

DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2026

Geth Rebuild: Verifiable Builds for Go Ethereum

Vivi Andersson

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Author

Vivi Andersson
vivia@kth.se
M.Sc. Computer Science, Software Engineering
Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Stockholm, Sweden

Examiner

Martin Monperrus
monperrus@kth.se
EECS TCS
KTH Royal Institute of Technology

Supervisor

Javier Ron Arteaga
javierro@kth.se
EECS TCS
KTH Royal Institute of Technology

Abstract

The increasing prevalence of software supply chain attacks emphasizes the urgent need to expand security efforts across all sectors. Blockchain ecosystems, which are critical to finance, healthcare, and governance, demand particularly robust protections. Despite their importance, the security of blockchain software supply chains, including the widely-used Ethereum platform, remains underexplored.

This thesis addresses this gap by examining the supply chain of Geth, the most widely adopted Ethereum client. We present a comprehensive model of Geth's supply chain, identifying 13 relevant attack vectors from the literature and assessing their specific impact on Geth. A key focus is placed on mitigating build-based supply chain threats. Through iterative static binary analysis, we *identify* and *resolve* five major challenges to achieving reproducible builds in Geth. Additionally, we develop `Geth Rebuild`, a tool for independent verification of Geth binaries, providing a practical defense against such threats.

This work concretely strengthens the security of Geth's supply chain, and in extension, enhances the resilience of the Ethereum ecosystem. Further, the contributions of this thesis offer practical insights for analyzing a software supply chain, and achieving reproducible builds.

Keywords

Open-Source Security, Software Supply Chain, Attack Vectors, Reproducible Builds, Blockchain Systems Security, Ethereum, Geth.

Abstract

Den ökande närvaron av attacker mot leveranskedjan (supply chain) av programvara understryker behovet av att utöka arbetet för att säkra dessa. Blockchains är underliggande teknologi inom kritiska områden som finans, sjukvård och samhällsstyrning, och kräver därför särskilt robusta skydd. Trots deras betydelse är säkerheten i blockchains leveranskedja, inklusive det brett använda Ethereum-nätverket, fortfarande underutforskad.

Denna avhandling adresserar detta genom att undersöka leveranskedjan för Geth, den mest använda Ethereum-klienten. Vi presenterar en omfattande modell av Geths leveranskedja, identifierar 13 relevanta attackvektorer från litteraturen och utvärderar deras specifika inverkan på Geth. Ett fokus läggs på att mitigera hot mot försörjningskedjan i kompileringsfasen. Genom iterativ statisk binär analys *identifierar* och *löser* vi fem nyckelorsaker för att uppnå reproducerbara kompileringar (reproducible builds) för Geth. Vi utvecklar även `Geth Rebuild`, ett verktyg för oberoende verifiering av Geth-binärer, vilket ger ett praktiskt försvar mot dessa hot.

Detta arbete stärker konkret säkerheten i Geths leveranskedja, och i förlängningen även Ethereum-nätverket. Vidare bidrar denna avhandling med praktiska insikter för att uppnå reproducerbara kompileringar.

Nyckelord

Säkerhet av öppen källkod, Programvaras leveranskedja, reproducible builds, Blockchainsäkerhet, Ethereum, Geth.

Acknowledgements

I would like to sincerely thank my supervisor, Javier Ron, for his invaluable guidance and support throughout this work, especially during our deep dives into x86. I am very grateful to my examiner, Professor Martin Monperrus, for giving me feedback, nudging me to set deadlines, and especially for introducing me to high-quality research. I would also like to thank everyone in the CHAINS research group, you both provided me with support and inspiration.

I would also like to give Martin Swende at Geth special recognition for his discussions with me, sharing insight and pointers. I appreciate his active efforts to implement reproducibility patches, which has enabled this work.

Lastly, I want to thank my partner Hjalmar for his unwavering encouragement and support throughout this process; his engagement with my work has been as committed as mine.

Acronyms

CI	Continuous Integration
CD	Continuous Deployment
ELF	Executable and Linkable Format
EVM	Ethereum Virtual Machine
GCC	GNU C Compiler
Geth	Go Ethereum
LOC	Lines of Code
OS	Operating System
OSS	Open Source Software
SBOM	Software Bill Of Materials
SLSA	Supply-chain Levels for Software Artifacts
VCS	Version Control System

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Research Questions 2

1.3 Contributions 3

2 Background and Related Work **4**

2.1 The Software Supply Chain 4

2.2 Software Supply Chain Attacks 6

2.3 Securing Software Supply Chains 8

2.3.1 Reproducible Builds 8

2.3.2 Software Bill of Materials 9

2.3.3 Diverse Double Compilation 10

2.3.4 Ethereum Supply Chain Analysis 10

2.4 The Go Programming Language 11

3 Geth Supply Chain **12**

3.1 Overview 12

3.2 Supply Chain Links 12

3.2.1 Source 13

3.2.2 Build 14

3.2.3 Package 14

3.2.4 Dependencies 16

3.3 Summary 17

4 Geth Supply Chain Attack Vectors **18**

4.1 Methodology 18

4.2 Attack Vectors 19

4.2.1 Source Threats 19

4.2.2 Build Threats 22

4.2.3 Package Threats 24
4.2.4 Dependency Threats 27
4.3 Summary 29

5 Geth Reproducibility 30
5.1 Methodology 30
5.2 Reproducibility Issues 31
5.3 Summary 39

6 Geth Rebuild 40
6.1 Threat Model and Mitigation 40
6.2 Requirements 41
6.3 System Design 42
6.3.1 Overview 42
6.3.2 Components 43
6.4 Implementation Details 44

7 Evaluation 45
7.1 Methodology 45
7.2 Experimental Results 46
7.3 Summary 47

8 Discussion 49
8.1 Research Questions 49
8.1.1 Geth Supply Chain Analysis (RQ1) 49
8.1.2 Reproducibility of Geth (RQ2) 50
8.1.3 Independent Reproducibility Verification (RQ3) 52
8.2 Ethics and Sustainability 53
8.3 Limitations 53
8.4 Future Work 54

9 Conclusions 55

References 56

Chapter 1

Introduction

In today's largely interconnected software ecosystem, the widespread practice of software reuse has expanded the threat landscape beyond the individual application [9]. The attack surface for software has become increasingly dispersed to sources external to a project: rather than attacking software directly, adversaries can leverage any point in the *supply chain* of components contributing to the final software product [45]. Recent years have seen an increasing number of such attacks [23, 54] indirectly targeting downstream components, marking the need to expand the lens in software security work.

Blockchain technology, a relatively young class of software, requires rigorous security and software integrity measures due to their inherent role as trustless platforms for managing transactions and executing smart contracts across critical sectors such as finance [12], healthcare [47], and governance [30]. Despite its critical nature, supply chain security for blockchain software remains relatively unexplored [56].

Ethereum [18] is a state-of-the-art blockchain processing over 1,000,000 transactions every day [22]. Go Ethereum (Geth) is the de-facto standard implementation of the Ethereum protocol, written in the *Go* programming language (Golang). As the most widely used Ethereum execution client [49], Geth [27] is an extremely appealing target for attackers looking for financial gain or disruptions in the blockchain network.

1.1 Problem Statement

Despite the growing number of supply chain attacks, the realm of supply chain security stays underexplored for blockchain software. For Geth, the potential consequences of a successful attack are substantial financial losses and undermining trust in the Ethereum network.

The Go programming language is notable for its work integrating supply chain security into their language ecosystem, including the adoption of *reproducible builds* [10]. However, their application and effectiveness for Geth remain largely unexplored.

In this thesis, we address these gaps by conducting a comprehensive analysis of security implications for the Geth supply chain, identifying obstacles to implementing reproducible builds and evaluating the viability of reproducible build verification as a defense against supply chain attacks.

1.2 Research Questions

In this thesis, we aim to answer the following research questions:

RQ 1. What are the most relevant supply chain attack vectors of Geth?

By answering this question, we aim to raise awareness of supply chain security within the blockchain, while also providing a foundation for addressing one such risk in RQ2 and RQ3. We will begin by mapping the supply chain of Geth. Following this, we will evaluate the potential risks posed by known supply chain attack patterns, assessing their impact on the security of the Geth supply chain.

RQ 2. What are the challenges in achieving reproducibility for Geth binary artifacts, and what measures can be implemented to address these?

By discovering the root causes of unreproducibility and developing strategies to address them, we establish a baseline for achieving reproducibility for Geth artifacts. We will systematically rebuild Geth to identify discrepancies in binary artifacts. The analysis will focus on understanding the causes of discrepancies and developing potential patches or solutions to address them.

RQ 3. To what extent can the reproducibility of Geth binaries be verified across different architectures and versions using an independent verifier?

By *independent verifier*, we refer to a process that enables third-party entities to validate the integrity of Geth binaries. This question evaluates the current practical feasibility of reproducible build verification for Geth binary artifacts for detecting potential build-based attacks. We will develop a tool to independently rebuild and verify the reproducibility of Geth binaries, incorporating the patches and solutions identified in RQ2. We will then run the rebuilder across a wider range of versions and architectures to assess the effectiveness and limitations of reproducible build verification in practice.

1.3 Contributions

The main contributions of this thesis are:

- The modeling of the Geth supply chain and a supply chain security analysis to identify potential threats.
- The identification of 5 reproducibility issues in Geth, proposed solutions to correct these, and the adoption of patches in Geth.
- The development of `Geth Rebuild`, an open-source tool for automating the reproducibility verification process for Geth binary artifacts, publicly available at `https://github.com/chains-project/geth-rebuild/`.

Chapter 2

Background and Related Work

In this chapter, we give a theoretical foundation to the work and present related research. We describe the software supply chain and its components, the supply chain attack pattern, and methods for securing it. Lastly, we give an overview of supply chain security in the context of the programming language Go.

2.1 The Software Supply Chain

A software supply chain is defined as all components involved in producing a final software artifact [23], including libraries and tools in the development, testing, and building phases [32]. The software supply chain comprises a sequence of links, forming a chain of sub-components that contribute to the final software. Each link in the supply chain includes the *artifacts* produced, the *operations* performed on these, and the *actors* managing the link and connections [46]. A link in a previous step in the software supply chain, i.e. an *upstream* link, affects links *downstream*. The supply chain of the software artifact consumed by an end-user consists of all prior links in the chain. Auditing a software's supply chain thus requires consideration of both its supply chain and its dependencies.

The software supply chain can be modeled as a sequence of four links: source, build, package, and dependencies [57]. These are illustrated in Figure 2.1.1, where solid arrows point to a downstream link, and actor relations are represented by dashed arrows. Additionally, each system within a link is overseen by an external system administrator.

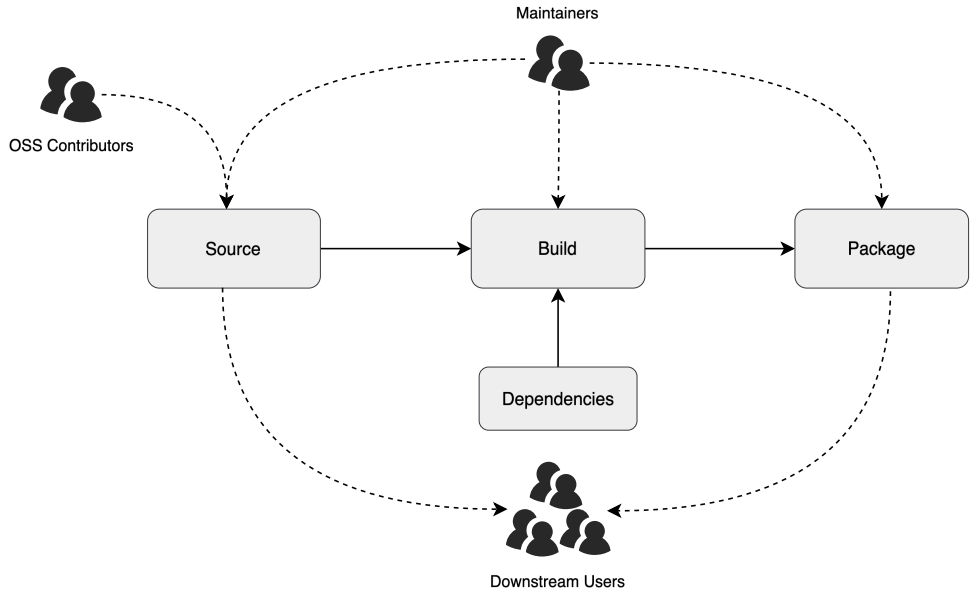

Figure 2.1.1: Supply Chain Model, adapted from [32, 57]

Source

Sources refer to artifacts authored or reviewed by people, for example, the source files of a project, or a specific *commit*, i.e. source code revision [57]. The actors affecting an Open Source Software (OSS) source component are mainly project maintainers and external contributors. Project maintainers are privileged and can configure the Version Control System (VCS) and integrate new code into the repository. External contributors have read access by default and can contribute through OSS sources through merge requests. A third actor, system administrators, are the entities with control over the VCS [32].

Build

The build is the link that transforms input artifacts (sources and dependencies) into outputs, such as executable files or archives [57]. This step includes compilation and packaging, often managed by a build automation platform in a Continuous Integration (CI)/Continuous Deployment (CD) system [32]. Commonly, actors for this link are project maintainers who configure and trigger builds, as well as potential administrators for the system.

Package

A package refers to bundled artifacts published for use to downstream consumers [57]. The packaged artifacts are distributed via distribution platforms, e.g. a tarball with source code on a hosting system or pre-compiled binaries via package managers [32]. Here, system administrators are any parties with control over the distribution platform, and project maintainers deploy artifacts to these.

Dependencies

Dependencies are project-external code components that input to a build, such as third-party OSS packages or a language runtime [57]. They are commonly downloaded and installed by a package manager [32], such as Go Modules (Sec. 2.4). As for the source link, actors interacting with the dependencies are project maintainers, contributors, and system administrators.

2.2 Software Supply Chain Attacks

An attack on the software supply chain targets a link downstream, which means at least two attacks are needed [46]. The first attack compromises a vulnerable link in the supply chain, which subsequently is leveraged for the second attack, on the main target. The entry point for of software supply chain attack is external to the exploited software's source code [1]. A software supply chain attack relies on the trust put in the supply chain, including its direct and transitive dependencies.

