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 that is used to create the system. Due to the financial capabilities of the Bitcoin Core project, there has been a general interest for security issues with its design and implementation. While most questions and issues have centered around Bitcoin Wallets and breaches that are 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 in which 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 itself has been described in several articles and studied in computer security and compiler security. During the years of research of 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 in order to reduce probability of undetected compiler vulnerabilities. This variety has not previously been described. In context of the Bitcoin Core build system an attack and compromise of it is possibly 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 use 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, 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

CPU	central processing unit
GDB	GNU Debugger
GCC	GNU Compiler Collection
TCC	Tiny C Compiler
PGP	Pretty Good Privacy
DDC	diverse double-Ccmpiling
SQL	Structured Query Language
NIST	National institute of Standards and Technology
RAM	random access memory
ACL2	A Computational Logic for Applicative Common Lisp
CPU	central processing unit

FPGA	field-programmable gate array
UEFI	unified extensible firmware interface
ASLR	address space layout randomization
OSS	open source software
VHDL	VHSIC hardware description language
ACL2	A Computational Logic for Applicative Common Lisp
VCS	Version Control System
NFT	non-fungible Token
MiTM	man-in-the-middle
NIST	National Institute for Standardization
HT	hardware trojan
NGO	non-government organization
MIPS	Microprocessor without Interlocked Pipeline Stages
DeFi	decentralized finance
XBI	cross-build injection

VM	virtual machine
YAML	YAML Ain’t Markup Language
ANSI	American National Standards Institute
MNC	multinational corporation
MULTICS	Multiplexed Information and Computing Service
CVE	Common Vulnerabilities and Exposures

Contents

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 compromised 1.2.

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

1.1 Hypothesis Statement

Trusting Trust - The hypothesis being worked with is: To what extent is Bitcoin Core 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 have to 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 analysed, for example:

1. stealing a Pretty Good Privacy (PGP) key from a maintainer and use 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, hide it. that would get caught by the verify signatures script

The motivation for this kind of attacks 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).

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

Link things together with references. This is a reference to a section: 1.2.

1.2 Background

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

Packages used in programming languages have in recent years been high profile targets for supply chain attacks. A popular package from the Ruby programming language used by web developers was compromised in 2019, where 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 credentials of the authors 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 can 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 initially have been spread through Baidu Wangpan [27]. This malicious version of Xcode replaces CoreServices object files with malicious object files. These object files are used for the compilation of 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 or 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 25the 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 attack 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 that mostly targeted the Chinese market [18]. Nevertheless, applications such as WeChat have been popular in larger regions of eastern Asia. The malware can therefore also have 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 by some 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 execution of 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 includes the ability to 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 such that it 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 quite 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 also 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 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 a 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 that are utilized as dependencies and libraries in the toolchain. Compiling Bitcoin Core locally for example takes 40 minutes with an Intel I7 CPU. Compiling GCC itself 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 that are needed to understand the degree project and thesis. (Still, keep in mind that this is an introductory part, which does not require too detailed description).

Use references[1]

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

This background presents background to writing a report in latex.

Look at sample table 1.2.1 for 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

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

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 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 the attacks was there because it couldn't be seen.

Listing 1.1: C example

```c
    ...
   /* 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 */
   ...
```

My aim is to give answers and elaborations for the following questions. What are the threats and the vulnerabilities that should be protected against with diverse double-Ccmpiling (DDC)? What are the technical benefits and shortcomings of DDC compared to other solutions to real and potential problems with compiler vulnerability and build 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 Preferable, 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 purpose of the project after purpose of the thesis).

1.4 Preliminaries and Definitions

Define FIRMWARE

According to the National Institute for Standardization (NIST) glossary, a trusted computer system is described as a system which is sufficient to perform its tasks, while a trustworthy system is described as a system which is reasonably secure and according to established standards [36]. 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 acknowledgement of solvable problems and the apparent lack of solutions [61]. Technologies in provable security have achieved increasing interest in software engineering and related disciplines.

The meaning of trust can be understood as a certain confidence that something will behave as expected, with or without a 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 a proof or formal verification for being verified, while a system that is simply trusted would not base incoming trust upon a proof or a 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 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 intended meaning and connotation of the work trust is 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 an objective verification, which would be more ideal [8].

On one side there's been described the concept of provable security with formal verification as a way to at least 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 intepreter. 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 a certain ports or sockets. Since hardware and software backdoor started to appear, there's been ongoing work both to protect against the 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 the 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 being mentioned for being the foundation of digital and software supply chain security, but will mostly be omitted in this essay for 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 to be 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 which will probably be able to automatically detect 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 what is 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 build security at first hand, but considering that many projects are today being built directly from an internet browser which 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 that are recommended are increased awareness, being careful, code review, testing and verifications [29]. (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 in the 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, even a complete analysis of a system's source code will not find the malware that is there.

Supply-chain attacks are those in which an adversary tries to introduce targeted vulnerabilities into deployed applications, operating systems, and software components [10]. Once pub- lished, such vulnerabilities are likely to persist within the affected ecosystem even if patches are later released [11]. Following a number of 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**] [19]. 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 which checks the dependencies of a source code project, such as Dependency-Track, is one which would, if it could, detect a compromised build system. An example would be if the build system named Maven itself was 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 strengthens 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 et al. software security tools are classified into different classes. (WHICH ONES?) Using this classification, a diverse double-compiling would fall into the vulnerability scanning class [20].

