Lec 04: On Trusting Trust

CSED415: Computer Security

Spring 2024

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Welcome survivors!
  - 12 students and one auditor

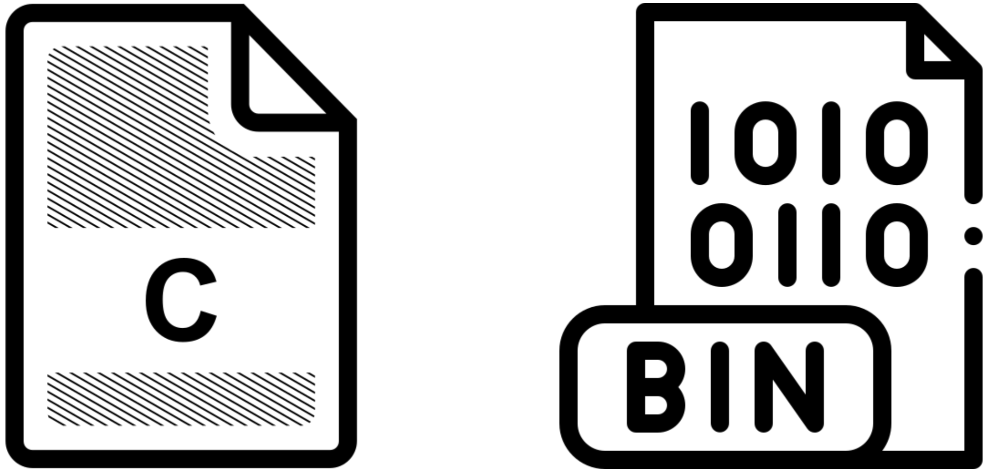
- Please form six teams for the team project
  - 2 students per team
  - Start searching for your teammate now
  - Finalize your teams by the end of next week (March 10th)
    - Refer to PLMS for instructions

Recap

- Defensive programming and secure coding guidelines
  - Buggy code is the root of evil
    - Do you remember any of the rule?
    - Have you tried reviewing your own code?
  - From Lec 03 slides:
    - Conforming to the secure coding standard is necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in C
    - Today we discuss why, and what should we do then

Source code vs. Binary

- Q) Given both source code and binary of a program, which one do you need to analyze if you want to know if the program is safe to run?



Source code is not always available

- Malware
- Commercial software
- Firmware binary extracted from a board
- …

What about open-sourced programs?

Fun fact

- Even when the source code is available, security experts often analyze **binaries**
  - Why? → today's topic

Key question

- You are given the entire source code of a program.
  Can you find all potential vulnerabilities in the program
  by analyzing its source code?

NO WAY!

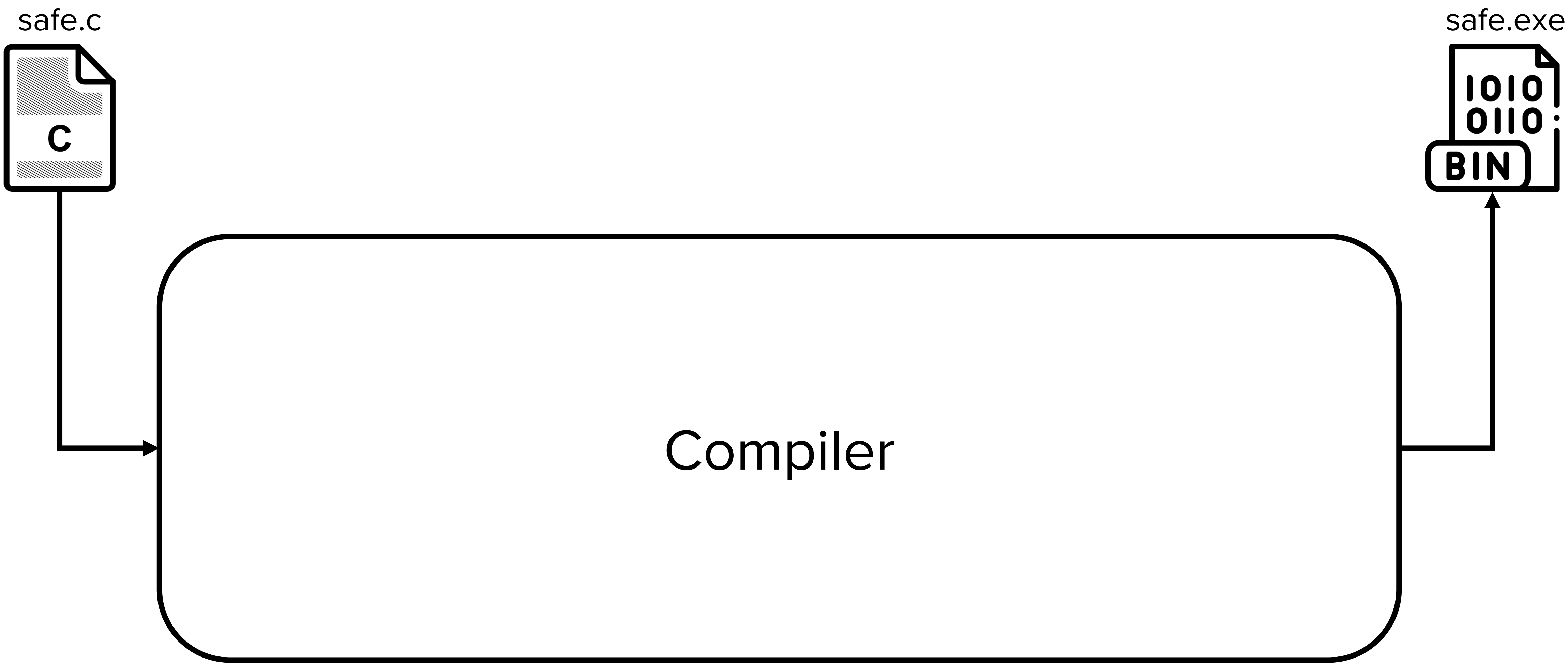
Ken Thompson

"Reflections on Trusting Trust"