A definition [46] of a software supply chain attack constitutes four stages:

1. **Compromising** a weakness in a supply chain
2. **Alteration** of the supply chain by leveraging the weakness
3. **Propagation** of the malicious alteration downstream
4. **Exploitation** in a downstream link

This definition distinguishes supply chain threats from other types of attacks. For example, simply exploiting a bug in a system (step 4) is not a supply chain attack if no deliberate, *nefarious*, alteration of a supply chain has been made (steps 1-3) [46].

To illustrate the variety of attacks in open-source and closed-source software, we

present three concrete examples. First, the canonical SUNBURST attack [24] showcases supply chain attacks' potential for widespread effects. Second, the significant vulnerabilities found in the XZ Utils package [11] highlight the risks associated with widely used open-source libraries. Finally, a 2023 supply chain attack on the Ledger Connect Kit [36] exemplifies an attack targeting the blockchain ecosystem.

SUNBURST. Following the four-step attack pattern, the 2020 SUNBURST attack on the IT management company SolarWinds affected several state and private organizations worldwide [24]. Through an unauthorized modification of the build supply chain [53], a dynamically linked library (DLL) in the sources was altered. When running, the library started a parallel thread calling an attacker-added malicious package [41] containing a backdoor, i.e. an entry point providing unauthorized access to a system. The backdoored software propagated to end-users through regular software updates, thus carrying valid SolarWinds certificates [24]. Ultimately, attackers succeeded in exploiting systems for numerous organizations, stealing high-value assets such as credentials [41].

XZ Utils. The 2024 XZ Utils [66] compression library attackers demonstrated an advanced supply chain attack approach, marking the first severe compromise of a widely used OSS [11]. Attackers under the name "Jia Tan" added a backdoor into an XZ Util package that the Secure Shell (SSH) protocol daemon `sshd` depended on. The `sshd` package was in turn used in several Linux Operating System (OS) systems managers, enabling unauthenticated remote code execution of a wide range of Linux systems if successfully compromising its supply chain.

Through sophisticated social engineering, the actors behind Jia Tan gained control over the XZ Utils project in 2021 [68]. Following two years of seemingly benign code contributions, the attack commenced at the start of 2024. The malicious changes inserting the backdoor were introduced in smaller steps, as to hide suspicious changes. This included altering the sources, for example, introducing changes to evade kernel security measures and adding obfuscated malicious binaries in an overlooked directory. Additionally, extra malicious files were included in the package, after building [11]. A manual investigation of performance problems led to early detection of the attack before its widespread implementation in Linux systems, limiting damages to numerous systems.

Ledger Connect Kit. In December 2023, attackers conducted a supply chain attack on the cryptocurrency wallet provider Ledger, resulting in stolen funds from decentralized application users [36]. Attackers gained access to a Ledger `npm` package registry through phishing session tokens, leading to malicious releases of the `connect-kit` package connecting third-party apps to Ledger cryptocurrency wallets. Although the malicious change was detected quickly, the malicious package propagated to several projects, facilitated by automatic dependency updates [37] and the absence of version pinning [32], i.e. locking a certain dependency version.

2.3 Securing Software Supply Chains

Various approaches to securing the software supply chain have been proposed, including enhancing transparency, validity, and separation [46]. Below, we discuss security approaches related to the contributions of this thesis.

2.3.1 Reproducible Builds

Insufficient transparency in build processes often leads to opaque binaries [8], whose contents are difficult to determine. The trust put in the source code of software might be misplaced if not verifying the validity of its executable binary counterpart [34]. Non-deterministic builds embed parameters that may change in every rebuild, further exacerbating the challenge of detecting insertions of malicious code into binary artifacts. Reproducible builds aim to establish the source-to-binary semantic correctness by ensuring that the same source code always compiles into the same binary [34].

Def 1. A build is *reproducible* if given the same source code, build environment, and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts [50].

By comparing cryptographic digests of the output artifact, independent builders can increase the trust in the artifact. More specifically, reproducible builds allow downstream users to verify that no vulnerabilities have been included during or after the build process [34]. A core assumption for reproducible builds is that an attacker

cannot subvert all build toolchains that reproduce a build [25].

Most software today is not reproducible [25]. Causes of unreproducible builds can be categorized as either *uncontrolled build inputs* or *build-nondeterminism* [34]. Uncontrolled build inputs occur when the surrounding environment affects the build process, for example by embedding time into a binary. Contrarily, build non-determinism occurs when the build process acts randomly and embeds this randomness into the artifact, for example, a non-deterministic file system ordering. Challenges to implementing reproducible builds in a system include technical aspects of achieving build determinism [34] as well as communication with upstream projects, which must also be reproducible [25].

Previous work on reproducibility investigates achieving reproducibility in theory and practice, with a focus on Linux environments. Fourné et al. [25] explore the motivations and challenges of reproducible builds across developer teams. Interviews with developers experienced in reproducibility provide recommendations for integrating reproducibility into other projects. Bajaj et al. [2] contribute an empirical study on the process of making software reproducible, extending the understanding of root causes for unreproducibility. Whereas these works provide general knowledge of reproducibility, this thesis focuses on a practical implementation of reproducibility and its verification.

Drexel, Hänggi, and Méndez [17] implement an independent verifier of reproducible build claims for Arch Linux packages. They present a verification pipeline and patchings of Linux packages. Our work contrasts with this by not specifically considering Linux software, thereby extending reproducibility research beyond Linux packages and environments. Bitcoin Core adopts reproducible builds through GUIX and allows verifiers to post their hashes to a shared repository[1] for comparison [5]. This work aims to achieve a similar verification but for the Ethereum ecosystem.

2.3.2 Software Bill of Materials

A Software Bill Of Materials (SBOM) is a tool for promoting transparency of the software supply chain [46]. The SBOM records all components that make up the software, principally its third-party dependencies [3]. The data is attached to a deployed artifact to provide insight into the software's internals and composition.

[1] See `https://github.com/bitcoin-core/guix.sigs`

SBOMs are used as a foundation for increasing software supply chain security, with use cases such as vulnerability tracking through a software's dependencies [3].

While SBOMs establish all components in a software product, reproducible builds focus on recreating a certain build. Reproducible builds guarantee that the same input (source code) consistently yields the same output (binary artifacts), to enhance the software integrity. In contrast, SBOMs do not inherently verify the integrity of those components or their builds but rather provide transparency.

2.3.3 Diverse Double Compilation

Diverse Double Compilation (DDC) is a technique presented by David A. Wheeler that can be used to detect compromised compiler toolchains [69]. The approach mitigates the canonical *trusting trust attack* as described by Ken Thompson [64], in which the binary of a compiler is modified to propagate malicious code into any compiled artifacts, such as a compiler itself when bootstrapping. Self-replication in the malicious binary ensures that the attack persists through rebuilds of the compiler toolchain from clean source code.

The DDC approach leverages a trusted, assumingly correct, compiler to compile the untrusted compiler twice [69]. Any self-replicating insertions of malicious logic after the first, clean build, results in a binary difference, thus detecting the presence of trusting trust attacks.

While reproducible builds verify the source-to-binary correctness of builds, they do not inherently detect whether the compiler is compromised. DDC specifically mitigates risks associated with trusting a potentially malicious compiler.

2.3.4 Ethereum Supply Chain Analysis

Aumasson, Kolegov, and Stathopoulou [1] conduct a security review of Ethereum *beacon clients* (consensus clients). Their work includes an analysis of the supply chain of the four largest clients, focusing on the impact of dependencies. Soto-Valero, Monperrus, and Baudry [56] analyze the supply chain of two Ethereum nodes (execution and consensus) written in Java. They analyze the evolution and diversity of dependencies across the two nodes and suggest actions for strengthening their supply chains. In contrast to these works, our work looks at the Ethereum *execution* client

Geth. We consider the whole supply chain of Geth, beyond their dependencies.

2.4 The Go Programming Language

The Go Programming Language Go (or Golang) is an open-source, statically typed, and compiled programming language developed by Google, initially released in 2009 [48]. It emphasizes supply chain security through its design [67].

Go dependency management is handled through Go Modules [60]. A Go module is a collection of versioned Go packages, which are released together. The central `go.mod` file defines the module path, the Go version, and dependencies. The module path serves as the module's identifier, corresponding to the repository's hosting location (e.g., GitHub) [67], enabling a decentralized package repository without a separate hosting system.

Listing 2.1: Example `go.mod` file

```
1 module github.com/ethereum/hive
2 go 1.21.0
3
4 require (
5     golang.org/x/net v0.21.0
6     github.com/ethereum/c-kzg-4844 v0.4.0 // indirect
7 )
```

Listing 2.1 shows an example `go.mod` file. Dependencies are specified as direct or indirect, pinned, or locked to a version, and can only be updated by calling the `go` directive [60], e.g., `go mod tidy`, which prevents attacks exploiting automatic dependency updates.

Go dependencies are built from source [67], mitigating attacks on an upstream build system [32]. The source code is fetched from its hosting location and verified by its SHA256 hash digest before download, with the hash stored globally in the Go checksum database (`sumdb`) and locally in the `go.sum` file [67]. This ensures that malicious changes in a package's source version are automatically detected.

Notably, the Go toolchain has achieved reproducibility from version 21.0 [10]. As Go bootstraps itself, each Go compiler produces deterministic outputs, i.e., reproducible builds. Furthermore, these compilers undergo a double compilation procedure (Sec. 2.3.3), ensuring the absence of the trusting trust attack.

Chapter 3

Geth Supply Chain

This chapter presents a comprehensive view of the Geth supply chain, later used as a foundation for analyzing attack vectors in Chapter 4.

3.1 Overview

The Go Ethereum (Geth) project [27] is an OSS developed primarily in Go, containing over 365,000 Lines of Code (LOC)[1]. The first stable version was released in 2015 [28], as an official implementation of the Ethereum protocol. Geth is an Ethereum execution client, one of two parts comprising a node, i.e. a computer running the Ethereum protocol [19]. An Ethereum execution client is mainly responsible for generating transactions for the network and validating them via execution in the Ethereum Virtual Machine (EVM), which defines the rules for state transitions between blocks.

3.2 Supply Chain Links

In the following sections, we present the Geth supply chain in terms of the four supply chain links described in Section 2.1: *source*, *build*, *package*, and *dependencies*.

[1] As of version 1.13.x

3.2.1 Source

The sources of Geth are the files comprising the project repository. Geth manages its source code with Git and hosts it on GitHub[2]. Table 3.2.1 provides an overview of the file count and LOC for each language in the Geth source code[3]. The main logic is written in Go (88.45%). Additionally, C (4.90%) is used, for example, in cryptography, and JavaScript (4.16%) for EVM tracing. The source code is versioned as a single Go module with around 900 distinct packages.

Table 3.2.1: Distribution of Files and LOC by Language in Geth

Language	Files	LOC	LOC %
Go	1,179	323,900	88.56
C	68	17,920	4.90
JavaScript	40	15,210	4.16
Assembly	7	4,280	1.17
Shell	10	900	0.25
Sage	3	890	0.24
Java	4	770	0.21
NSIS	5	690	0.19
Solidity	2	350	0.09
Python	2	340	0.09
M4	3	330	0.09
Dockerfile	2	70	0.02
Makefile	1	50	0.01
HTML	1	50	0.01
Total	1,327	365,750	100

The primary actors interacting with the source link are OSS contributors and project maintainers. OSS contributors, who by default have read access, submit changes to the codebase through pull requests [20]. Since 2015, the project has seen over 1,000 contributors, with 198 active within the past year.

Project maintainers, primarily Geth core developers, play a more central role. They contribute to the codebase, configure services, and review pull requests. Their identities are either openly listed on their GitHub profiles or pseudonymous, with personal connections that can be derived from the profile. In contrast, OSS contributors can remain anonymous, for example by creating new accounts devoid of personal information.

[2]Available at `https://github.com/ethereum/go-ethereum`
[3]Scripts for numeric data at `https://github.com/vivi365/ssc-eth-clients/`

3.2.2 Build

Geth uses a containerized CI/CD pipeline to automate package builds and deployments. The pipeline uses Travis CI [29], with configurations defined in a `.travis.yml` file, specifying *when* and *how* to trigger actions. Figure 3.2.1 illustrates the general CI pipeline for unstable software versions.

The configurations also define the build environment, which is recreated from scratch for each build, or *job*. This includes retrieving and installing open-source components like the Go and C compilers. After initializing the Ubuntu Linux environment, used as the base image, the GNU C Compiler (GCC) compiler is installed via `apt-get`, with its version determined by the default of the specific Ubuntu release. During the build[4], Geth downloads the Go toolchain from the Go releases page[5], and is verified by its checksum. Although all contributors can modify the source code and some build configurations, the maintainers primarily oversee build configurations.

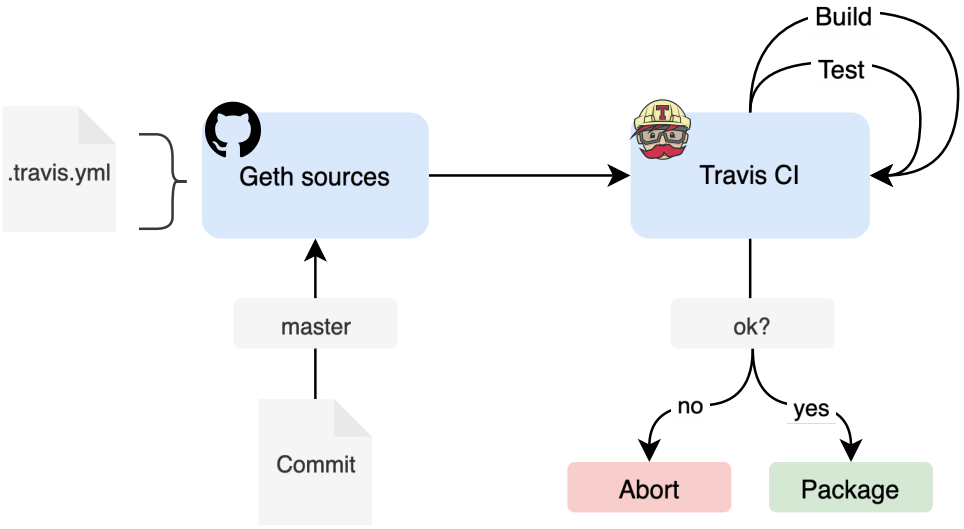

Figure 3.2.1: Geth Travis CI Pipeline for Unstable Builds

3.2.3 Package

Geth supports three main distribution methods: installing via package managers, downloading pre-built binaries, and running in a Docker container [21].