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 was able to do this by changing the code that generated Bitcoin addresses. The second case is that of 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 was able to do 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 was able to do this by compromising the server that hosted the website. In all of these cases, the attackers were able to 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 were able to 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. In all of these cases, the attackers were able to 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 [35] [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

How can we trust that an executable (compiled) object is a legitimate representation of the source code of the intended program? Do we have a good reason to believe that the compiled code doesn't contain malware which was embedded during compilation, malware that can possibly reproduce itself forever? These questions were asked by Ken Thompson in his Turing Award lecture [67]. The idea originates from an Air Force evaluation of the Multiplexed Information and Computing Service (MULTICS) system that was carried out by Karger and Schell and published in a technical report in 1974 [32]. In 1985, a decade after the work by Karger and Schell, Ken Thompson specified the vulnerability. Thompson posed the questions along with example 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 of the compiler. 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: Any program that handles another program can be compromised in the way that is 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 both from academic researchers and from commercial businesses. Cybersecurity is today considered as more important than ever. The potential for deceptive malware to propagate in object form without being seen implies that the risk for enormous damage is technically possible. The build system being used in most of these cases relies on the GNU Compiler Collection (GCC) in most cases [66]. If the particular instance of the GCC would contain a self-replicating malware, being used to build the next version of itself and being used to build Bitcoin Core, a single individual, Government organization or some capable non-government organization (NGO) would become able to manipulate the cryptocurrency centrally and arbitrarily. While the ideas and executions were already studied in the 1970s first by Karger & Schell, then by Thompson, there was almost no research about the topic between the mid-1980s to the mid 1990s, partly due to the abundance of ideas and abundance of techniques for mitigation [32, 67]. Only in 1998, Henry Spencer suggested a countermeasure [64].

While the original trusting trust attack was created with a malware compiler, more cent reports in systems security put more emphasis on the input data. Bratus et al. write that the input being 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 itself. Therefore, it is argued that input data could and should be treated as a potential program itself, because the input is changing the state of the machine. Validation of input of verification of source code are two measures that can be appropriate to reduce the risk of an attacker getting control of the flow of execution.

Before double-compilation, all previously known countermeasures were known to be insufficient. Security experts even claimed that there was no defense at all against the trusting trust attack: Bruce Schneier has claimed that there is no defense if the build systems have been compromised [55]. Consequently, given that a trusting trust attack cannot be countered, attackers would quietly be able to subvert entire classes of computer systems, gaining complete control over financial, infrastructure, military, and/or business systems worldwide.

The related work that is discussed in this chapter mostly covers supply-chain security and the vulnerabilities resulting from third-party software, or software dependencies. There are several technologies in use for the purpose of detecting binary differences [26, 47]. Detecting binary differences through use of analysis methods for binary object has been the main idea behind DDC and the other means of binary analysis.

Two main practices that have been used to improve the security of the supply-chain are, firstly, the practice of testing and verification. Testing and verification of systems have been done extensively for many years to minimize the risk for including any vulnerabilities in the release version of the system. Secondly, and more recently, more emphasis has been put on a practice that is referred to as secure development and application security, to avoid including vulnerabilities as early as possible in the supply-chain. The latter practices are included is a work method referred to as shifting left, meaning that software development should shift the work on security to the left, i.e. earlier, in the production line so that cybersecurity is being worked on and considered as early as possible instead of waiting with it to be fixed later [16]. It is reasonable that these two practices complement each other, instead of one being always superior to the other. In many cases, a technique for testing and verification will depend on and require a certain development practice being done prior to the test e.g. using checksums, and 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. conducted a review of supply-chain attacks, with emphasis on software backdoors [48]. Their results give measures and specific statistics of dependencies, modules and packages in JavaScript (npm), Ruby (Gems), Java (Maven), Python (PyPI) and PHP.

It has been considered a weakness that critical systems and critical activities rely on trust in the people who delivered the systems, as previously described by Ken Thompson. Zboralski, a technical writer, states that this potential vulnerability is the central problem in network security [76]. We must either build the entire computer system ourselves only with our own hardware and software, or we must 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

Backdoors can be inserted into software and hardware during several steps of construction and development. The vulnerabilities can be created, by mistake or deliberately, both during the design and architecure of a system as well as during the actual construction. Every use of backdoors has not been out of malice, since backdoors have historically been used for legitimate administrators and maintenance staff to be able to unlock every device of a certain kind for repair and maintenance [25]. This feature of undocumented ways to unlock any device has been a trade-off, partly to being able to solve problems at the expense of compromised security. The main problem is the impracticality of trying to verify an entire system, which would take too long due to the enormous combinatorial number of possible states of the circuits.

2.1.2 Self-Replicating Compiler Malware

Software which is supposed to cause miscompilation so that the resulting executable contains a vulnerability is sometimes referred to as compiler malware. John Regehr states that countermeasures can be taken to mitigate the threat of such compiler malware [53]. Two of the questions posed by are:

Will this kind of attack ever be detected? Whose responsibility is it to protect the system: The end-user or the system designer?

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 certain source code, for example the login program login.c, and conditionally embedding a backdoor into the login program so that a special sequence of input will always authenticate a user. This vulnerability is the one was described and demonstrated by Thompson's Turing Award Lecture. A number of hypothetical attacks are described in their report. One of the possible attacks that is described is the possibility of exploiting the compiler and build system of the platform being evaluated. The authors Karger and Schell returned more recently with a follow-up article that describes the progress during the last 30 years, concluding that the scenario is still a problem which still has not been completely solved [33].

2.1.3 Deceptive Hardware Trojans

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

A technology named DeTrust is proposed [79]. DeTrust proves that a hardware Trojan can be included during construction. It will avoid detection by the commonly used verification systems. These two verification systems are named FANCI and VeriTrust. FANCI performs functional analysis of the logic in the circuit that is unlikely to affect the output, and then flag it as potentially suspicious [68]. Another technology named VeriTrust tries to find malware circuitry by checking the extreme values and corner cases of the circuit [78].

Zhang et al. also recognize that verifying all possible states of a hardware often becomes unfeasible in practice because of the quick growth of combinations of possible states as bitlength becomes longer and the functionality of the system is becoming 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 make some practical proposals how to defend against the attack technique they first described. Their suggestions are 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 it does today in order to detect an implicitly triggered hardware Trojan.

