Improving Trust in Software through Diverse Double Compilation and Reproducible Builds

Yrjan Skrimstad

10th September 2018

University of Oslo

Introduction

- Trust is *the belief* that someone or something is good, honest, safe or reliable.
- Parallel trust combinations is a way to say that there is an increase in trust by transitivity if there are multiple trusted paths to the target.
- Will discuss two techniques using *deterministic build processes* to help gain trust in compiled code:
    - *Reproducible builds*: used to gain trust in distributed compiled code, where the source code is available.
    - *Diverse Double Compiling*: used to gain trust in compilers.

The Problem

- How can we trust that compiled code accurately reflects the source code?
- We are here interested in malicious behaviour.
- Comparing source code and compiled code for equivalent behaviour is an *undecidable problem* for Turing-complete languages.
- Major problem for distributors of open-source software, however it can also be a problem for distributors of proprietary software.

The Problem - for open-source software

- Linux distributions (and other projects) can distribute software in compiled form to millions of users.
- It has often been said that open-source software is easier to trust, as the source code is available. This makes it somewhat possible to review the software.
- Software is often distributed in binary form.
- If we cannot trust that the compiled code is a non-malicious result of the reviewed source code. How does having the source code actually help us?

The Problem - for proprietary software

- Perhaps seen as less of a problem for the end-user, as everything is more based on a ‘blind trust’. The users will typically not have any access to the source code, anyway.
- However, attacks such as malicious compiler attacks may still be a problem.
- Malicious software may be spread from the software distributor, while the distributor is unaware of the problem.
- The ability to use techniques to verify that source code and compiled code are identical is necessarily moved to the distributor.

The Problem - how can compiled code be subverted

- Pre-compilation: it is trivial to alter source code and then claim that the compiled code is based on non-altered source code.
- In-compilation: a malicious compiler can change the result of the compilation, and thus create malicious compiled code.
- Post-compilation: it is possible to alter the compiled code itself.

Malicious compilers

- One way to create malicious compiled code *in-compilation*.
- Purposefully miscompiles software.
- Can have great consequences: allows for software distributors to unknowingly spread malware.
- It is not a fantasy. We have seen multiple compiler attacks ‘in the wild’, such as: W32.Induc and XcodeGhost.

Malicious compilers - XcodeGhost

- Example of a real world compiler attack.
- Discovered in September 2015. Targeted Apple Xcode.
- Spread in China through local file sharing services.
- Believed to have infected at least 3418 different iOS applications.
- Most well-known infected application was WeChat, version 6.2.5. At the time WeChat had about 570 million active users daily, it is not known how many used the iOS version of the application.

Malicious compilers - self-replicating attack

- Old idea, first known mention in 1974 by Karger and Schell in a Multics security evaluation. Called a ‘compiler trap door’.
- Most famously discussed by Thompson in 1984, from there often known as the ‘Trusting Trust attack’.
- Hypothetical attack, the attack has not been documented ‘in the wild’.

Malicious compilers - self-replicating attack: general idea

- The attack will target some part of the system, perhaps a login daemon where it can insert a backdoor when compiling.
- It will also target the compiler itself. Infecting the compiler upon compilation.
- Through this the attack can self-replicate when the user attempts to recompile the compiler.
- The attack does not require the compiler to be self-hosted, and can also create a cycle based on multiple different compilers.

Malicious compilers - self-replicating attack: illustration

$$\begin{tikzcd}
 & c_P \arrow[d, "\text{(Compiler with trap door)}"] \\
s_C \arrow[r] \arrow[dr] \arrow[dddr, "\text{(Source code without trap door)}"'] & \boxed{1} \arrow[d, "\text{(Compiler with trap door)}"] \\
 & \boxed{2} \arrow[d, "\text{(Compiler with trap door)}"] \\
 & \cdots \arrow[d, "\text{(Compiler with trap door)}"] \\
 & \boxed{n} \arrow[d, "\text{(Compiler with trap door)}"] \\
 & {}
\end{tikzcd}$$

Deterministic build processes

- Requirement for reproducible builds and diverse double compiling.
- Goal is to create bit-for-bit identical builds.
- Often less trivial than one might think, some typical issues are:
    - Parallel compilation: where order of completion can create differing results.
    - Build path: storing of paths in compiled code.
    - Pseudo-random behaviour: typically generation of identifiers.
    - Included timestamps: often included for version or debug information.

Reproducible builds

- Attempt to create a verifiable path between source code and compiled code using *deterministic build processes*.
- Typically requires specific information about the build environment to create the same result across different systems.
- Allows any third-party to verify that they get the same result when compiling the source code.

Reproducible builds (continued)

- Removes the build and distribution process as a single point of failure and allows for increased trust through *parallel trust combinations*.
- Removes some of the incentive to attack developers to insert malicious software.
- Can also help detect some bugs (and has), such as varying constants set at compile time.