Package Managers. Geth can be installed automatically via various package managers [21]. These generally distribute the software in binary form, for example,

[4]Defined in `go-ethereum/build/ci.go`
[5]`https://go.dev/dl/`

Ubuntu Personal Package Archives (PPAs) and Homebrew Bottles.

Package managers generally build binaries on-site using source code and build instructions the project maintainers provide. Some allow users to build Geth locally, for example, Nix OS and Homebrew (when using the `--build-from-source` flag). Thus, distribution from package managers often includes both the build and distribution steps.

Listing 3.1: Excerpt from `.travis.yml` on Build Signing

```
% go run build/ci.go archive -type tar
        -signer OSX_SIGNING_KEY
        -signify SIGNIFY_KEY -upload gethstore/builds
```

Pre-built Binaries. Users can download stable and unstable binary builds for Linux, macOS, and Windows from the Geth website[6]. After successful compilation and testing (Fig. 3.2.1), the CI system archives the binaries to the respective package hosting service. These releases are signed with Geth public keys via OpenBSD's Signify, as shown in listing 3.1, allowing users to verify downloads with encryption programs like Pretty Good Privacy (PGP).

Docker Images. Geth can also be run in a Docker container for both stable and unstable builds. The base image is defined in source code, in `Dockerfile`, and secret variables needed are kept in Travis. For example, before pushing an image to Docker Hub, Travis authenticates to the Docker registry by retrieving Base64-encoded keys from the environment. These keys are then passed to the `docker login` command via standard input to prevent logging passwords, as shown in Listing 3.2.

Listing 3.2: Docker Authentication in `/build/ci.go`

```
// Retrieve the upload credentials and authenticate
user := getenvBase64("DOCKER_HUB_USERNAME")
pass := getenvBase64("DOCKER_HUB_PASSWORD")

if len(user) > 0 && len(pass) > 0 {
    auther := exec.Command("docker", "login", "-u", string(user), "--password-
    stdin")
    auther.Stdin = bytes.NewReader(pass)
}
```

[6] `https://geth.ethereum.org/downloads`

3.2.4 Dependencies

The dependencies of Geth, are primarily the required third-party libraries defined in the `go.mod` file. At the time of analysis[7], the project has 137 dependencies, of which 69 are direct and 68 indirect. The dependencies are other Go projects, but may also include other languages such as C, Java, and JavaScript. The dependencies are either hosted on GitHub or self-hosted.

Table 3.2.2: Dependencies and Unique Owners by Type

Owner Type	Total Deps	Owners	Direct	Indirect
Organization	80 (58.4%)	37 (45.1%)	39 (56.5%)	41 (60.3%)
Individual	47 (34.3%)	40 (48.8%)	21 (30.4%)	26 (38.2%)
Core developer	10 (7.3%)	5 (6.1%)	9 (13.0%)	1 (1.5%)
Total	137	82	69	68

In Table 3.2.2, we outline the distribution of dependencies determined by their ownership type: core developers, organizations, and individual contributors. The majority (80 out of 137) of dependencies belong to GitHub organizations or communities with multiple owners. The largest portion of these (21 out of 80) originates from either Go or Google, excluding standard built-in packages. Additionally, Amazon Web Services accounts for a large portion of these (13 out of 80), which is associated with the `devp2p` debugging package.

On the other hand, the dominant (40 out of 82) ownership type is repositories managed by single individuals. These 40 individuals together account for 47 dependencies. Over half (26 out of 47) of these are indirect, or transitive, dependencies. No single individual owns more than three dependencies. Among owners categorized as individuals, certain accounts may hold reputable statuses, such as Google employees or longstanding contributors to the Go community. However, the identities of some owners are not easily discernible. Lastly, 10 dependencies are created by the core development team themselves.

Go dependencies are managed by Go modules (Sec. 2.4). For Geth, the maintainers review updates to dependencies as other pull requests, and vulnerabilities in dependencies are detected through advisories or automatically e.g. by Dependabot

[7]Version 1.13.14

[13] alerts. The `go.mod` file defining their dependencies has been updated by 18 users in 57 commits in the past year, of which a majority are core developers.

3.3 Summary

The Geth supply chain comprises four key links, as outlined above. Each link involves various services, including GitHub, Travis CI, and Azure Blob Storage. As an OSS software, contributors can influence each link in distinct ways, ensuring accurate privileges for these is paramount. To secure the software supply chain effectively, it is essential to consider all links and connections.

Chapter 4

Geth Supply Chain Attack Vectors

In Chapter 3, we provided a comprehensive description of the Geth supply chain. In this chapter, we use this model to assess the impact of known supply chain attack vectors, addressing RQ1. The analysis is based on a 2023 taxonomy of attacks [32].

4.1 Methodology

To evaluate potential security risks in a software supply chain, various methodologies can be employed. Threat modeling is a general term for the systematic approach of assessing a system's security by describing a subject, finding potential threats, and actions to mitigate these [16]. Strategies to identify threats include using assessment mnemonics such as STRIDE [40], constructing attack trees describing methods of exploit, and considering concrete attack patterns collected from libraries of past exploits [52]. For software supply chain security, domain-specific approaches such as the MITRE Deliver Uncompromised framework [7], Google's Supply-chain Levels for Software Artifacts (SLSA) [57], and the Cloud Native Computing Foundation Software Supply Chain Best Practices [51] can be leveraged for specialized analysis.

Here, a 2023 taxonomy of software supply chain attacks presented by Ladisa et al. [32] serves as a framework for identifying potential threats to the Geth software supply chain. The taxonomy (attack tree) categorizes 107 attack vectors (i.e. attacker methods) which are derived from literature, past supply chain attacks, and proofs-of-concept. This taxonomy serves as a comprehensive state-of-the-art framework

for assessing supply chain-specific threats. The stakeholders of this analysis are downstream Geth users and the broader Ethereum ecosystem, both of whom could face significant consequences from a successful attack on Geth. In this analysis, we do not account for the attacker's ultimate goal. Instead, we focus on the *method of injecting malicious code* in the Geth supply chain.

We apply this taxonomy to the four Geth supply chain links outlined in Chapter 3, making three main modifications. First, we restructure the taxonomy from a package-based to a supply chain link-based perspective, organizing it into four branches. Second, we exclude certain attack vectors based on four criteria, described in Appendix A.1. Finally, we rename selected attack vectors to enhance clarity within this specific context. For each identified attack vector, we provide their description and an analysis of their threat to Geth. The analysis includes identifying known safeguards [32], which undergo a similar filtering (App. A.2)

4.2 Attack Vectors

Given the critical role of Geth in the Ethereum network, its software needs strong security measures. In the following sections, we present 13 unique supply chain attack vectors relevant to Geth, each mapped to one of the four supply chain links (Ch. 3). For each link, we provide two tables: one detailing the attack vectors and another mapping the corresponding safeguards. The safeguards table specifies which attack vectors they address and their current adoption status in Geth.

4.2.1 Source Threats

A threat to the Geth sources targets their code as hosted on GitHub. More specifically, attackers can tamper with code in any of the project's programming languages, including lower-level, less easily readable languages like C, Assembly, and Shell (Sec. 3.2.1). Adversarial changes in sources can propagate across all distribution forms, including package-managed distributions, pre-compiled binaries, and Docker images (Sec. 3.2.3). As a result, this link offers a broad attack surface. To address these risks, we have identified three primary attack vectors, along with corresponding mitigation strategies. These are summarized in Table 4.2.1 and their safeguards are detailed in Table 4.2.2.

Table 4.2.1: Supply Chain Attack Vectors Targeting Geth Sources

ID	Attack Vector
S1	Hypocrite Merge Request
S2	Contribute as Privileged Maintainer
S3	Exploit Vulnerability in VCS

Table 4.2.2: Safeguards for Source Threats (S1-S3) and Adoption Status

			Applies to			
Id	**Safeguard**	**Type**	*S1*	*S2*	*S3*	**Adopted**
SF1	Pull Request Review	Preventive	•	•		Yes
SF2	Protect Sensitive Branches	Preventive, Detective	•			Partially
SF3	User Account Management	Preventive, Corrective		•		Yes
SF4	Use Dedicated VCS	Preventive			•	Yes

S1. Hypocrite Merge Request

Attack Vector. Hypocrite merge requests leverage various tactics to conceal a malicious code integration into open-source repositories [32]. A simple approach exploits *rendering weaknesses* in version control systems, such as GitHub. For example, a GitHub pull request automatically hides code changes in files when a nullable character (eg., undefined ones) is added [42]. More advanced attacks transform immature vulnerabilities into exploitable ones through incremental modifications across multiple commits, potentially over a long time. This method was employed in the XZ-Utlis attack, as described in Section 2.2.

Analysis. Large and active OSS projects, such as Geth, receive numerous daily pull requests, often from external, anonymous contributors. While anonymity encourages collaboration, it also poses risks, as tracing malicious actions is challenging. To mitigate hypocrite merge request attacks, two safeguards can be employed, as listed in Table 4.2.2. Firstly, *pull request review* (*SF1*), serves as a preventative measure, to assess code changes, often conducted by humans. A robust review can detect code smells, potentially an immature vulnerability, and prevent it from developing further. Geth employs thorough manual code review[1]. Secondly, *protection of sensitive branches* (*SF2*) serves as both a preventive and detective measure. Maintainers can

[1]For example: `https://github.com/ethereum/go-ethereum/pull/29146`

enforce extra security checks for all users or first-time contributors, such as requiring specific status checks or signed commits. While Geth does not fully adopt these safeguards, they can enhance security, although, potentially at the cost of contribution ease.

S2. Contribute as Privileged Maintainer

Attack Vector. Project maintainers possess elevated privileges in their projects. By gaining the privileges of these users, attackers can merge malicious code in pull requests. Methods for acquiring elevated privileges may be by exploiting vulnerabilities in the developer's workstation, such as insecure systems or locally used open-source software components. This type of attack was observed in 2021, when attackers compromised the accounts of two PHP maintainers through a vulnerability in their VCS server, allowing them to insert a backdoor into the PHP source code via a malicious commit [31].

Analysis. As privileged users, Geth project maintainers are inherently attractive targets for attackers. Their public identities can exacerbate risks, as more personal information may be accessible. Attackers could use their elevated privileges to modify branch protection rules, for example allowing direct pushes to the repository for malicious actors. These attacks are highly visible in the VCS, making preventative pull request reviews (*SF1*) an effective safeguard Secure user account management (*SF3*) also prevents and corrects such attacks. While the extent of Geth maintainers' cyber hygiene practices is unclear, their professionalism and use of verified signatures for commits suggest awareness of these threats.

S3. Exploit Vulnerability in VCS

Attack Vector. Instead of gaining elevated privileges to push a malicious change, attackers can exploit vulnerabilities in the VCS itself. These vulnerabilities allow attackers to directly modify where the files are stored. For instance, a 2010 attack on the open-source file transfer service PROFTPD targeted a vulnerable server, and successfully altered the repository source files, introducing a backdoor [43].

Analysis. Compared to committing malicious changes in pull requests, this attack vector may be harder to detect due to reduced transparency. Past attacks on VCSs have targeted privately served solutions. To prevent such threat, a *dedicated version*

control system (*SF4*) should be used. Geth uses GitHub as their VCS, which decreases the direct control they have over their source code hosting, yet increases transparency for users and avoids potential vulnerabilities associated with custom VCS server implementations.

4.2.2 Build Threats

Build threats target the Geth compilation process to compromise the resulting package. These attack vectors exploit vulnerabilities in Geth's build system, Travis CI (see Sec. 3.2.2), which is responsible for compiling and distributing Geth binaries. Attacks on the build process can affect several distribution formats and are less transparent than source-level attacks, as the compilation process lacks visibility. We have identified three key attack vectors (Tab. 4.2.3) and their respective safeguards (Tab. 4.2.4).

Table 4.2.3: Supply Chain Attack Vectors Targeting Geth Builds

ID	Attack Vector
B1	Exploit Shared Build Resources
B2	Tamper with Build as Privileged Maintainer
B3	Exploit Vulnerability in Build System

B1. Exploit Shared Build Resources

Attack Vector. Exploiting shared build resources poses a significant threat when build environments or tools are compromised, allowing malicious code to propagate across multiple software projects. This type of attack leverages shared dependencies, tools, or infrastructure used by different programs or build jobs [32]. A notable example is the 2015 XCodeGhost attack, where a malicious version of the iOS Integrated Development Environment (IDE) XCode was distributed [70]. Developers using XCodeGhost unknowingly embedded malicious code such as password stealers into their software during compilation, resulting in 39 deployed malicious applications. Here, a single attack on the distribution of XCode mounted widespread exploitation across numerous applications.

Analysis. In Geth, containerized processes prevent the local propagation of malicious components by creating new, clean environments for each build. This use of *ephemeral*

Table 4.2.4: Safeguards for Build Threats (B1-B3) and Adoption Status

			Applies to			
ID	**Safeguard**	**Type**	*B1*	*B2*	*B3*	**Adopted**
SF5	Ephemeral Build Environment	Preventive	•			Yes
SF6	Isolation of Builds	Preventive	•			Partially
SF7	Reproducible Builds	Detective, Corrective	•	•	•	No
SF3	User Account Management	Preventive, Corrective		•		Yes

build environments (*SF5*) ensures that no artifacts from previous builds persist, reducing the risk of contamination. *Isolation of builds* (*SF6*) ensures each build remains fully independent, preventing any interaction between concurrent or previous builds. This isolation also protects sensitive information, such as signing keys, and blocks any influence between builds running simultaneously or sequentially. In Geth, different target architecture builds (e.g. for Linux) often share the same build job, increasing the risk of shared vulnerabilities. Lastly, *reproducible builds* (*SF7*), as described in Section 2.3.1 effectively detects and corrects this attack. As of writing, Geth does not support reproducible builds.