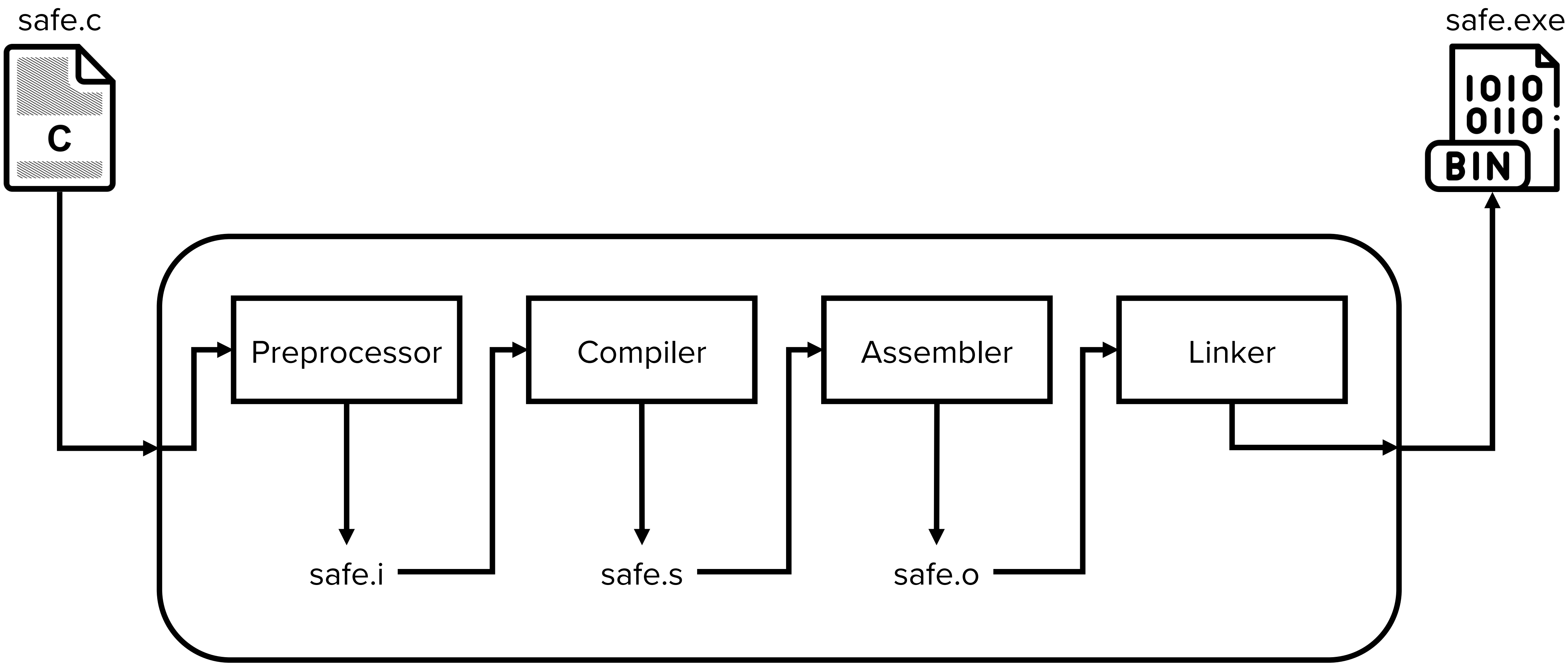
Communications of the ACM, 1984

Trusting Trust

Compiler 101

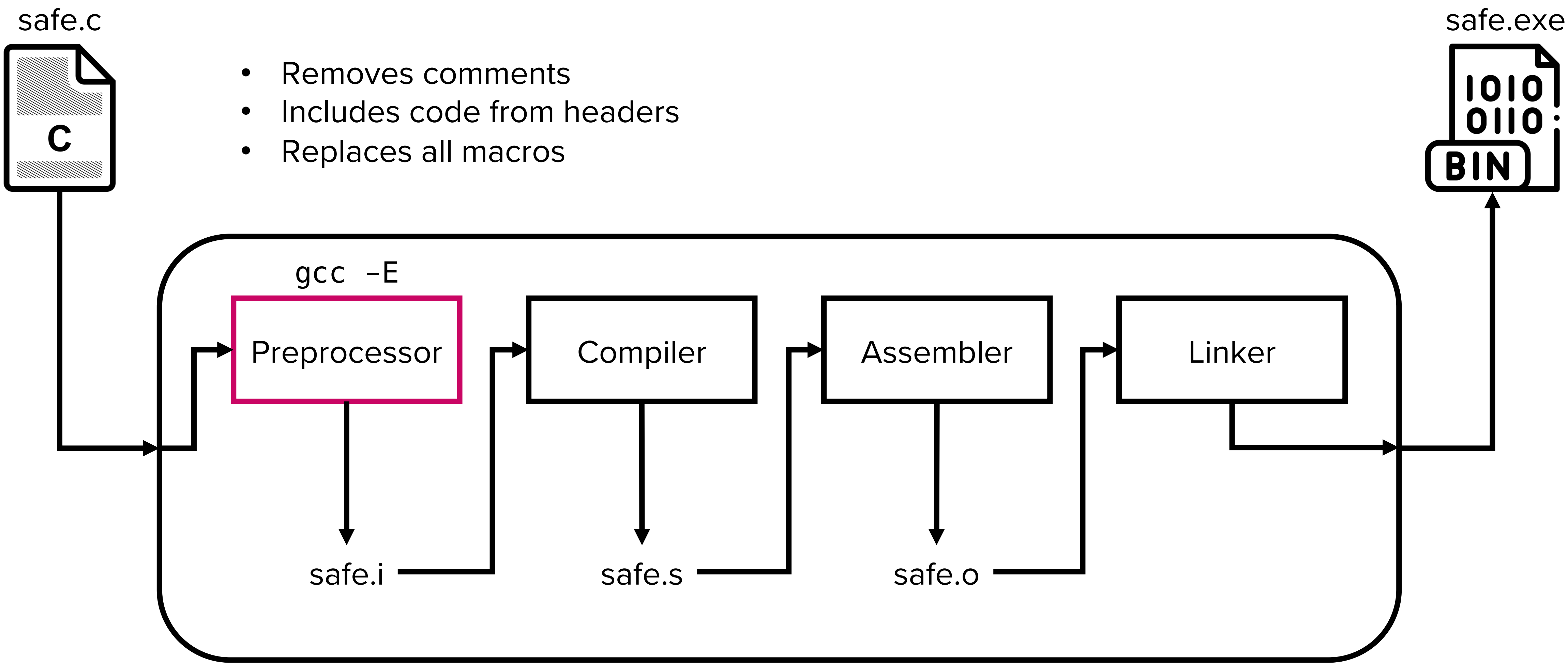


Compiler 101



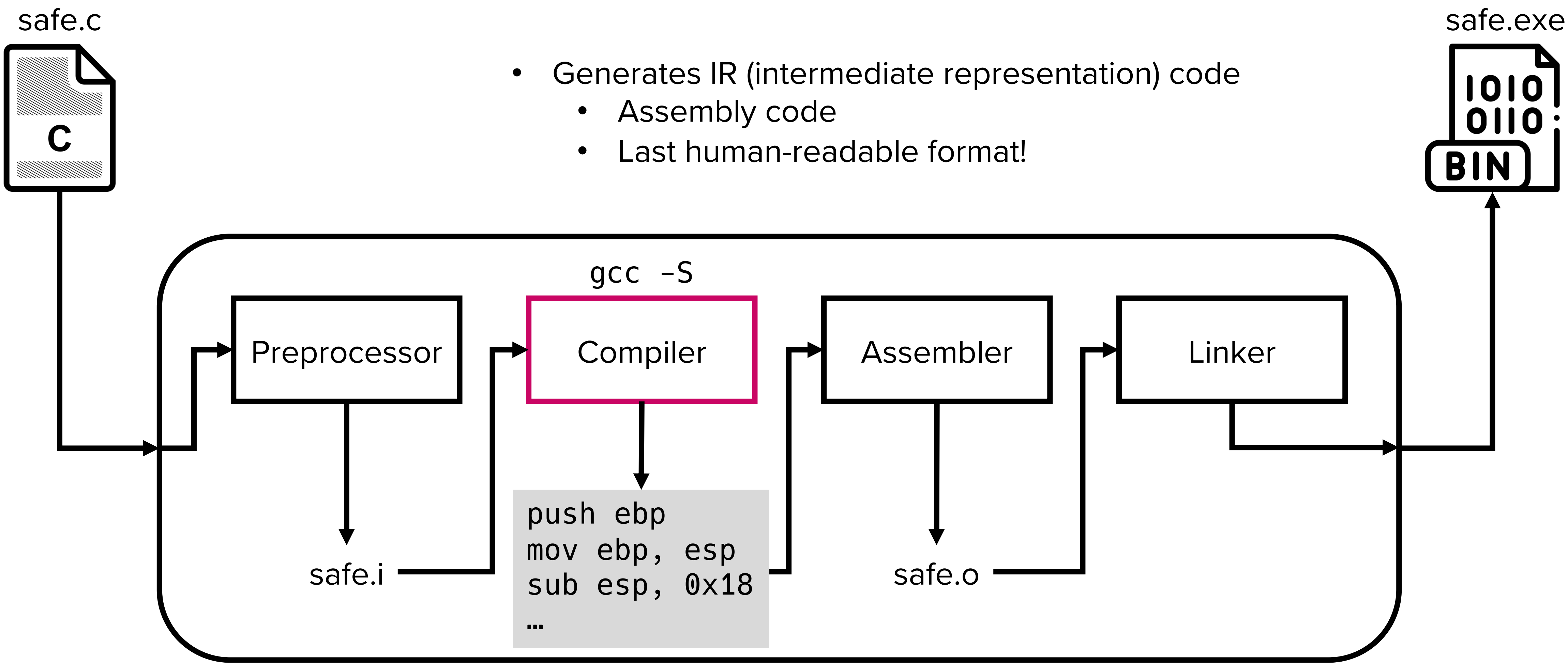
Compiler 101