2.2 Countermeasures to Backdoors

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, there are fewer parts that can go wrong. 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 large damage if they were compromised. Attempts have been made to measure and economically quantify these types of risks and potential damages, notably the Gordon-Loeb model for IT security [28].

2.2.1 Response-Computable Authentication

Dai et al. created a framework for response-computable authentication (RCA) that can reduce the risk for including undetected authentication backdoors [17]. Their work extends the Google Native Client (NaCl) [75]. 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 idea is to separate and perform checks of the authentication system and its subsystems, such as checking the cryptographic password-check function for collisions, and detect side effects or find 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 cryptosystems in general: It should be assumed that an attacker has complete knowledge of the mechanisms in place, but no knowledge of the actual passwords or secret codes or keys that are being used. Commonly referred to as Kerckhoffs's principle, this assumption was reformulated (or possibly independently formulated) by American mathematician Claude Shannon as "the enemy knows the system", and consequently, hardware and software developers must design and construct all systems assuming that an enemy will be knowledgeable of how the machine works [34, 59]. The work by Dai et al. did not include any real testing or checking of their framework which could have been done, for example testing the framework with 30 different authentication mechanism where one of them is deliberately vulnerable to see if it can be strengthened that the framework actually detects the true 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 is centered around specific debugging techniques that utilize decision trees of binary code [57, 77]. They introduced a debugging technique that they call delta-debugging, making it possible to detect which parts of a system are possibly compromised and then perform steps for further analysis. Their software makes use of the GNU Debugger (GDB) and is capable of analyzing binary code for x86, x64 and Microprocessor without Interlocked Pipeline Stages (MIPS) architectures. Their software, named WEASEL, is available for the public at GitHub [56]. Their article then gives the results from practical test cases to show that the WEASEL can detect and disable both malware that was found in real incidents and malware that was deliberately created for their 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 how to prevent the backdoors proactively [58]. These extend the previous work with the NaPu that was proposed by Dai et al. The result is a system which has reduced risk for vulnerabilities through use of virtualization and isolation.

2.2.3 Firmware Analysis

Shoshitaishvili et al. describe a system named 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 the purpose of analysis, there is a significant difference between openly available source code and proprietary source code. The reason they give is that 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 is able to find vulnerabilities that other analysis systems cannot, namely the one from Schuster et al. which is based on certain assumptions that Firmalice does not need [57].

Their article describes how to detect backdoors. Since a proprietary source code often isn't available for direct analysis, the Firmalice system makes use of existing techniques of disassembly, then identifies what the privileged states 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 about several cases of real products which had vulnerabilities in the object code, and 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 be aimed at making compilers that preserve the security properties of the source programs they take as input in the target programs they produce as output [50]. Secure development is broad in scope, it targets languages with a variety of features (including objects, higher-order functions, memory allocation, concurrency) and employs a range of different techniques to ensure that security of the source code is preserved in the generated executable and at the target platform.

The attacks against the compiler which were described have been called deniable due to the fact that the attack cannot be seen when viewing the source code of the compiler. This term is taken from the legal expression “plausible deniability” and the technology known as deniable cryptography [3, 71]. The property of being deniable is obviously a main characteristic of the most difficult supply-chain attacks and constituted a major challenge.

2.3.1 Debootstrapping

The technique named debootstrapping has been suggested and put in practice to avoid trust in a single build system [38]. The idea is to always use very different compiler implementations so that one compiler can test the other, and avoid self-compiling compilers which compile different versions of themselves. The need for debootstrapping has been described in work by Courant et al [38]. 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 other language than the language of the compiler-under-test. The result, called the debootstrapped binary, may be very different from the bootstrapped binary (different or no optimisation, etc., as our debootstrapped compiler produces worse code), but it should have the same semantics.

For the purpose of 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 could implement only a subset and doesn’t need to meet critical performance requirements or be optimized, since it is only used for correctness. Two such compilers that have been used a Java compiler named Jikes, and a minimal C compiler called TCC that has been used to debootstrap

GCC [4].

2.3.2 Self-Hosted Systems

There have been findings of an undocumented extra microprocessor in certain systems [23]. There are claims that the only system that can be fully 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. This, the author means, increases the complexity of the problem of checking whether a system is trustworthy. The approach that is described includes steps performing DDC but also takes a much broader scope of creating a whole self-hosted independent computer system with FPGA to check another system so that the system that performs the check also reduces the risks of being compromised in the linker, loader, assembler, operating system or its mainboard.

2.4 Testing and Verification

Several sources write that it's better to verify than to trust [46, 65]. It is recommended that a zero-trust policy should be practiced as much as practically possible [1]. During verification, one major challenge has been the exponentially increasing number of states of the system that need to be verified, and that 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. Though the question in general is undecidable, the authors describe certain techniques (model checking and more) that can be used with approximation and a reduced state space with annotations and assertion in the source code that should be checked.

A related technology uses an approach with proofs at the machine-code level of compilers. There has been work on that using the system named A Computational Logic for Applicative Common Lisp (ACL2) which can be used as a theorem-prover for LISP. Formal verification of compilers has been described in [73]. Wurthinger writes that even if a compiler is correct at source level and passes the bootstrap test, it may be incorrect and produce wrong or even harmful output for a specific source input [18]. Attempts have also been made to directly analyze binary (executable) code to detect vulnerabilities. Some methods to perform the analysis make use of techniques from graph theory [22, 26].

2.4.2 Diverse Double-Compilation

DDC is a technique proposed by David A. Wheeler in 2005 to use an alternative implementation to gain trust in a bootstrapped binary, proving the absence of trusting trust attack [69]. (DDC needs reproducible/deterministic compiler builds.) First, a bootstrap binary is used to build a reference implementation from 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 ever using 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 trusting trust attack; if it is not, 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 decribed procedure, unless multiple compilers were compromised.