B2. Tamper with Build as Maintainer

Attack Vector. An attacker with elevated privileges can tamper with the build process, such as altering an installer or modifying a CI configuration. This mirrors the risk seen in source threats (Sec. 4.2.1) and can be achieved by compromising systems managed by the maintainer.

Analysis. While much of Geth's build configuration is defined in its source code, Travis CI retains control over critical settings and secret variables. An attacker who gains access to these configurations could, for instance, manually trigger a build that is uploaded with compromised signatures under the guise of legitimate Geth builds. Safeguards include prevention through user account management (*SF3*) and detection and correction through reproducible builds (*SF7*).

B3. Exploit Vulnerability in Build System

Attack Vector. Build systems are also susceptible to attacks from lower-privileged attackers. For example, in a 2019 attack, the server manager Webmin's vulnerable

build server was exploited, which in turn included a malicious change of their source code, enabling remote code execution on affected systems [6].

Analysis. Geth relies on secure cloud-based environments provided by its CI service, Travis CI. As a closed-source platform, Geth depends on the security of the CI provider since it lacks direct control or visibility into the internal mechanisms of the system. If the Travis build script were compromised, reproducible builds would serve as a crucial safeguard by allowing discrepancies to be detected.

4.2.3 Package Threats

Package threats target the distribution of built source code, representing the final stage in the supply chain 2.1.1) before reaching end-users unless they build the source code themselves. Given the different distribution forms (Sec. 3.2.3) and their distinct package archiving approaches, a single package attack only affects a limited range of distribution forms. As a result, the scope of package threats is generally narrower than that of source and build threats. We identify 5 unique attack vectors (Tab. 4.2.5) and 4 safeguards for these (Tab. 4.2.6).

Table 4.2.5: Supply Chain Attack Vectors Targeting Geth Packages

ID	Attack Vector
P1	Package Name Confusion
P2	Mask Legitimate Package
P3	Prevent Update to Non-Vulnerable Version
P4	Malicious Distribution as Privileged Maintainer
P5	Exploit Vulnerability in Hosting System

P1. Package Name Confusion

Attack Vector. Confusion over the name of a legitimate software package can lead users to inadvertently download a malicious version instead [32]. Attackers may exploit this by using *typosquatting* - for instance, changing `ethereum/go-ethereum` to `etherum/go-ethereum` - or by altering the owner/repository/package word order. Other techniques exploit broken code suggestions from generative AI services, i.e. LLM hallucinations [35].

Table 4.2.6: Safeguards for Package Threats (P1-P5) and Adoption Status

			Applies to					
ID	**Safeguard**	**Type**	*P1*	*P2*	*P3*	*P4*	*P5*	**Adopted**
SF8	Code Signing	Detective	•	•		•	•	Yes
SF9	Use Encrypted Channels	Preventive		•				Yes
SF10	Preventive Squatting	Preventive, Detective	•					Partially
SF3	User Account Management	Preventive, Corrective				•		Yes

Analysis. For Geth packages, the binary distributions are generally referenced by their location (via their webpage), rather than their name. Therefore, this attack vector is more relevant for the distribution forms that prompt users to search for the package by its name, for example, when downloading a PPA or Homebrew package. This attack vector may be extra relevant for the Docker distribution, considering its diverging name (`ethereum/client-go`) from other package forms. To mitigate these attacks, *preventive squatting* can be employed (*SF10*), which involves claiming similar package names, to prevent them from registration. This safeguard may already be in place, and in extension, can be adopted by package managers as well. Additionally, the use of *code signing* (Sec. 3.2.3)is a detective safeguard that effectively mitigates this, and a majority of all package threats, under the assumption of users verifying signatures.

P2. Mask Legitimate Package

Attack Vector. A legitimate package can be masked, i.e. hidden or substituted by a malicious one through various attacks [32]. This can happen through attacks like man-in-the-middle (MITM), where an attacker intercepts and alters the communication between the user and the server, or DNS cache poisoning, which corrupts the domain name resolution process and redirects the user to a malicious source. The user believes they are downloading the legitimate package, while in reality, they are retrieving a tampered version.

Analysis. The general risk of these attacks depends on the vulnerability of the communication channels used for package distribution. *Using encrypted channels* (*SF9*) prevents this threat. For instance, Geth requires the use of `https` for binary downloads, and their hosting solutions, such as Docker, also mandate encrypted

channels [15].

P3. Prevent Update to Non-Vulnerable Version

Attack Vector. This attack vector exploits the update mechanisms of software to prevent users from updating to secure, non-vulnerable versions of software. By keeping users on vulnerable versions, attackers increase the risk of exploiting known security flaws.

Analysis. Users rely on vulnerability advisories from Geth to stay informed about security updates. However, the update process for downstream users who rely on binary releases can be compromised if the official website does not reference or provide access to the latest secure binaries. For instance, during a website hosting bug, or if a CI build fails (Sec. 3.2.2), the latest version could be inaccessible. No clear mitigation for this risk is identified; however, users seeking a missing security version update are likely to report the issue, prompting a fix.

P4. Malicious Distribution as Privileged Maintainer

An attacker obtaining elevated privileges can push a malicious package version to the hosting for downstream users to download. This attack vector reflects the compromise of a privileged maintainer for sources (4.2.1) and the build (4.2.2). In this case, there may be more accounts to target, one for each service.

Analysis. For Geth, archiving a package only occurs during the build process (Sec. 3.2.2), making a malicious package upload unrelated to a triggered build process suspicious. Such an occurrence should raise concerns, increasing the visibility of this attack vector. This vector is prevented by the previously described safeguards code signing (*SF8*) and user account management (*SF3*).

P5. Exploit Vulnerability in Hosting System

Attack Vector. This attack vector targets the hosting infrastructure. An attacker could exploit vulnerabilities in these hosting systems to inject malicious packages or modify legitimate binaries, bypassing security measures set by the package maintainers. This could result in the distribution of compromised software to unsuspecting users who trust the hosting platform.

Analysis. For hosting systems, such as Azure for binary releases or Launchpad for Ubuntu PPAs, vulnerabilities in the infrastructure itself can expose the distribution to risks. If an attacker gains control over these systems, they could manipulate hosted binaries or alter the repository metadata, enabling the distribution of malicious or outdated packages. Users detect this attack by verifying package signatures (*SF8*).

4.2.4 Dependency Threats

Each software project has its distinct supply chain, so every attack vector targeting the source, build, and package also applies to every upstream dependency of Geth. In the case of Go, dependencies are fetched by their source code and built locally (Sec. 2.4). Therefore, an upstream dependency whose build or packaging has been compromised does not affect Geth. Additionally, some dependency-specific attacks, such as dependency confusion, which exploits name confusion in the dependency resolution process [4], and masking the download of dependencies, do not apply to Go. In the former case, the Go dependency ecosystem mitigates it by referencing dependencies by their concrete hosting location rather than by name, and in the latter, by cryptographically verifying the fetched source code (Sec. 2.4).

Given this, we identify 2 general dependency-specific threats, summarized in Table 4.2.7, and their safeguards in Table 4.2.8.

Table 4.2.7: Supply Chain Attack Vectors Targeting Geth Packages

ID	Attack Vector
D1	Source Code Exploit in Dependency
D2	Advertise Malicious Package as Benign

Table 4.2.8: Safeguards for Dependency Threats (D1-D3) and Adoption Status

			Applies to		
ID	**Safeguard**	**Type**	*D1*	*D2*	**Adopted**
SF11	Version Pinning	Preventive	•	•	Yes
SF12	Vulnerability Scanners in CI/CD	Corrective	•	•	No

D1. Source Code Exploit in Dependency

Attack Vector. This attack covers all potential source code exploits at the dependency site. Attacking a dependency of a target project rather than the target itself is an effective obfuscation technique [33]. This is especially true for large dependency trees, where the detection of such exploits is more complex to detect.

Analysis. These source threats from dependencies are generally mitigated through source code review, first when introducing a new dependency, and later, when updating the version. The use of *version pinning* (*SF11*) as is done in Go (Sec. 2.4) allows for auditing such upgrades of dependencies before integrating them into the project. This process should also consider transitive dependencies, why a small and shallow dependency tree is preferable. For reported vulnerabilities, Geth relies mainly on community-reported advisories. Here, the integration of *vulnerability scanners in the CI/CD* (*SF12*) could automatically stop builds or dependency integration that contains a known vulnerability.

D2. Advertise Malicious Package as Benign

Attack Vector. An attacker can create a malicious package and present it as benign, waiting for downstream users to incorporate it into their projects [32]. Past attacks [44] have shown that attackers may include the malicious code into the project later on, waiting until it is used by a specific target or has a wider use reach. This means that this attack vector can impact the current and future dependencies of a project, possibly leveraging a lack of transparency in dependency updates. Unlike *D1*, this vector involves intentional actions by the upstream dependency.

Analysis. To mitigate this risk, it is essential to carefully select a minimal dependency tree developed and maintained by reputable individuals or organizations. Ensuring that the individuals responsible for dependencies are trustworthy is crucial, especially for the individual owners further up the dependency tree (see Tab. 3.2.2). Considering the Geth maintainers practices, we generally observe careful choices in dependency updates[2]. Additionally, safeguards *SF11* and *SF12* also apply.

[2]For example: `https://github.com/ethereum/go-ethereum/pull/30402`

4.3 Summary

Exploits in the *sources* propagate to all distribution forms of Geth, yet source code changes are transparent in the VCS and can be mitigated by careful code review. The *build* process is more opaque, and therefore, a malicious change in this link is more challenging to detect. Building Geth in containerized processes is important but does not mitigate all build threats. For Geth *packages*, the use of digital signatures allows users to detect the majority of threats introduced after compilation. This makes a compromise in this link harder to go unnoticed. Finally, the *dependencies* of Geth poses 137 additional potential attack surfaces, why careful dependency management and auditing is an essential practice. Here, a lack of transparency in the dependency tree can pose threats to the supply chain.

Chapter 5

Geth Reproducibility

In Chapter 4, we have provided a comprehensive view of the potential impact of supply chain threats to Geth. In this Chapter, we focus on strengthening the build link of the supply chain. We address RQ2 by finding the concrete challenges to reproducibility in Geth binaries and finding measures for overcoming these.

5.1 Methodology

Recall Definition 2.3.1, stating that a *reproducible build* is a build of a specific artifact that can be recreated bit-for-bit, given the same source code, build environment, and build instructions. In this experiment, we scope our reproducibility investigation to Geth binary *Linux* releases of the `geth` command. These artifacts are built and released through the Geth CI pipeline (Sec. 3.2.2). We call the official binary the *reference artifact*, and the attempt to reproduce it through the *rebuilding* process the *rebuilt artifact*.

To rebuild an artifact, we need to retrieve the correct build configurations used for the reference artifact. We configure 4 general parameters, listed in Table 5.1.1 along with their retrieval source. The Git commit hash determines the source code version, which we retrieve from the Go binary buildinfo (`go version -m geth`). We retrieve the remaining parameters from build configurations specified in the source code, mainly in the `.travis.yml` file. We use a containerized environment to control these parameters during the rebuild.

Table 5.1.1: Configurations for Rebuilding Geth Linux Binary Artifacts

Configuration Type	Source	Defined by
Source Code Version	Binary Buildinfo	Git commit hash
Go Toolchain Version	`build/checksums.txt`	Version specification
C Toolchain	`.travis.yml`	Package installation
Build Command and Flags	`.travis.yml`	Build script

The rebuilding process of the reference artifact produces the rebuilt artifact. Next, we check the two artifacts for equality by comparing their *hash digests*. We strip the symbol tables of each artifact before the comparison, which prevents any varying debug information from affecting the reproducibility status. A comparison match indicates a reproducible reference artifact, i.e. a *successful rebuild*. Conversely, a comparison mismatch indicates an unsuccessful reproduction *attempt* or an unreproducible reference *artifact*, which we called an *unsuccessful rebuild*.

We analyze unsuccessful rebuilds for the root causes of their discrepancy. This is done through static analysis of the binaries, aided by comparison tools and manual inspection of the artifact's raw and disassembled formats. After identifying the root cause, we propose and implement *fixes* to correct the reproducibility issue, one at a time until achieving a fully reproducible Geth binary artifact.

5.2 Reproducibility Issues

We identified 5 reproducibility issues leading to discrepancies in rebuilt artifacts, we list these in Table 5.2.1 along with their corresponding fixes. These issues are defined as follows:

Def 2. A *Reproducibility Issue* refers to a condition causing a binary discrepancy in rebuilds that prevents an independent entity from verifying an official Geth binary.

This definition encompasses the two reproducibility categories *uncontrolled build inputs* and *build-nondeterminism*, as described in Section 2.3.1.

Table 5.2.1: Five Reproducibility Issues and their Fixes

ID	Reproducibility Issue	Status	Fix
RI1	Unreproducible Artifact Build IDs	Fixed	Exclude Build IDs from Verification
RI2	Conditional Date Embedding	Fixed	Fix Conditional Statement
RI3	System-Specific Paths	Fixed	Use another Ubuntu Distribution
RI4	Random Ubuntu Distribution in Build	Fixed	Fetch Actual Distribution from Build Logs
RI5	Modified VCS State in Metadata	Fixed	Untrack Archive Tarballs

In the following sections, we describe each case. We present the binary differences manifesting the issue as displayed by Diffoscope [14], where the left side of each figure shows the official Geth reference artifact and the right side shows the rebuilt artifact. The accompanying tables summarize the identified *potential fixes* for each case, along with their implementation status: *Adopted*, *Proposed*, or *Excluded*.

RI1. Unreproducible Artifact Build IDs

The Go and GCC compilers used for building Geth each embed *build identifiers* into the final binary artifact. The GCC compiler passes a GNU build ID to the linker, which stores it in the Executable and Linkable Format (ELF) note section `.note.gnu.build-id`. This is used to aid debugging of crashes [38]. Similarly, a unique Go build ID is stored in the `.note.go.buildid` section, which is in part used to speed up the build time through caching [63].

Issue. Both the Go and GNU build IDs claim to be reproducible [38, 63]. Whereas we found this to be true for the Go build ID alone, when we enabled C code compilation (`CGO_ENABLED=1`), it resulted in unreproducible build IDs for both. Figure 5.2.1 illustrates one such difference between a Geth reference artifact and the rebuilt artifact. In this case, all other sources of non-determinism are eliminated. This suggests the GNU build ID embeds some system-specific parameters.

