdot A_{2,2}$	**0.71**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.27 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:H/UI:N
$D_{2,2} \cdot A_{2,3}$	**0.00**	-	-
$D_{3,1} \cdot A_{3,1}$	**0.51**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.27 \cdot 0.62$	CVSS:3.1/AV:N/AC:H/PR:H/UI:R
$D_{3,2} \cdot A_{3,3}$	**0.00**	-	-
$D_{4,1} \cdot A_{4,1}$	**0.00**	-	-
$D_{4,2} \cdot A_{4,1}$	**0.00**	-	-
$D_{4,2} \cdot A_{4,2}$	**1.62**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.85 \cdot 0.62$	CVSS:3.1/AV:N/AC:H/PR:N/UI:R
$D_{5,2} \cdot A_{5,1}$	**1.62**	$8.22 \cdot 0.85 \cdot 0.44 \cdot 0.62 \cdot 0.85$	CVSS:3.1/AV:N/AC:H/PR:L/UI:N

Table C.2: CVSS vector strings after mitigation. Taken from our paper [101].