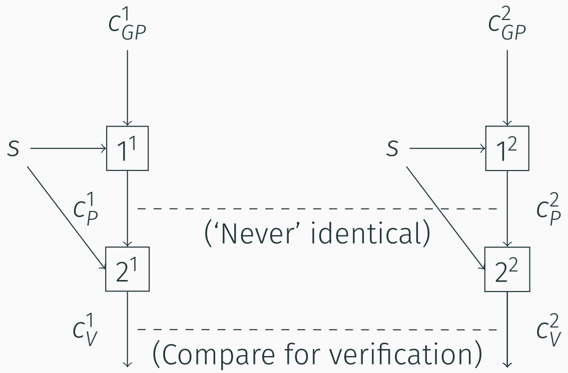
Reproducible builds (continued)

- New and major effort in multiple open-source projects such as Debian, openSUSE, Bitcoin and The Tor Project.
- openSUSE at 95% reproducible packages, Debian at 94% reproducible.
- Great help to be able to prove that distributed binary packages are not maliciously modified.
- Weakness to malicious compiler attacks, if the malicious compiler has already spread to the user.

Diverse Double Compiling

- Technique to verify the absence of self-replicating behaviour in compilers. Can detect a self-replicating compiler attack.
- First properly described by Wheeler in 2005, previously mentioned on mailing lists as an idea.
- Works by generating the same compiler from two different compilers. This will result in two compilers that should be semantically identical, however the compiled code will be different.
- These two semantically identical compilers will then be used to generate two new compilers, these compilers should be identical.

Diverse Double Compiling: self-hosting compiler



Diverse Double Compiling (continued)

- If they are identical we have shown that the two original compilers either inserted the same self-replicating attack or no self-replicating attacks.
- If they are not identical it can be hard to say which compiler has inserted self-replicating behaviour.
- Does not prove that the ‘grand-parent compilers’ are without self-replicating behaviour, only proves that it was not inserted.
- Sometimes done with a simpler compiler that is easier to trust than an industry-strength compiler.

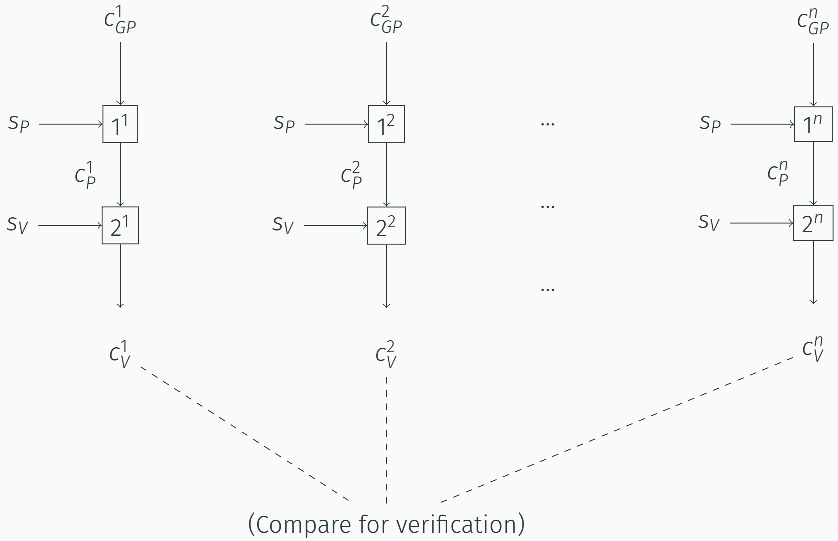
Diverse Double Compiling: general

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

Diverse Double Compiling: more than 2 grand-parent compilers

- Not previously described, but mentioned as a possibility by Wheeler in 2010.
- Gives some added abilities compared to ‘regular’ Diverse Double Compiling:
    - Added trust in the final generated compiler, utilising parallel trust combinations.
    - Some ability to say which grand-parent inserted self-replicating behaviour.
- Nevertheless, it requires more compilers.

Diverse Double Compiling: more than 2 grand-parent compilers



Implementation: language selection

- Rust was a bit too slow to compile for iterative development.
- Haskell was not deterministic, which makes Diverse Double Compiling impossible.
- Go was deterministic and very fast to compile:
    - Very fast to compile, could recompile the entire compiler in minutes.
    - Deterministic, as long as the compiler was compiled from the same working directory.
    - Curiously enough, authored by in part by Ken Thompson.
- I chose Go.

Implementation: quine

- Used techniques known from quines to implement the attack.
- Quines comes from Quine’s paradox, named after the American logician Quine.
- Indirectly self-referential, according to Hofstadter in the book Gödel, Escher, Bach (1979).
- Example: ‘“Is a sentence fragment” is a sentence fragment.’
- A quine in computer programming is a program that can output itself, without any inputs.

