DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2022

Diverse Double-Compiling for Bitcoin Core

Niklas Rosencrantz

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Author

Niklas Rosencrantz
Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
KTH Royal Institute of Technology

Examiner

Professor Benoit Baudry
Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Supervisor

Professor Martin Monperrus

Electrical Engineering and Computer Science

KTH Royal Institute of Technology

Abstract

A supply chain attack is a type of attack that targets the software or hardware that is used to create a system. By compromising the supply chain, an attacker can insert malicious code into the system that can be used to attack the system. Supply chain attacks have been used to attack the Bitcoin network in the past, and they are a serious threat to the security of Bitcoin. To secure a system against supply chain attacks, it is important to trust every piece of software and hardware used to create the system. Due to the financial capabilities of the Bitcoin Core project, there has been a general interest in security issues with its design and implementation. While most questions and issues have centered around Bitcoin Wallets and breach not because of Bitcoin itself but a bug in a web framework for the Cryptocurrency Exchange etc., research and information are scarce about how Bitcoin Core has applied secure development or application security in the past and currently. The first Bitcoin white paper was aimed at solving the problem of double-spending. It didn't specifically address the security of its own implementation, and nothing was written about a trusting trust attack. This article discusses the various ways a supply chain attack can occur and how it can be used to attack Bitcoin. It also discusses the steps that can be taken to secure a system against such attacks. A backdoor in a login system can take the form of a hard-coded user and password combination which gives access to the system. This backdoor may be inserted by the compiler that compiles the login program, and that compiler might have been subverted by the ancestor compiler that compiled the compiler. I have implemented a proof-of-concept attack on a C compiler and show that techniques can detect and mitigate the attack. Such a problem with deceptive compilers introducing malware into the build system has been described in several articles and studied in computer security and compiler security. During the years of research on this class of attacks, it was implied that there are several additional sub-problems to analyze and discover, such as actually creating a working example of the attack and also the different methods to reduce the probability of such an attack. One mitigation that has been described is to use diverse double-compilation as a countermeasure. I introduce a variant of DDC that will build. with two compilers twice, the second time switching the roles of the compilers to reduce the probability of undetected compiler vulnerabilities. This variety has not previously been described. In the context of the Bitcoin Core build system, an attack and compromise of it are possible by a few attack methods and exploits that can be tried and analyzed, for example:

(describe the problems that are solved)

1. Stealing a PGP key from a maintainer and using that to sign new versions of Bitcoin-Core
2. Conducting a domain-specific attack where the compiler inserts code that changes the recipient address in 1/10000 Bitcoin transactions.
3. A malevolent maintainer could upload malicious code and hide it. that would get caught by the verify signatures script

(The presentation of the results should be the main part of the abstract. Use about ½ A4-page.)

(Use probably one sentence for each chapter in the final report.) Write an abstract.

Introduce the subject area for the project and describe the problems that are solved and described in the thesis. Present how the problems have been solved, methods used and present results for the project.

English abstract

Keywords

Application security, Computer security, Compiler security, Software Supply Chain, Malware, Build security, Network security, Trojan Horse, Computer Virus, Cybersecurity, Information Security, Merkle tree, Bitcoin Code, Cryptocurrency, Cryptography, Signed code, attack, software, malicious, security, Trojan

Abstract

Svenskt abstract Svensk version av abstract – samma titel på svenska som på engelska.

Problemet med vilseledande kompilatorer som introducerar skadlig kod är relevant för datorsäkerhet och en svår klass av attacker att analysera och upptäcka. En lösning för detta är att använda olika dubbelkompilering som motåtgärd. Ändå är det svårt att ändra kompileringspipelinen för riktig programvara. Jag lär mig och skriver om designen av, och jag implementerar och utvärderar olika dubbelkompilering för bitcoin-core. För att attackera och kompromissa med Bitcoins kärna kan man överväga några attackmetoder och exploateringar som kan prövas och analyseras, till exempel:

Nyckelord

Civilingenjör examensarbete, ...

Acknowledgements

I would like to thank everyone who encouraged and inspired me, especially my professors Martin and Benoit, Rand Walzmann, Basile Starynkevitch, Terrence Parr, Fredrik Lundberg, Johan Håstad, Roberto Guanciale, David A. Wheeler, Bruce Schneier, Johan Petersson and Holger Rosencrantz, both who reviewed early versions of this text and my lovely wife Ecaterina who contributed with artwork for the illustrations and diagrams in this report.

Acronyms

ACL2 A Computational Logic for Applicative Common Lisp

ANSI American National Standards Institute

ASLR address space layout randomization

CPU central processing unit

DDC diverse double-compiling

DoS denial of service

FPGA field-programmable gate array

GCC GNU Compiler Collection

GDB GNU Debugger

HT hardware trojan

MiTM man-in-the-middle

NFT	non-fungible token
NGO	non-government organization
NIST	National Institute for Standardization
NIST	National institute of Standards and Technology
OSS	open source software
PGP	Pretty Good Privacy
RAM	random access memory
SQL	Structured Query Language
TCC	Tiny C Compiler
XBI	cross-build injection

Contents

1 Introduction 1

1.1 Hypothesis Statement 1

1.2 Background 3

1.3 Problem Statement 6

1.3.1 Trojan Horses 6

1.4 Preliminaries and Definitions 7

1.5 Examples of Key Attack Techniques 9

1.5.1 XcodeGhost 9

1.5.2 Win32/Induc.A 9

1.5.3 ProFTP Login Backdoor 10

1.5.4 SUNBURST Malware 10

1.5.5 Supply Chain Security 10

1.6 Methodology 13

1.7 Contributors 14

2 Related Work 15

2.1 The Threat Landscape 17

2.1.1 Software and Hardware Backdoors 17

2.1.2 Self-Replicating Compiler Malware 18

2.1.3 Deceptive Hardware Trojans 18

2.2 Countermeasures to Backdoors 19

2.2.1 Response-Computable Authentication 20

2.2.2 Isolating Backdoors with Delta-Debugging 20

2.2.3 Firmware Analysis 21

2.3 Secure Compilation 21

2.3.1 Debootstrapping 22

2.3.2 Self-Hosted Systems 22
2.4 Testing and Verification 23
2.4.1 Verification of Source Code and of Compiler 23
2.4.2 Diverse Double-Compilation 24
2.4.3 Reproducible Builds 26
2.4.4 Fuzz Testing 27
2.5 Secure Development for Cryptocurrency 28
2.6 Summary 30

3 Method **31**
3.1 Qualitative Investigation with Interviews 31
3.1.1 Interview Protocol from Interview with David A. Wheeler 32
3.1.2 Discussion with Committers to GCC 35
3.2 Feasibility of the Attack 35
3.3 Creating a Real Example Attack 35
3.4 Executing an Example Attack 39
3.5 The Defense 39
3.5.1 Theoretical Background 39
3.5.2 Countering Trust Attacks with DDC 43
3.6 Summary 45

4 The work **47**
4.1 Implementation details of self-replicating compiler malware 51
4.2 Demonstration 51
4.3 Interview Protocol 52
4.3.1 Interviewees 52
4.3.2 Interview Questions 52
4.4 Results 52
4.4.1 Learning the background and techniques 52
4.4.2 Practical usage 53
4.4.3 What will future work be like? 54
4.4.4 Alternative approaches 54
4.4.5 What is the critique or drawbacks? 54
4.4.6 What can we expect from the future? 55

4.4.7 What would a software be like that could perform the checks against supply-chain attacks? 55
4.4.8 Why should the build system be self-hosting in the first place (build itself)? 55
4.5 Main Findings 55

5 Result **57**

6 Conclusions **59**
6.1 Discussion 59
6.1.1 There is more than Code 61
6.1.2 Future Work 61
6.2 Extending DDC 61
6.2.1 Final Words 61

References **62**

Chapter 1

Introduction

Supply-chain attacks have become a problem with computer systems and networks when the system will inherently trust its creator i.e. the hardware constructors and the software authors, the upstream development team, and the supply chain. Software developers normally don't expect an attack on their code while it is being built by the build system. An attacker could put in a backdoor (Trojan horse) in a program or a system as it is being built.

My work with this project, as described in this report, has been to investigate and understand the vulnerabilities and the means of mitigation.

Changing the compilation pipeline of real software without it being seen might sound difficult, but it can be done. In my research, I examine the design of such attacks, and I implement and evaluate the technique named diverse double compilation for bitcoin core. I also suggest a few completely novel ideas to reduce the assumptions and reduce the probability even more that a system is compromised1.2.

Cybersecurity can be accomplished primarily in two complementary ways: secure software development, a NIST initiative, and software vulnerability protection.

1.1 Hypothesis Statement

Trusting Trust - The hypothesis being worked with is: To what extent Bitcoin Core is secure from supply-chain vulnerabilities? There are methods to reduce the attack surface of a computer system in general.

The key idea is that to believe you are 100% secure, you must trust every piece of software and hardware that has ever been used to create your system. During my research of the topic, I've studied, created, and evaluated certain new and established techniques of diverse double-compiling in the context of the build system for Bitcoin Core [44]. To attack and compromise Bitcoin core one might consider a few attack methods and exploits that can be tried and analyzed, for example:

1. stealing a Pretty Good Privacy (PGP) key from a maintainer and using that to sign new versions of Bitcoin-Core
2. a domain-specific attack where the compiler inserts code that changes the recipient address in 1/10000 Bitcoin transactions.
3. a malevolent maintainer could upload malicious code and hide it. that would get caught by the verify signatures script

The hypothesis is: To what extent is Bitcoin Core secure from supply-chain vulnerabilities? Malware can encrypt itself, spread, and replicate as a virus on computers. Supply-chain attacks have become a problem with computer systems and networks when the system inherently trusts its creator, i.e., the hardware constructors and the software authors, the upstream development team, and the supply chain. An attacker could design a backdoor (Trojan horse) in a program or a system as the build system processes the build. Software developers typically do not expect an attack on their code while compiling it. The motivation for this kind of attack is quite easy to understand: One can control entire organizations and even governments and countries if one controls the means to compromise the computer networks and their supply chains. Or acquire enormous amounts of financial funds by manipulating the financial systems or the decentralized financial systems such as cryptocurrency blockchains and non-fungible tokens (NFTs).

The motivation for this kind of attack is relatively easy to understand: One can control entire organizations and even governments and countries if one holds the means to compromise the computer networks and their supply chains. An attacker could also acquire enormous monetary funds by manipulating the economic systems or the decentralized financial systems of cryptocurrency blockchains such as NFTs.

Provide a general introduction to the area for the degree project. Use references!

Link things together with references.

1.2 Background

A common situation is that a new compiler will be installed on a system. That compiler is being compiled, and a Trojan horse might have targeted a specific input, such as the compiler itself.

Packages used in programming languages have recently been high-profile targets for supply chain attacks. A popular package from the Ruby programming language used by web developers was compromised in 2019 when it started to include a snippet of code allowing remote code execution on the developer's machine [60]. What is interesting about this case is that the authors' credentials were compromised, and the malicious code was never found in the code repository. It was only available from the downloaded version installed by the Ruby package manager. https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem/

In another incident, a compiler attack referred to as Xcodeghost (2015) constituted a malware Xcode compiler for Apple macOS that can inject malware into output binary. n September 2015 a new compiler malware attack was discovered in China. This attack targeted Apple's Xcode development environment, the official development environment for iOS and OSX development. The attack was a malicious compiler that the user could download. As users in China had slow download speeds when downloading large files from Apple's servers, many users would instead download Xcode from other colleagues or from Baidu Wangpan, a Chinese cloud service created by Baidu. The malicious compiler is believed to have been spread initially through Baidu Wangpan [27]. This malicious version of Xcode replaces CoreServices object files with malicious object files. These object files are used to compile many iOS and OSX applications. It is unknown to the author if the attack did anything to OSX applications. Nevertheless, the attack infected many iOS applications. These applications were then spread through the Apple App Store to the end-users, while neither the users of the applications nor the developers of the applications knew of this. Pangu Team claims the attack infected at least 3418 different iOS applications [25]. Amongst the apps infected were WeChat version 6.2.5, a very popular instant messaging application [22]. In September 2015 WeChat had 570 million active users daily [26]. As iOS had a mobile phone market share of roughly 25At the same time, it is to be expected that the virus can have reached many millions of users [23]. 15 The infected iOS applications will gather system data, encrypt it and send it to a remote server using HTTP [27]. The application also contains the ability to attempt to trick the user into giving away their iCloud password through a crafted dialogue box. Further, the attacker can read and write to and from the clipboard. It can also craft and open malicious URLs, this can also be used for malicious behavior through crafting specific URLs that open other apps with security weaknesses [43]. The attack seems to have spread mostly in China, as the applications infected were mostly Chinese-developed applications targeting the Chinese market [18]. Nevertheless, applications such as WeChat have been popular in larger regions of eastern Asia. The malware can therefore also spread to larger regions [22]. Pangu Team also released an application to detect malicious applications created through XcodeGhost infected compilers [25].

A relatively recent third incident of this type has been the Win32/Induc. A that was targeting Delphi compilers. W32/Induc is a self-replicating virus that works similarly to the compiler trap door attack. The compiler trap door attack will be further explained in Chapter 3. The virus inserts itself into the Delphi source 13 libraries upon execution, infecting the compiler toolchain. It then inserts itself into all produced executables from the infected toolchain. The virus targets Delphi installations running on a Windows platform. The virus has come in three known variants named Induc-A, Induc-B and Induc-C [39]. It is believed that the two initial runs were testing versions to test the insertion of the virus building up to the release of the more malicious Induc-C virus. Upon executing an infected file, the virus will check for the existence and location of a Delphi installation [39]. Early versions of the virus looked for Delphi installations by looking for a specific registry subkey. Later versions will instead search the hard drive for a compatible Delphi installation. Once the installation is found, the virus will create a backup of the original SysConst.dcu (for earlier versions of the virus) or SysInit.dcu (for later versions of the virus) used for all produced executables [39, 44]. After this, it will copy the SysConst.pas or SysInit.pas file from the object library, modify the source code to include the malicious behavior, and compile the file. It will then be inserted so that it is used instead of the original SysConst.dcu or SysInit.dcu. At this point, the Delphi compiler is infected, and all produced executables from the compiler will also include the virus. Induc-C also can infect any .exe files on the computer [39]. This greatly increases the virus' ability to spread to other computers. The initial versions of the virus (Induc-A and Induc-B) seem not to include any malicious behavior other than self-reproduction [39]. In contrast, Induc-C includes behavior where it downloads and runs other malware. It does this by downloading specific JPEG files containing encrypted URLs in the EXIF sections. It will then download and execute the malware at these locations. Amongst known malware executed is a password stealer. It is also reported that Induc-C includes behavior that can be used for botnets. The known defense against the attack is antivirus software, which can detect infected executables or infected object files [44]. As of September 2011, over 25were recorded in Russia [39]. For Induc-C most of the detected infections occurred in Russia and Slovakia. 14 This attack is similar to the compiler trap door attack, as they both attack compilers and include self-replicating behavior. The main difference between this and the compiler trap door attack is that instead of attaching the virus to the compiler executable, it inserts itself into object files used for the compilation of all programs using the infected toolchain. The compiler executable itself does not get infected unless it is produced using this toolchain. As the malware isn't specifically attached to the compiler executable, it can be easily delivered through any infected executable and will then further spread itself to all compiled executables. It is reasonable to believe that this the method will resort to a virus that spreads itself faster to more computers, however, it might also be easier to detect

In another recent attack, there was a Trojan horse attack on cryptocurrency usage in a codebase that was compromised from a GitHub repository [60].