- Removes comments
- Includes code from headers
- Replaces all macros

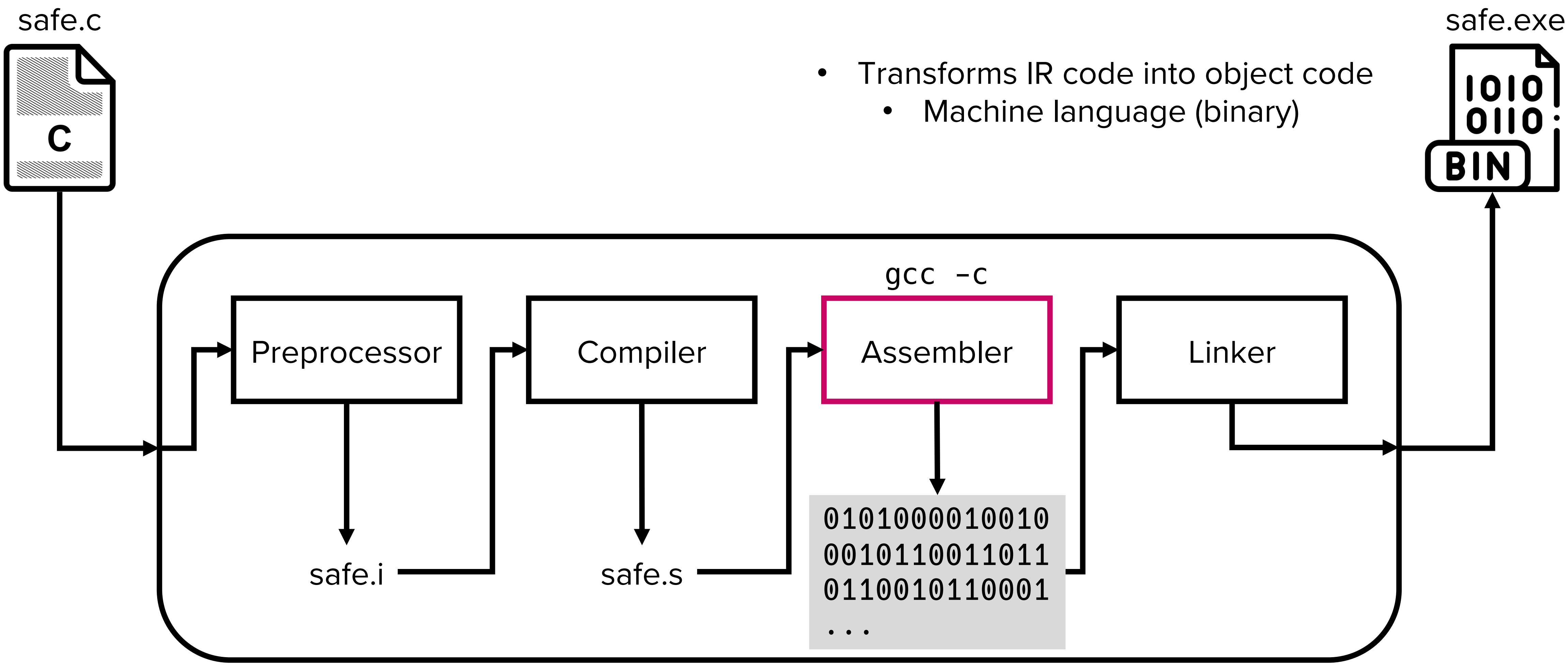


Compiler 101

- Generates IR (intermediate representation) code
  - Assembly code
  - Last human-readable format!