Historically, Henry Spencer was the first to suggest a comparison of binaries from different compilers [64]. His idea was inspired by McKeeman et al. who had written about techniques for detecting compiler defects, and provided a formal treatment of verifying self-compiling compilers [42]. McKeeman also introduced the T-diagrams to illustrate compilation techniques. Spencer remarked that compilers are a special case of computer programs for the purpose of 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 a given input.

However, one can apply a further level of indirection.

What was suggested was to compile the compiler using both itself and a different compiler, generating two executables. These two executables can be assumed to behave the same under test, because both of them came from the same source code. They won't be bit-by-bit identical though because the two different compiler manufacturers have used different techniques to generate the compiled executables. Now, one can use both those binaries to compile the compiler source again, generating two outputs. Since the binaries should be the same, the outputs should be bit-by-bit identical. Any difference indicates either a critical defect in the procedure, or that there is malware in at least one of the original compilers. [64].

Wheeler put the idea into practice and coined the technique called DDC in 2005 in hist first report about it [69]. Wheeler then elaborated in more detail about DDC in his 2010 Ph.D. dissertation [70]. The compilation procedure of GCC was not as described in its own documentation. It's mentioned in the thesis of Wheeler how the GCC compiles itself: GCC is built with a three-stage bootstrap procedure. The committers of GCC have been using DDC for some time, although not using the term DDC for what they're doing [11]. The GCC compiler documentation explains that its normal full build process, called a bootstrap, can be broken into stages. The command "make bootstrap" is supposed to build GCC three times: When GCC is built, there is a procedure with three stages. First the compiler is built with some C compiler, that might be an older GCC , or might be a different compiler entirely. This task is called "stage 1". Next, GCC is built again, by the "stage 1" compiler that it previously compiled, to produce "stage 2".

Finally, GCC is built with the "stage 2" compiler it generated the second time and the result is referred to as stage 3 [66]. The two final stages should produce the same two outputs (besides from small differences in the timestamps in object files) and that is checked with the command "make compare". If the two outputs are not bit-for-bit 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. The checking is done for every build of GCC [11]. Now hypothetically, if the particular instance of GCC had included some malware of the trusting trust 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 too. The more compilers that are included, the stronger the result will be: Either there is no trusting trust attack, or every C compiler we tried contained the malware. Since the compilers that have been used are both Sun's proprietary compiler and GCC to build GCC on Solaris, Wheeler meant that his proof shows that either GCC is legitimate or Sun's proprietary compiler contains malware, where the latter event is considered highly unlikely.

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. This has been the case with the U.S. military, who standardized software development with the Ada programming language and hardware development with the VHSIC hardware description language (VHDL) language. For the standardized languages, there are many compilers. This diversity of compilers will reduce the probability of compromised and subverted build systems. If there is only one compiler for a certain programming language, then it will not be possible to perform the DDC. Wheeler described the three parts that are central to the attack: trigger, payloads and non-discovery. With trigger, it is meant that a certain 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 the binary A if the binary B is cooperating 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 larger and more diverse systems under test and relaxing the requirements of DDC, something that is detailed later in section 2.3.2.

Several sources including Wheeler's dissertation explain that the security of a computer system isn't 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 the same argument can be applied to the linker, the loader, the operating system, firmware, unified extensible firmware interface (UEFI) and even the hardware of the computer system 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. To achieve reproducible builds, all non-determinism such as randomness and build-time timestamps need to be removed or handled in a deterministic manner so that the builds give bit-by-bit identical results every build for the same version [37]. To check that a build is the legitimate version, a simple check of a checksum is done. The authors note, however, that there is still no obvious consensus 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) Tiny C Compiler (TCC) with a minimal amount of trusted code.

Linderud examined build systems with independent and distributed builds to increase the probability of a secure result from the build. He suggests metadata to make it easier to see whether the integrity of the build has been compromised [39]. The methods described in that thesis are based on validation of build integrity. The methods include signed code and Merkle trees to validate the downloaded packages against available metadata [43]. The methods would have the same vulnerability as any other reliance on an external supplier, so it will protect against third-party tampering of the system, but it will not protect against any attack from a previous version of the build system itself, such as in the case of a trusting trust attack. Supply-chain attacks have even been implemented in analog hardware [74].

2.4.4 Fuzz Testing

Wheeler claims in his writing about DDC that randomized testing or fuzz testing would be unlikely to 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). In the case of testing a compiler, the compiler-under-test will be compared with a reference compiler. The outcome of the test will depend on whether the two compilers produced different binaries. This approach is described by Faigon [24]. The approach has been successful in finding many software bugs and compiler errors, but it will be extremely unlikely to detect maliciously corrupted compiler. If such a corrupted compiler diverts from its specification in only 1/1000 executions of its target (as would be the case in a cryptocurrency system that sends every thousand transaction to a different receiver), it becomes evident that tests are unlikely to detect the bug in the compiler. For randomized testing to work for compiler-compilers, the input would need to be a randomized new version of the compiler which has not been attempted by anyone. The situation will be that there's a compiler being compiled and that the Trojan horse only targets very specific input, such as the compiler itself. On the other hand, Wheeler's analysis doesn't go into detail why we cannot input the source code of the compiler to a compiler binary and then do fuzz testing with variations on that.

2.5 Secure Development for Cryptocurrency

Due to the financial capabilities of the Bitcoin Core project, there has been a general interest for security issues with its design and implementation. While most questions and issues have centered around Bitcoin Wallets and breaches that are 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 [44]. It didn't specifically address the security of its own implementation, and nothing was written about a trusting trust attack.

The reason for trusting the blockchain of Bitcoin is mostly due to the integrity of being able to verify the entire blockchain can be verified from the hash value of the first block. That value of the first block is written as a constant in the source code of Bitcoin Core. Developers and users trust that it's secure. But compromising the security of a digital currency is of course an activity that many individuals and organizations would like to do. Not only the single individual "malicious hacker" who tries to rob the digital bank or steal someone's digital wallet, but also large multinational corporation (MNC) and governments that have interests in compromising the cryptocurrency and manipulating the blockchain.