A sufficiently complicated project often takes a lengthy and complex build process, with many binaries from several vendors utilized as dependencies and libraries in the toolchain. Compiling Bitcoin Core locally for example takes 40 minutes with an Intel I7 CPU. Compiling GCC takes several hours, and compiling Firefox takes a whole day.

What is diverse double-compiling? (TODO) [62].

Present the background for the area. Give the context by explaining the parts needed to understand the degree project and thesis. (Still, remember that this is an introductory part, which does not require too detailed description).

Use references[1]

A detailed description of the area should be moved to Chapter 2, where detailed information about the background is given together with related work.

[1]You can also add footnotes if you want to clarify the content on the same page.

This background presents the background to writing a report in latex.

Look at the sample table 1.2.1for a table sample.

Table 1.2.1: Sample table. Make sure the column with adds up to 0.94 for a nice look.

SAMPLE	**TABLE**
One	Stuff 1
Two	Stuff 2
Three	Stuff 3

Boxes can be used to organize content

Development environment for prototype

```
Operating systems
computer: Linux - kernel 4.18.5-arch1-1-ARCH
android phone: 8.1.0

Build tools
exp (build tool): version 55.0.4

...
```

1.3 Problem Statement

I present the problems found in the area. I prefer to use and end this section with a question as a problem statement. The purpose of the degree project/thesis is the purpose of the written material, i.e., the thesis. The thesis presents the work / discusses/illustrates and so on. The goal means the goal of the degree project. Present the following: the goal(s), deliverables, and results of the project.

1.3.1 Trojan Horses

Self-reproducing compiler attacks have been called deniable, taken from the expression of plausible deniability [71]. The concept has been applied to cryptography, as in the expression deniable cryptography. It can be used for attacks as well, where the attack can be denied that attacks were there because it couldn't be seen.

Listing 1.1: C example

```
    ...
    /* intercept the login for a specific username */
int login(char *user) {
    /* start of enemy code */
if(strcmp(user, "hackerken") == 0) return 1;
    /* end of enemy code */
    ...
```

I aim to give answers and elaborations to the following questions. What are the threats and vulnerabilities that should be protected against with diverse double-compiling (DDC)? What are the technical benefits and shortcomings of DDC compared to other solutions to real and potential problems with compiler vulnerability and building security? What critique is known against DDC and what other critique is reasonable to give? What does a real demonstration of DDC look like? Show (don't tell) the audience a real example of an actual procedure. How can DDC be applied to the build system of Bitcoin Core, and what can we learn from it? What would be the challenges and benefits? Which part of the build process is more likely to be vulnerable than other parts? Can I recompile Bitcoin Core as the system-under-test, and what will that result be? What possible and provable improvements can be made compared to the current state of the art, and how can the technology become portable and easily available for software engineering teams and researchers? Can a compiler binary be made more auditable and examined and if yes, how??

Use references Preferably, state the problem to be solved as a question. Do not use a question that can be answered with yes and/or no.

Use acronyms: The central processing unit (CPU) is very nice. It is a CPU

It is not "The project is about" even though this can be included in the purpose. If so, state the project's purpose after the purpose of the thesis).

1.4 Preliminaries and Definitions

Define FIRMWARE

According to the National institute of Standards and Technology (NIST) glossary, a trusted computer system is described as a system that is sufficient to perform its tasks, while a trustworthy system is described as a system that is reasonably secure and according to established standards [37]. The use of the terms may differ between different contexts.

There is a fundamental difference between physical security and digital security in a manner of acknowledgment of solvable problems and the apparent lack of solutions [61]. Technologies in provable security have increased interest in software engineering and related disciplines.

The meaning of trust can be understood as certain confidence that something will behave as expected, with or without real proof that it will. The security researcher and author Ross Anderson has pointed out the difference between a trusted system and a trustworthy system [2]. A trustworthy system will typically base trust upon proof or formal verification for being verified, while a system that is simply trusted would not base incoming trust upon proof or verification but some other characteristic such as origin or proximity. The trusted and the trustworthy systems might even be mutually exclusive. It can be the case that none of the trustworthy systems are parts of the systems that are actually trusted by a certain individual(s), for example, the case when all computers in a closed network have been compromised. The terminology of a trusted system will be used in the succeeding sections according to the aforementioned definitions. What's being discussed is often a form of cross-build injection (XBI).

One might ask what makes one system more trustworthy than the other and if the question of trustworthiness is only a matter of trust in the meaning of a user's perceived understanding of reliability and security. The security researcher Caspar Bowden pointed out that the meaning and connotation of the word trust are completely different in different environments [8]. According to Bowden, for a policy-maker, trust means that it's part of the goals and objectives for a system to be trusted. For a security engineer, trust means that the system isn't perfect when it's based on trust instead of objective verification, which would be ideal [8].

On one side there's been described the concept of provable security with formal verification is a way to theoretically verify all parts and states of a system, even if it takes a very long time with the methods and equipment available today.

The word "binary" is often used to distinguish between executable files (instructions) and text or media (data files), where the latter type of files is not directly executable by the processor or an interpreter. A compiler is expected to accept source code and generate object code. Hence, by definition, the contents of a computer's RAM (or peripheral storage) can be either (or both) of:

Executable: Data that can be executed by a computing environment. Compilers produce executables, and compilers themselves are executables.

Source: Data that can be compiled by a compiler to produce an executable. Any source (aka source code) is written in some language.

A software backdoor is an undocumented way of gaining access to a computer system. A backdoor is a potential security risk. For instance, a malicious process can potentially be listening for commands on certain ports or sockets. Since hardware and software backdoors started to appear, there's been ongoing work to protect against known backdoor attacks. Researchers have also deliberately created new attacks against their own systems to find and mitigate vulnerabilities before the vulnerabilities become exploited.

While a relatively large amount of research has been done on the topic of supply chain security, not much has been covered about supply chain security in the context of cryptocurrency and digital payment solutions. Security of hardware, network security, and physical security (e.g. ”analog” security of buildings and physical locks, etc.) are briefly mentioned as the foundation of digital and software supply chain security but will mostly be omitted in this essay for the sake of narrowing the scope.

1.5 Examples of Key Attack Techniques

1.5.1 XcodeGhost

In 2015 a compiler for Apple Xcode appeared that seemed more easily available in Asia. It was compromised with malware that would inject malware into the output binary. It was named XcodeGhost and was a malware Xcode compiler for Apple macOS

1.5.2 Win32/Induc.A

Another malware targeting Delphi compilers was called Win32/Induc.A

1.5.3 ProFTP Login Backdoor

In 2010 there was an injection of a login backdoor in the software named ProFTP. This was a supply chain attack that will probably be able to automatically detected in the not-so-distant future, while the attack was not merely in the binary code but also in the source code. What happened was that a malware hacker got access to the repository and made a commit that included a backdoor to the next version of the ProFTP software.

1.5.4 SUNBURST Malware

Certain attacks, such as the SUNBURST attack against the SolarWinds system, happen because malware was put into a vendor's trusted software; then an enemy may attack all the vendor's client organizations at once. This was a supply chain attack that happened because of trust [51].

1.5.5 Supply Chain Security

In general, a security model should make clear the resource that should be protected and what is the type of threats and vulnerabilities that should be protected against. It has been reported that 70% of Chrome vulnerabilities are memory safety issues and the same number, 70% of Microsoft vulnerabilities are memory safety issues[14] [15]. This might seem not directly related to building security at first hand, but considering that many projects are today being built directly from an internet browser that activates a CI/CD pipeline, it's possible that build security could be compromised from a vulnerability in an internet browser.

Some general good practices recommended are increased awareness, carefulness, code review, testing, and verifications [30]. (a paragraph is always at least 3 sentences) It's been argued that zero-trust should be practiced as much as practically possible [1].

Extensive work has been carried out in academic research and commercial industries to make software engineering and system development as secure as possible. Despite these efforts, the main problems have remained unsolved for decades, and supply-chain malware has been an increasing problem that has led to enormous damages and possibly could compromise the infrastructure of entire countries and governments, with recent experience from the SolarWinds incident where malware from the supply-chain corrupted numerous systems that were critical for security at Governmental organizations [51].

The results show that computer systems can be compromised by increasingly deceptive and stealthy malware. It is expected that vulnerabilities of computer systems can and will stay undetected during a source code review or during some other routine inspection. After Thompson showed in practice that compilers can be subverted to insert malicious Trojan horses into critical software, including themselves, it became more clear that the malware can be expected to stay undetected. If this kind of Trojan horse goes undetected, a complete analysis of a system's source code will not find the malware.

Supply-chain attacks are when an adversary tries to introduce targeted vulnerabilities into deployed applications, operating systems, and software components [10]. Once published, such vulnerabilities are likely to persist within the affected ecosystem even if patches are later released [11]. Following several attacks that compromised multiple firms and government departments, supply-chain attacks have gained urgent attention from the US White House [12].

There are several specialized software systems for the purpose of analyzing and finding vulnerabilities in source code and their dependencies[**SonarQube**] [20]. These systems are in their current capabilities unable to detect malware that exists in binaries because the scanning is only or mostly done at the source code level. there is a large listing about static source analysis on Wikipedia, but no listing of binary code analysis tools or dependency analysis tools.

A system that checks the dependencies of a source code project, such as Dependency-Track, would it could detect a compromised build system. An example would be if the build system named Maven itself were compromised or if the JAR provided from Maven Central had been compromised or something as easy as replacing a "safe" version of log4j with the vulnerable version to introduce a vulnerability that the dependency check will not find because the dependency check does not check the actual binary code itself but only the list of dependencies in the project.

An overview and online searches strengthen the hypothesis that there is no analysis software today that can detect a supply chain attack such as a compromised binary. The way Dependency-Track works is that it takes for granted that what is listed in, for example, pom.xml is the versions of the dependencies that should be looked up. It doesn't perform any binary scan inside the libraries.

In the recent classification by Dimov and Dimotrov, software security tools are classified into different classes. (WHICH ONES?) Using this classification, a diverse double-compiling would fall into the vulnerability scanning class [21].

In the context of Bitcoin, there have been a few notable cases of supply chain attacks. The first case is the Bitcoin-Core project, where a malicious developer uploaded a version of the Bitcoin-Core software that included a backdoor. This backdoor allowed the developer to steal funds from any Bitcoin address. The developer did this by changing the code that generated Bitcoin addresses. The second case is the Bitcoin-Core project, where a malicious developer uploaded a version of the Bitcoin-Core software that included a backdoor. This backdoor allowed the developer to steal funds from any Bitcoin address. The developer did this by changing the code that generated Bitcoin addresses. The third case is that of the Bitcoin.org website, where a malicious attacker was able to insert code that would redirect users to a phishing website. The attacker did this by compromising the server that hosted the website. In all of these cases, the attackers could compromise the systems because they had control of the supply chain. In the first two cases, the attackers were able to upload malicious code to the Bitcoin-Core project's code repository. In the third case, the attacker was able to compromise the server that hosted the Bitcoin.org website. In all of these cases, the attackers could compromise the systems because they had control of the supply chain. In the first two cases, the attackers were able to upload malicious code to the Bitcoin-Core project's code repository. In the third case, the attacker was able to compromise the server that hosted the Bitcoin.org website. These cases show it can attack the Bitcoin network by compromising the supply chain. In all of these cases, the attackers could compromise the systems because they had control of the supply chain. In the first two cases, the attackers were able to upload malicious code to the Bitcoin-Core project's code repository. In the third case, the attacker was able to compromise the server that hosted the Bitcoin.org website. All of these cases show that it is possible to attack the Bitcoin network by compromising the supply chain.

Maynor (WHO?) pointed out that we should trust no one, not even yourself or the weak link might be the build tools [41]. It was also told that a possible attack vector that often is not explored is attacking the program as it is built [40]. David Maynor means that the risk of trusting-trust attacks is increasing [40]. Maynor states that a compiler itself will invoke several other processes besides itself to translate a source file into a binary executable.

A large amount of discussion, research, questions being posed, and answers have resulted from the topic, but still no definite conclusion has been drawn. Much work has also been done both to avoid including backdoors as early as possible in the development pipeline, and to check, test and verify a finished release version to make it as likely as possible to verify that no backdoor was included.

Security models based on a zero-trust policy have been described in the literature and at NIST [36] [65]. Reproducible builds are a set of software development practices that create an independently-verifiable path from source to binary code [54]. Some of the questions that need answers are the following:

What is the range of a variable? Under what conditions are some code reachable? Any path, all paths? What dangerous actions does this program perform? Security scanning scripts can then ask questions such as: Can the source string buffer size of a particular unbounded string copy be larger than the destination buffer size? Was the return value from a security-critical function call tested for success before acting on the results of the function call? Is untrusted user input used to create the file name passed to a create-file function?

What is Zero-trust?

The three works that are often being cited about Bitcoin are first two mainly technical texts by two different research groups [7] [45]. The third is an article written more from the economical perspective [6]

. At the time of writing, these are getting out of date, so in what follows I will concentrate on developments since then. I'll assume you know the detail, or can look it up, or are not too bothered.

1.6 Methodology

Introduce, theoretically, the methodologies and methods that can be used in a project and, then, select and introduce the methodologies and methods that are used in the degree project. Must be described on the level that is enough to understand the contents of the thesis.

Use references!

Preferably, the philosophical assumptions, research methods, and research approaches are presented here. Write quantitative / qualitative, deductive / inductive / abductive. Start with theory about methods, choose the methods that are used in the thesis and apply.

Detailed description of these methodologies and methods should be presented in Chapter 3. In chapter 3, the focus could be research strategies, data collection, data analysis, and quality assurance.

Present the stakeholders for the degree project.

Explain the delimitations. These are all the things that could affect the study if they were examined and included in the degree project. Use references!

1.7 Contributors

In text, describe what is presented in Chapters 2 and forward. Exclude the first chapter and references as well as appendix.

Chapter 2

Related Work

Why should anybody trust that an executable (compiled) object is a legitimate representation of the source code of the intended program? Do we have good reason to believe that the compiled code does not contain malware embedded during compilation, which can reproduce itself forever? Ken Thompson asked these questions during his Turing Award lecture [67]. The idea originates from an Air Force evaluation of the MULTICS system carried out by Karger and Schell and published in a technical report in 1974 [33]. In 1985, a decade after work by Karger and Schell, Ken Thompson specified the vulnerability in more concrete detail. Thompson posed questions and an example of the source code in C.

Thompson's article has become a seminal work [9, 52, 69]. Thompson provided a detailed explanation of the attack, including source code to prove the concept with a hidden Trojan horse in the ancestor compiler, which is or was used to compile the next version. Thompson asked how much one can trust a running process (with one or more threads) or if it is more important to trust the people who wrote the source code. Thompson also stated that the vulnerability is not limited to the compiler or even ends with the build system: A supply-chain attack can compromise practically any program that handles another program in the way described, such as an assembler, linker, ar, Libtool, a loader, or firmware, and hardware microcode.