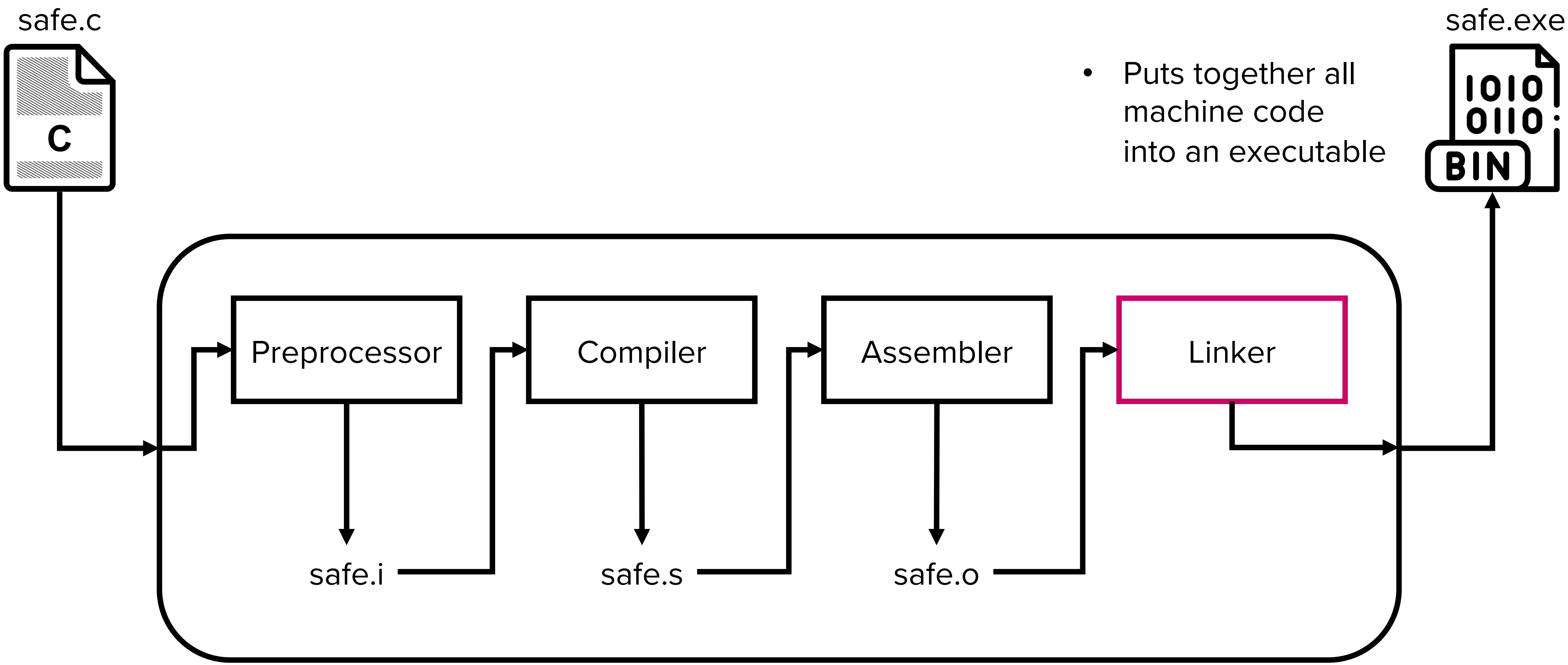
Compiler 101



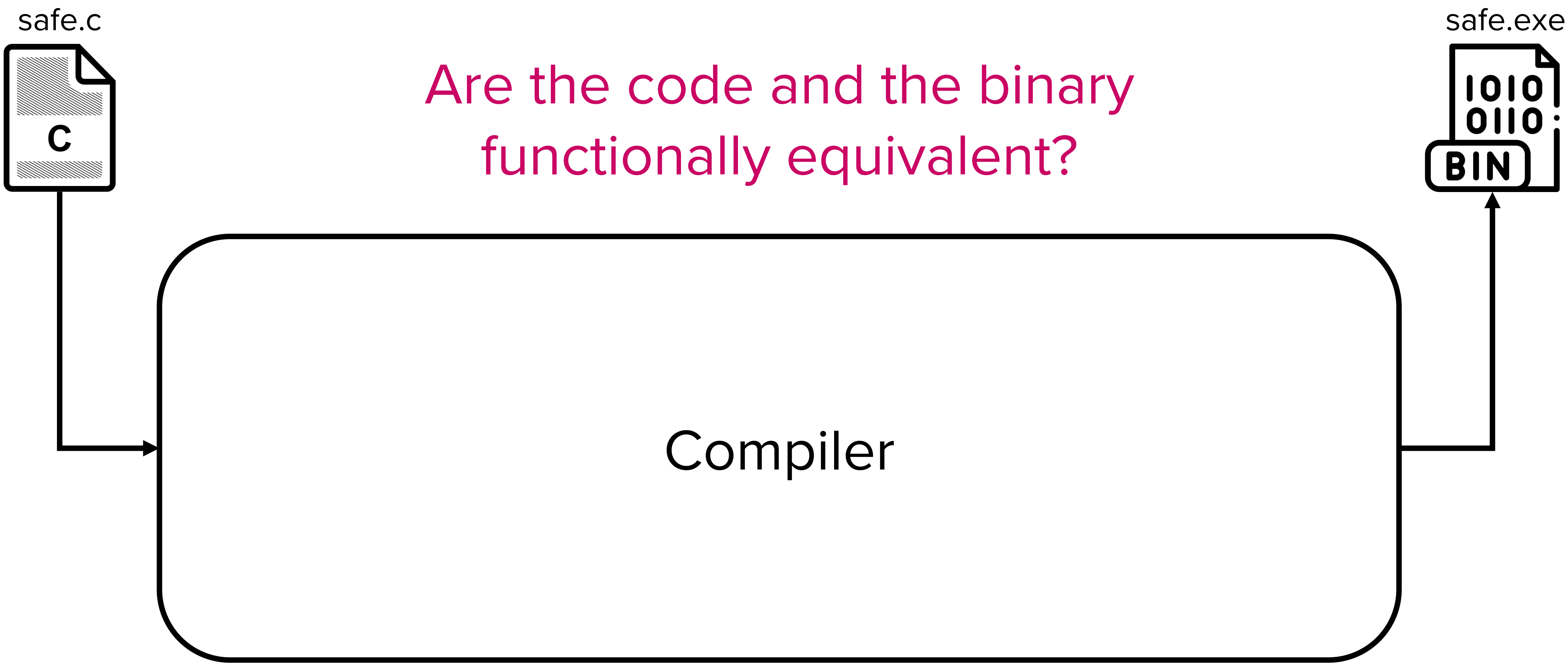
- Transforms IR code into object code
  - Machine language (binary)

Compiler 101

- Puts together all machine code into an executable



Question: Can we trust compilers?



Chasing down a rabbit hole

- How do we know a program is safe?
  - Inspect the program's source code 😎
- But isn't the code compiled by a compiler?
  - Inspect the compiler's source code 🤨
- But isn't the compiler compiled by another compiler?
  - Now what? How deep do we go down? 🤯

Reflections on Trusting Trust

- Ken Thompson and Dennis Ritchie
  - Won Turing Award in 1983 for their work on Unix Operating System
- Thompson presented *"Reflections on Trusting Trust"* in his acceptance speech (1984)
  - https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf
  - Also available on PLMS

TURING AWARD LECTURE

Reflections on Trusting Trust

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.

Attack objectives

1. Create a malicious compiler that injects a backdoor to a specific program
2. Do not leave a trace in the source code of the compiler

Thompson Compiler – stage 1

- Writing a self-reproducing program (Quine)

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};
```

```
/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
```

// print the array definition

// prints the rest (comments and main)

Thompson Compiler – stage 2

- "Training" a compiler

[Compiler v1 code]

```
c = next();

if(c == '\\') {
  c = next();
  if(c == 'n')
    return('\n');
}
```

→ Escape sequence handler
(processes a two-char sequence, '\' and 'n', and turns it into '\n')

compiler

comp v1

Thompson Compiler – stage 2

- "Training" a compiler

[Compiler v2 code]

```
c = next();

if(c == '\\') {
  c = next();
  if(c == 'n')
    return('\n');
  if(c == 'v')
    return('\v');
}
```

→ Added handling logic for vertical tab ('\v')

compiler

comp v1

Thompson Compiler – stage 2

- “Training” a compiler

[Compiler v2 code]

```
c = next();

if(c == ‘\\’) {
  c = next();
  if(c == ‘n’)
    return(‘\n’);
  if(c == ‘v’)
    return(‘\v’);
}
```

→ Added handling logic for vertical tab (‘\v’)

compiler

comp v1

Parse error: ‘\v’ unknown

Result

Thompson Compiler – stage 2

- “Training” a compiler

[Compiler v2’ code]

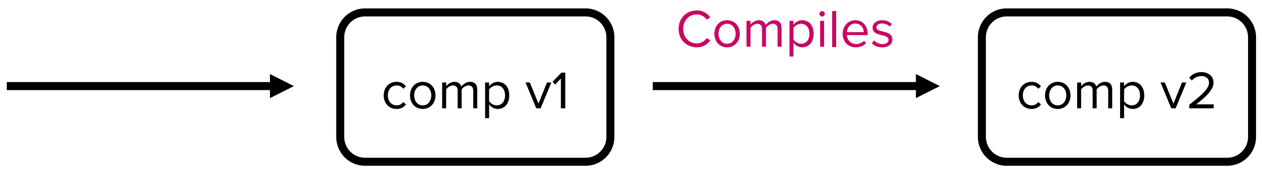
```
c = next();

if(c == ‘\\’) {
  c = next();
  if(c == ‘n’)
    return(‘\n’);
  if(c == ‘v’)
    return(11);
}
```

→ use 11 instead (corresponds to ascii char ‘\v’)

compiler

Result



Learned what \v means

Thompson Compiler – stage 2

- "Training" a compiler

[Compiler v2 code] → compiler → Result

```
c = next();

