Chapter 2

Related Work

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

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

In the years after Thompson's article, the technologies of compiler security, dependency tracking, and supply-chain cybersecurity receives even more interest from academic researchers and commercial businesses. The potential for deceptive malware to propagate in object form without being seen implies that the risk of enormous

damage is technically possible. The build system in many cases relies on the GNU Compiler Collection (GCC) [56]. A possible scenario is that a particular instance of the GCC contains self-replicating malware and that instance compiles itself to the next version. If that compiler compiles Bitcoin Core, a single individual, government organization, or some capable non-government organization (NGO) would be able to manipulate transactions centrally and arbitrarily. The ideas and executions were studied during the 1970s by Karger & Schell. Later, Ken Thompson pointed out the attack in the 1980s. Since then, practically no known researchers have made progress on the topic between the mid-1980s and the mid-1990s. The slow progress was partly due to the abundance of ideas and techniques for mitigation [27, 57]. Only in 1998 did Henry Spencer suggest a countermeasure [54].

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

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

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

38].

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

2.1 The Threat Landscape

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

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

2.1.1 Software and Hardware Backdoors

Mistakes or malice can result in software vulnerabilities and compromised system security. It can happen during the design or during construction. Mistakes and malice sometimes cause backdoors in software and hardware during several steps of

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

2.1.2 Self-Replicating Compiler Malware

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

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

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

2.1.3 Deceptive Hardware Trojans

Zhang et al. conducted research mostly about hardware trojan (HT) [69]. They state that vulnerabilities lead to the risk of novel and modified exploits on computer systems. Despite extensive work to prevent the attacks, the attacks are feasible through compromised hardware. Existing trust verification techniques are not effective enough

to defend against hardware Trojan horse backdoors found in field-programmable gate array (FPGA), for example, used in military technology.

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

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

2.2 Countermeasures to Backdoors

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

2.2.1 Response-Computable Authentication

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

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

2.2.2 Isolating Backdoors with Delta-Debugging

Schuster and Holz have written about ways to reduce the risk of software backdoors. Their work emphasizes specific debugging techniques that utilize decision trees in binary code [48, 67]. They introduced a debugging technique called delta-debugging, making it possible to detect which system parts are possibly compromised and perform further analysis steps. Their software uses GNU Debugger (GDB) and can analyze binary code for x86, x64, and MIPS architectures. Their software, named WEASEL, is available to the public on GitHub [47]. Their article then gives the results from practical test cases to show that the WEASEL can detect and disable both malware found in

actual incidents and malware deliberately created for specific testing purposes. The authors do not, however, describe how to detect a possible vulnerability in their dependence on the GDB.

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

2.2.3 Firmware Analysis

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

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

2.3 Secure Compilation

Secure compilation can ensure that compilers preserve the security properties of the source programs they take as input in the target programs they produce [41]. Secure development is broad in scope; it targets languages with various features (including

objects, higher-order functions, memory allocation, and concurrency) and employs various techniques to ensure preserving the security of the source code in the generated executable and at the target platform.

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

2.3.1 Debootstrapping

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

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

2.3.2 Self-Hosted Systems

There have been findings of an undocumented extra microprocessor in specific systems [19]. There are claims that the only system that can be entirely trustworthy is the one where everything used to create the system is available in the system itself. Somlo describes such a self-hosted system in a recent research report [53]. Somlo notes that

the lengths of supply chains are getting longer and longer. Consequently, according to Somlo, it increases the complexity of the problem of checking whether a system is trustworthy. Somlo describes an approach with steps to perform DDC. Somlo takes a broader scope and suggests an entirely self-hosted independent system with FPGA capable of checking another system. The idea is to have the system that performs the check also reduce the risks of being compromised in the linker, loader, assembler, operating system, or mainboard.

2.4 Testing and Verification

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

2.4.1 Verification of Source Code and of Compiler

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

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

conduct an analysis [18, 22].

2.4.2 Diverse Double-Compilation

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

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

It was suggested to compile the compiler using itself and a different compiler, generating two executables. These two executables can be assumed to behave under test because they came from the same source code. However, they will not be identified as equal because the two different compiler manufacturers have used different techniques to generate the compiled executables. Now, one can use both binaries to recompile the compiler source, generating two outputs. Since the binaries

should be identical, the outputs should be bit-by-bit identical. Any difference indicates either a critical defect in the procedure or malware in at least one of the original compilers [54].

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

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

David A. Wheeler then states that one consequence of the vulnerability and countermeasures is that organizations and governments may insist on using

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

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

Listing 2.1: Example of condition for DDC

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

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

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

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

2.4.3 Reproducible Builds

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

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

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

2.4.4 Fuzz Testing

In his writing about DDC, Wheeler claims randomized testing or fuzz testing would unlikely detect a compiler Trojan [60]. Fuzz testing or randomized testing tries to find software defects by creating many random test programs (compared to numerous monkeys at the keyboard). When testing a compiler, the compiler-under-test gets compared with a reference compiler. The test outcome will depend on whether the two compilers produced different binaries. Faigon describes this approach [20]. The approach has found many software bugs and compiler errors, but it is improbable to detect maliciously corrupted compilers. Suppose such a corrupted compiler diverts from its specifications in only 1/1000 executions of its target (as would be the case in a cryptocurrency system that sends every thousand transactions to a different receiver). In that case, it becomes evident that tests are unlikely to detect the bug in the compiler. For randomized testing to work on compiler-compilers, the input would need to be a randomized new version of the compiler, which no one has attempted. The situation

will be that a compiler gets compiled, and the Trojan horse only targets particular input, such as the compiler itself. However, Wheeler's analysis does not explain why we cannot input the compiler's source code into a compiler binary and then do fuzz testing with variations.

2.5 Secure Development for Cryptocurrency

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

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

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

In a recent news article, Sharma writes that supply-chain attacks have recently occurred against the blockchain [51]. The software supply chain attacked a DeFi

platform for cryptocurrency assets. A committer to the codebase had included a vulnerability in the platform's production version. It raised questions about quality assurance for source code contributions and whether the review process was the real problem in this case.

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

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

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

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

the Bitcoin Core project and that there is room for further improvement.

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

2.6 Summary

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