In the years after Thompson's article, the technologies of compiler security, dependency tracking, and supply-chain security have received even more interest from academic researchers and commercial businesses. Cybersecurity is today considered more critical than ever. The potential for deceptive malware to propagate in object form without being seen implies that the risk of enormous damage is technically possible. The build system used in most of these cases relies on the GNU Compiler Collection (GCC) [66]. A possible scenario is that a particular instance of the GCC contains self-replicating malware and that instance compiles itself to the next version. If that compiler compiles Bitcoin Core, a single individual, government organization, or some capable non-government organization (NGO) would be able to manipulate transactions centrally and arbitrarily. The ideas and executions were studied during the 1970s by Karger & Schell. Later, Ken Thompson pointed out the attack in the 1980s. Since then, practically no known researchers have made progress on the topic between the mid-1980s and the mid-1990s. The slow progress was partly due to the abundance of ideas and techniques for mitigation [33, 67]. Only in 1998 did Henry Spencer suggest a countermeasure [64].

Validation of input and source code verification are measures that can be appropriate to reduce the risk of an attacker getting control of the flow of execution. While a miscompilation caused the initial trust attack, more recent reports on systems security emphasized the input data. The authors Bratus et al. write that the input processed by the machine should be considered at least as important as the executable [9]. One observation is that input to a machine changes the state of the running process and the machine as if the input data were a program. Therefore, the authors meant that input data could and should be treated as a potential program because the input changes the machine's state.

Before the idea of double-compilation, nobody had suggested any countermeasures. There was no defense against the trust attack, or the defenses were insufficient. Security experts even claimed that there was no defense against the trust attack; Bruce Schneier has asserted that there is no defense if an attack causes the production systems to be compromised [55]. Consequently, given that there is no defense or countermeasure for a trust-based attack, attackers would quietly be able to subvert entire classes of computers and operating systems, gaining complete control over financial, infrastructure, military, and business systems worldwide.

Detecting binary differences through analysis methods for binary objects has been the main idea behind DDC and the other means of binary analysis. There are several technologies in use to detect binary differences. The related work discussed in this chapter primarily covers supply-chain security and the vulnerabilities resulting from third-party software or software dependencies [27, 47].

The two leading practices for improving the security of the supply chain are, firstly, the practice of testing and verification. Testing and verifying systems have been done extensively for many years to minimize the risk of including any vulnerabilities in the system's release version. Secondly, more recently, more emphasis has been put on a practice referred to as secure development and application security to avoid including vulnerabilities as early as possible in the supply chain. The latter practices a work method referred to as shifting left. Shifting left means software development should work on security from the beginning in the production line. The idea is that cybersecurity should be worked on and considered as early as possible instead of waiting for somebody to fix it later [16]. It is reasonable that these two practices complement each other instead of always being superior to the other. In many cases, a technique for testing and verification will depend on and require a specific development practice done before the test, e.g., using checksums. Therefore the two practices should be seen as complementary to each other.

2.1 The Threat Landscape

To give an overview of the threat landscape, Ohm et al. reviewed supply-chain attacks, emphasizing software backdoors [48]. Their results show measures and specific statistics of dependencies, modules, and packages in JavaScript (npm), Ruby (Gems), Java (Maven), Python (PyPI), and PHP.

Zboralski, a technical writer, states that this potential vulnerability is the central problem in network security [76]. As previously described by Ken Thompson, critical systems and activities rely on trust in the people who deliver the systems. We must either build the entire computer system ourselves only with our hardware and software or implicitly trust those who created the system. Zboralski further claims that the problems of trusting trust are good reasons to work in security engineering.

2.1.1 Software and Hardware Backdoors

Mistakes or malice can result in software vulnerabilities and compromised system security. It can happen during the design or during construction. Mistakes and malice sometimes cause backdoors in software and hardware during several steps of construction and development. All use of backdoors has not been out of malice since legitimate administrators and maintenance staff have historically used backdoors to be able to unlock every device of a certain kind for repair and maintenance [26]. This feature of undocumented ways to unlock any device has been a trade-off, partly to solve problems at the expense of compromised security. Today's computers cannot verify an entire system; it would take too long due to the enormous combinatorial number of possible circuit states.

2.1.2 Self-Replicating Compiler Malware

Software that is supposed to cause miscompilation so that the resulting executable contains a vulnerability is sometimes called compiler malware. John Regehr suggests mitigating the threat with countermeasures against such compiler malware [53]. Two of the questions posed are:

Will this kind of attack ever be detected? Who is responsible for protecting the system: The end-user or the system designer?

The secure compilation aims to protect against the threat of the compiler potentially getting compromised by someone who inserted malware into it. The canonical example is making the compiler check if it is compiling a specific source code, for example, the authentication part of a system, and conditionally embedding a backdoor into the login program so that a particular input sequence will always authenticate the user. This vulnerability was described and demonstrated by Thompson's Turing Award Lecture. The report describes some hypothetical cyberattacks. One of the attack scenarios concerns the possibility of exploiting an IT platform's compiler and production system. Karger and Schell returned more recently with a follow-up article describing the progress over the last 30 years, concluding that the scenario is still a problem that nobody has wholly solved [34].

2.1.3 Deceptive Hardware Trojans

Zhang et al. conducted research mostly about hardware trojan (HT) [79]. They state that vulnerabilities lead to the risk of novel and modified exploits on computer systems. Despite extensive work to prevent the attacks, the attacks are feasible through compromised hardware. Existing trust verification techniques are not effective enough to defend against hardware Trojan horse backdoors found in field-programmable gate array (FPGA), for example, used in military technology.

The authors propose a technology they call DeTrust [79]. DeTrust proves that somebody can include hardware Trojans during construction. These Trojans can also avoid detection by commonly used verification systems. These two verification systems are called FANCI and VeriTrust. FANCI performs a functional analysis of the logic in the circuit that is unlikely to affect the output and then flags it as potentially suspicious [68]. Another technology named VeriTrust tries to find malware circuitry by checking the circuit's extreme values and corner cases [78].

Zhang et al. also recognize that verifying all possible hardware states often becomes unfeasible in practice because of the rapid growth of combinations of possible states as bitlength becomes longer and the system's functionality becomes increasingly advanced [79]. After concluding that formal verification of non-trivial circuits is becoming unfeasible in practice, the authors describe techniques to defeat the two validation systems. Finally, the research group made some practical proposals for defending against the attack technique they first described. Their suggestion is to extend FANCI and VeriTrust, mainly consisting of ways to make the verification system able to follow and trace a signal in the hardware through more levels than today to detect an implicitly triggered hardware Trojan.

2.2 Countermeasures to Backdoors

The logical choice from companies and researchers is often to harden security from the parts of the system that are most exposed to the public or the outside, for example, the authentication systems, which would cause significant damage if they were compromised. The Gordon-Loeb model measures and economically quantifies these risks and potential damage to IT security [29]. In general, what is being done is often referred to as a reduction of the exposed surface of the system; according to common engineering principles that with fewer parts that are exposed, fewer parts can go wrong.

2.2.1 Response-Computable Authentication

Dai et al. created a framework for response-computable authentication (RCA) that can reduce the risk of including undetected authentication backdoors [18]. Their work extends to the Google Native Client (NaCl) [75]. The idea is to separate and perform checks of the authentication system. Part of the system consists of an isolated environment that works as a restricted part between the authentication system and the other parts of a system. The system is, in its turn, divided into subsystems. These subsystems include checking the cryptographic password-check function for collisions, detecting side effects, or finding other hidden defects or embedded exploits that could otherwise have gone undetected and compromised the authentication.

The underlying assumptions of the work by Dai's research group are similar to the assumptions of cryptographic systems in general: The premise is that an attacker has complete knowledge of the mechanisms in place but no knowledge of the actual passwords or secret codes, or keys in use. This principle descends from Kerckhoffs's principle. The assumption was reformulated (or possibly independently formulated) by American mathematician Claude Shannon. Shannon formulated it as ”the enemy knows the system.” Consequently, hardware and software developers must design and construct all systems assuming that an enemy will know how the machine works [35, 59]. The work by Dai et al. did not include actual testing or checking of their framework. Testing and checks could further prove the benefit. For example, somebody could test the framework with 30 different authentication mechanisms, where one of them is deliberately vulnerable. Then observe if the framework detects the actual vulnerability with as few false positives as possible.

2.2.2 Isolating Backdoors with Delta-Debugging

Schuster and Holz have written about ways to reduce the risk of software backdoors. Their work emphasizes specific debugging techniques that utilize decision trees in binary code [57, 77]. They introduced a debugging technique called delta-debugging, making it possible to detect which system parts are possibly compromised and perform further analysis steps. Their software uses GNU Debugger (GDB) and can analyze binary code for x86, x64, and MIPS architectures. Their software, named WEASEL, is available to the public on GitHub [56]. Their article then gives the results from practical test cases to show that the WEASEL can detect and disable both malware found in actual incidents and malware deliberately created for specific testing purposes. The authors do not, however, describe how to detect a possible vulnerability in their dependence on the GDB.

Schuster et al. also describe techniques on how to prevent backdoors proactively [58]. They extended the previous work with Napu proposed by Dai et al. The result is a system that has reduced the risk of vulnerabilities through virtualization and isolation.

2.2.3 Firmware Analysis

Shoshitaishvili et al. describe a system called Firmalice [61]. Firmalice is an analysis system for firmware. The authors note that Internet of Things (IoT) devices are becoming more common not in many environments and that there have been mistakes that caused vulnerabilities of the software and firmware. Shoshitaishvili et al. state that for analysis, there is a significant difference between openly available source code and proprietary source code. They give because there is no direct availability to review or check the source and dependencies of an embedded system built from closed and proprietary source code. The authors claim that their system can find vulnerabilities that other analysis systems cannot, namely the one from Schuster and Holz, which rests on certain assumptions that Firmalice does not need [57].

Their article describes how to detect backdoors. Proprietary source code is often unavailable for direct analysis, so the Firmalice system uses existing disassembly techniques. It then identifies what the privileged state of the program could be and generates certain graphs (dependency graphs and flow graphs) so that the analysis identifies what instructions lead to the privileged state. The authors then report several cases of real product vulnerabilities in the object code. The authors can evaluate the accuracy of the analysis system and find out if the numbers of any false positives or false negatives are within the acceptable range [61].

2.3 Secure Compilation

Secure compilation can ensure that compilers preserve the security properties of the source programs they take as input in the target programs they produce [50]. Secure development is broad in scope; it targets languages with various features (including objects, higher-order functions, memory allocation, and concurrency) and employs various techniques to ensure preserving the security of the source code in the generated executable and at the target platform.

Attacks against the described compiler have been called deniable since the attack and undetected when viewing the compiler's source code. The term descends from the legal expression ”plausible deniability” and the technology known as deniable cryptography [3, 71]. The property of being deniable is a primary characteristic of the most brutal supply-chain attacks and constitutes a significant challenge.

2.3.1 Debootstrapping

A technique called debootstrapping has been suggested and put into practice to avoid trusting the software production system [17]. The idea is always to use very different compiler implementations so that one compiler can test the other and avoid self-compiling compilers that compile different versions. The need for debootstrapping has been described in work by Courant et al. The motivation for debootstrapping is to remove the self-dependency and be able to check for a compromised compiler. It involves creating a new compiler in some programming languages other than the language of the compiler-under-test. The result, called the debootstrapped binary, may be very different from the bootstrapped binary (with different or no optimization, as our debootstrapped compiler produces worse code than before debootstrapping); it should have the same semantics.

For Debootstrapping, it will need a second independent compiler where the requirements are somewhat different from the production-level compiler that should be released. The compiler used for checking can only implement a subset and does not need to meet critical performance requirements or be optimized since its purpose is only correctness. Two such compilers have used a Java compiler named Jikes to debootstrap a Java compiler and a minimal C compiler called Tiny C Compiler (TCC) to debootstrap GCC [4].

2.3.2 Self-Hosted Systems

There have been findings of an undocumented extra microprocessor in specific systems [24]. There are claims that the only system that can be entirely trustworthy is the one where everything used to create the system is available in the system itself. Somlo describes such a self-hosted system in a recent research report [63]. Somlo notes that the lengths of supply chains are getting longer and longer. Consequently, according to Somlo, it increases the complexity of the problem of checking whether a system is trustworthy. Somlo describes an approach with steps to perform DDC. Somlo takes a broader scope and suggests an entirely self-hosted independent system with FPGA capable of checking another system. The idea is to have the system that performs the check also reduce the risks of being compromised in the linker, loader, assembler, operating system, or mainboard.

2.4 Testing and Verification

Several sources write that it is better to verify than to trust [46, 65]. The recommendation is to adhere to a zero-trust policy as much as possible [1]. During verification, one major challenge has been the exponentially increasing number of states of the system that need to be verified, and the tools available for verification have not been able to keep up with the advances in more complicated computer systems.

2.4.1 Verification of Source Code and of Compiler

Formal verification techniques consist of mathematical proofs of the correctness of a system that can be either hardware or software, or both [49]. The authors describe how Frama-C has been used to prove the correctness of some properties annotated into a critical C code embedded into aircraft. Functional equivalence is generally undecidable. The authors describe specific techniques (model checking and more) that are approximate solutions to the equivalence problem. It only requires a reduced state space with annotations and assertions in the source code for checking.

A related technology uses an approach with proofs at the machine-code level of compilers. For this purpose, researchers use a system named A Computational Logic for Applicative Common Lisp (ACL2) as a theorem-prover for LISP. The literature contains a plethora of descriptions of formal verification of compilers [73]. Wurthinger writes that even if the compiler is correct at the source level and passes the bootstrap test, it may be incorrect and produce incorrect or harmful outputs for a specific source input [19]. There were also attempts to analyze binary (executable) code to detect vulnerabilities directly. Some methods utilized techniques from graph theory to conduct an analysis [23, 27].

2.4.2 Diverse Double-Compilation

DDC is a technique proposed by David A. Wheeler in 2005. (DDC needs reproducible/deterministic compiler builds.) David A. Wheeler used an alternative implementation to gain trust in a bootstrapped binary, proving the absence of a trust attack [69]. First, a bootstrap binary compiles a reference implementation from the source, and we check that the resulting binary is identical to the bootstrap binary. Second, we use our debootstrapped implementation to build the reference implementation under test. Finally, we use the debootstrapped binary to compile the reference implementation again, getting a final binary. The final binary is produced without the bootstrap binary, using the compiler sources from the reference implementation. If it is bit-for-bit identical to the bootstrapped binary, then we have proved the absence of a trusting trust attack. If it differs, there may be a malicious backdoor or a self-reproducing bug, but there may also be a reproducibility issue in the toolchain. The main point is that DDC will detect a compromised compiler through the previously described procedure unless multiple compilers are compromised.