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

Sharma writes in a recent news article that supply-chain attacks have occurred recently against the blockchain [60]. That was a software supply chain attack against a decentralized finance (DeFi) platform for cryptocurrency assets. A committer to the codebase had included a vulnerability that was included in the production version of the platform. This has led to questions about quality assurance for contributions to source code and if the review process was the real problem in this case.

Rosic discusses a number of different hypothetical scenarios to compromise a cryptocurrency and a blockchain [5]. The majority of the scenarios that are described are issues and problems with collaboration between Bitcoin users and miners. None of the scenarios that are described involve binary Trojan horse backdoors or a compromise of the build system itself.

In 2022, the researchers Choi et al. have 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 the security vulnerabilities on average took significantly long time before they were mitigated.

In general, any software that includes third party dependencies must be checked and tracked so that a dependency doesn't contain a vulnerability or an exploit, and the same reasoning about the build system. At first sight, there is no clear policy available and mentioning of any mechanism in practice for secure development and/or testing and verification of the security, including the dependencies and the build system. There is also a clear lack of Common Vulnerabilities and Exposures (CVE) information available for Bitcoin and Blockchain projects. The authors of Reproducible Builds mention in the article that early development of Bitcoin Core was done in a “jail” [37]. The authors of the article most likely referred to a system named Gitian that’s been used to check the integrity of Bitcoin builds [72]. Gitian creates this control by doing this deterministic build inside a specific virtual machine (VM), that is being fed the instructions through a specific declaration written in YAML Ain’t Markup Language (YAML). Bitcoin Core has since then changed its build system to GUIX [27].

The authors Groce et al. published their research about the effectiveness of fuzz testing for Bitcoin Core fuzzing. They examine to what extent it has been possible to conduct fuzz testing for finding bugs in the software of Bitcoin Core [30]. A common problem with fuzzing is that the fuzzing becomes saturated, the project being tested soon becomes “resistant” and further fuzzing finds almost nothing, although having been effective in the beginning. he authors conclude that there are possibilities to utilize fuzz testing for the Bitcoin Core project and that there is room for further improvement.

For the purpose of simplicity, it is preferable to conduct academic research and tests with a minimal distribution of a software, at least to begin with. Otherwise, there is a risk that build duration and other unnecessary complications slow down the pace of the work. For example, the build duration of large projects written in C++ are often rather long. So instead of the Bitcoin Core that is written in C++, there’s another implementation of Bitcoin called Mako that is written in American National Standards Institute (ANSI) C with fewer external dependencies [31]. This project can be compiled to a Bitcoin binary that can be conceptually easier to prove it is free from malware. As would be the case in a cryptocurrency system that sends every thousand transaction to a different receiver. For randomized testing to work for compiler-compilers, the input would need to be a randomized new version of the compiler. It’s been said that multiple implementations of a protocol are good practices. 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 even though extensive research has been done for a long time about software vulnerabilities, there is so far relatively little about the supply-chain security of cryptocurrency in general. While diverse double-compiling has been studied and used to rather large efforts, there is also relatively little or nothing that puts DDC in the context of cryptocurrency and Bitcoin.

Chapter 3

(WORK IN PROGRESS) Method

The methods described in the subsequent chapters describe the details of, and prove the feasibility of, a trust-based attack. It also includes an implementation of this type of attack. One task has been to create a real version of the attack for demonstration purposes.

The subsequent chapters will explain why certain methods were chosen and not others. It will also explain how certain novel methods can be useful to detect and identify a trust-based attack. It also discusses how to prevent such attacks from compromising the build system in the first place.

3.1 Qualitative Investigation with Interviews

I have conducted an interview with David A. Wheeler. I also corresponded with committers to much-used compilers GCC and TCC.

3.1.1 Feasibility of the Attack

A common situation is that a new compiler or a new build system is going to be installed on a system. That compiler is itself being compiled and built. The usual procedure to compile the compiler is illustrated here.

A Trojan horse in the compiler-compiler might have targeted a very specific input, such as the compiler itself.

That the attack is feasible has already been proven in practice. Thompson did release

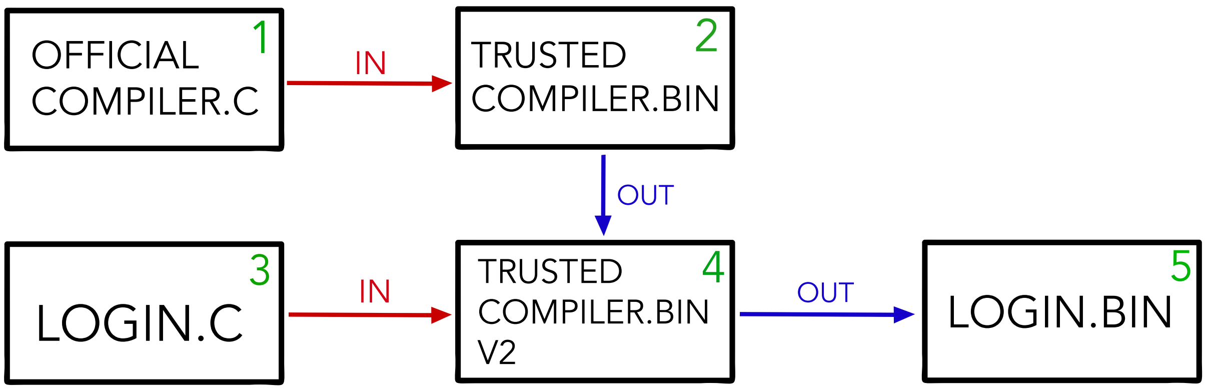

Figure 3.1.1: Compilation of a compiler that compiles the login program

a malware compiler internally at Bell Labs. My own implementation is also available to prove that the implementation is feasible in practice.