if(c == '\\') {
  c = next();
  if(c == 'n')
    return('\n');
  if(c == 'v')
    return('\v');
}
```

comp v2

→ Now comp v2 is "trained" and understands '\v'

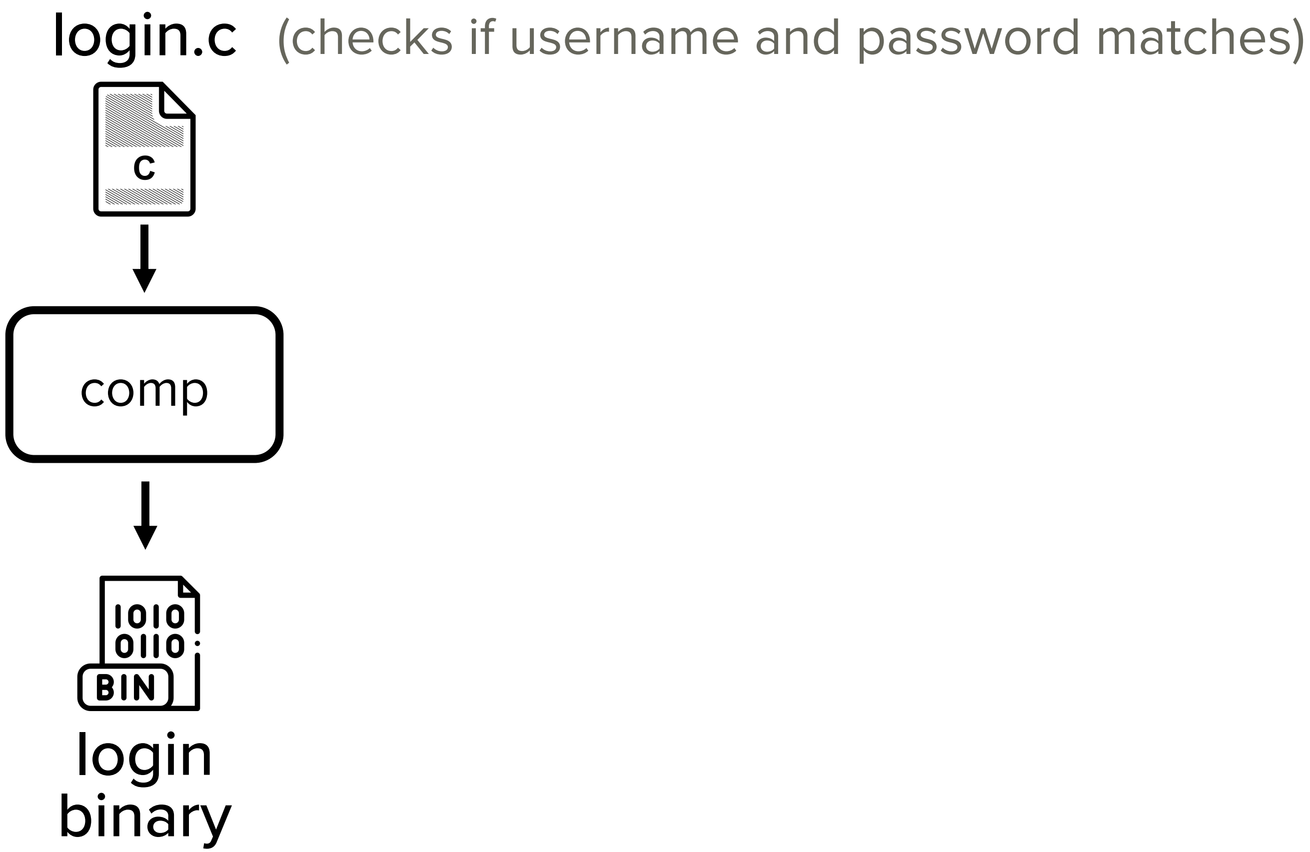
→ Note: What it learned ('\v' == 11) does not appear on the code

Thompson Compiler – stage 3

- Injecting a backdoor to a compiler

[Normal compiler code]

```
void compile(char *s)
{
  /* ... */
}
```

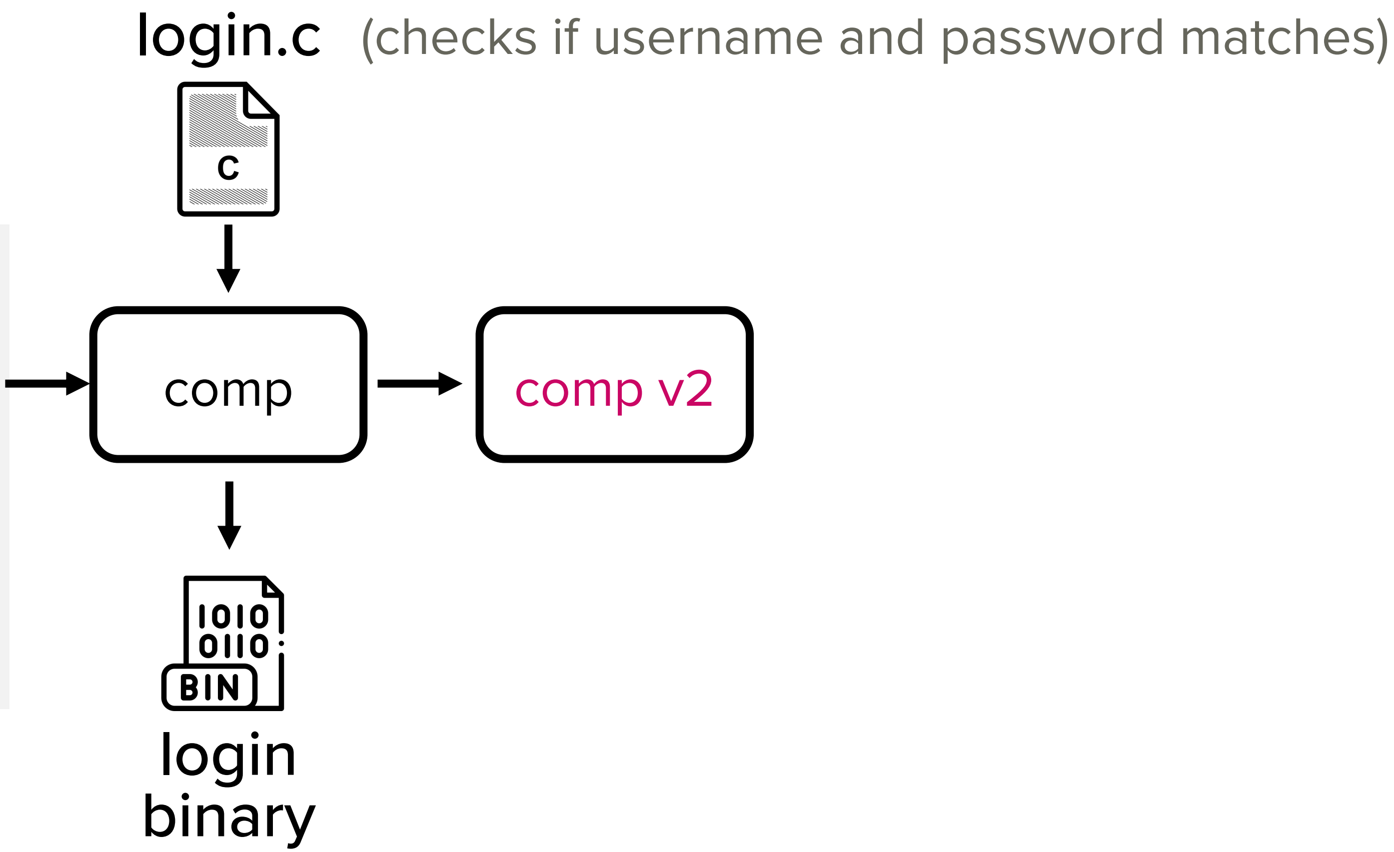


Thompson Compiler – stage 3

- Injecting a backdoor to a compiler

[Compiler v2 code]

```
void compile(char *s)
{
  if(match(s, “login”)) {
    compile(“backdoor”);
    return;
  }
  /* ... */
}
```



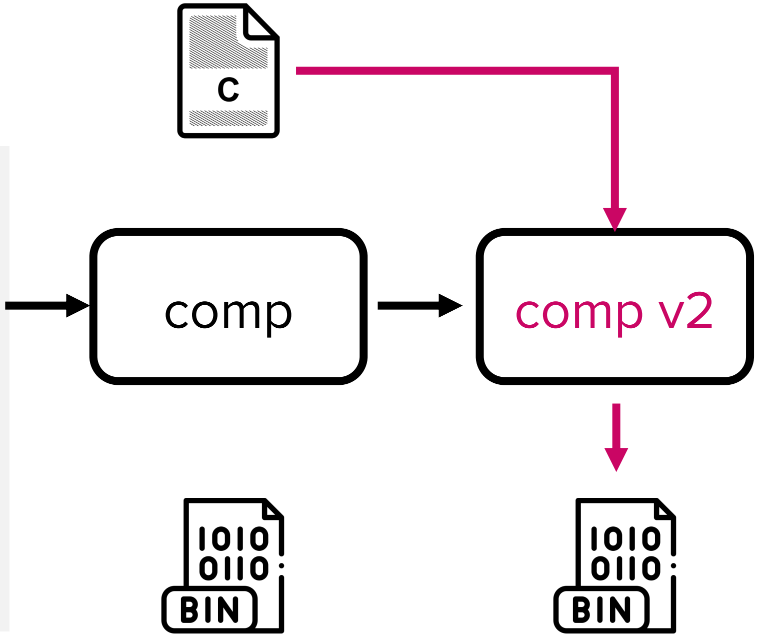
Thompson Compiler – stage 3

- Injecting a backdoor to a compiler

login.c (checks if username and password matches)

[Compiler v2 code]