Historically, Henry Spencer was the first to suggest a comparison of binaries from different compilers [64]. The idea was originated by McKeeman and Wortman., who had written about techniques for detecting compiler defects and provided a formal treatment for verifying self-compiling compilers [42]. McKeeman also introduced T-diagrams to illustrate compilation techniques. Spencer remarked that compilers are a particular case of computer programs establishing a trustworthy and honest system. It will not work to simply compile the compiler using a different compiler and then compare it to the self-compiled code because two different compilers – even two versions of the same compiler – typically compile different code for the given input. However, one can apply a different level of indirection.

It was suggested to compile the compiler using itself and a different compiler, generating two executables. These two executables can be assumed to behave under test because they came from the same source code. However, they will not be identified as equal because the two different compiler manufacturers have used different techniques to generate the compiled executables. Now, one can use both binaries to recompile the compiler source, generating two outputs. Since the binaries should be identical, the outputs should be bit-by-bit identical. Any difference indicates either a critical defect in the procedure or malware in at least one of the original compilers [64].

In his first report about it, Wheeler put the idea into practice and coined the DDC technique in 2005 [69]. Wheeler then elaborated more on DDC in his 2010 Ph.D. dissertation [70]. The committers of GCC have been using very similar techniques for some time, although they are not using the term DDC for their actions [11]. The compilation procedure of GCC was not as described in its documentation. Wheeler's thesis mentions how the GCC compiles itself: It is a three-stage bootstrap procedure. The committers of GCC have been using DDC for some time, although they are not using the term DDC for their actions. The GCC compiler documentation explains that its normal complete build process, called a bootstrap, can be broken into stages. The command ”make bootstrap” is supposed to build GCC three times: When the system compiles GCC, it follows a procedure in three stages. First, a C compiler, which might be an older GCC or a different compiler, compiles the new GCC. This task is called ”stage 1”. Next, GCC is built again by the ”stage 1” compiler it previously compiled to produce ”stage 2”.

Finally, the ”stage 2” compiler compiles GCC a second time. The result is called stage 3 [66]. The final stages should produce the same two outputs (besides minor differences in the object files' timestamps), which are checked with the command ”make compare.” If the two outputs are not identical, the build system and engineers should report a failure. The idea is that a build system with this kind of checking should be made the final compiler independent of the initial compiler. Every build of GCC gets checked [11]. If a particular instance of GCC had included some malware of the trusted type, the check is done by the three-stage bootstrap, starting with a different compiler. The proof is that there is either no malware or that the other compiler has the malware. The more compilers included, the stronger the result will be: Either there is no trust attack or every C compiler we tried contained the malware. Both SUN's proprietary compiler and GCC have compiled GCC on Solaris. David A. Wheeler claims that this verifies that either GCC is legitimate or SUN's proprietary compiler contains malware, where the latter event is considered unlikely.

David A. Wheeler then states that one consequence of the vulnerability and countermeasures is that organizations and governments may insist on using standardized languages instead of customized languages. The U.S. military effectively did this with the standardized software development of the Ada programming language and standardized hardware development with the VHDL language. For standardized languages, there are many compilers. This diversity of compilers will reduce the probability of compromised and subverted build systems. If only one compiler exists for a specific programming language, it will not be possible to perform the DDC. Wheeler described three parts central to the attack: triggers, payloads, and non-discovery. A trigger means a specific condition inserts the enemy code (the payload) [70].

The test that is performed can be described in the following condition. If the condition holds, then there can only be an attack hidden in binary A if binary B cooperates with A. So either the compiler isn't compromised, or both of them are.

Listing 2.1: Example of condition for DDC

```
/* compile_with(x,y) means that we compile y with compiler x,*/

if (compile_with(compile_with(A,source),(source))
== compile_with(compile_with(B,source),(source)))
```

Furthermore, Wheeler suggested several possible future potential improvements, such as more extensive and diverse systems under test and relaxing the requirements of DDC, detailed later in section 2.3.2. (LINK TO SECTION!)

Several sources, including Wheeler's dissertation, explain that the security of a computer system is not a yes-or-no hypothesis but rather a matter of extent. Even if we could completely solve the compiler and software backdoor problems, the same vulnerability and argument apply to the linker, the loader, the operating system, firmware, UEFI, and even the computer system hardware and the microprocessor.

2.4.3 Reproducible Builds

Reproducible builds, which have been part of the Debian Linux project, have been a relatively successful attempt to guarantee as much as possible that the generated executable code is a legitimate representation of the source code and vice versa. All non-determinism, such as randomness and build-time timestamps, must be removed to achieve reproducible builds. Alternatively, the non-determinism turns deterministic. The objective is that the builds give identical results for every build for the same version [38]. A simple checksum checks that a build is a legitimate version. The authors note, however, that there is still no apparent consensus on which checksum should be considered the right one for any specific build. Finally, trusting the compiler itself has become a catch. Therefore, an instance of GCC is bootstrapped from a minimal (6 kilobyte) TCC with a minimal amount of trusted code.

Linderud examined software production systems with independent and distributed builds to increase the probability of a secure output from the build. He suggests metadata to make it easier to see whether the integrity of the build is compromised [39]. The methods described in that thesis are about the validation of software integrity. The methods include signed code and Merkle trees to validate the downloaded packages against available metadata [43]. The method would have the same vulnerability as any other reliance on an external supplier to protect against third-party tampering with the system. However, it will not protect against any attack from a previous version of the production system. In the case of a trust attack, it is technically possible. Supply-chain attacks were even proven to exist in analog hardware [74].

2.4.4 Fuzz Testing

In his writing about DDC, Wheeler claims randomized testing or fuzz testing would unlikely detect a compiler Trojan [70]. Fuzz testing or randomized testing tries to find software defects by creating many random test programs (compared to numerous monkeys at the keyboard). When testing a compiler, the compiler-under-test gets compared with a reference compiler. The test outcome will depend on whether the two compilers produced different binaries. Faigon describes this approach [25]. The approach has found many software bugs and compiler errors, but it is improbable to detect maliciously corrupted compilers. Suppose such a corrupted compiler diverts from its specifications in only 1/1000 executions of its target (as would be the case in a cryptocurrency system that sends every thousand transactions to a different receiver). In that case, it becomes evident that tests are unlikely to detect the bug in the compiler. For randomized testing to work on compiler-compilers, the input would need to be a randomized new version of the compiler, which no one has attempted. The situation will be that a compiler gets compiled, and the Trojan horse only targets particular input, such as the compiler itself. However, Wheeler's analysis does not explain why we cannot input the compiler's source code into a compiler binary and then do fuzz testing with variations.

2.5 Secure Development for Cryptocurrency

Due to the financial capabilities of the Bitcoin Core project, there has been a general interest in security issues with its design and implementation. Many questions and issues centered around Bitcoin Wallets and their potential breaches and thefts. Theft of a Bitcoin Wallet is not an issue with Bitcoin itself. However, it is because of a bug in a web framework for the Cryptocurrency Exchange. Research and information are scarce about how Bitcoin Core has applied secure development or application security in the past. The first Bitcoin white paper aimed at solving the problem of double-spending [44]. It did not specifically address the security of its implementation and did not address a supply-chain attack.

The reason for trusting the blockchain of Bitcoin is primarily due to the integrity of verifying that anybody can verify the entire blockchain from the hash value of the first block. The first block's value is a constant in the source code of Bitcoin Core. Developers and users trust that it is secure. Nevertheless, compromising a digital currency's security is an activity that many individuals and organizations would like to do. It is not just the single individual "malicious hacker" who tries to rob the digital bank or steal someone's digital wallet but also large MNCs and governments that have an interest in compromising the cryptocurrency and manipulating the blockchain.

Chipolina describes in a recent article several techniques and social engineering practices that could make it possible to manipulate a cryptocurrency and compromise its production line and security [12]. One of the scenarios described is the potential risk of having the supply chain compromised if one or more maintainers or developers get their personal or physical security compromised. The development relies on properly using PGP keys, which can get stolen or handled insecurely by mistake.

In a recent news article, Sharma writes that supply-chain attacks have recently occurred against the blockchain [60]. The software supply chain attacked a DeFi platform for cryptocurrency assets. A committer to the codebase had included a vulnerability in the platform's production version. It raised questions about quality assurance for source code contributions and whether the review process was the real problem in this case.

Rosic discusses several different hypothetical scenarios to compromise cryptocurrency and blockchain [5]. Most of the scenarios described are issues and problems with collaboration between Bitcoin users and miners. None of the scenarios described involve binary Trojan horse backdoors or a compromise of the production system.

In 2022, researchers Choi et al. submitted their study of several cryptocurrencies, including specific findings of many security vulnerabilities [13]. Their findings include many duplicated vulnerabilities across different projects, seemingly due to the majority of the cryptocurrencies appearing to have been copies of Bitcoin to begin with and therefore included the same vulnerabilities. They also noted that security vulnerabilities generally take long before somebody mitigates them.

At first sight, there is no clear policy available and no mechanism in practice for secure development and testing and verification of the security, including the dependencies and the build system. In general, any software that includes third-party dependencies must be checked and tracked so that a dependency does not contain a vulnerability or an exploit, and the same reasoning about the build system. There is also an apparent lack of CVE information for Bitcoin and Blockchain projects. The authors of Reproducible Builds mention in the article that the early development of Bitcoin Core was in a "jail." [38] The article's authors most likely referred to a system called Gitian that checked the integrity of Bitcoin builds [72]. Gitian creates this control by doing this deterministic build inside a specific VM, which feeds the instructions through a declaration in YAML. Bitcoin Core has since then changed its build system to GUIX [28].

The authors, Groce et al., published their research about the effectiveness of fuzz testing for Bitcoin Core fuzzing [31]. They examine to what extent it has been possible to conduct fuzz testing to find bugs in the software of Bitcoin Core. A common problem with fuzzing is that the fuzzing becomes saturated, the project under test soon becomes resistant, and further fuzzing finds almost nothing, although having been successful in the beginning. The authors conclude that there is a possibility to utilize fuzz testing for the Bitcoin Core project and that there is room for further improvement.

For simplicity, it is preferable to conduct academic research and tests with minimal software distribution, at least in the beginning. Otherwise, there is a risk that the duration of the build process and other unnecessary complications slow down the pace of the work. For example, the build duration of large projects written in C++ is often relatively long. So instead of the Bitcoin Core written in C++, there is another implementation of Bitcoin called Mako written in ANSI C with fewer external dependencies [32]. Compiling the project into a Bitcoin binary makes it conceptually easier to prove that it is free from malware, as would be the case in a cryptocurrency system that sends one out of every thousand transactions to a different receiver. For randomized testing to work on compiler-compilers, the input must be a randomized new version of the compiler. A recommendation is to keep multiple implementations of a protocol as good practice. In the case of BTC, they are necessary to mitigate the harm of developer centralization.

2.6 Summary

Looking at the related work in summary, it is worth noting that despite extensive research, there is relatively little about the supply-chain security of cryptocurrency in general. While diverse double-compiling has been studied and used in rather great efforts, relatively little or nothing puts DDC in the context of cryptocurrency and Bitcoin.

Chapter 3

Method

This chapter describes the methods known to prove a trust-based attack's feasibility. It provides a thorough description of how to create and execute a self-reproducing attack against an American National Standards Institute (ANSI) C compiler. The trust attack references a complete implementation, unique in its capabilities. The subsequent sections explain why specific methods were chosen and not others.

The chapter further describes how specific novel approaches can help detect and identify a trust-based attack. It also discusses how to prevent attacks from compromising a production system in the first place. The chapter describes the methods for validation, proof based on the theoretical study, and based on related work, for example, how repeated runs are possible. The figures in this chapter visualize a trust-based attack for demonstration purposes. The methods are applied to Bitcoin Core in a case study.

3.1 Qualitative Investigation with Interviews

One activity has been to conduct an interview with David A. Wheeler. There was also discussion about the subject and about the techniques with committers to the well-known and popular compilers GCC and TCC. to keep a narrow and well-defined scope, some technologies were considered but finally not chosen, being deemed out of scope (Compcert, Formal Methods, Symbolic Logic, Complexity Theory, AI methods).

3.1.1 Interview Protocol from Interview with David A. Wheeler

The following questions and answers were discussed with Dr David A. Wheeler, Director of Open Source Supply Chain Security at the Linux Foundation. Wheeler is presumably the most knowledgeable person about diverse double-compiling. The interview protocol has been divided into sections, and the answers from Wheeler are quoted verbatim.

Learning the background and techniques

Question: What is the relevant knowledge and how can we learn the topic? What kind of background would help a programmer to learn and work with compiler security and build security? Will it help the most to have a background as a C programmer, an expert in compiler technology, a security engineer, a cryptanalyst...?

Answer: You should know about compilers (take a class!), computer security, and build systems. You should definitely know how to program in at least one programming language; it doesn't need to be C. Once you learn a few programming languages, learning more is easy; I know over 100 (I counted).

There's no need to be a cryptanalyst. That's a very specialized field & not really relevant for this work. You need to know how to create cryptographic hashes and what decent algorithms are, but that basically boils down to ”run a tool to create an SHA-256 hash”.

Question: Would it be alright to study the C programming language and C compilers to be prepared? There have been reports that most security breaches exploit some bug in some source code written in C, is that still the case and a reason to concentrate on C? If not, what are the other options?

Answer: To study the area, you don't need to learn C, though C is still a good language. Many compilers are written in C, and a vast amount of low-level code is in C.

C is not a ”big” programming language. However, it has few ”guard rails” - almost any mistake becomes a serious bug & often a security vulnerability. Unlike almost all other languages, C & C++ are memory-unsafe, that is, they provide no memory safety protection. That has resulted in a lot of vulnerabilities:

70% of Chrome vulnerabilities are memory safety issues; https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

70% of Microsoft
vulnerabilities are memory safety issues: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Almost any other language is memory-safe & resolves this problem. If performance is irrelevant, you can use other languages like Python, Ruby, etc. If you want decent performance, you can use decently-performing languages like Java, C#, and Go. The big challenge is if performance is critical. Historically Ada & Fortran were your closest realistic options for performance, but Rust has dramatically risen recently. You should certainly check out Rust at least.

Practical usage

Question: What are the most ambitious uses of DDC you are aware of?

Answer: That would be various efforts to rebuild & check GNU Mes. A summary, though
a little old, is here: https://reproducible-builds.org/news/2019/12/21/reproducible-bootstrap-of-mes-c-compiler/

Question: Are there some public papers or posts you can recommend to read and refer to for my thesis?

Answer: Well, my paper :-). For the problem of supply chain attacks, at least against open source software (OSS), check out: Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier https://arxiv.org/abs/2005.09535

Website: https://reproducible-builds.org

It's mentioned that the technique of the attack is not limited to the compiler, it could also be part of the operating system or hardware itself. What would be en example?

I focused on the "self-perpetuation" part & discussed that in my paper to some extent. E.g., if the OS detected recompilation of the compiler or itself, it could insert the malicious result instead.

Alternative approaches

Question: What other types of mitigations have there been apart from DDC?

Answer: Main one: bootstrappable builds http://bootstrappable.org/ There, the idea is that you start from something small you trust & then go.

But they actually work well together. How do you trust the small thing? How can you test its results? One approach is... DDC :-). So they are really complementary.