Fix. After unsuccessful attempts to reproduce the build IDs, we identified two possible fixes, as outlined in Table 5.2.2. The first option, Omit Embedding GNU build ID (*F1.1*), can be done using specific compiler flags. However, this option was excluded as it risks losing debug information. Instead, we chose to Exclude Build IDs from Verification (*F1.2*) by removing their designated ELF sections before comparing their hashes.

This method effectively only removes metadata, while preserving the integrity of the executable portion of the binary artifact for verification. However, to ensure proper execution of the Geth binary, it may be necessary to retain a copy of the unstripped binary.

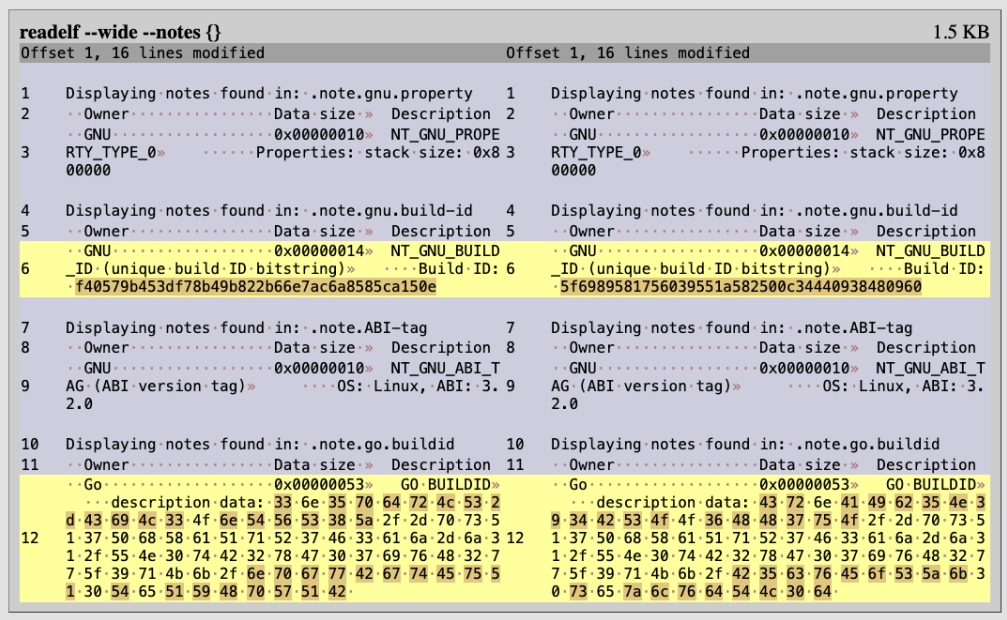

Figure 5.2.1: Binary Difference showing Inconsistent Build IDs

Table 5.2.2: Potential fixes for Unreproducible Artifact Build IDs and their Adoption Status

ID	Fix	Details	Status
F1.1	Omit Embedding GNU build ID	Unfavorable for debugging	Excluded
F1.2	Exclude Build IDs from Verification	Binary strip before comparison	Adopted

RI2. Conditional Date Embedding

Geth's `/build/ci.go` script sets build flags for the `go build` command. Any flags with the prefix `-ldflags` will be passed to the Go linker `cmd/link` [58], the last step of the build producing the final executable. The linker flags can be used to embed dynamic version information in the artifact. For example, the `-X` linker flag lets you assign a specified value to a string variable which is defined in the source code [62]. Geth uses this flag to embed the Git *commit hash* and *commit date* variables in each binary artifact, as seen in Listing 5.1.

Listing 5.1: Excerpt of the `buildFlags` Function passing Git Commit Hash (line 4) and Date (line 5) as Flags.

```
1 // buildFlags returns the go tool flags for building.
2 func buildFlags(env build.Environment, staticLinking bool, buildTags []string)(
      flags []string){
3 var ld []string
4 ld = append(ld, "-X","github.com/ethereum/go-Ethereum/internal/version.gitCommit
      ="+env.Commit)
5 ld = append(ld, "-X","github.com/ethereum/go-ethereum/internal/version.gitDate="
      +env.Date)
6
7 flags = append(flags, "-ldflags", strings.Join(ld, " "))
8 return flags
9 }
```

Issue. The two linker flags are automatically set by retrieving their values from the environment during the build process. The commit date is obtained using the `getDate(commit)` function, which executes `git show` to fetch the commit date. However, this function is not called when building Geth from a Git *detached HEAD* state, meaning it operates on an unnamed branch [26]. As a result, if an artifact is rebuilt without explicitly checking out the commit to a branch, the linker will not receive a commit date. Figure 5.2.2 illustrates this binary difference.

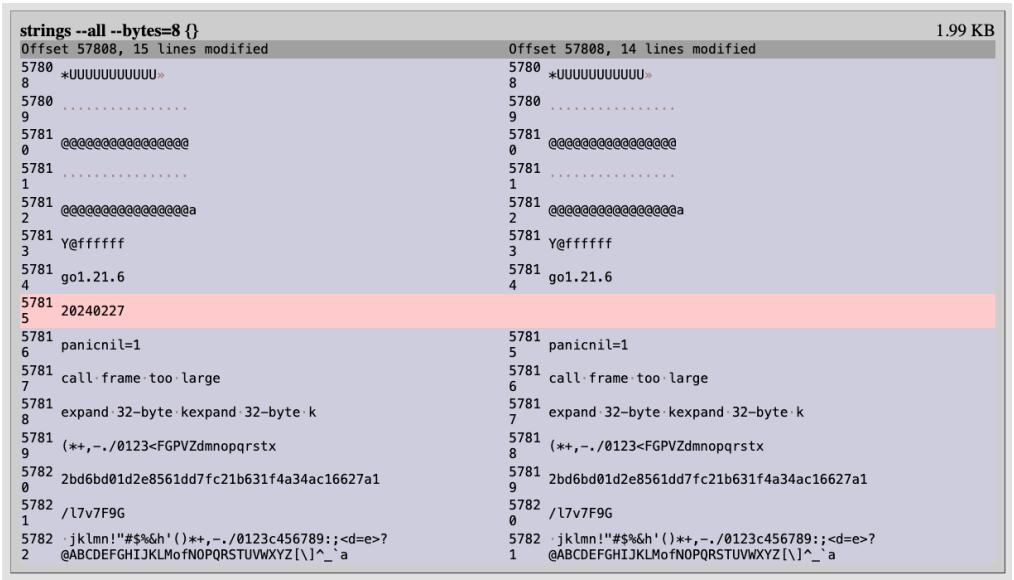

Figure 5.2.2: Binary Difference showing Release Date only in Reference Artifact (left)

Fix. Table 5.2.3 lists two options for fixing this case of unreproducibility. The apparent solution, Fix Conditional Statement (*F2.1*), was adopted by a small source

code change. Yet, the Go compiler already embeds equivalent data in the Go *build info* fields `vcs.revision` and `vcs.time` variables. These variables are set during package loading in a Go compilation, and they are retrieved from the VCS logs, analogous to the Geth approach. Consequently, the Geth custom linker flags are rendered obsolete. Therefore, we also propose to Avoid Double Embedding of Metadata (*F2.2*) by removing these linker flags.

Table 5.2.3: Potential fixes for Conditional Date Embedding and their Adoption Status

ID	Fix	Details	Status
F2.1	Fix Conditional Statement	Geth pull request (merged)	Adopted
F2.2	Avoid Double Embedding of Metadata	Communication with Maintainers	Proposed

RI3. System-Specific Paths

Absolute file paths differ across systems, and the embedding of these is one of the most common causes of unreproducibility in software [2, 34]. Go accounts for this by trimming system-specific paths from the resulting binary artifact when providing the `trimpath` build flag [59]. Instead, paths will start with the module name or import path of the file.

strings --all --bytes=8 {} 1.62 KB

Offset 58136, 18 lines modified		Offset 58136, 18 lines modified	
58136	Southern Sotho	58136	Southern Sotho
58137	Swedish - Sweden	58137	Swedish - Sweden
58138	Swedish - Finland	58138	Swedish - Finland
58139	Ukrainian	58139	Ukrainian
58140	Uzbek - Cyrillic	58140	Uzbek - Cyrillic
58141	Vietnamese	58141	Vietnamese
58142	Linux usbfs	58142	Linux usbfs
58143	/home/travis/gopath/pkg/mod/github.com/karalabe/usb@v0.0.2/libusb/libusb/os/linux_netlink.c	58143	/root/go/pkg/mod/github.com/karalabe/usb@v0.0.2/libusb/libusb/os/linux_netlink.c
58144	/home/travis/gopath/pkg/mod/github.com/karalabe/usb@v0.0.2/libusb/libusb/os/linux_usbfs.c	58144	/root/go/pkg/mod/github.com/karalabe/usb@v0.0.2/libusb/libusb/os/linux_usbfs.c
58145	LIBUSB_SUCCESS / LIBUSB_TRANSFER_COMPLETED	58145	LIBUSB_SUCCESS / LIBUSB_TRANSFER_COMPLETED
58146	/home/travis/gopath/pkg/mod/github.com/karalabe/usb@v0.0.2/libusb/libusb/io.c	58146	/root/go/pkg/mod/github.com/karalabe/usb@v0.0.2/libusb/libusb/io.c
58147	ctx->pollfds_cnt >= internal_nfds	58147	ctx->pollfds_cnt >= internal_nfds
58148	Access denied (insufficient permissions)	58148	Access denied (insufficient permissions)
58149	No such device (it may have been disconnected)	58149	No such device (it may have been disconnected)
58150	System call interrupted (perhaps due to signal)	58150	System call interrupted (perhaps due to signal)
58151	Operation not supported or unimplemented on this platform	58151	Operation not supported or unimplemented on this platform
58152	Toegang geweigerd (onvoldoende toegangsrechten)	58152	Toegang geweigerd (onvoldoende toegangsrechten)
58153	Apparaat bestaat niet (verbinding met apparaat verbroken?)	58153	Apparaat bestaat niet (verbinding met apparaat verbroken?)

Figure 5.2.3: Binary Difference showing System-Specific Paths

Issue. Comparing a Geth reference artifact with its rebuilt version revealed a discrepancy in system-specific file paths. In Figure 5.2.3, we see full path embedding for three C libraries that are dependencies of Geth. The left side displays paths specific to Travis, while the rebuilt artifact on the right shows system paths typical of a Linux Ubuntu environment. Further investigation uncovered a `trimpath` bug specific to Ubuntu Bionic, which was the distribution used for building the Geth artifacts, including the reference artifact.

Fix. Table 5.2.4 presents two potential fixes for the issue. One instinctive solution is to implement Patch Go Bug in Ubuntu Bionic (*F3.1*), for which we have submitted a bug report to Go[1]. However, given the approaching end of life for Bionic, prioritizing a patch for this specific bug is not prioritized. Instead, we recommend adopting Use another Ubuntu Distribution for Geth builds. Starting with version 1.14.2, Geth has transitioned to using Ubuntu Noble for their builds to address this reproducibility issue.

Table 5.2.4: Potential fixes for System-Specific Paths and their Adoption Status

ID	Fix	Details	Status
F3.1	Patch Go Bug in Ubuntu Bionic	Go bug report (unresolved)	Proposed
F3.2	Use another Ubuntu Distribution	Upgrade to Ubuntu Noble in v.1.14.2	Adopted

RI4. Random Ubuntu Distribution in Build

Geth utilizes GCC as the C compiler for its binary releases, as detailed in Section 3.2.2. The specific version of the GCC compiler used is determined by the default version in the underlying Ubuntu distribution. Since compiler toolchains directly influence both the compilation process and the resulting binary, and are subject to version-specific changes, achieving reproducible builds requires an exact match of the compiler version.

Issue. Travis CI provides Ubuntu distributions to use as the base environment during builds [65]. However, a bug in Travis CI lets developers specify a certain Ubuntu distribution in the `.travis.yml` configuration, while another random distribution may be used during the build process. For example, a Geth build specifying `dist:noble`

[1] `https://github.com/golang/go/issues/67011`

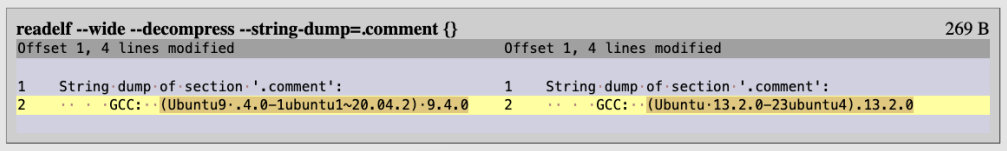

Figure 5.2.4: Binary Difference showing Mismatching GCC Versions

resulted in building with `dist:focal` [2]. We identified this issue while attempting to reproduce Geth builds configured for Ubuntu Noble, which resulted in large arbitrary binary differences. Additionally, we discovered inconsistencies in the `.comment` ELF section of the binary, indicating mismatched GCC versions, as illustrated in Figure 5.2.4. The varying availability of GCC versions across different Ubuntu distributions practically requires using the same distribution used for the build of the reference artifact.

Fix. Table 5.2.5 presents three potential solutions to address this issue. Given its trivial nature and relevance, we recommend that the team behind the closed-source software implement to Patch Travis CI Bug (*F4.1*). We detail the problem and propose two viable fixes[3]: aborting the build or issuing clear warnings when an undefined Ubuntu distribution is specified. However, in the absence of a fix, we identified a workaround and we Fetch Actual Distribution from Build Logs (*F4.2*). While this does not fully resolve the issue, it allows for rebuilding without the consistently inconsistent GCC compiler versions. Finally, due to the lack of response and support from Travis CI, we concur with the Geth team to Migrate to another Build Service (*F4.3*) in the long term.