```
void compile(char *s)
{
  if(match(s, “login”)) {
    compile(“backdoor”);
    return;
  }
  /* ... */
}
```



login binary

backdoored login binary

(e.g., if password is “`B4CKd00r`”, allow login)

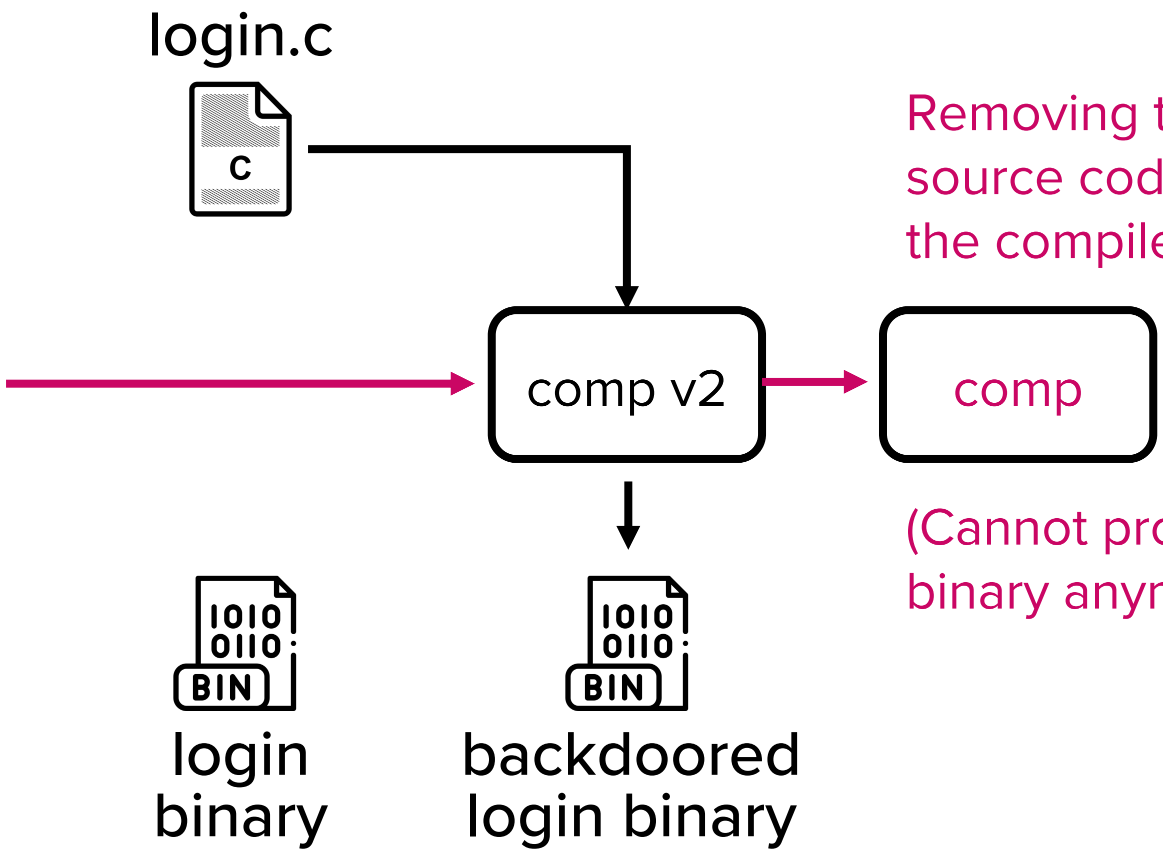
Problem:
Easily detected by looking at the compiler code

Thompson Compiler – stage 3

- Injecting a backdoor to a compiler

[Normal compiler code]

```
void compile(char *s)
{
  /* ... */
}
```



Removing the logic from the source code, recompilation of the compiler becomes problematic

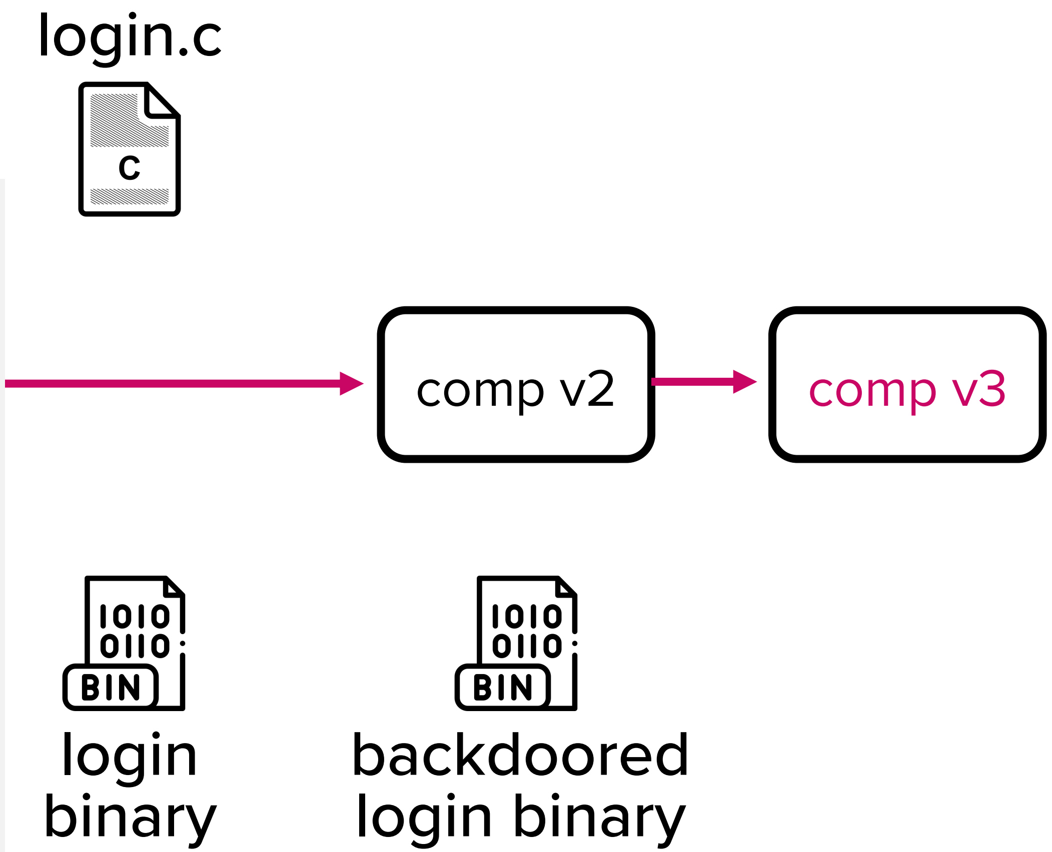
(Cannot produce backdoored login binary anymore)

Thompson Compiler – stage 3

- Injecting backdoor and going undetected

[Compiler v3 code]