Another approach is proofs at the machine-code level of compilers. There's been some ACL2-related work on that. But proofs are hard anyway; doing them at that level is even harder.

What is the critique or drawbacks?

Question: What has been the most valid critique against DDC?

Answer: "That's not the primary problem today."

I actually agree with this critique. The primary problem is software filled with vulnerabilities. This has many causes: most software developers don't learn how to write secure software (it's not taught), their tools don't protect them by default (C/C++ in particular), & they don't have tools in their CI pipeline to check for things. Software, even when fixed, is often not updated in a timely way in production.

Once those are resolved, simple supply chain attacks come to the fore, especially dependency confusion, typo squatting, and insertion of malicious code into source code. None of them are trusting trust attacks.

But that's okay. Academic research is supposed to expand our knowledge for the longer term. Once those other problems are better resolved, trusting trust attacks become more likely. I think they would have been more likely sooner if there was no known defense. Even if people don't apply DDC, the fact that DDC exists reduces the incentives for an attacker to use a trusting trust attack (because they can now be detected after the fact). I look forward to a time when DDC is increasingly important to apply because we've made progress on the other problems.

3.1.2 Discussion with Committers to GCC

During discussion with committers to GCC, a number of research reports and techniques were suggested. Many of them were in-scope and included), while some techniques were deemed out-of-scope (CompCert) for this particular project. It would have been possible to interview with more C expert and compiler expert, but the particular expertise were not exactly in scope.

3.2 Feasibility of the Attack

The attack scenario is possible when an owner has agreed to install a new compiler or system. The software engineer will compile and install that compiler. A Trojan horse in the compiler-compiler can target and get triggered by a particular input, such as the compiler itself. It is helpful to describe with illustrations the procedure to compile the compiler.

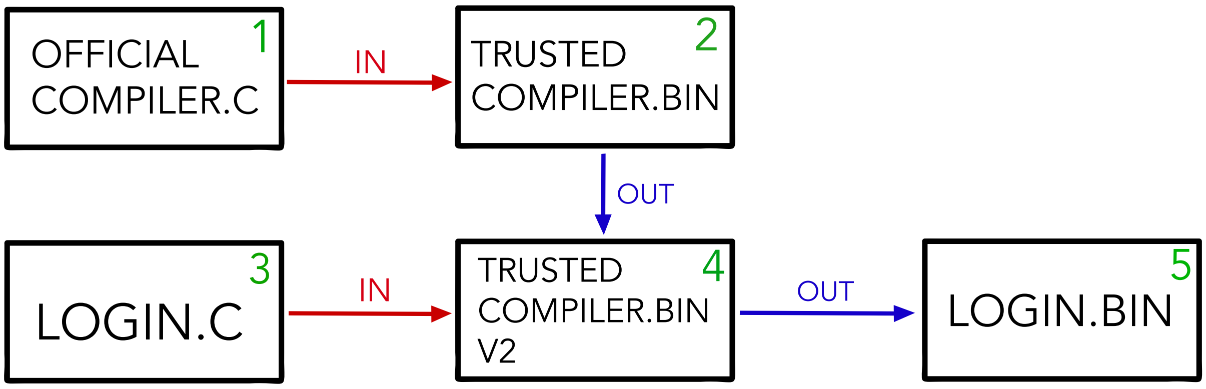

Figure 3.2.1: Compilation of a compiler that compiles the login program

Several implementations already demonstrated the feasibility of the attack. For example, Ken Thompson did release a malware compiler internally at Bell Labs. The compiler was accepted and used even though it caused miscompilation. The attack implementation in this chapter confirms that the attack and the defense are possible in practice.

3.3 Creating a Real Example Attack

This section describes the implementation of the attack. The scenario can happen when an owner has agreed to install a new compiler or even upgrade a current system. The software engineer will compile and install that compiler. A Trojan horse in the ancestor compiler can target and get triggered by a particular input, such as the compiler itself. This chapter describes, with illustrations where appropriate, the method of attack.

The attempts to create a successful attack in this study were initially flawed. The flaw made the attack unrealistic at first: The malware would replicate into all future versions of the compiler, but only when explicitly given a new header file with future versions of the compiler. The header file will not be available in the live environment, so the attack was not realistic initially. The efforts resulted in a reproducible complete example attack in an ANSI C compiler is made available online at the time of this writing. (https://github.com/montao/tinycc/blob/ddc/.github/workflows/action.yml).

Listing 3.1: Example in C how to pull in arbitrary code anywhere

```
...
/* open the file */
#include "attack.c"
fd = _tcc_open(s1, filename);
...
```

For example, suppose that a perpetrator tries to manipulate the TinyC to insert the attack code into future versions of itself. In that case, the perpetrator can put an #include statement right before it opens the file to compile.

The file attack.c contains malware that will include malware in itself for all future versions. The attack code detects the dummy login program and lets the username hackerken log in without a password. The attack code is replicated in all future versions of the compiler when a self-hosting compiler compiles itself, given that the preprocessor includes the attack in future versions. How is the attack supposed to take place without providing the header file? It seems that there is a circular dependency in the attack code.

Listing 3.2: C example

```
#include "attackvector.c"


// use naive signatures for now
// needs to ./generate-attackvector < attack.c > attack-array.h
static char compile_sig[] = "/* open the file */";
static char login_sig[] = "int login(char *user) {";
static char login_attack[] = "if(strcmp(user, \"hackerken\") == 0) "
                             "return 1;";

 /* attack code follows */
 ...
```

However, that makes the backdoor require that the programmer compile all future versions of the compiler, including the header file attack-array.h making the backdoor dependent on the header file present, which will not be from a direct clone of the compiler.

The improvement of not needing the extra header file was to compile the .c file twice and paste it into the char array, and like that, it will self-reproduce any (finite) number of generations without being seen in the source code. The code pipeline for Bitcoin Core helped run and check new builds quickly.

During the work, new and established techniques of diverse double-compiling proved their feasibility in the context of the build system for Bitcoin Core. Even a program such as nm can check the compiler-generated symbol table, and a program such as objdump can check object code; the next time the preprocessor pulls in header files, they can still cause miscompilation.

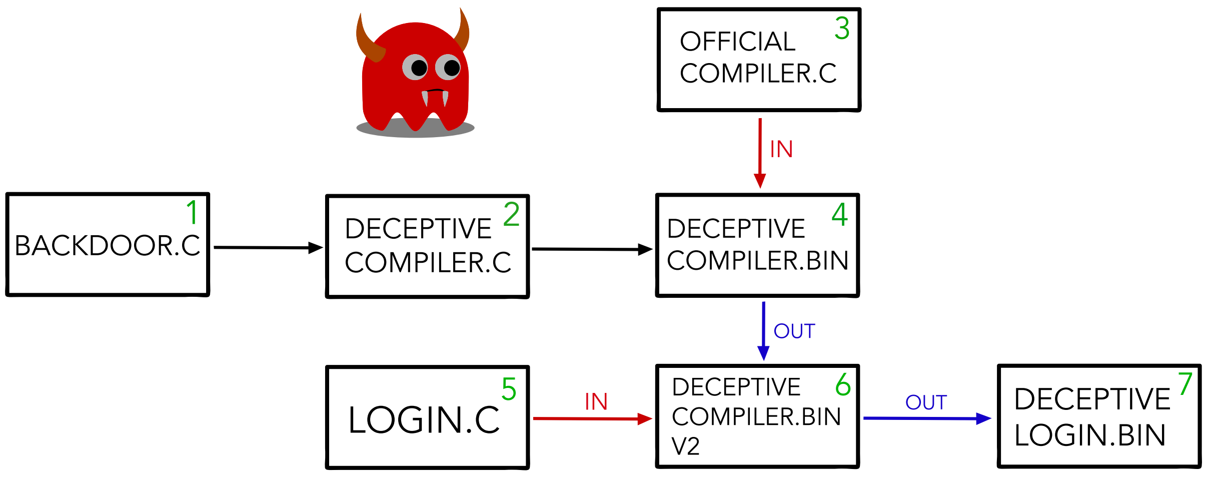

Figure 3.3.1: How to perform a trust attack

Listing 3.3: C example

```
...
/* open the file */
#include "attack.c"
fd = _tcc_open(s1, filename);
...
```

The file attack–array.c is generated from running generate-attack-array

Listing 3.4: C example

```
#include <stdio.h>

int main(void) {

  printf("static char compile_attack[] = {\n");
  int c;
  while ((c = fgetc(stdin)) != EOF) {
    printf("\t%d,\n", c);
  }
  printf("\t0\n};\n\n");
  return 0;
}
```

The result will make the attack–array.h contain the array with the attack.c source code.

Listing 3.5: C example

```
static char compile_attack[] = {
47,
47,
32,
....
```

3.4 Executing an Example Attack

This section describes the implementation and execution of the attack. The attack vector of the compromised compiler will appear in the binary executable code. Consequently, any malicious embedded code injected into the login program or the compiler will be possible to find. As a last resort, it is reasonable to check the CPU's machine instructions, but it is preferred to perform the analysis before execution if that is feasible.

Technically, the malware could have been found by looking at the machine instructions that the CPU executes. The malware appears in the executable object, even though it is difficult to read or see. Nevertheless, a programmer cannot prevent the trust attack by writing everything in assembly code. It is doable to create a similar miscompilation from the assembler that translates the assembly code to machine code.

Defeating a Code Review

Reviewers will not see an embedded self-replicating Trojan malware during code review, given that the attacker has embedded it into the compiler's executable. The malware will only appear in the executable (binary) object(s) in a form that is neither visible nor readable for the majority of reviewers.

3.5 The Defense

This section describes the implementation of the defense. It will explain the details of producing a general system based on DDC. One can observe systems and functionality of programs in repeated runs to exclude the probability of sheer luck.

3.5.1 Theoretical Background

In general, comparing source code and compiled code for equivalence is an undecidable problem. If it had been decidable, it would have solved the Halting problem. However, two bit-sequences or files are fundamentally equivalent if the programmer checks two bit-for-bit executable identical files. The minor complication is that we have several compilers and programming languages.

After many years, the idea of countermeasures against the Trojan horse using diverse double-compilation emerged. Other methods, such as verification techniques, reduction of the attack surface, application security, and secure development methods, can be helpful. Nevertheless, nobody can verify every piece of software and hardware used to create a design. Such a task is intractable, so we have to make do with heuristics and approximations. In the context of Bitcoin, there are methods to increase the system's security. Owners of systems can limit their exposure to the public internet. Typically, security engineers referred to this activity as reducing the attack surface.

The first method is to use several compilers to compile the source code. The technique is effective against attacks where the attacker has control of a single compiler. The idea is that if the attacker can only control one compiler, they can only insert malicious code into one of the compiled binaries. This method is ineffective against attacks where the attacker has control of several compilers.

Part of the work with this project has been to investigate and understand both the vulnerabilities and the possibilities of a defense. Changing the compilation pipeline of existing software without reviewers being able to see it might sound complicated, but somebody can do it. The study examines the design of such attacks by implementing and evaluating a technique called diverse double compilation for Bitcoin Core. It suggests a few novel ideas to reduce the assumptions and the probability even more that a system is compromised.

A hardened cybersecurity can be accomplished in two complementary ways: secure software development, a NIST initiative, and software vulnerability protection. There are methods to reduce the attack surface of a computer system in general. Trojan Horse, man-in-the-middle (MiTM) attacks, and denial of service (DoS) are examples of common exploits. These ways of attacking and exploiting systems and networks to gain a profit for the attacker. Structured Query Language (SQL) injection and buffer overflow are examples of attacks that can result in controlled systems injecting a backdoor. Such a backdoor in a login system can take the form of a hard-coded user and password combination that gives the system access. The compiler may have inserted this backdoor into the login program, and the ancestor compiler could have subverted the next version of the compiler. The project includes the implementation of a proof-of-concept attack targeting a C compiler, showing that techniques can detect and mitigate the attack. There is such a problem with deceptive compilers introducing malware into the build system, as described in several articles and studied in computer security and compiler security. Several additional sub-problems were the reason for analysis and discovery in this attack class. For example, there are no readily available working examples of the attack. Different methods to reduce the probability of such an attack are also not readily available. One mitigation is to use diverse double-compilation as a countermeasure. Here is an introduction to a variant of DDC that is beneficial. With two compilers twice, the second time switching the roles of the compilers to reduce the probability of undetected compiler vulnerabilities. This variety is novel. In the context of the Bitcoin Core build system, an attack and compromise are possible by a few attack methods and exploits that can be tried and analyzed, for example. Given the probability of a successful attack, the methods here reduce the probability of the attack going undetected.

It is conceivable that somebody can conduct a domain-specific attack where the compiler inserts code that changes the recipient address in 1/10000 Bitcoin transactions. It is also feasible that a malevolent maintainer could upload malicious code and hide it. Such an activity would get caught by the verify signatures script, but could also go unnoticed.

Software engineers must keep dependencies and sources updated and verified. The same should be applied to compilers to reduce risk and make mitigation feasible. The attack vector of the compromised compiler will appear in the binary executable code. Consequently, any malicious embedded code injected into the login program or the compiler will be possible to find. As a last resort, one can check the machine instructions that the CPU executes, but it is preferred to perform the analysis before execution if that is feasible. Some can use compromised compilers to insert malicious code into binaries. The binaries can then exploit systems. This vulnerability is critical as it allows attackers to access systems and data. It is, therefore, essential to ensure that compilers are kept up to date and depend only on verified sources. Observation of procedures and functionality of programs in repeated runs can exclude the probability of sheer luck and provide proof of the feasibility of the trust attack.

This chapter provides illustrations for the attack and defense for an audience who does not need prior knowledge of formal methods or symbolic logic. This chapter provides an attack visualization to visualize the attack according to the literature study and including some formal theoretical background. Assuming that we have written the source code of a compiler C for a new system. The compiler is written in the language A with the capability to compile the language A. For the sake of argument, the compiler is presumably bug-free. Now there is a need to compile the source code.

Various organizations have, during recent years, suffered over $40 MUSD in losses due to compromised security in cryptocurrency exchanges and transactions. In the future, it is likely to expect these types of attacks to target vulnerabilities in decentralized cryptocurrency blockchains. To attack and compromise the Bitcoin Core, one might consider a few attack methods and exploits that can be tried and analyzed, for example: Somebody is stealing a Pretty Good Privacy (PGP) key from a maintainer and using that to sign new versions of Bitcoin-Core. The belief that a computer system is 100 % secure can only result from verifying all software and hardware that designers and hardware constructors used to create the system.

A domain-specific attack is a compiler inserting code that changes the recipient's address in 1/10000 Bitcoin transactions. A malevolent maintainer could upload malicious code and then hide it. Such an activity would get caught by a script that verifies signatures commonly catches such an attempt, but may fail to detect the malware in some instances, or even produce false positives.

Packages used in programming languages have recently been high-profile targets for supply chain attacks. A popular software package from the Ruby programming language used by web developers was compromised in 2019 when it started to include a snippet of code allowing remote code execution on the developer's machine. The author's credentials were compromised, and nobody ever found the malicious code in the code repository.