3.1.2 Correspondence with Committers to TCC

3.1.3 Correspondence with Committers to GCC

3.2 The Method of the Trust-Based Attack

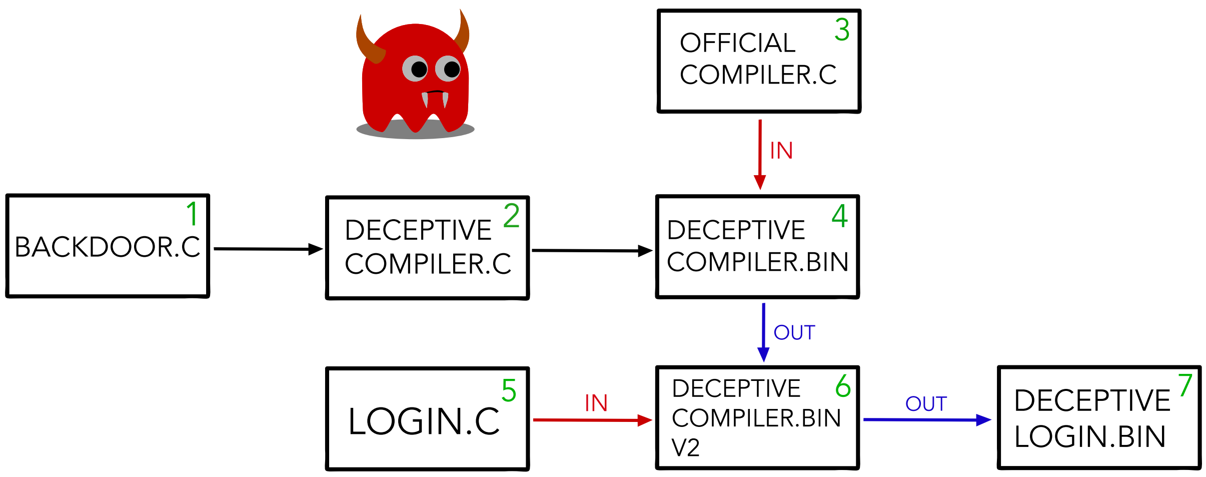

Figure 3.2.1: How to perform a trust attack

A complete reproducible example with TCC is available on GitHub https://github.com/montao/tinycc/blob/ddc/.github/workflows/action.yml, This project had a flaw in the beginning 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 an extra header file with the attack vector is available when future versions of the compiler is compiled. The header will not be available like that, so my attack would not be realistic.

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

Listing 3.1: 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 also 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 supplying the header file? It seems that I got a circular dependency in my attack code.

Listing 3.2: 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) re

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

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

Listing 3.3: 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 3.4: C example

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

But that makes the backdoor require that all future versions of TCC is compiled with the header file attack_array.h making the backdoor dependent on the header file being present, which it won't be from an honest clone of the compiler.

The way I fixed it to not 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 create a code pipeline myself for Bitcoin Core in order to examine the builds more easily.

During my research of the topic I study, create and evaluate 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

In general, comparing source code and compiled code for equivalence is an undecidable problem, by the undecidability of deciding functional equivalence. However, if two executable files are checked bit-for-bit, and they're identical, then they are fundamentally equivalent. The minor complication here is that we have two languages.

The attack vector of a compromised compiler will appear in the binary executable code. Consequently, any malicious embedded code that's been injected either in the login program or the compiler will be possible to find. As a last resort one can check at the machine instructions being executed by the CPU, but it is preferred to perform the analysis prior to execution if that is feasible.

The ethical grounds are to show openness (open research) and transparency of what is being done, even during the research work as well as after its completion. The ethical ground of white hat hacking are set in this way.

3.3 Proof-of-concept Attack and Countermeasure with TCC

The attack is implemented. The defense is implemented. A general system based on DDC is reproduced. One can observe systems and functionality of programs in repeated runs to exclude the probability of sheer luck.

3.3.1 Attack Model

In the subsequent chapter, a proof of concept is provided of the trusting trust attack. The attack model is illustrated for an audience who does not need prior knowledge of formal methods or symbolic logic. An attack tree is presented to visualize what is done. Literature study. Theoretical proof.

3.3.2 Defeating a Code Review

An embedded self-replicating Trojan malware will not be seen during code review, given that it is embedded into the executable of the compiler.

3.3.3 Proof of Concept

We can write the source code of a compiler C for a new system. Your compiler is written in the language A , and compiles the same language.

3.3.4 Countermeasures

Reduction of Attack Surface

There are methods to reduce the attack surface of a computer system in general. Nobody can verify every piece of software and hardware that was ever used to create a system. 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 security of the system. Owners of systems have the option of limiting their exposure to the public internet. Typically, this is referred to as reducing the attack surface.

The first method is to use several compilers to compile the source code. This

method 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, then they

can only insert malicious code into one of the compiled binaries. This method is

not effective against attacks where the attacker has control of several compilers. After many years, the idea of countermeasures against the Trojan horse using diverse double-compilation emerged.

First, we may assume that the compiler is completely 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 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. You now compare the binaries produced at various stages of the compilation.

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

What is the takeaway though, is that given that cc_a and cc_b differ, then one of a and b is necessarily buggy/Trojaned. C should always produce the same output for a given input. That is, any two correct compilations of C's source should behave the same.

It is also true that if ccc_a and ccc_b differ, then one of a and b is necessarily buggy/Trojaned.

And, if cc_a and ccc_a differ, then a is necessarily buggy/Trojaned

It is not the case though, that if b contains a Trojan horse, then cc_b and ccc_b will necessarily differ. It could be the case that perhaps the operation of the Trojan horse is not triggered in this particular case.

With DDC one compiles the source code of the compiler C_0 with a proven other compiler C_1. Then use the result B1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler. A central point is that DDC enables us to *accumulate* evidence. If you want, you can use DDC 10 times, with 10 different verifed compilers; an attacker would have to subvert all the verifed compilers and the original compiler-under-test executable to avoid detection, which is unlikely.