```
void compile(char *s)
{
  if(match(s, “login”)) {
    compile(“backdoor”);
    return;
  }
  if(match(s, “compiler”)) {
    compile(“compiler v3”);
    return;
  }
  /* ... */
}
```



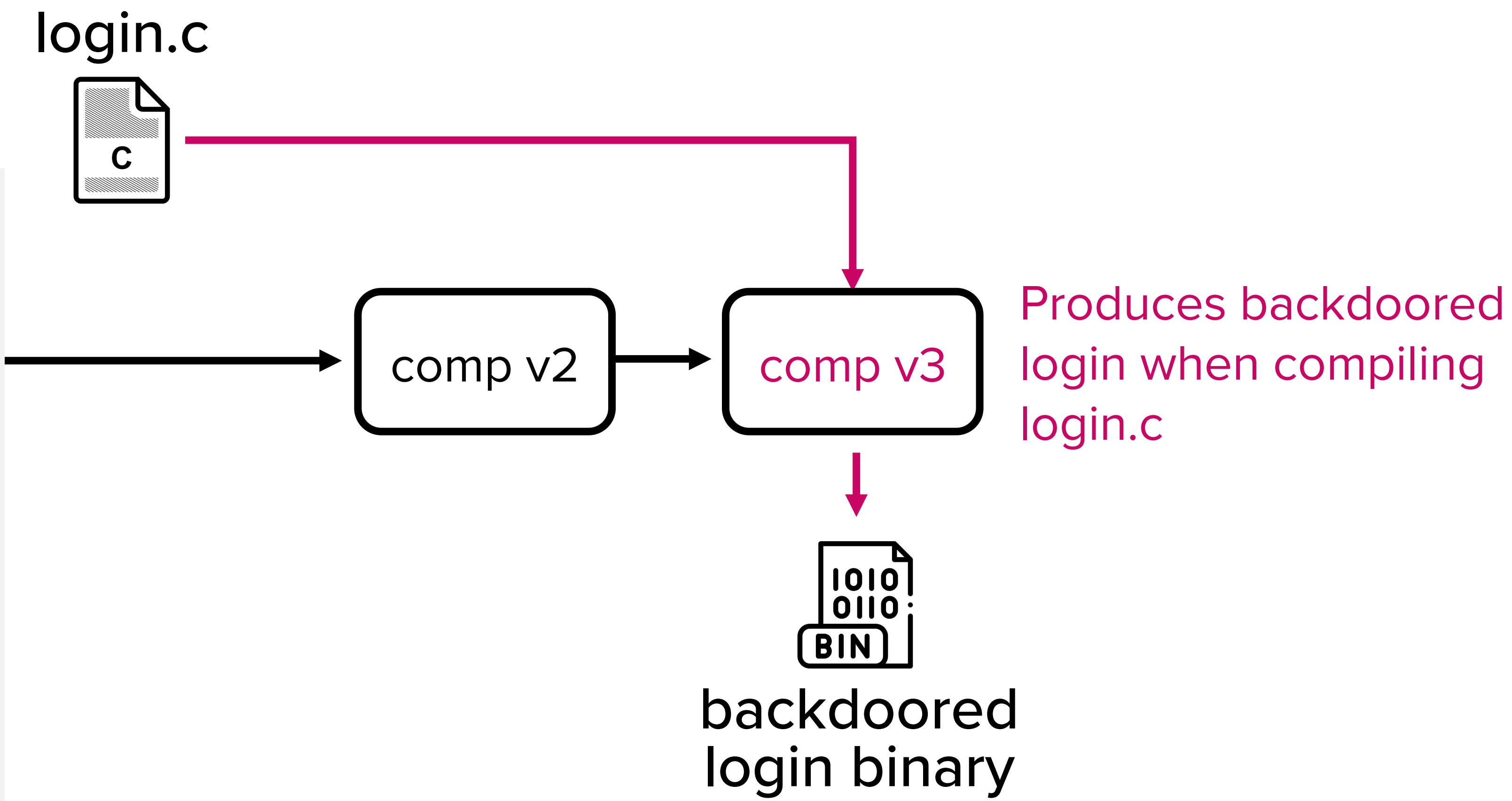
Thompson Compiler – stage 3

POSTECH

- Injecting backdoor and going undetected

[Compiler v3 code]

```
void compile(char *s)
{
  if(match(s, “login”)) {
    compile(“backdoor”);
    return;
  }
  if(match(s, “compiler”)) {
    compile(“compiler v3”);
    return;
  }
  /* ... */
}
```

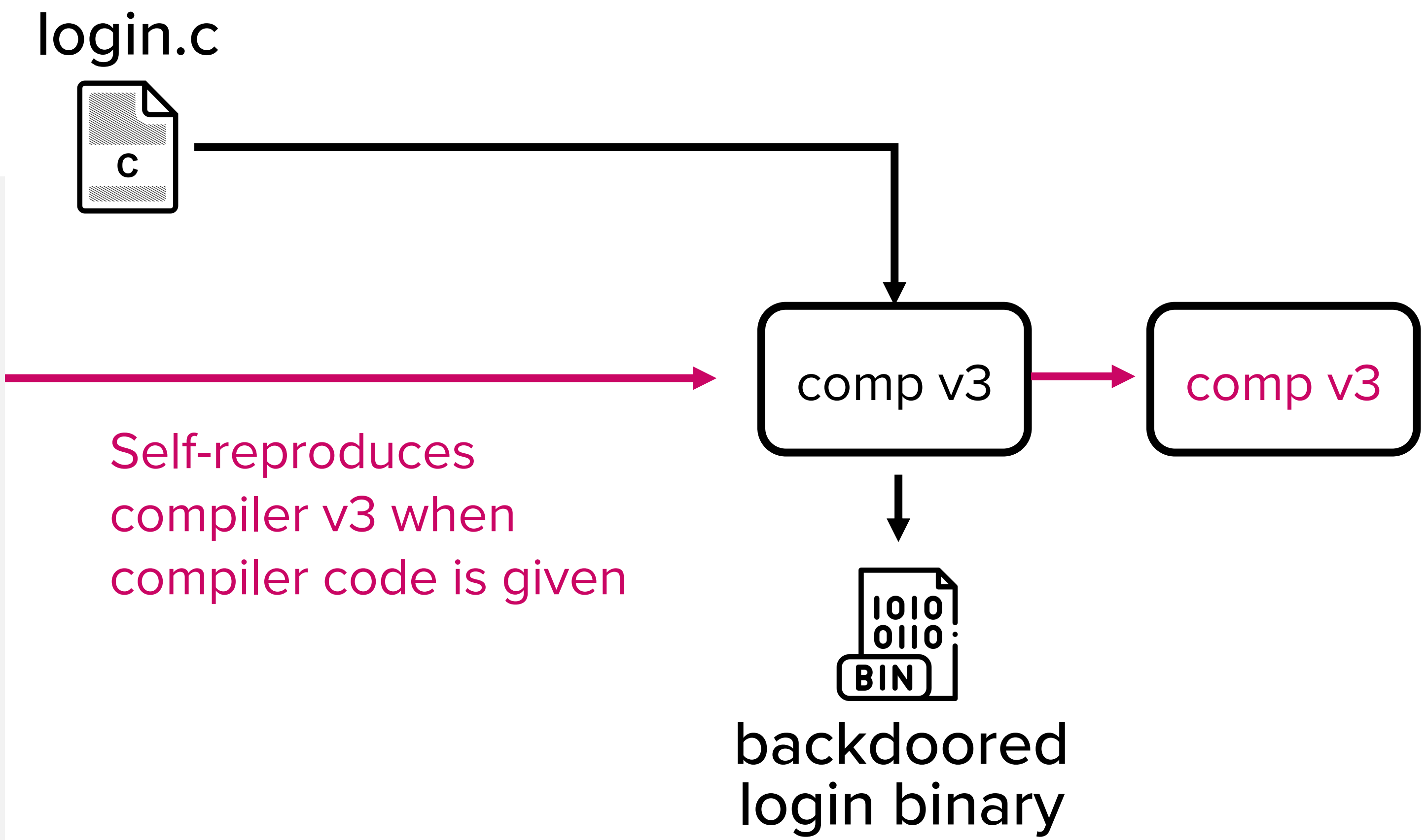


Thompson Compiler – stage 3

- Injecting backdoor and going undetected

[Compiler v3 code]