Ethical Considerations

The ethical grounds are to show openness (open research) and transparency in research, even during the research work and after its completion. It is the moral foundation of white hat hacking. It is not the case that somebody should not trust self-reproducing programs. While a trust attack hides in programs that produce programs, it does not say anything about this, making them more or less trustworthy. There is nothing language-specific about this attack. Somebody could have hidden a Trojan horse in a compiler for a more modern language like Java.

Thompson experimented with putting malware into the compiler. The malware could have been found by looking at the machine instructions that the CPU executes. The malware appears in the executable object, even though it is difficult to read or see. Nevertheless, a programmer cannot prevent the trust attack by writing everything in assembly code. It is doable to create a similar miscompilation from the assembler that translates the assembly code to machine code.

3.5.2 Countering Trust Attacks with DDC

With diverse double-compiling, one compiles the compiler's source code C_0 with another, verified compiler C_1. Then use the result B1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler. The central point is that DDC enables us to accumulate and strengthen the evidence. If needed, diverse double-compiling can run ten times with ten different verified compilers. In this scenario, an attacker would have to subvert all the verified compilers and the original compiler-under-test executable to avoid detection, which is unlikely.

As described by David A. Wheeler, with DDC, one compiles the source code of the compiler C_0 with a verified compiler C_1. Then use the result B_1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler. The central point is that DDC allows the tester to accumulate evidence. If the testers choose, they can use DDC 10 times with ten different verified compilers. Then an attacker would have to subvert all the verified compilers to make the compiler-under-test executable to avoid detection, which is highly unlikely to be achieved. The illustration describes the procedure. The central question is whether the executable of (7) represents the source code of (1), and nothing else.

A key idea here is that (3) and (5) should necessarily produce the same output for the same input. Even though different compilers may have made them, the two executables are supposed to behave the same or be buggy or Trojaned. It is essential to understand that the program in box 4 (LOGIN.C) could have been a compiler, so there are three compilers in the actual test: First, we may assume that the compiler in box 4 is entirely perfect and bug-free.

Now the programmers need to compile the source code. They receive two executable compilers that can compile the programming language A, a, and b. They installed the compilers in the new system but warned that one might be buggy or contain a Trojan horse. Having a Trojan horse can be the only potential problem in a or b. Considering the programming language A, the three compilers implement A correctly and entirely according to specification.

Now the programmers compile C using both a and b to produce executables c_a and c_b respectively. You then compile C using c_a to produce cc_a, and compile C using c_b to produce cc_b. Finally, you compile C using cc_a to produce ccc_a, and compile C using cc_b to produce ccc_b. Now somebody can compare the generated executable objects at various stages of the compilation.

It is not the case that, given that c_a and c_b differ, one of a and b is necessarily buggy or Trojaned. Because even if both a and b are bug-free, they will likely compile the same input code to different output machine instructions. However, given that cc_a and cc_b differ, one of a and b is necessarily buggy or Trojaned. After being generated correctly, C guarantees to produce the same output for the same input every time, deterministically. If there are two correct compilations of C's source, these two will be indistinguishably equal bit-by-bit.

If ccc_a and ccc_b differ, then one of a and b is necessarily buggy or Trojaned. And, if cc_a and ccc_a differ, then a is necessarily buggy or Trojaned. It is not the case, though, that if b contains a Trojan horse, then cc_b and ccc_b will necessarily differ. The attacker could have decided not to activate the Trojan backdoor for this particular case.

It is feasible to apply the methods in the Bitcoin Core context. Both the attack and the defense are possible. With DDC, one compiles the compiler's source code C_0 with a verified compiler C_1. Then use the result B1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler. The central point is that DDC enables us to accumulate and strengthen the evidence. Some can use DDC 10 times with ten different verified compilers; an attacker would have to subvert all the verified compilers and the original compiler-under-test executable to avoid detection, which is unlikely.

A key idea here is that (3) and (5) should necessarily produce the same output for the same input. Even though different compilers may have made them, the two executables are supposed to behave the same or be buggy or Trojaned. It is essential to understand that the program in box 4 (LOGIN.C) could have been a compiler, so there are three compilers in the actual test: First, we may assume that the compiler in box 4 is entirely perfect and bug-free. Now you need to compile your source. You're given two executable A compilers, a, and b, on the new system but warned that one might be

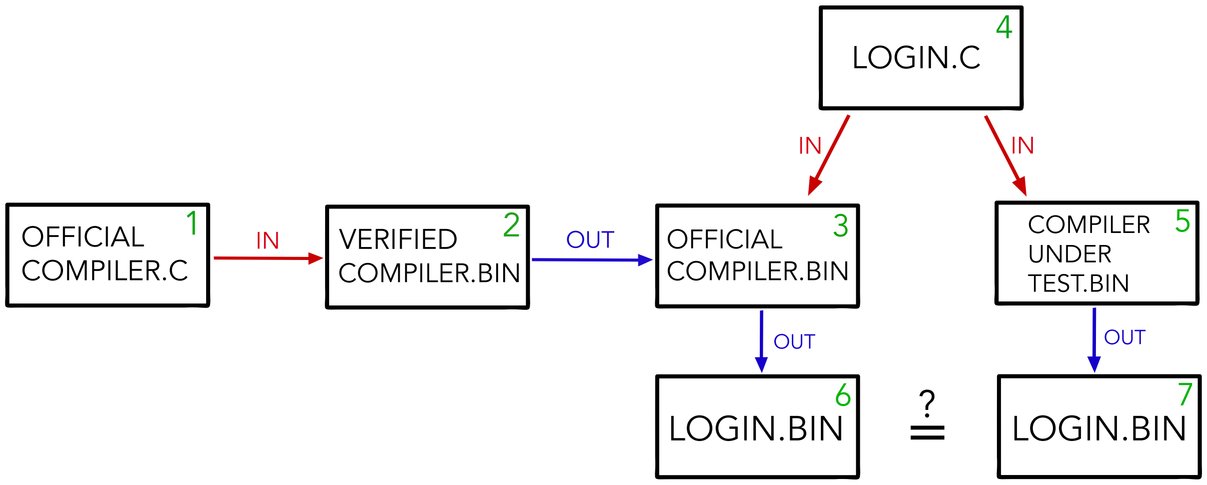

Figure 3.5.1: Does the executable of (7) represent the source code of (1), or has the login program been Trojaned? How to detect a trust attack with diverse double-compiling.

buggy or contain a Trojan horse. Except for this potential problem in a or b, the three compilers all implement A according to its specification, which is complete and exact. You compile C using both a and b to produce executables c_a and c_b respectively. You then compile C using c_a to produce cc_a, and compile C using c_b to produce cc_b. Finally, you compile C using cc_a to produce ccc_a, and compile C using cc_b to produce ccc_b. Now we compare the binaries produced at various stages of the compilation.

It is not the case that, given that c_a and c_b differ, one of a and b is necessarily buggy or Trojaned. Because even if a and b are bug-free, they will likely compile the same input code to different output machine instructions.

The conclusion is that given the difference of cc_a and cc_b, one of a and b is necessarily buggy or Trojaned. C should always produce the same output for a given input. Any two correct compilations of C's source should behave the same.

If ccc_a and ccc_b differ, then one of a and b is necessarily buggy or Trojaned. And, if cc_a and ccc_a differ, then a is necessarily buggy or Trojaned. It is not the case, though, that if b contains a Trojan horse, then cc_b and ccc_b will necessarily differ. The attacker could have decided not to activate the Trojan backdoor for this particular case.

3.6 Summary

One can apply the methods above to Bitcoin Core. Both the attack and the defense. With DDC, one compiles the compiler's source code C_0 with a verified compiler C_1.

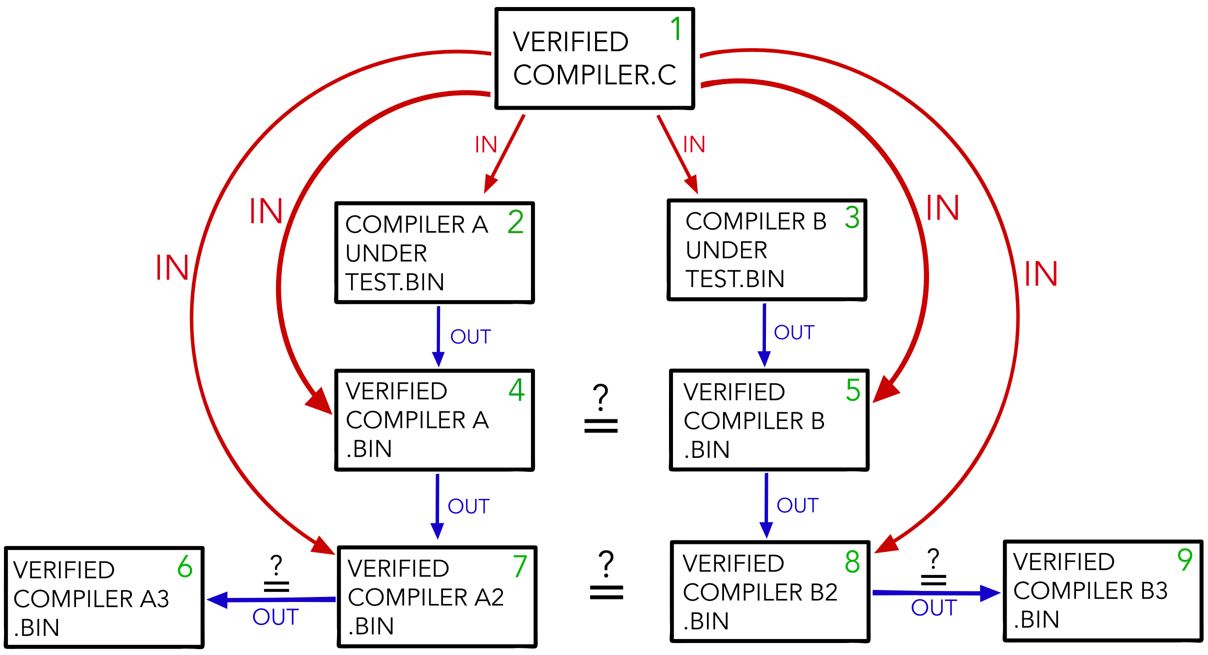
VERIFIED COMPILER.C 1
COMPILER A UNDER TEST.BIN 2
COMPILER B UNDER TEST.BIN 3
VERIFIED COMPILER A .BIN 4
VERIFIED COMPILER B .BIN 5
VERIFIED COMPILER A3 .BIN 6
VERIFIED COMPILER A2 .BIN 7
VERIFIED COMPILER B2 .BIN 8
VERIFIED COMPILER B3 .BIN 9
IN
OUT
?

Figure 3.5.2: Do the executables of (4), (5), (6), (7), (8), and (9) represent the source code of (1)? Note that (1), (3), and (5) are supposed to be the same compiler. What conclusions can somebody draw from comparing the results at the different stages?

Then use the result B1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler. The central point is that DDC enables us to accumulate* evidence *. You can use DDC 10 times with ten different verified compilers;. An attacker would have to subvert all the verified compilers and the original compiler-under-test executable to avoid detection, which is unlikely.

Chapter 4

The work

I have successfully performed diverse double-compilation using TCC and GCC implementations and checked that the generated object code is free of trust attacks. I have also performed diverse double-compilation on a compromised version of TCC, which contained a trust attack, and found that it was detected during diverse double-compiling. In the C programming language, it has been possible to directly access the locations in random access memory (RAM) where the machine byte codes of a function are stored. One possible trick is manipulating what's in RAM in C to step around in the available memory. An attack can be made by changing what's in RAM, and the address space layout randomization (ASLR) does little or nothing to prevent or detect it because a process can retrieve the address of its variables even if the address is randomized and different every time.

The injector of injector of malware.

There have been many claims about the Thompson compiler backdoor (https://stackoverflow.com/questions/781718/thompsons-trojan-compiler/782677) without anyone, or almost nobody has shown that the attack can be made in practice.

I put a complete reproducible example with TCC on GitHub https://github.com/montao/tinycc/blob/ddc/.github/workflows/action.yml. Still, it had a flaw, to begin with, which did make the attack unrealistic at first: I could make the Trojan replication to all future versions of the compiler, but that is only possible if a different header file with the attack vector is available when future versions of the compiler are compiled. The header will not be available, so my attack would not be realistic.

For example, if I try to corrupt the TinyC so that it inserts the attack code to future versions, I can put an include statement right before it opens the file to compile:

Listing 4.1: C example

```
 ...
/* open the file */
#include "attack.c"
fd = _tcc_open(s1, filename);
...
```

The file attack.c contains the attack code and will be included by the preprocessor. The attack code detects my dummy login program and lets the username hackerken login without a password. The attack code is replicated to all future versions of the compiler when TCC compiles itself, but only if the attackvector.h is supplied to future generations. How is the attack supposed to take place without providing the header file? It seems that I got a circular dependency in my attack code.

Listing 4.2: C example

```
#include "attackvector.c"

// use naive signatures for now
// needs to ./generate-attackvector < attack.c > attackvector.h
static char compile_sig[] = "/* open the file */";
static char login_sig[] = "int login(char *user) {";
static char login_attack[] = "if(strcmp(user, \"hackerken\") == 0) retu

/* attack code follows */
...
```

The file attackvector.c is generated from running generate-attack-array

Listing 4.3: C example

```
#include <stdio.h>
```

```
int main(void) {

  printf("static char compile_attack[] = {\n");
  int c;
  while ((c = fgetc(stdin)) != EOF) {
    printf("\t%d,\n", c);
  }
  printf("\t0\n};\n\n");
  return 0;
}
```

...so that the attackvector.h contains the array with the attack.c source code itself.

Listing 4.4: C example

```
  static char compile_attack[] = {
  47,
  47,
  32,
  ....
```

, But that makes the backdoor require that all future versions of TCC are compiled with the header file attack-array.h making the backdoor dependent on the header file present, which won't be from a direct clone of the compiler.

The way I fixed it not to need the extra header file was to compile the .c file twice and paste it into the char array, and like that, it will self-reproduce any (finite) number of generations without being seen in the source code.

I used a code pipeline or created a code pipeline myself for Bitcoin Core to examine the builds more quickly.

During my research of the topic, I studied, created, and evaluated some new and established techniques of diverse double-compiling in the context of the build system for Bitcoin Core [44].

Modern OSs randomize memory section addresses (it makes some attacks more difficult), so if anyone restarts the process, the addresses of the instructions might be different.

Check compiler generated symbol table with nm, check code with objdump Header files that are pulled in... can be compromised

The build process of Bitcoin Core is my main applied DDC. I will find out about relevant frameworks, build systems and platforms, such as GUIX (https://guix.gnu.org), maybe Frama-C (https://frama-c.com), and/or the CompCert project (https://compcert.org/).

Checks and provisioning can be done to a system in a Docker container for portability and reproducibility. I have access to the Bitcoin build toolchain and can build Bitcoin Core with GCC. I can create the demonstration of a compiler Trojan horse in the source code of a compiler to learn and show the concepts for real. I can perform the checks and verify that the results are according to expectations [**compile**]. I will learn about and use the Bitcoin core build system and its compiler [**btccore**]. The 4 main attacks that can happen directly due to the 51% attack are Selfish mining. Canceling transactions. Double Spending. Random forks. I will learn and use the GUIX build system. I will also explore some ideas that are new or rarely used in compiler and build security, such as comparing checksums from known "safe" builds with new builds of the same source [46].