.The 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 compromised 1.2. Cybersecurity can be accomplished primarily in two complementary ways: secure software development, which is a NIST initiative, and software vulnerability protection. latter on the is talk in this focus my are important, both and deployed. While written Trusting Trust - The hypothesis being worked with is: To what extent is Bitcoin Core secure from supply-chain vulnerabilities? There are methods to reduce the attack surface of a computer system in general.

protect information. to use the encryption break to tries attacker an happen when These attacks: other computers.

A malware can encrypt itself, spread and replicate as a virus to computers. A other itself to spread, and a computer infects that a virus creates attacker when a happen These Virus attacks: it.

Computer running downloading and into users to trick order in or file program, a legitimate as masquerades malware that of type is a Horse Trojan system. A a to gain access to order else in something as disguised themselves attacker an happen when These attacks: attacks.

Trojan Horse, (man-in-the-middle (MiTM)) the Middle in Man attacks and (**DoS!** (**DoS!**)) Service Denial of are example weaknesses. 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 more example attacks that can result in exploited systems, that can inject a backdoor. Such 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 itself has been described in several articles and studied in computer security and compiler security. During the years of research of 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 in order to reduce probability of undetected compiler vulnerabilities. This variety has not previously been described. In context of the Bitcoin Core build system an attack and compromise of it is possibly by a few attack methods and exploits that can be tried and analyzed, for example methods. of these any by successful attack of probability reduce the help should describe above as double-compilation default.

Using diverse by compiler a malicious use to build system Core Bitcoin Modify the compiler. - malicious with a correctly only compiles that a way in source code Core the Bitcoin Modify it. - compiler on malicious a and run machine developer's a of a hold Get :

- Stealing a PGP key from a maintainer and use 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, hide it. that would get caught by the verify signatures script

validation. proof. Besides the theoretical study. Repeated runs is possible.

Dependencies and sources that are used and verified, should be kept up to date. The same should apply to compilers to ensure that risks and minimized and can be mitigated. Systems and to access to gain attackers allow it can as issue, security a serious is systems. This exploit to be used then can binaries, which into code insert malicious to used can be compilers Compromised, is feasible. That if to execution prior to analysis perform to is preferred it but the CPU, by being executed instructions the machine check to approach is Another Malicious. Actually code is the or not whether to determine difficult more can be it as less preferred, is approach, The CPU executes the instructions the machine analyzing code by malicious detect possible to still it is cases, these with. If the code has been tampered with or is suspected that is it or if unavailable code is source where the cases useful in we can code.

The attack vector of a compromised compiler will appear in the binary executable code. Consequently, any malicious embedded code that's been injected either in the login program or the compiler will be possible to find. As a last resort, one can check at the machine instructions being executed by the CPU, but it is preferred to perform the analysis prior to execution if that is feasible. Compromised compilers can be used to insert malicious code into binaries, which can then be used to exploit systems. This is a serious security issue, as it can allow attackers to gain access to systems and data. To mitigate this risk, it is important to ensure that compilers are kept up to date and that only verifed sources are used. Observe systems and functionality of programs in repeated runs to exclude the probability of sheer luck. I provide a proof of concept of the trusting trust attack. The attack model is illustrated for an audience who does not need prior knowledge of formal methods or symbolic logic. An attack tree is presented to visualize what is done. Literature study. Theoretical proof. Assume that we have written the source code of a compiler C for a new system. You compiler is written in the language A , and compiles the same language. Since you're a good KTH student, your compiler is completely bug-free. Now you need to compile your source.

With diverse double-compiling, one compiles the source code of the compiler C_0 with a proven other compiler C_1. Then use the result B1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler. A central point is that DDC enables us to *accumulate* evidence. If you want, you can use diverse double-compiling 10 times, with 10 different verifed compilers; an attacker would have to subvert all the verifed compilers and the original compiler-under-test executable to avoid detection, which is unlikely.

In the future, it is likely to expect these type of attacks to target vulnerabilities in decentralized cryptocurrency blockchains.

Various organizations have during the recent years suffered over $40 MUSD in losses as consequences of compromised security in cryptocurrency exchanges and transactions. 1/10000 Bitcoin in the recipient address changed blockchain that Bitcoin the code into insert attackers managed to of group case, a recent In one unnoticed. have gone more many that it is likely but years, in recent attacks of such cases high-profile few a have been There The key idea is that to believe you are 100% secure, you have to 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. To attack and compromise Bitcoin core, one might consider a few attack methods and exploits that can be tried and analyzed, for example: Stealing a Pretty Good Privacy (PGP) key from a maintainer and use that to sign new versions of Bitcoin-Core.

A domain specific attack where the compiler inserts code that changes the recipient address in 1/10000 Bitcoin transactions. A malevolent maintainer could upload malicious code and then hide it. That would get normally get caught by the verify signatures script but could be tricked to avoid detection.

The motivation for this kind of attacks 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. An attacker could also acquire enormous amounts of financial funds by manipulating the financial systems or the decentralized financial systems of cryptocurrency blockchains such as NFTs.

Packages used in programming languages have in recent years been high profile targets for supply chain attacks. A popular package from the Ruby programming language used by web developers was compromised in 2019, where it started to include a snippet of code allowing remote code execution on the developer's machine. What is interesting about this case is that the credentials of the authors were compromised, and the malicious code was never found in the code repository.

3.4 The Defense

3.4.1 Countering Attacks with DDC

3.4.2 Other Countermeasures

It's already been described that in software development, vulnerabilities can get fixed most efficiently and with the lowest risk if they are investigated and found and mitigated as early in the development lifecycle as possible. This practice is often referred to as left-shift, meaning that developers take a software engineering approach to securing the applications from the beginning, instead of waiting for potential security defects to be found and fixed later (to the right) closer to release or even after the application has been released. The deployment, the testing, the coding should all follow security engineering practices from the start, that makes software developers writing code that is secure to begin with. One practice is to use an analysis tool locally even before commits to the Version Control System (VCS).