Implementation: quine (continued)

```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)
}
```

Implementation: insertion of attack

- Needed a way to detect what program we are compiling.
- Needed a way to modify the compilation of the program.
- Simplest way: read and modify source code before the tokenisation.
- PoC: modify ‘Hello, World!’.

Implementation: modify ‘Hello, World!’

```go
if strings.Contains(base.filename, "hello.go") {
    fileContent, err := readFile(base.filename)
    (...)
    modContent := strings.Replace(fileContent,
        "Hello, World!", "There is a trap door.", 1)
    err = writeFile(modPath, modContent)
    (...)
    newFile, err := os.Open(modPath)
    (...)
    var p parser
    p.init(base, newFile, errh, pragh, mode)
    p.next()
    return p.fileOrNil(), p.first
}
```

Implementation: self-replicate

```
modifiedString := `(The code to duplicate)`
(... Read the file into the variable 'content', iff syntax.go ...)
if !strings.Contains(content, "sNzrBzaIxgSNMmMuPaE3") {
    (...)
    addition := fmt.Sprintf("}()\n\n\tmodifiedString "
                            + ":= %c%s%c\n\n\t%s",
                            96, modifiedString,
                            96, modifiedString)
    content = strings.Replace(content, "}()",
                              addition, 1)
    (...)
    err = writeFile(modPath, content)
    (...)
    newFile, err := os.Open(modPath)
    (...)
    var p parser
    p.init(base, newFile, errh, pragh, mode)
    p.next()
    return p.fileOrNil(), p.first
}
```

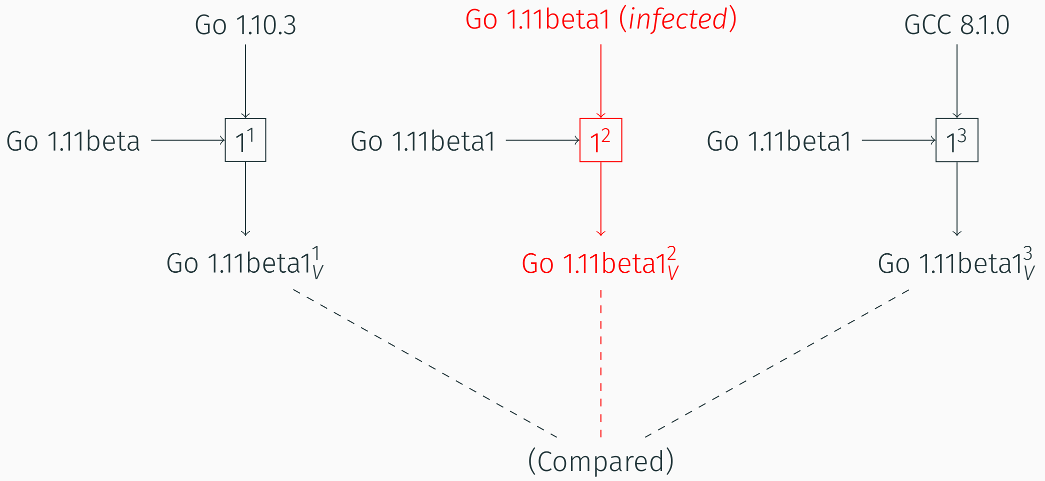
Implementation: mutation of programs

- Used technique known from quines to replicate into a new compiler.
- It works, both self-replication and modification of another program.
- Full implementation available on Github.

Diverse Double Compiling: real attempt

- Used a variant of DDC with three grand-parent compilers:
  - Go 1.10.3
  - Go 1.11beta1
  - GCC 8.1.0
- Go 1.11beta1 had self-replicating attack inserted.
- The target compiled from source was Go 1.11beta1.
- Go has a multi-stage compilation process, so it already compiles the compiler again using the result from the first compilation.
- Checked for identical results using a cryptographic hashing algorithm.
- Could see which grand-parent had inserted the self-replicating attack.

Diverse Double Compiling: real attempt illustration



Go 1.10.3
Go 1.11beta1 (*infected*)
GCC 8.1.0
Go 1.11beta
1^1
Go 1.11beta1
1^2
Go 1.11beta1
1^3
Go $1.11\text{beta}1_V^1$
Go $1.11\text{beta}1_V^2$
Go $1.11\text{beta}1_V^3$
(Compared)

Summary

In this thesis I have:

- Explained how compiled code can be altered so that it is *not* equivalent to source code.
- Discussed two real-world examples of malicious compiler attacks.
- 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 capable of identifying which compiler introduced an eventual self-replicating attack.
- Discussed how reproducible builds and Diverse Double Compiling can increase trust in compiled code, by utilising parallel trust combinations.

Questions?