Parallel builds with GNU make -j 4 or 8. Get the number of cores.

In general, it is often desirable to make every technical system secure. The specific problem, in this case, is the trick of hiding and propagating malicious executable code in binary versions of compilers - a problem that was not mitigated for 20 years between 1983 and 2003.

In the context of security engineering and cryptocurrency, a security engineer might face a reversed burden-of-proof. The engineering would need to prove security which is known to be hard, both for practical reasons and for the reason of which and what kind of security models to use. For example, a security engineer might want to demonstrate a new cryptographic alternative to BTC and blockchain. Typically the engineer might get the question if he can prove that there is no security vulnerability.

At least 7 or 8 different operating systems and at least two very different compilers can build Bitcoin Core (https://github.com/bitcoin/bitcoin/tree/master/doc)

Describe the degree project. What did you actually do? This is the practical description of how the method was applied.

Modern OSs randomize memory section addresses (it makes some attacks more difficult), so if anyone restarts the process, the addresses of the instructions might be different.

I will use the build process of Bitcoin Core as my main applied DDC. I will find out about relevant frameworks, build systems and platforms, such as GUIX (https://guix.gnu.org), maybe Frama-C (https://frama-c.com), and/or the CompCert project (https://compcert.org/).

Checks can be created and provisioning a system in a Docker container for portability and reproducibility. I have access to the Bitcoin build toolchain and can build Bitcoin Core with GCC [**compile**]. I can create the demonstration of a compiler Trojan horse in the source code of a compiler to learn and show the concepts for real. I can perform the checks and verify that the results are according to expectations [**compile**]. I will learn about and use the Bitcoin core build system and its compiler [**btccore**]. The 4 main attacks that can happen directly due to the 51% attack are Selfish mining. Canceling transactions. Double Spending. Random forks. I will learn and use the GUIX build system. I will also explore some ideas that are new or rarely used in compiler and build security, such as comparing checksums from known ”safe” builds with new builds of the same source [46].

4.1 Implementation details of self-replicating compiler malware

–

4.2 Demonstration

I present results from TCC (a Compiler).

4.3 Interview Protocol

4.3.1 Interviewees

INTERVIEW WITH DAVID A. WHEELER

4.3.2 Interview Questions

How can we learn the topic? What is relevant knowledge?

4.4 Results

4.4.1 Learning the background and techniques

What kind of background would help a programmer to learn and work with compiler security and build security? Will it help the most to have a background as a C programmer, an expert in compiler technology, a security engineer, a cryptanalyst...?

You should know about compilers (take a class!), computer security, and build systems. You should definitely know how to program in at least one programming language; it doesn't need to be C. Once you learn a few programming languages, learning more is easy; I know over 100 (I counted).

There's no need to be a cryptanalyst. That's a very specialized field & not really relevant for this work. You need to know how to create cryptographic hashes and what decent algorithms are, but that basically boils down to ”run a tool to create an SHA-256 hash”.

Would it be alright to study the C programming language and C compilers to be prepared? There have been reports that most security breaches exploit some bug in some source code written in C, is that still the case and a reason to concentrate on C? If not, what are the other options?

To study the area, you don't need to learn C, though C is still a good language. Many compilers are written in C, and a vast amount of low-level code is in C.

C is not a ”big” programming language. However, it has few ”guard rails” - almost any mistake becomes a serious bug & often a security vulnerability. Unlike almost all other languages, C & C++ are memory-unsafe, that is, they provide no memory safety protection. That has resulted in a lot of vulnerabilities:

70% of Chrome vulnerabilities are memory safety issues; https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

70% of Microsoft vulnerabilities are memory safety issues: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Almost any other language is memory-safe & resolves this problem. If performance is irrelevant, you can use other languages like Python, Ruby, etc. If you want decent performance, you can use decently-performing languages like Java, C#, and Go. The big challenge is if performance is critical. Historically Ada & Fortran were your closest realistic options for performance, but Rust has dramatically risen recently. You should certainly check out Rust at least.

4.4.2 Practical usage

Q: What are the most ambitious uses of DDC you are aware of?

A: That would be various efforts to rebuild & check GNU Mes. A summary, though a little old, is here: https://reproducible-builds.org/news/2019/12/21/reproducible-bootstrap-of-mes-c-compiler/

Q: Are there some public papers or posts you can recommend to read and refer to for my thesis?

A: Well, my paper :-). For the problem of supply chain attacks, at least against OSS, check out: Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier https://arxiv.org/abs/2005.09535

Website: https://reproducible-builds.org

It's mentioned that the technique of the attack is not limited to the compiler, it could also be part of the operating system or hardware itself. What would be en example?

I focused on the "self-perpetuation" part & discussed that in my paper to some extent.

E.g., if the OS detected recompilation of the compiler or itself, it could insert the malicious result instead.

4.4.3 What will future work be like?

4.4.4 Alternative approaches

Q: What other types of mitigations have there been apart from DDC?

A: Main one: bootstrappable builds http://bootstrappable.org/ There, the idea is that you start from something small you trust & then go.

But they actually work well together. How do you trust the small thing? How can you test its results? One approach is... DDC :-). So they are really complementary.

Another approach is proofs at the machine-code level of compilers. There's been some ACL2-related work on that. But proofs are hard anyway; doing them at that level is even harder.

4.4.5 What is the critique or drawbacks?

Q: What has been the most valid critique against DDC?

A: "That's not the primary problem today."

I actually agree with this critique. The primary problem is software filled with vulnerabilities. This has many causes: most software developers don't learn how to write secure software (it's not taught), their tools don't protect them by default (C/C++ in particular), & they don't have tools in their CI pipeline to check for things. Software, even when fixed, is often not updated in a timely way in production.

Once those are resolved, simple supply chain attacks come to the fore, especially dependency confusion, typo squatting, and insertion of malicious code into source code. None of them are trusting trust attacks.

But that's okay. Academic research is supposed to expand our knowledge for the longer term. Once those other problems are better resolved, trusting trust attacks become more likely. I think they would have been more likely sooner if there was no known defense. Even if people don't apply DDC, the fact that DDC exists reduces the incentives for an attacker to use a trusting trust attack (because they can now be detected after the fact). I look forward to a time when DDC is increasingly important to apply because we've made progress on the other problems.

4.4.6 What can we expect from the future?

4.4.7 What would a software be like that could perform the checks against supply-chain attacks?

4.4.8 Why should the build system be self-hosting in the first place (build itself)?

Q: Is GCC already being checked with DDC (https://lwn.net/Articles/321225/) and if yes, since how long?

4.5 Main Findings

There are multiple languages and compilers I can use to implement the attack. One of the main requirements for me is that the compiler is self- hosted. This means that I require the language compiler to be compiled using itself as the compiler, if the compiler is not compiled using this compiler it would be harder to implement a self-replicating attack as it would require the attack to be able to modify multiple compilers. It is in fact possible to do this attack against all compilers capable of compiling other compilers, however to be able to have an infinite cycle of 45 perpetuation of the attack we require some sort of cycle. The easiest cycle to create is one where we compile the compiler using itself, and it is therefore the cycle we aim to create. It is perhaps more interesting to show the attack in the setting of a major, or the major, compiler for a language. This is to show that the attack is not limited to small specifically written compilers. To further be able to detect the attack using Diverse Double-Compiling (DDC), I will add two secondary requirements: It also has to be possible to deterministically compile the compiler for the language I want to implement the attack in. If there is different output for every compilation of the compiler it will be impossible to use DDC as it, as a technique, relies on deterministic compiler output. I want at least one completely different secondary compiler, capable of compiling the compiler to be verified. This requirement is to be able to show that the technique of DDC works with a compiler with a completely different code base than the verified compiler. If I did not have a different secondary compiler I would have had to write a secondary compiler to use, or use an earlier known clean compiler. It is possible to write a secondary compiler for most languages, especially a basic compiler that contains no advanced features not required by the language specification or for compiler extensions needed by the compiler to verify. Nevertheless, writing this secondary compiler could be time-consuming, which is why I have avoided doing this here. Using a known clean compiler, that is one that has not been subjected to the specific attack, it is also a possibility to show DDC. However, this might limit us if we want to use DDC to give us an added feeling of safety with regard to attacks other than our own. As I am in control of my attack, I could therefore always create a version without the attack. A positive effect of DDC is to force a would-be attacker to attack multiple different compilers with a self-replicating attack to hide the attack from verification. I would therefore like to show the technique of DDC using as diverse compilers as possible. The language compiler should not be too slow to compile, to enable iterative development. It is a clear negative, when attempting to show multiple techniques requiring the recompilation of the compiler, if the main compiler takes very long to compile. A fast compiler will be both a positive 46 for me, when writing the attack and showing the detection of it, and for anyone who would like to reproduce the work. Therefore, I am very positive towards a faster compiler. Further, it is a positive, if the language is not a language which already has been used previously for examples of DDC in an academical context. As there is very little written on DDC, this only eliminates C as the implement- ation language. To sum up the requirements: • The compiler should be self-hosted. • The compiler needs to be able to deterministically compile itself. • I want diverse working compilers to be able to perform DDC. • I want a language that is not C. • I prefer a major compiler for the language as the compiler to infect and verify. • I prefer a language with a compiler that is not too slow to compile.

In this section I will explain what a quine is and how to implement a quine in the C programming language [38]. We will later use the techniques from the implementation of the quine to implement the self-replicating attack.

Chapter 5

Result

Here I describe the results of the project. It was possible to implement the trusting trust attack based on Thompson's original description.

For the results, make sure to include some quantitative metrics if possible, as well as compare them with other methods if possible (ideally showing why my way is better)

In the context of Bitcoin, there have been a few notable cases of supply chain attacks. The first case is the Bitcoin-Core project, where a malicious developer uploaded a version of the Bitcoin-Core software that included a backdoor. This backdoor allowed the developer to steal funds from any Bitcoin address. The developer did this by changing the code that generated Bitcoin addresses. The second case is the Bitcoin-Core project, where a malicious developer uploaded a version of the Bitcoin-Core software that included a backdoor. This backdoor allowed the developer to steal funds from any Bitcoin address. The developer did this by changing the code that generated Bitcoin addresses. The third case is that of the Bitcoin.org website, where a malicious attacker was able to insert code that would redirect users to a phishing website. The attacker did this by compromising the server that hosted the website.

In all of these cases, the attackers could compromise the systems because they had control of the supply chain. In the first two cases, the attackers were able to upload malicious code to the Bitcoin-Core project's code repository. In the third case, the attacker was able to compromise the server that hosted the Bitcoin.org website.

These cases show it can attack the Bitcoin network by compromising the supply chain.

Secure compilation aims at protecting against the threat of the compiler potentially getting compromised by someone who inserted malware into it. The canonical example is making the compiler check if it is compiling a particular source code, for example, the login program login.c, and conditionally embedding a backdoor into the login program so that a specific sequence of input will always authenticate a user. This vulnerability was described and demonstrated by Thompson's Turing Award Lecture.

The supply chain is a vulnerable point for Bitcoin Core and other cryptocurrencies. The secure compilation is one way to protect against potential attacks, but it is not a complete solution. More research is needed to understand how to protect against all possible supply-chain attacks.

Chapter 6

Conclusions

This thesis has shown that the supply chain can be compromised for Bitcoin Core and that the attack can be mitigated. If we observe that the two diverse double-compiled compiler binaries are identical, we know that our build system is safe. But if they are not identical, we haven't proved that our build system is compromised.

In conclusion, the paper shows that it is possible to reduce the attack surface of the system by using diverse double-compiling. With this technique, it is possible to improve the security of Bitcoin Core, but it is not possible to achieve 100The reason for this is that it is impossible to trust all the software and hardware that has been used to create the system.

6.1 Discussion

Can we do better than trusting the persons and believing that the build systems did not put malware into the Bitcoin binary? For example, the genesis block's hash value (0x000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f) has to be hard-coded in Bitcoin's source code and can always be "sniffed" for by the compiler. To make sure that the compiler has no such sniffing code, we can look at its source and compile the compiler first – but where are we going to find a clean compiler for that? One that has never been touched by a program touched by Ken Thompson sometime in the past? One way out of this is to write a basic C compiler from scratch using assembly language and use that to bootstrap a C compiler. That is not an activity that happens on average [1]

Investigating a scenario of a complicated security breach requires a security model that is simplified compared to the real scenario because a sufficiently complicated project, as for example Bitcoin Core or its compiler GCC or Clang, often takes a lengthy and complex build process with many binaries from several vendors that are utilized as dependencies and libraries in the toolchain [10]. Compiling Bitcoin Core locally, for example, takes 40 minutes with an Intel I7 CPU. Compiling GCC takes several hours, and compiling Firefox takes a whole day [22]. Vice-versa double-compiling: Switch the compiler under test, which will relax the assumption that one is trusted.

Generated many new randomized versions of the compilers

Describe who will benefit from the degree project, the ethical issues (what ethical problems can arise), and the sustainability aspects of the project.

One interesting question that came up during my meetings with supervisors and the audience was: Why not always use the trusted compiler and nothing else?

Firstly if one used only one trusted compiler for everything, we're back to the original problem, which is a total trust in a single compiler executable without a viable verification process.

As we will see later, the assumption can be relaxed from having a trusted compiler to the assumption that not all compilers are corrupted and use the technique I call vice-versa compiling: First, take compiler C_1 as trusted and perform DDC. Then switch compilers so that C_1 is the trusted one.

Also, as explained in section 4.6, there are many reasons the trusted compiler might not be suitable for general use. It may be slow, produce slow code, generate code for a different CPU architecture than desired, be costly, or have undesirable software license restrictions. It may lack many useful functions necessary for general-purpose use. In DDC, the trusted compiler only needs to be able to compile the parent; there is no need for it to provide other functions.

Finally, note that the "trusted" compiler(s) could be malicious and still work well for DDC. We just need justified confidence that any triggers or payloads in a trusted compiler do not affect the DDC process when applied to the compiler-under-test. That is much, much easier to justify.

6.1.1 There is more than Code

Data-driven attacks (input-processing)

Use references!

6.1.2 Future Work

I had one or two ideas that seem new:

6.2 Extending DDC

Observe systems and functionality of programs in repeated runs to exclude the probability of sheer luck. (1) One new idea would be to use two compilers without knowing which one is trusted and perform DDC twice, first trust compiler A and then trust compiler B. Then it wouldn't matter which one is trustworthy as long as both of them are not compromised. It could be used as a future research direction.

(2) Analyse the machine instructions during actual execution dynamically and check if the machine instructions of the running process have a different number of states (typically at least one more state) than the source code of the process. Typically a corrupted process will include at least one additional logic state for the backdoor of the Trojan horse, for example, the additional logic state of not asking for a password for a specific username.