Table 5.2.5: Potential fixes for Random Ubuntu Distribution in Build and their Adoption Status

ID	Fix	Details	Status
F4.1	Patch Travis CI Bug	Travis bug report (unresolved)	Proposed
F4.2	Fetch Actual Distribution from Build Logs	Access Travis API during rebuild	Adopted
F4.3	Migrate to another Build Service	Communication with Maintainers	Proposed

[2] `https://app.travis-ci.com/github/ethereum/go-ethereum/jobs/621721567#L7-L20`
[3] `https://travis-ci.community/t/travis-uses-unexpected-ubuntu-distribution`

RI5. Modified VCS State in Metadata

In addition to the `vcs.revision` and `vcs.date` fields discussed in *RI2*, the Go build information includes the `vcs.modified` field, which indicates whether any changes have been made to the version-controlled repository since the last commit. This is determined by running the `git log` command [61].

Issue. During our experiments, we discovered that all Geth Linux binary builds, *except* those targeting the AMD64 architecture, had the `vcs.modified` field set to *true*. This discrepancy complicates the rebuilding process, requiring the verifier to also modify the repository worktree for certain rebuilds to achieve a reproducible artifact. Figure 5.2.5 illustrates an example of this discrepancy.

strings --all --bytes=8 {} 949 B

Offset 53010, 15 lines modified		Offset 53010, 15 lines modified	
53010	build» DefaultGODEBUG=httplaxcontentlength=1,httpmuxgo121=1,tls10server=1,tlsrsakex=1,tlsunsafeekm=1	53010	build» DefaultGODEBUG=httplaxcontentlength=1,httpmuxgo121=1,tls10server=1,tlsrsakex=1,tlsunsafeekm=1
53011	build» CGO_ENABLED=1	53011	build» CGO_ENABLED=1
53012	build» GOARCH=arm64	53012	build» GOARCH=arm64
53013	build» GOOS=linux	53013	build» GOOS=linux
53014	build» vcs=git	53014	build» vcs=git
53015	build» vcs.revision=aadddf3a6e5e889e96859132f115cc44dd2ecfbd	53015	build» vcs.revision=aadddf3a6e5e889e96859132f115cc44dd2ecfbd
53016	build» vcs.time=2024-07-02T13:19:57Z	53016	build» vcs.time=2024-07-02T13:19:57Z
53017	build» vcs.modified=true	53017	build» vcs.modified=false
53018	!*.47>DJN	53018	!*.47>DJN
53019	''(()***	53019	''(()***
53020	!#%(+/1269,	53020	!#%(+/1269,
53021	#(+189;<==>»	53021	#(+189;<==>»
53022	#(,.16I1	53022	#(,.16I1
53023	%)+-1"2	53023	%)+-1"2
53024	!$%),---3	53024	!$%),---3

Figure 5.2.5: Binary Difference showing Mismatching `vcs.modified` Fields

All Linux binaries are built in the same job within the CI/CD process, meaning they share the system state. If a build adds files to the repository, it modifies the status of the worktree, which is then reflected in the Go build info field `vcs.modified`. In this particular case, the issue was caused by the archiving procedure of writing tarballs to the repository root, as discovered by the Geth maintainers.

Fix. Table 5.2.6 lists two viable fixes for rectifying this reproducibility issue. The less sophisticated approach is to Replicate Modified VCS State (*F5.1*) by adding a placeholder file to the Geth repository before rebuilding, as to also alter the repository.

Table 5.2.6: Potential fixes for Modified VCS State in Metadata and their Adoption Status

ID	Fix	Details	Status
F5.1	Replicate Modified VCS State	Provisional fix	Excluded
F5.2	Untrack Archive Tarballs	Patched in v.1.14.9 unstable	Adopted

Ultimately, the Geth developers adopted a patch to Untrack Archive Tarballs (*F5.2*), effectively correcting this issue.

5.3 Summary

We identified and analyzed 5 distinct reproducibility issues in Geth Linux binary artifacts and proposed solutions to each. By addressing these 5 cases, we confirmed that all Linux binary artifacts can be made reproducible. This work supports more rigorous verification of source-to-binary correctness for Geth artifacts. In the next chapter, we implement one such approach.

Chapter 6

Geth Rebuild

In Chapter 5, we established the steps required for achieving reproducible Geth binary artifacts. In this chapter, we present `Geth Rebuild`[1], a tool that automates the process of rebuilding and verifying Geth binaries. This tool serves as an independent, external verifier for Geth binary artifacts, and lays the groundwork for the reproducibility evaluation in Chapter 7. In the following sections, we outline the threat and mitigation model, followed by the tool's requirements and design.

6.1 Threat Model and Mitigation

The threat model addressed by `Geth Rebuild` revolves around adversaries seeking to compromise the integrity of binary artifacts. These adversaries may either be authorized individuals or external attackers. They all aim to introduce malicious changes during the build process or the distribution of Geth binaries.

There are two threat vectors:

1. **Build Link Tampering**: An adversary could manipulate the build process to introduce malicious code or alter the built binaries.

2. **Package Link Tampering**: After the binaries are built, an adversary could modify them by inserting malicious alterations in the distribution step.

[1] `https://github.com/chains-project/geth-rebuild/`

These threat vectors are consistent with the build- and package supply chain attack vectors detailed in Chapter 4.

The mitigation of the threat vectors works as follows:

1. **Artifact Rebuild**: An independent verifier rebuilds a reference binary artifact published by Geth.
2. **Discrepancy Detection**: During comparison, the verifier detects a discrepancy between the reference and the rebuilt artifact.
3. **Attack Detection**: The verifier reports or analyzes it. The analysis uncovers malicious code present in the reference binary artifact.

This procedure assumes that both the reference artifact's and the rebuilt artifact's build environment cannot be subverted simultaneously. Therefore, employing more independent verifiers is beneficial for strengthening the claim of a correct source-to-binary semantic for a Geth artifact.

6.2 Requirements

To meet the tool's intended purpose, we identify three main requirements:

Build Configuration Retrieval. The first step to reproducing a binary artifact is to retrieve the build configurations used in the original build of the reference artifact. These parameters include aspects such as compiler versions, environment flags, and dependencies. `Geth Rebuild` should automate the process of inferring these parameters.

Artifact Rebuild. After retrieving the build configurations, the tool should rebuild the reference artifact. The rebuild should mirror the original build process by using the retrieved build configurations. This step produces a new binary, the rebuilt artifact.

Rebuild Analysis. Lastly, the tool should analyze the result of the rebuild by comparing the reference artifact and the rebuilt artifact for bit-by-bit equality. This step should effectively report the reproducibility of the Geth reference artifact.

6.3 System Design

6.3.1 Overview

Figure 6.3.1 provides an overview of the architecture of `Geth Rebuild`. Three main components are shown in blue, each addressing one of the requirements described in Section 6.2. In the first step, the *build configuration retrieval* component takes a description of the target reference artifact to be rebuilt. Using source code- and build log analysis, all parameters needed for a rebuild are inferred. The second component, the *artifact rebuild*, takes the inferred build configurations and the Geth source code and rebuilds the reference artifact in a containerized environment. In this step, we also retrieve the reference artifact, as published by Geth. In the third step, the *rebuild analysis* component compares the two binary artifacts for bit-by-bit equality. The results are recorded in a results report. A difference report is produced in case of a comparison mismatch, indicating unreproducibility.

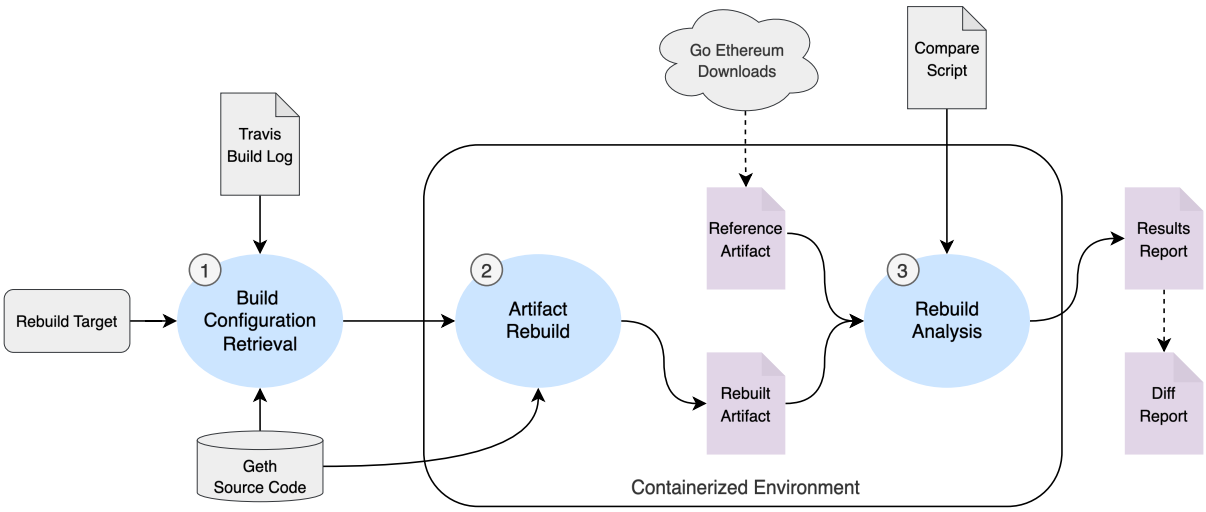

Figure 6.3.1: Overview of `Geth Rebuild` Architecture

While rebuilding a Geth binary artifact is trivial, achieving reproducibility in the build process is not. As shown in Chapter 5, unexpected variables can affect the build reproducibility. `Geth Rebuild` simplifies the task of rebuilding by automating the retrieval of appropriate build configurations. Additionally, `Geth Rebuild` allows rebuilding for all Geth-supported Linux architectures, each requiring distinct build configurations.

6.3.2 Components

Build Configuration Retrieval

The build configurations are categorized into three specifications. The *artifact specification* records the unique attributes of the reference artifact that will be rebuilt. The *toolchain specification* configures the compilation tools used for rebuilding the artifact. Lastly, the *environment specification* includes all general settings necessary for configuring the containerized environment for a rebuild. Table 6.3.1 provides an overview of the build configuration parameters and their retrieval method for each specification.

Table 6.3.1: Build Configurations used in `Geth Rebuild`

Artifact Specification		Toolchain Specification		Environment Specification	
Param	**Retrieval**	**Param**	**Retrieval**	**Param**	**Retrieval**
OS	PI	Go Version	SCA	Ubuntu Dist	BL
Architecture	PI	Build Cmd	SCA	Env. Flags	DF
Version	PI	C Deps	DF	Utility Deps	DF
Commit	PI/GL				
PI Program Input		**GL** Git Logs		**SCA** Source Code Analysis	
DF Defaults		**BL** Travis Build Logs			

The artifact specification defines the target artifact for the rebuild. Therefore, these parameters are program inputs (PI) to `Geth Rebuild`. The parameters *OS*, *Architecture*, *Version* are mandatory, whereas the *Commit* parameter is only used for rebuilds of unstable versions. For stable builds, the Commit parameter is inferred through Git logs (GL).

For the toolchain specification, source code analysis (SCA) is used to retrieve the *Go Version* and *Build Command* parameters. This is done through regular expression matching of configuration files. The *C Dependencies* parameter is configured as a default value (DF) for the specific target OS and Architecture.

In the environment specification, the *Environment Flags* and the *Utility Dependencies* are configured similarly, through default values. Environment flags include, for example, ARM optimizations (`GOARM`), and the enabling or disabling of CGO (`CGO_ENABLED`). Lastly, the *Ubuntu Distribution* configures the environment for the

rebuild and is replicated from the reference build. To avoid the Travis configuration bug (*RI4*) we retrieve the distribution from the Travis build log (BL) rather than using source code analysis on its configuration file.

Artifact Rebuild

The artifact rebuild component leverages containerization to rebuild the reference artifact in the configured Ubuntu environment. By doing this, we ensure that the rebuild process itself is reproducible, that is, verifiable. First, the specified dependencies and Go toolchain are downloaded. Next, the reference artifact is retrieved from the Geth downloads website[2], and the symbol table is stripped. Then, the Geth repository is copied from the local machine, the reference binary is re-compiled, and finally, also stripped of its symbol table. This procedure fetches the reference artifact and produces its rebuilt version.

Rebuild Analysis

The artifact rebuild component generates a container image that includes the reference artifact, the rebuilt artifact, and a binary comparison script. Executing the comparison script within the container determines the result of the rebuild. The comparison is performed by calculating the artifact's SHA-256 hash digests and checking for equality.

The results are recorded in a log on the local machine. The log indicates the status of the rebuild: a comparison *match* (successful rebuild), *mismatch* (unsuccessful rebuild), or *error* (erroneous rebuild process). In the case of an unsuccessful rebuild, we also output an HTML file displaying the binary differences.

6.4 Implementation Details

`Geth Rebuild` is written in Go, leveraging shell scripting for OS-command-heavy functions. Docker [39] facilitates containerization, and Diffoscope [14] generates the difference reports.

[2] `https://geth.ethereum.org/downloads`

Chapter 7

Evaluation

Building on the design and implementation of `Geth Rebuild` presented in Chapter 6, this chapter focuses on evaluating the reproducibility of Geth binaries across a larger scope of versions and architectures when evaluated through `Geth Rebuild`. By doing this, we aim to answer RQ3, assessing the extent to which reproducibility of Geth binary artifacts can be independently verified, and identifying any remaining challenges.

7.1 Methodology

To analyze the reproducibility of Geth binary artifacts, we run `Geth Rebuild` across 7 stable and 5 unstable releases of Geth. Table 7.1.1 lists the versions evaluated. All stable versions from the present (1.14.8) back until 1.14.2 are analyzed, as this version apparently fixes the System-Specific Paths issue (*RI3*) by bumping the Ubuntu version. Furthermore, we run the tool on unstable builds to distinguish any differences in behavior compared to stable releases. We randomly select a sample of 5 commits from a pool of 33 available unstable artifacts, which all belong to the latest unstable version (1.14.9).

We run the experiments for all Linux architectures supported by Geth: *AMD64*, *386*, *ARM5-7*, and *ARM64*. For unreproducible reference artifacts, we manually analyze the produced difference report to determine the potential root causes of the unreproducibility.