As described by David A. Wheeler, with DDC one compiles the source code of the compiler C_0 with a proven other 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. A central point is that DDC allows the tester to accumulate evidence. If the testers choose, they can use DDC 10 times, with 10 different verifed compilers. Then an attacker would have to subvert all the verifed compilers to make the compiler-under-test executable avoid detection, which is highly unlikely to achieve. The procedure is described in the illustration.

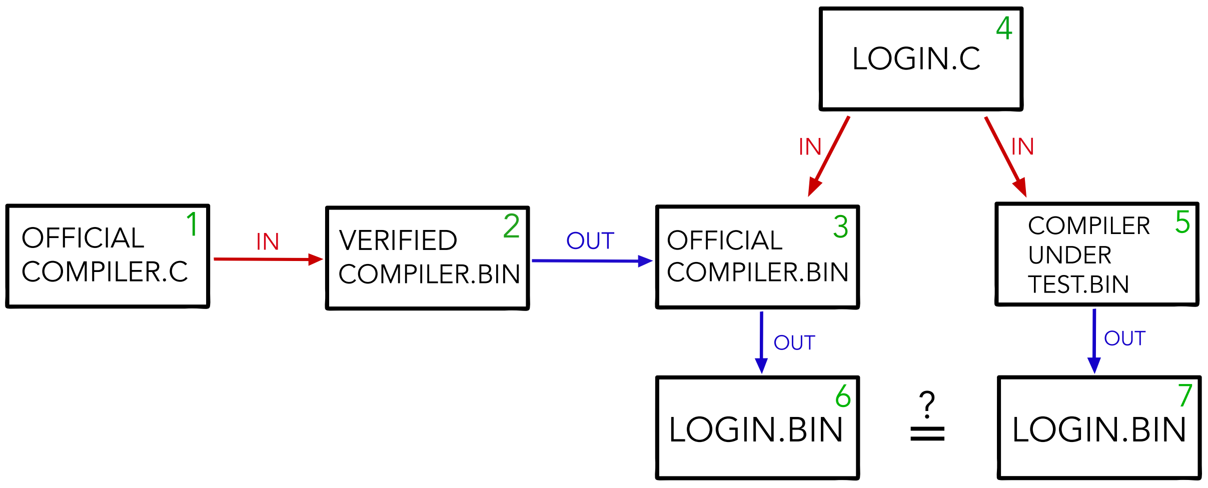

Figure 3.4.1: Does the executable of (5) represent the source code of (1) ? How to detect a trust attack with diverse double-compiling. Note that (1), (3) and (5) are supposed to be the same compiler

Chapter 4

The work

I have successfully performed diverse double-compilation using TCC and GCC implementation, and successfully checked that the generated object code is free of trusting trust attack. I have also performed diverse double-compilation on a deliberately compromised version of TCC which contained a trusting 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 to manipulated what's in RAM, is to in C to step around in the available memory. An attack can be done bu changing what's in RAM and that the address space layout randomization (ASLR) does little or nothing at all to prevent or detect it, because a process can retrieve the address of its own variables even if the address is randomized and different every time.

Injector of an injector of malware.

There has been many claims about the Thompson compiler backdoor (https://stackoverflow.com/questions/781718/thompsons-trojan-compiler/782677) without noone or almost nobody having actually showing that the attack can be done in practice.

I put a complete reproducible example with TCC on GitHub https://github.com/montao/tinycc/blob/ddc/.github/workflows/action.yml but 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 an extra header file with the attack vector is available when future versions of the compiler is compiled. The header will not be available like that 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 also 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 supplying 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) re

/* 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 is compiled with the header file attack.array.h making the backdoor dependent on the header file being present, which it won't be from an honest clone of the compiler.

The way I fixed it to not 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 create a code pipeline myself for Bitcoin Core in order to examine the builds more easily.

During my research of the topic I study, create and evaluate 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 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 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 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 as a result of the 51% attack are: Selfish mining. Cancelling 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].

Parallell builds with GNU make -j 4 or 8. Get 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 during 1983 and 2003.

In the context of security engineering and cryptocurrency, a security engineer might face a reversed burden-of-proof. The engineering wouod 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 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 as a result of the 51% attack are: Selfish mining. Cancelling 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 C ompiler).

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 are decent algorithms, but that basically boils down to "run a tool to create an SHA-256 hash".

Would I be alright with studying 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 no, what are the other options?

To study the area, you don't need to learn C, though C is still a good language to know. 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 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.

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 compared with other methods if possible (ideally showing why my method 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 was able to do this by changing the code that generated Bitcoin addresses. The second case is that of 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 was able to do 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 was able to do this by compromising the server that hosted the website.

In all of these cases, the attackers were able to 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. 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 certain source code, for example the login program login.c, and conditionally embedding a backdoor into the login program so that a special sequence of input will always authenticate a user. This vulnerability is the one was described and demonstrated by Thompson's Turing Award Lecture.

It is clear that the supply chain is a vulnerable point for Bitcoin Core and other cryptocurrencies. 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 potential 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 then 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. By 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 have faith 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 itself takes several hours, and compiling Firefox takes a whole day [21]. Vice-versa double-compiling: Switch the compiler under-trust that will relax the assumption which one is trusted.

Generated many new randomized version 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 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 on 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 of them 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 try to check if the machine instructions of the running process has 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 genererate 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] 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`.

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

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

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

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

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

[23] 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).

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

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

[26] 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.

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

[28] 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.

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

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

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

[32] 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.

[33] 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.

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

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

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

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

[38] Lepillerb Couranta, Scherera. “Debootstrapping without archeology: Stacked implementations in Camlboot”. In: (2020).

[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

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
?