(3) Randomly generate many different versions of a new compiler and compare them (this idea needs to be developed further)

6.2.1 Final Words

If you are using mendeley to manage references, you might have to export them manually in the end as the automatic ways removes the ”date accessed” field

Bibliography

[1] Amaral, Thiago Melo Stuckert do and Gondim, João José Costa. “Integrating Zero Trust in the cyber supply chain security”. In: *2021 Workshop on Communication Networks and Power Systems (WCNPS)*. IEEE. 2021, pp. 1–6.

[2] Anderson, Ross. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2020.

[3] Bauer, Scott. *Deniable Backdoors Using Compiler Bugs*. 2015. URL: `https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf`.

[4] Bellard, Fabrice. “Tcc: Tiny c compiler”. In: *URL: http://fabrice. bellard. free. fr/tcc* (2003).

[5] blockgeeks. *Hypothetical Attacks on Cryptocurrencies*. blockgeeks, 2021. URL: `https://blockgeeks.com/guides/hypothetical-attacks-on-cryptocurrencies/`.

[6] Böhme, Rainer, Christin, Nicolas, Edelman, Benjamin, and Moore, Tyler. “Bitcoin: Economics, technology, and governance”. In: *Journal of economic Perspectives* 29.2 (2015), pp. 213–38.

[7] Bonneau, Joseph, Miller, Andrew, Clark, Jeremy, Narayanan, Arvind, Kroll, Joshua A, and Felten, Edward W. “Sok: Research perspectives and challenges for bitcoin and cryptocurrencies”. In: *2015 IEEE symposium on security and privacy*. IEEE. 2015, pp. 104–121.

[8] Bowden, Caspar. “Reflections on mistrusting trust”. In: *QCon London* (2014). URL: `http://qconlondon.com/london-2014/dl/qcon-london-2014/slides/CasparBowden_ReflectionsOnMistrustingTrustHowPolicyTechnicalPeopleUseTheTWordInOppositeSense`
`pdf`.

[9] Bratus, Sergey, Darley, Trey, Locasto, Michael, Patterson, Meredith L, Shapiro, Rebecca bx, and Shubina, Anna. “Beyond planted bugs in” trusting trust”: The input-processing frontier”. In: *IEEE Security & Privacy* 12.1 (2014), pp. 83–87.

[10] Buck, Jack. *GCC build process*. 2006. URL: `https://lwn.net/Articles/321225/`.

[11] Buck, Joe. *Ken Thompson’s Reflections on Trusting Trust*. lwn.net, 2009. URL: `https://lwn.net/Articles/321225/`.

[12] Chipolina, Scott. *A Hypothetical Attack on the Bitcoin Codebase*. 2021. URL: `https://decrypt.co/51042/a-hypothetical-attack-on-the-bitcoin-codebase`.

[13] Choi, Jusop, Choi, Wonseok, Aiken, William, Kim, Hyoungshick, Huh, Jun Ho, Kim, Taesoo, Kim, Yongdae, and Anderson, Ross. “Attack of the Clones: Measuring the Maintainability, Originality and Security of Bitcoin’Forks’ in the Wild”. In: *arXiv preprint arXiv:2201.08678* (2022).

[14] Cimpanu, Catalin. *Chrome: 70% of all security bugs are memory safety issues*. 2020. URL: `https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/`.

[15] Cimpanu, Catalin. *Microsoft: 70% of all security bugs are memory safety issues*. 2019. URL: `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`.

[16] Committee, IEEE Standards et al. “IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment: IEEE Standard 2675-2021”. In: (2021).

[17] Courant, Nathanaëlle, Lepiller, Julien, and Scherer, Gabriel. “Debootstrapping without Archeology: Stacked Implementations in Camlboot”. In: *arXiv preprint arXiv:2202.09231* (2022).

[18] Dai, Shuaifu, Wei, Tao, Zhang, Chao, Wang, Tielei, Ding, Yu, Liang, Zhenkai, and Zou, Wei. “A Framework to Eliminate Backdoors from Response-Computable Authentication”. In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 3–17. DOI: `10.1109/SP.2012.10`.

[19] Dave, Maulik A. “Compiler verification: a bibliography”. In: *ACM SIGSOFT Software Engineering Notes* 28.6 (2003), pp. 2–2.

[20] *Dependency-Track*. `https://dependencytrack.org`. Accessed: 2022-04-18.

[21] Dimov, Aleksandar and Dimitrov, Vladimir. “Classification of Software Security Tools”. In: (2021).

[22] Dong, Carl. *Reproducible Bitcoin Builds*. Youtube, 2019. URL: `https://www.youtube.com/watch?v=I2iShmUTEl8`.

[23] Dullien, Thomas and Rolles, Rolf. “Graph-based comparison of executable objects (english version)”. In: *Sstic* 5.1 (2005), p. 3.

[24] Ermolov, Mark and Goryachy, Maxim. “How to hack a turned-off computer, or running unsigned code in intel management engine”. In: *Black Hat Europe* (2017).

[25] Faigon, Ariel. *Testing for zero bugs*. 2005. URL: `https://www.yendor.com/testing/`.

[26] Farsole, Ajinkya A, Kashikar, Amurta G, and Zunzunwala, Apurva. “Ethical hacking”. In: *International Journal of Computer Applications* 1.10 (2010), pp. 14–20.

[27] Gao, Debin, Reiter, Michael K, and Song, Dawn. “Binhunt: Automatically finding semantic differences in binary programs”. In: *International Conference on Information and Communications Security*. Springer. 2008, pp. 238–255.

[28] *Gitian*. 2022. URL: `https://gitian.org/`.

[29] Gordon, Lawrence A and Loeb, Martin P. “The economics of information security investment”. In: *ACM Transactions on Information and System Security (TISSEC)* 5.4 (2002), pp. 438–457.

[30] Graff, Mark and Van Wyk, Kenneth R. *Secure coding: principles and practices*. ” O’Reilly Media, Inc.”, 2003.

[31] Groce, Alex, Jain, Kush, Tonder, Rijnard van, Tulajappa, Goutamkumar, and Le Goues, Claire. “Looking for Lacunae in Bitcoin Core’s Fuzzing Efforts”. In: (2022).

[32] Jeffrey, Christopher. *Mako*. `https://github.com/chjj/mako`. 2022.

[33] Karger, Paul A and Schell, Roger R. *Multics Security Evaluation Volume II. Vulnerability Analysis*. Tech. rep. Electronic Systems Div., L.G. Hanscom Field, Mass., 1974.

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

[35] Kerckhoffs, Auguste. *La cryptographie militaire. 9: 5–38*. 1883.

[36] Kerman, Alper, Borchert, Oliver, Rose, Scott, and Tan, Allen. "Implementing a zero trust architecture". In: *National Institute of Standards and Technology (NIST)* (2020).

[37] Kissel, Richard. *Glossary of key information security terms*. Diane Publishing, 2011.

[38] Lamb, Chris and Zacchiroli, Stefano. "Reproducible builds: Increasing the integrity of software supply chains". In: *IEEE Software* 39.2 (2021), pp. 62–70.

[39] Linderud, Morten. "Reproducible Builds: Break a log, good things come in trees". MA thesis. The University of Bergen, 2019.

[40] Maynor, David. "The compiler as attack vector". In: *Linux Journal* 2005.130 (2005), p. 9.

[41] Maynor, David. "Trust no one, not even yourself, or the weak link might be your build tools". In: *The Black Hat Briefings USA* (2004).

[42] McKeeman, William M and Wortman, David B. *A compiler generator*. Tech. rep. 1970.

[43] Merkle, Ralph C. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[44] Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized Business Review* (2008), p. 21260.

[45] Narayanan, Arvind, Bonneau, Joseph, Felten, Edward, Miller, Andrew, and Goldfeder, Steven. "Bitcoin and cryptocurrency technologies". In: *Curso Elaborado Pela* (2021).

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

[47] Oh, Jeongwook. “Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries”. In: *Blackhat technical security conference*. 2009.

[48] Ohm, Marc, Plate, Henrik, Sykosch, Arnold, and Meier, Michael. “Backstabber’s knife collection: A review of open source software supply chain attacks”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2020, pp. 23–43.

[49] Pariente, Dillon and Ledinot, Emmanuel. “Formal verification of industrial C code using Frama-C: a case study”. In: *Formal Verification of Object-Oriented Software* (2010), p. 205.

[50] Patrignani, Marco, Ahmed, Amal, and Clarke, Dave. “Formal approaches to secure compilation: A survey of fully abstract compilation and related work”. In: *ACM Computing Surveys (CSUR)* 51.6 (2019), pp. 1–36.

[51] Peisert, Sean, Schneier, Bruce, Okhravi, Hamed, Massacci, Fabio, Benzel, Terry, Landwehr, Carl, Mannan, Mohammad, Mirkovic, Jelena, Prakash, Atul, and Michael, James Bret. “Perspectives on the SolarWinds incident”. In: *IEEE Security & Privacy* 19.2 (2021), pp. 7–13.

[52] Pfleeger, Charles P and Pfleeger, Shari Lawrence. *Analyzing computer security: A threat/vulnerability/countermeasure approach*. Prentice Hall Professional, 2012.

[53] Regehr, John. *Defending Against Compiler-Based Backdoors*. 2015. URL: `https://blog.regehr.org/archives/1241`.

[54] *Reproducible Builds Website*. 2022. URL: `https://reproducible-builds.org`.

[55] Schneier, Bruce. *Countering trusting trust*. 2006. URL: `https://www.schneier.com/blog/archives/2006/01/countering_trus.html`.

[56] Schuster, Felix. *WEASEL*. `https://github.com/flxflx/weasel`. 2013.

[57] Schuster, Felix and Holz, Thorsten. “Towards reducing the attack surface of software backdoors”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 851–862.

[58] Schuster, Felix, Rüster, Stefan, and Holz, Thorsten. “Preventing backdoors in server applications with a separated software architecture”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2013, pp. 197–206.

[59] Shannon, Claude E. “Communication theory of secrecy systems”. In: *The Bell system technical journal* 28.4 (1949), pp. 656–715.

[60] Sharma, Ax. *Cryptocurrency launchpad hit by $3 million supply chain attack*. 2021. URL: `https://arstechnica.com/information-technology/2021/09/cryptocurrency-launchpad-hit-by-3-million-supply-chain-attack/`.

[61] Shoshitaishvili, Yan, Wang, Ruoyu, Hauser, Christophe, Kruegel, Christopher, and Vigna, Giovanni. “Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.” In: *NDSS*. Vol. 1. 2015, pp. 1–1.

[62] Skrimstad, Yrjan. “Improving Trust in Software through Diverse Double-Compiling and Reproducible Builds”. MA thesis. 2018.

[63] Somlo, Gabriel L. “Toward a Trustable, Self-Hosting Computer System”. In: *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2020, pp. 136–143.

[64] Spencer, Henry. “November 23, 1998.“Re: LWN-The Trojan Horse (Bruce Perens)””. In: *Robust Open Source mailing list* ().

[65] Stafford, VA. “Zero trust architecture”. In: *NIST Special Publication* 800 (2020), p. 207.

[66] Stallman, Richard M et al. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation, 1999.

[67] Thompson, Ken. “Reflections on trusting trust”. In: *ACM Turing award lectures*. 2007, p. 1983.

[68] Waksman, Adam, Suozzo, Matthew, and Sethumadhavan, Simha. “FANCI: identification of stealthy malicious logic using boolean functional analysis”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 697–708.

[69] Wheeler, David A. “Countering trusting trust through diverse double-compiling”. In: *21st Annual Computer Security Applications Conference (ACSAC’05)*. IEEE. 2005, 13–pp.

[70] Wheeler, David A. “Fully countering trusting trust through diverse double-compiling”. PhD thesis. George Mason University, 2010.

[71] Wikipedia. *Plausible deniability.* `https://en.wikipedia.org/wiki/Plausible_deniability`. 2021.

[72] Wirdum, Aaron van. *What Is Gitian Building? How Bitcoin’s Security Processes Became a Model for the Open Source Community*. 2016. URL: `https://bitcoinmagazine.com/technical/what-is-gitian-building-how-bitcoin-s-security-processes-became-a-model-for-the-open-source-community-1461862937`.

[73] Würthinger, Thomas and Linz, Juli. “Formal Compiler Verification with ACL2”. In: *Institute for Formal Models and Verification* (2006).

[74] Yang, Kaiyuan, Hicks, Matthew, Dong, Qing, Austin, Todd, and Sylvester, Dennis. “A2: Analog malicious hardware”. In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, pp. 18–37.

[75] Yee, Bennet, Sehr, David, Dardyk, Gregory, Chen, J Bradley, Muth, Robert, Ormandy, Tavis, Okasaka, Shiki, Narula, Neha, and Fullagar, Nicholas. “Native client: A sandbox for portable, untrusted x86 native code”. In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 79–93.

[76] Zboralski, Anthony C. *Things To Do in Ciscoland When You’re Dead*. 2000. URL: `http://phrack.org/issues/56/10.html`.

[77] Zeller, Andreas. “Isolating cause-effect chains from computer programs”. In: *ACM SIGSOFT Software Engineering Notes* 27.6 (2002), pp. 1–10.

[78] Zhang, Jie, Yuan, Feng, Wei, Linxiao, Liu, Yannan, and Xu, Qiang. “VeriTrust: Verification for hardware trust”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.7 (2015), pp. 1148–1161.

[79] Zhang, Jie, Yuan, Feng, and Xu, Qiang. “Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans”. In: *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 2014, pp. 153–166.

Appendix - Contents

A First Appendix 71

B Second Appendix 72

Appendix A

First Appendix

This is only slightly related to the rest of the report

Appendix B

Second Appendix

Functional equivalence is undecidable. It means there is no algorithm that, given arbitrary (source) code in programming language A and arbitrary (compiled) code in another programming language B, can decide whether both code always function the same if given the same input. Here, we assume both programming languages are arbitrarily fixed and Turing-complete. ”behave the same” means either both loop forever or both return the same string.

proof.

Suppose there is such an algorithm M . Let us solve the halting problem in language A using M

.

Let S_a be some arbitrary source code in A and w

be some arbitrary string.

Since S_a is Turing complete, we can write S_b, a program in B that behaves the same as S_a

.

Write code S_b' in language B so that it is the same as S_b except when it halts. At the time when it halts before returning the result, it will check whether the given input is w first. If it is, S_b' will loop forever. Otherwise, it will return the result as usual. That is, S_b' is the same as S_b except possibly that S_b' always loop forever if the input is w

.

Now we use M to check whether S_a and S_b'

always behave the same if given the same input.

If M's verdict is yes, then S_a on w does not halt. Otherwise, S_a on w halts. We have solved the halting problem in language A

.

However, it is known that the halting problem in a Turing-complete language is undecidable. This contradiction implies that M

does not exist.

In fact, any nontrivial behavior property of a program in a Turing-complete language is undecidable. That is Rice's theorem.

What are the nontrivial behavior properties?

A property of a program is a behavior property if, for any two programs that behaves the same, either both has that property or neither does.

A behavior property is nontrivial if at least one program has it while at least one program does not.

The following are some nontrivial behavior properties. Will the program output a specific output if it runs without any input? Is there an input that makes the program not function according to specification? Will the process connect to the internet
?