Table 7.1.1: Evaluated Stable and Unstable Geth Versions

Stable Versions		Unstable Versions	
Version	**Commit**	**Version**	**Commit**
1.14.8	a9523b64	1.14.9	c4b01d80
1.14.7	aa55f5ea	1.14.9	2f2e5b08
1.14.6	aadddf3a	1.14.8	83e70aa3
1.14.5	0dd173a7	1.14.8	dbc1d04f
1.14.4	5550d839	1.14.8	cf8aa31e
1.14.3	ab48ba42		
1.14.2	35b2d07f		

7.2 Experimental Results

Our experiments identified both unreproducible and reproducible Geth binary artifacts. We show the results from our analysis for stable and unstable versions in Tables 7.2.1 and 7.2.2, respectively[1]. In the tables, red cells represent unreproducible reference artifacts, while green ones represent reproducible ones. We also include abbreviated letters for unsuccessful rebuilds to indicate the identified root cause of the discrepancies.

Table 7.2.1: Results from Rebuilding Stable Geth Artifacts

		Stable Version						
		1.14.2	1.14.3	1.14.4	1.14.5	1.14.6	1.14.7	1.14.8
Architecture	AMD64	✓	✓	✓	✓	✓	✓	B
	386	✓	✓	✓	✓	✓	✓	B
	ARM5	✓	✓	✓	✓	✓	✓	B
	ARM6	✓	✓	✓	✓	✓	✓	B
	ARM7	✓	✓	✓	✓	✓	✓	B
	ARM64	✓	✓	✓	✓	✓	✓	B

B Ubuntu Bionic Bug

[1]All results are available at: `https://github.com/chains-project/geth-rebuild/tree/results/internal/experiments/res/`

The findings reveal that 6 out of 7 stable Geth versions were reproducible, while none of the evaluated unstable builds were successfully rebuilt. Notably, all unsuccessful rebuilds exhibit the same root cause: the previously identified reproducibility issue of System-Specific Paths (*RI3*) in binary artifacts.

Recall that in *RI3*, we attributed this issue to a Go bug unique to Ubuntu Bionic, and that this was fixed by upgrading the Ubuntu version (Use another Ubuntu Distribution). Curiously, the second reproducibility issue connected to a bug, Random Ubuntu Distribution in Build (*RI4*), reintroduced this issue.

Table 7.2.2: Results from Rebuilding Unstable Geth Artifacts

		Unstable Version				
		cf8aa3	dbc1d0	83e70a	2f2e5b	c4b01d
Architecture	AMD64	B	B	B	B	B
	386	B	B	B	B	B
	ARM5	B	B	B	B	B
	ARM6	B	B	B	B	B
	ARM7	B	B	B	B	B
	ARM64	B	B	B	B	B

B Ubuntu Bionic Bug

For all evaluated unstable versions, as well as one stable version (1.14.8), we found that the Travis CI bug resulted in the use of the Ubuntu *Bionic* distribution, instead of the intended, *Noble*. Consequently, this rendered most evaluated Geth artifacts unreproducible. Following discussions with core Geth developers, a new patch[2] was developed to downgrade the Ubuntu distribution specified in the Travis configuration.

7.3 Summary

This chapter assessed the reproducibility of Geth binary artifacts across various versions and architectures, focusing on the feasibility of independent verification

[2] https://github.com/ethereum/go-ethereum/pull/30319

as outlined in RQ3. The findings indicate that while the `Geth Rebuild` verifier itself demonstrates the feasibility of independent verifiers for Geth, reproducibility is currently partly hindered by the reintroduction of a Go bug via Travis CI, causing inconsistent build outputs. Consequently, full reproducibility, and by extension, reliable third-party verification, could not be consistently achieved across all evaluated artifacts. However, for future versions, we anticipated the patching to fix the final discrepancies.

Chapter 8

Discussion

In this chapter, we consolidate the findings of the thesis, reflecting on the study's implications, limitations, and potential future directions.

8.1 Research Questions

In this thesis, we address the understudied area of supply chain security within blockchain applications, with a particular focus on the supply chain of Geth. We aimed to identify and alleviate potential supply chain risks for Geth. To achieve this, we explored three research questions, we discuss these in the following.

8.1.1 Geth Supply Chain Analysis (RQ1)

In RQ1, we evaluated the impact of known supply chain attacks on Geth by modeling its supply chain (Ch. 3) and conducting a security analysis (Ch. 4). Given Geth's critical role and the rise in supply chain attacks [23, 54], proactively addressing these threats is crucial. A single vulnerability in the supply chain can allow an attacker to compromise the system.

Our findings indicate that Geth has several safeguards in place, including rigorous pull request reviews, containerized ephemeral build environments, and software package signing. The Go dependency management system also mitigates several dependency-related attack vectors (Sec. 4.2.4).

Ideally, safeguards should include both pre- and post-exploit measures. Preventive

safeguards avert attacks before they occur (to various extents), detective measures ensure timely detection, and corrective measures effective mitigation.

In our analysis, we identified the safeguard *reproducible builds* (*SF7*) as having the greatest potential to strengthen the Geth supply chain, particularly for reliably detecting build-based threats, with the additional potential of correcting attacks, e.g. by aborting a software build. This finding shaped the direction of the rest of the study, focusing on its implementation for Geth.

8.1.2 Reproducibility of Geth (RQ2)

In RQ2 (Ch. 5), we examined the challenges of achieving reproducible builds for Geth and the steps to overcome them. Through systematic rebuilding, analysis, and fixing of reproducibility issues, we were able to achieve reproducible builds.

Challenges

While Go builds are designed to be reproducible (Sec. 2.4), and in fact, should have addressed the most common reproducibility issues reported in prior work [2, 17, 34], we encountered specific instances of non-determinism. Achieving reproducibility in complex, real-world projects like Geth is compounded by additional challenges, despite the reproducibility support of the Go toolchain. We attribute these issues to three main causes: *C Interoperability*, *Project-induced Non-determinism*, and *Software Bugs*.

C Interoperability. C's integration in Go, originally as its bootstrap compiler [58], provides performance advantages [55] but introduces increased build complexity, which resulted in reproducibility challenges. For Geth, CGO issues raised in the form of unreproducible build IDs (*RI1*) and an intricate `trimpath` bug (*RI3*) that occurred selectively with certain C dependencies in specific environments. While these issues were ultimately surmountable, they highlight a broader trade-off in polyglot environments like CGO.

Project-induced Non-determinism. Non-determinism also arose from project-specific factors, including build configurations and source code logic. For instance, the conditional embedding of the commit date via linker flags (*RI2*) and the storage of tarballs in the repository root (*RI5*) led to discrepancies between reference and

rebuilt artifacts. These subtle differences highlight how seemingly minor project configuration decisions can impact reproducibility, which may not necessarily be easily discernible.

Software bugs. Finally, software bugs were a notable obstacle. In our work, we discovered two bugs interfering with reproducibility. A particularly concerning issue was a Travis CI bug (*RI4*) affecting the reproducibility throughout a wide range of Geth versions (Sec. 7.2). The lack of support from Travis CI further exacerbated the problem, highlighting the need to consider alternative build services to ensure a robust supply chain.

Implications

In practice, achieving reproducibility involved in-depth analysis of binary artifacts, often requiring examination of hundreds of lines of disassembled code to identify discrepancies. While analyzing lower-level code further adds complexity to the reproducibility effort, it also offers insights into rarely scrutinized code parts, revealing bugs and unintentional behavior that might otherwise go unnoticed. In fact, our work reinforced the notion that reproducible builds are not only a tool for verifying binary integrity but also an effective means of improving overall software quality, a finding supported by prior research [25].

As a result of our investigation, we discussed with Geth core developers to address these issues, leading to the adoption of several patches, summarized in Table 8.1.1.

Table 8.1.1: Geth's Adopted Patches for Our Discovered Reproducibility Issues

Issue	Patch
Unreproducible Artifact Build IDs (*RI1*)	`github.com/ethereum/go-ethereum/pull/30321`
Conditional Date Embedding (*RI2*)	`github.com/ethereum/go-ethereum/pull/30320`
System-Specific Paths (*RI3*)	`github.com/ethereum/go-ethereum/pull/29723`
Random Ubuntu Distribution in Build (*RI4*)	`github.com/ethereum/go-ethereum/pull/30319`
Modified VCS State in Metadata (*RI5*)	`github.com/ethereum/go-ethereum/issues/30324`

8.1.3 Independent Reproducibility Verification (RQ3)

In RQ3, we evaluated the practical feasibility of verifying the reproducibility of Geth binary artifacts as a means to detect build-based attacks. To this end, we developed a tool that automates the rebuilding and analysis of Geth artifacts, applying it across multiple versions to assess reproducibility. The evaluation revealed that some artifacts were only partially reproducible, primarily due to the reintroduction of a software bug within the CI system. We anticipate these issues to be resolved in short, resulting in full reproducibility for Geth Linux artifacts.

Independent Verification

Whereas reproducibility is beneficial for the Geth build process itself, allowing the developers to control the integrity of builds, we want to emphasize the advantages of *independent* verification in our work. This approach allows users to validate the integrity of the binaries they are using, without relying solely on the developers' assurances. Furthermore, an increasing number of verifiers agreeing on a certain version of a binary, independently, strengthens the claims of its correctness.

We identify two additional measures to further enable the independent verification of their binary artifacts:

Reproducibility Claim. A formal claim from the Geth team asserting that their builds are fully reproducible and verifiable by third parties is crucial. Such a claim would incentivize the broader adoption of independent verifiers, as seen in other open-source projects like Linux [17] and Bitcoin [5]. Continuous monitoring and patching of reproducibility issues will be required and we note that the Geth developers have shown a drive to address such tasks. In turn, such endeavors could eventually be handled by independent verifiers.

Verification instructions. Our analysis revealed that reproducing builds often required meticulous investigation into both source code and binaries to identify the correct build parameters. For example, a non-obvious issue emerged where disabling CGO during cross-compilation was necessary, a detail uncovered through careful review of build scripts and prior GitHub discussions. Additionally, inconsistencies across Ubuntu distributions demanded static analysis of binaries and build logs to determine the appropriate GCC version. To simplify future verification efforts, such

configurations should be documented, for example automatically through a tailored SBOM for each build.

8.2 Ethics and Sustainability

The ethical considerations of this study primarily revolve around the responsible disclosure of potential vulnerabilities within the Geth client. Although no concrete vulnerabilities were identified during the research, given the critical role of Geth in the Ethereum ecosystem, we have ensured that Geth maintainers have the opportunity to review this report prior to publication.

This study also contributes to the sustainability of the software ecosystem in two main ways. First, it enhances the security of Geth by providing both the prerequisites (Ch. 5) and a method (Ch. 6) for detecting build-based attacks on its binary artifacts. This not only strengthens Geth itself but also helps secure the broader Ethereum network.

Second, our research supports the growing movement toward adopting reproducible builds across the software industry. By offering practical insights into achieving reproducibility, we contribute to facilitating future efforts in this area. While much of the existing research [2, 17, 25, 34] has focused on Linux, our work adds momentum to exploring reproducibility in other software ecosystems.

8.3 Limitations

Study scope. In this study, we focused on the reproducibility of Geth binary artifacts targeting Linux operating systems. However, as noted in Section 3.2.3, this is only a part of their software distribution. We do not investigate the reproducibility of other packaging forms, which may face new challenges to reproducibility.

Threat Mitigations. We acknowledge that reproducibility verification as done by *Geth Rebuild* does not prevent a malicious change from *altering* the supply chain. However, it can stop its *propagation* downstream, by detecting inconsistencies between the source code and the resulting binary artifacts. While *Geth Rebuild* does not address attacks that directly target the source code or dependencies, it effectively identifies discrepancies introduced in the build and distribution. More specifically, *Geth Rebuild* mitigates *build-based* attacks.

As an example, for the XZ Utils attacks described in Section 2.2, the rebuilder would not detect the obfuscated malicious files already present in the source code. The malicious source code would consistently compile to the same binary. However, the addition of malicious files that produce divergence between local and distributed builds is an attack mitigated by this approach.

8.4 Future Work

We developed a supply chain model of Geth and analyzed potential threats to it based on a 2023 taxonomy [32]. This approach was experimental, and future work to validate these findings is important. In particular, engaging with Geth developers to assess the utility of such a supply chain security assessment would provide valuable insights to refine the methodology for comprehensively evaluating supply chain risks.

In this study, we analyzed the entire supply chain of Geth but focused primarily on build-based threats. Future work should aim to strengthen the remaining links in the Geth supply chain: the source, package, and dependencies. For example, improving transparency in the dependency tree and its evolution may prove useful for preventing emerging supply chain risks.

Continuing our work on reproducibility, establishing an infrastructure for independent verifiers to compare Geth artifacts and binaries is an important next step to enhance confidence in the build process. This should also ensure the integrity and correctness of such reproducibility claims.

Lastly, from the perspective of the broader Ethereum ecosystem, strengthening the supply chains of other Ethereum clients is valuable future work. Specifically, hardening minority clients can enhance their trustworthiness and encourage wider adoption, promoting client diversity and resilience in the Ethereum network.

Chapter 9

Conclusions

The increasing prevalence of supply chain attacks emphasizes the need to secure all components of modern software systems. For Ethereum, operating in a highly security-critical domain, ensuring the integrity of its clients, such as Geth, is crucial. This thesis has focused on the relatively underexplored area of supply chain security in blockchain software, specifically targeting the Geth client.

Through a comprehensive analysis of Geth's supply chain, we identified 13 relevant attack vectors and examined their potential impact on Geth. Our work further focused on build-based threats to strengthen the Geth supply chain. We identified and addressed 5 distinct challenges to reproducibility for Geth, finally reaching reproducible binary builds. Furthermore, by the development of `Geth Rebuild`, we demonstrated the practical feasibility of independent verification of reproducibility with the concrete potential to detect and mitigate attacks introduced in the build and distribution of Geth Linux binary artifacts.

This work not only strengthens the security posture of Geth and Ethereum but also contributes to the broader discourse of adopting reproducible builds across the software industry at large.

Bibliography

[1] Aumasson, Jean-Philippe, Kolegov, Denis, and Stathopoulou, Evangelia. *Security Review of Ethereum Beacon Clients*. Tech. rep. 2021.

[2] Bajaj, Rahul, Fernandes, Eduardo, Adams, Bram, and Hassan, Ahmed E. *Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors*. *Empirical Softw. Engg*. 29.1 (2023). DOI: 10.1007/s10664-023-10399-4. https://doi.org/10.1007/s10664-023-10399-4.

[3] Balliu, Musard, Baudry, Benoit, Bobadilla, Sofia, Ekstedt, Mathias, Monperrus, Martin, Ron, Javier, Sharma, Aman, Skoglund, Gabriel, Soto-Valero, César, and Wittlinger, Martin. *Challenges of Producing Software Bill of Materials for Java*. *IEEE Security Privacy* 21.6 (2023), 12–23. DOI: 10.1109/MSEC.2023.3302956.

[4] Birsan, Alex. *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies*. Medium, 2021. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610 Online; accessed 09/27/2024.

[5] Bitcoin Core. *Download - Additional verification with reproducible builds*. https://bitcoincore.org/en/download/ Online; accessed 05/17/2024.

[6] Cameron, Jamie. *Webmin Security – Remote Command Execution [CVE-2019-15231]*. 2023. https://webmin.com/security/#remote-command-execution-cve-2019-15231 Online; accessed 04/11/2024.

[7] Clancy, Charles, Ferraro, Joseph, Martin, Robert, Pennington, Adam, Sledjeski, Christopher, and Wiener, Craig. *Deliver uncompromised: Securing critical software supply chains*. *MITRE Technical Papers* 24 (2021), 01.

[8] Courant, Nathanaëlle, Lepiller, Julien, and Scherer, Gabriel. *Debootstrapping without Archeology: Stacked Implementations in Camlboot*. *The Art, Science, and Engineering of Programming* 6.3 (2022). DOI: 10.22152/programming-journal.org/2022/6/13.

[9] Cox, Russ. *Our Software Dependency Problem*. 2019. https://research.swtch.com/deps Online; accessed 04/17/2024.

[10] Cox, Russ. *Perfectly Reproducible, Verified Go Toolchains*. Aug. 28, 2023. https://go.dev/blog/rebuild Online; accessed 05/17/2024.

[11] Cox, Russ. *Timeline of the xz open source attack*. 2024. https://research.swtch.com/xz-timeline Online; accessed 04/18/2024.

[12] *Decentralized Finance (DeFi) market set to soar to $48.02bn by 2031, study shows*. https://ibsintelligence.com/ibsi-news/decentralized-finance-defi-market-set-to-soar-to-48-02bn-by-2031-study-shows/ Online; accessed 08/22/2024.

[13] *Dependabot*. GitHub. https://github.com/dependabot.

[14] Diffoscope. *diffoscope: in-depth comparison of files, archives, and directories*. Reproducible Builds Project. https://diffoscope.org/ Online; accessed 05/20/2024.

[15] Docker. *Docker Engine security*. https://docs.docker.com/engine/security/ Online; accessed 09/27/2024.

[16] Drake, Victoria. *Threat Modeling*. 2021. https://owasp.org/www-community/Threat_Modeling Online; accessed 02/29/2024.

[17] Drexel, Joshua, Hänggi, Esther, and Veiga, Iyán Méndez. *Reproducible Builds and Insights from an Independent Verifier for Arch Linux* (2024). DOI: 10.18420/SICHERHEIT2024_016. https://dl.gi.de/handle/20.500.12116/43956 Online; accessed 04/29/2024.

[18] Ethereum. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Tech. rep. 2014. https://ethereum.org/en/whitepaper/ Online; accessed 08/19/2024.

[19] Ethereum. *Node Architecture*. 2023. https://ethereum.org/en/developers/docs/nodes-and-clients/node-architecture/#execution-client Online; accessed 03/10/2024.

[20] Ethereum, Go. *Contribution*. 2024. https://github.com/ethereum/go-ethereum/blob/master/README.md#contribution Online; accessed 03/04/2024.

[21] Ethereum, Go. *Installing Geth*. 2023. https://geth.ethereum.org/docs/getting-started/installing-geth Online; accessed 03/01/2024.

[22] etherscan.io. *Ethereum Daily Transactions Chart*. https://etherscan.io/chart/tx Online; accessed 01/30/2024.

[23] European Union Agency for Cybersecurity. *ENISA Threat Landscape for Supply Chain Attacks*. Technical Report. LU: Publications Office, 2021.

[24] FireEye. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor*. 2020. https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor Online; accessed 01/25/2024.

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

[26] Git. *git-checkout Documentation*. https://git-scm.com/docs/git-checkout#_detached_head Online; accessed 06/05/2024.

[27] *Go Ethereum*. 2024. https://geth.ethereum.org/.

[28] Go Ethereum. *Release Frontier (1.0.0) ethereum/go-ethereum*. 2015. https://github.com/ethereum/go-ethereum/releases/tag/v1.0.0 Online; accessed 03/11/2024.

[29] *Home – Travis-CI*. https://www.travis-ci.com/.

[30] Kaaru, Steve. *The Philippines is moving toward blockchain-based governance: DICT's David Almirol*. 2024. Online; accessed 08/22/2024.

[31] Karunaratne, Ayesh. *'git.php.net' server compromised*. *PHP.Watch* (2021). https://php.watch/news/2021/03/git-php-net-hack Online; accessed 04/10/2024.

[32] Ladisa, Piergiorgio, Plate, Henrik, Martinez, Matias, and Barais, Olivier. *SoK: Taxonomy of Attacks on Open-Source Software Supply Chains*. *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, 1509–1526. DOI: 10.1109/SP46215.2023.10179304.

[33] Ladisa, Piergiorgio, Sahin, Merve, Ponta, Serena Elisa, Rosa, Marco, Martinez, Matias, and Barais, Olivier. *The Hitchhiker's Guide to Malicious Third-Party Dependencies*. *Proceedings of the 2023 SCORED*

Workshop. arXiv:2307.09087. 2023, 65–74. DOI: 10.1145/3605770.3625212. http://arxiv.org/abs/2307.09087 Online; accessed 04/22/2024.

[34] Lamb, Chris and Zacchiroli, Stefano. *Reproducible Builds: Increasing the Integrity of Software Supply Chains* (2021). DOI: 10.1109/MS.2021.3073045. http://arxiv.org/abs/2104.06020%20http://dx.doi.org/10.1109/MS.2021.3073045.

[35] Lanyado, Bar. *Can you trust ChatGPT's package recommendations?* Vulcan. https://vulcan.io/blog/ai-hallucinations-package-risk Online; accessed 09/27/2024.

[36] Ledger. *Security Incident Report*. Tech. rep. 2023. https://www.ledger.com/blog/security-incident-report Online; accessed 04/20/2024.

[37] Lorepunk. *Major Ledger Connect-Kit Exploit Compromises Multiple DApps*. 2023. https://nftnow.com/news/major-ledger-connect-kit-exploit-compromises-multiple-dapps/ Online; accessed 05/07/2024.

[38] McGrath, Roland. *RolandMcGrath/BuildID*. 2008. https://fedoraproject.org/wiki/RolandMcGrath/BuildID Online; accessed 05/20/2024.

[39] Merkel, Dirk. *Docker: lightweight Linux containers for consistent development and deployment*. *Linux J.* 2014.239 (2014).

[40] Microsoft. *STRIDE*. 2022. https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats Online; accessed 02/29/2024.

[41] Microsoft Threat Intelligence. *Analyzing Solorigate*. 2020. https://www.microsoft.com/en-us/security/blog/2020/12/18/analyzing-solorigate-the-compromised-dll-file-that-started-a-sophisticated-cyberattack-and-how-microsoft-defender-helps-protect/ Online; accessed 01/25/2024.

[42] Mortenson, Samuel. *mortenson/pr-sneaking: A repository demonstrating how you can sneak malicious code into Github PRs*. Apr. 11, 2024. https://github.com/mortenson/pr-sneaking.

[43] Naraine, Ryan. *Open-source ProFTPD hacked, backdoor planted in source code*. 2010. https://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code/ Online; accessed 04/11/2024.

[44] NPM Blog. *Plot to steal cryptocurrency foiled by the npm security team*. 2019. https://blog.npmjs.org/post/185397814280/plot-to-steal-cryptocurrency-foiled-by-the-npm Online; accessed 02/20/2024.

[45] Ohm, Marc, Plate, Henrik, Sykosch, Arnold, and Meier, Michael. *Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2020, 23–43.

[46] Okafor, Chinenye, Schorlemmer, Taylor R., Torres-Arias, Santiago, and Davis, James C. *SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. Proceedings of the 2022 ACM SCORED Workshop*. SCORED'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 15–24. DOI: 10.1145/3560835.3564556.

[47] Pessarlay, Wahid. *Blockchain market share in healthcare to reach $1,029 billion by 2034: report*. 2024. https://coingeek.com/blockchain-market-share-in-healthcare-to-reach-1029-billion-by-2034-report/ Online; accessed 08/22/2024.

[48] Pike, Rob. *Go at Google: Language Design in the Service of Software Engineering*. 2012. https://go.dev/talks/2012/splash.article Online; accessed 04/16/2024.

[49] Rannou, Sébastien. *Ethereum Client Diversity Part 2: Execution-layer Diversity*. Kiln. https://www.kiln.fi/post/ethereum-client-diversity-part-2-execution-layer-diversity Online; accessed 08/21/2024.

[50] Reproducible Builds project (Lamb, Chris). *Definitions*. 2019. https://reproducible-builds.org/docs/definition/ Online; accessed 04/16/2024.

[51] S. T. A. Group. *Software Supply Chain Best Practices*. Tech. rep. Cloud Native Computing Foundation, 2021.

[52] Shostack, Adam. *Threat modeling : designing for security*. 1st edition. Indianapolis, IN: John Wiley and Sons, 2014 - 2014.

[53] SolarWinds. *Setting the New Standard in Secure Software Development: The SolarWinds Next-Generation Build System*. Tech. rep. 2021. https://www.solarwinds.com/resources/whitepaper/setting-the-new-standard-in-secure-software-development-the-solarwinds-next-generation-build-system/delivery Online; accessed 05/07/2024.

[54] Sonatype. *9th Annual State of the Software Supply Chain*. Tech. rep. 2023. https://www.sonatype.com/state-of-the-software-supply-chain/Introduction Online; accessed 01/26/2024.

[55] Sorniotti, Alessandro, Weissbacher, Michael, and Kurmus, Anil. *Go or No Go: Differential Fuzzing of Native and C Libraries. 2023 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2023.

[56] Soto-Valero, César, Monperrus, Martin, and Baudry, Benoit. *The Multibillion Dollar Software Supply Chain of Ethereum. IEEE Computer* (2022). DOI: 10.1109/MC.2022.3175542. http://arxiv.org/pdf/2202.07029.

[57] T. L. Foundation. *SLSA: Supply-chain Levels for Software Artifacts*. 2023. https://slsa.dev Online; accessed 02/29/2024.

[58] The Go Programming Language. *compile command*. https://pkg.go.dev/cmd/compile Online; accessed 06/05/2024.

[59] The Go Programming Language. *go command*. https://pkg.go.dev/cmd/go Online; accessed 06/09/2024.

[60] The Go Programming Language. *Go Modules Reference*. https://go.dev/ref/mod Online; accessed 04/16/2024.

[61] The Go Programming Language. *go/cmd/internal/vcs/vcs.go*. https://cs.opensource.google/go/go/+/master:src/cmd/go/internal/vcs/vcs.go;l=329;drc=27093581b2828a2752a6d2711def09517eb2513b Online; accessed 08/21/2024.

[62] The Go Programming Language. *link command*. https://pkg.go.dev/cmd/link Online; accessed 06/05/2024.

[63] The Go Programming Language. *Source file work/buildid.go*. https://go.dev/src/cmd/go/internal/work/buildid.go Online; accessed 05/21/2024.

[64] Thompson, Ken. *Reflections on Trusting Trust. Commun. ACM* 27.8 (1984), 761–763. DOI: 10.1145/358198.358210. https://doi.org/10.1145/358198.358210.

[65] Travis CI. *The Ubuntu Linux Build Environments*. https://docs.travis-ci.com/user/reference/linux/ Online; accessed 06/08/2024.

[66] tukaani-project. *XZ Utils*. https://github.com/tukaani-project/xz Online; accessed 04/18/2024.

[67] Valsorda, Filippo. *How Go Mitigates Supply Chain Attacks*. 2022. https://go.dev/blog/supply-chain Online; accessed 04/16/2024.

[68] Vaughan-Nichols, Steven. *This backdoor almost infected Linux everywhere: The XZ Utils close call*. ZDNET, 2024. https://www.zdnet.com/article/this-

backdoor-almost-infected-linux-everywhere-the-xz-utils-close-call/ Online; accessed 04/18/2024.

[69] Wheeler, David A. *Countering Trusting Trust through Diverse Double-Compiling (DDC). Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society Order Number P2461. Tucson, Arizona: IEEE Computer Society, 2005, 28–40.

[70] Xiao, Claud. *Update: XcodeGhost Attacker Can Phish Passwords and Open URLs through Infected Apps*. 2015. https://unit42.paloaltonetworks.com/update-xcodeghost-attacker-can-phish-passwords-and-open-urls-though-infected-apps/ Online; accessed 04/11/2024.

Appendix - Contents

A Attack Vector Taxonomy Modifications **64**

A.1 Criteria for Excluding Attack Vectors 64

A.2 Criteria for Excluding Safeguards 65

Appendix A

Attack Vector Taxonomy Modifications

In this appendix, we present the modifications to the attack taxonomy and safeguards [32] used in our attack vector analysis in Chapter 4.

A.1 Criteria for Excluding Attack Vectors

The original supply chain attack taxonomy includes 107 unique attack vectors. We compiled these into 13 relevant to Geth by filtering attack vectors on the following criteria:

1. Social Engineering Attack: The attack is a social engineering vector without clear mitigation strategies.

2. Fine-grained: The child attack vector is sufficiently explained by its parent, making it redundant.

3. Package-Centric View: The attack vector is specific to a package-centric perspective.

4. Inapplicability: The attack vector is irrelevant to the Go programming language or Geth.

A.2 Criteria for Excluding Safeguards

The taxonomy is accompanied by 43 unique safeguards for mitigating supply chain attacks. We compiled these into 12 relevant to our attack vectors. We filtered the safeguards based on the following criteria:

1. General Security Measure: The safeguard is too general and lacks specificity to the attack vectors.
2. End-User Focused: The safeguard is intended for end-users rather than developers or projects.
3. Inapplicability: The safeguard does not apply to the Go programming language or Geth.