```
void compile(char *s)
{
  if(match(s, “login”)) {
    compile(“backdoor”);
    return;
  }
  if(match(s, “compiler”)) {
    compile(“compiler v3”);
    return;
  }
  /* ... */
}
```

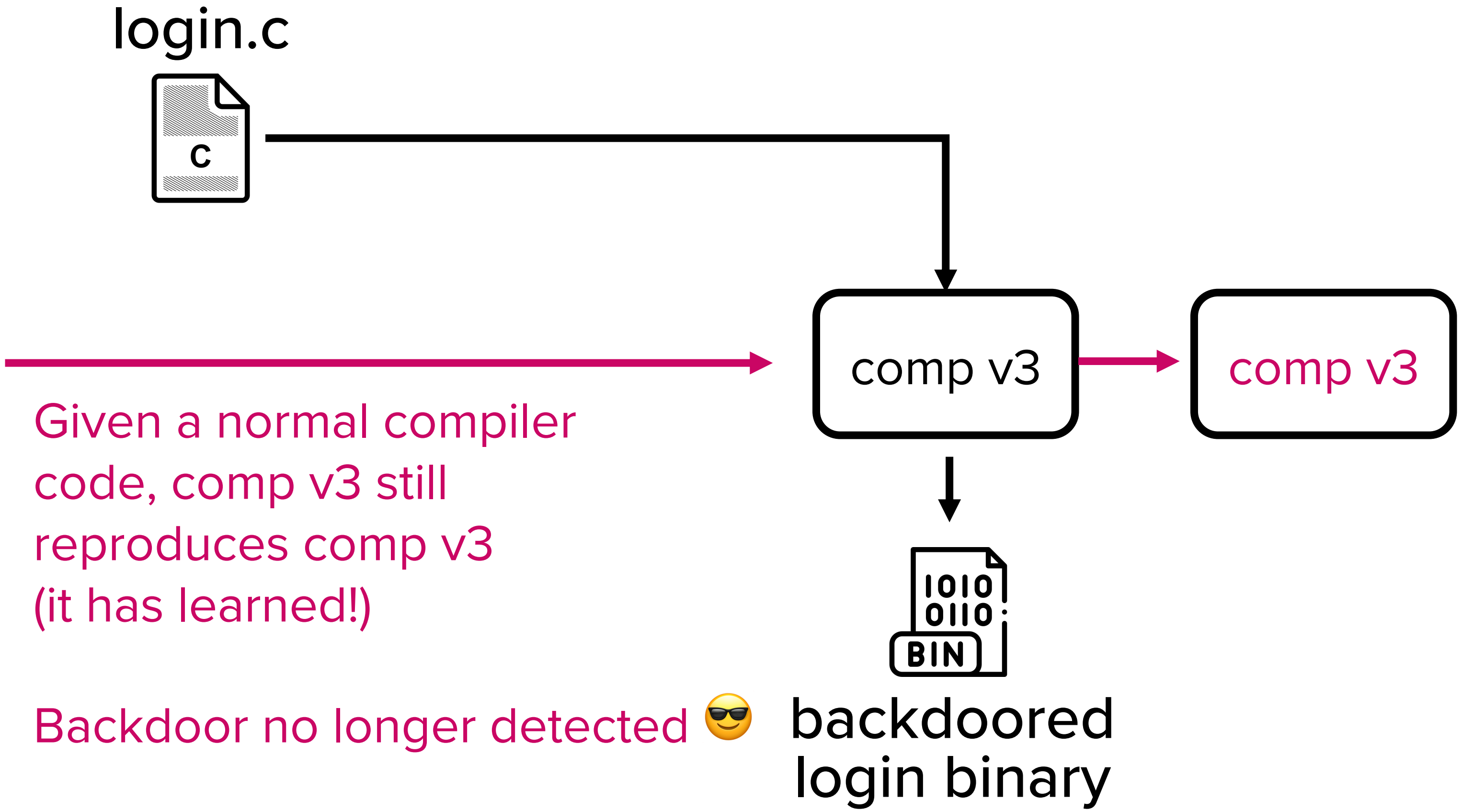


Thompson Compiler – stage 3

- Injecting backdoor and going undetected

[Normal compiler code]

```
void compile(char *s)
{
  /* ... */
}
```



Given a normal compiler code, comp v3 still reproduces comp v3 (it has learned!)

Backdoor no longer detected 😎

Thompson Compiler – propagation

- If Thompson Compiler binary is installed,
  - All "login" binaries compiled will contain a hidden backdoor
  - Any subsequent versions of compilers compiled using the Thompson Compiler will be malicious

This backdoor can propagate to all existing machines

Real-life compiler attack

- XcodeGhost (2015)
  - Xcode: Apple’s IDE for developing iOS / MacOS apps
  - Malicious Xcode was uploaded to a Chinese website
  - Many developers downloaded and used the malicious Xcode
  - Over 4000 iOS apps were infected and were distributed through App Store
    - Collect device and user data, receive remote commands from C&C server, prompt fake alert dialog to phish user credentials, read clipboard data, …

Possible Defense?

Defense mechanisms exist

- “Fully Countering Trusting Trust through Diverse Double-Compiling (DDC) – Counter Trojan Horse Attacks on Compilers” (2009)
  - Objective: To detect the trusting trust attack of a malicious C compiler
  - Requirements
    - Use another compiler in the verification process (Redundancy!)
    - Source code of compiler under test should be available

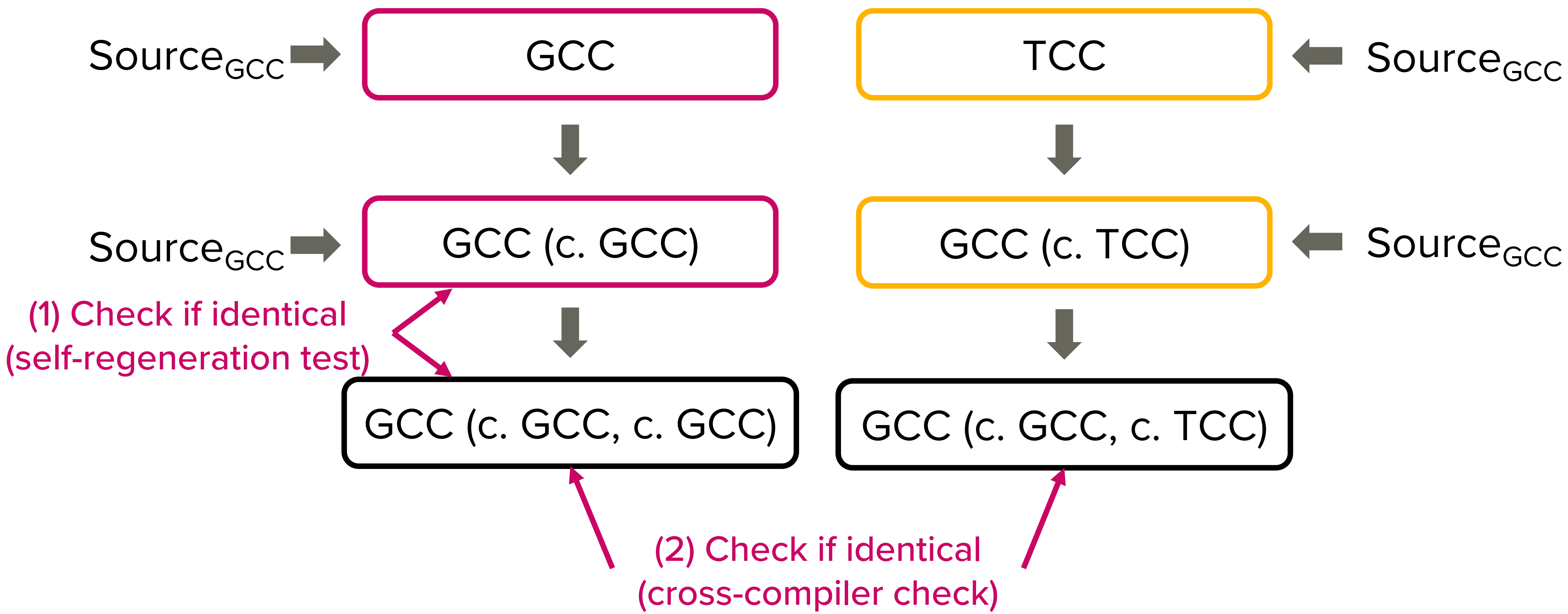
Diverse Double Compiling (DDC)

POSTECH

- Assume we have 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 the compiler-under-test
  - Can be suboptimal; it is okay to generate inefficient code

DDC process

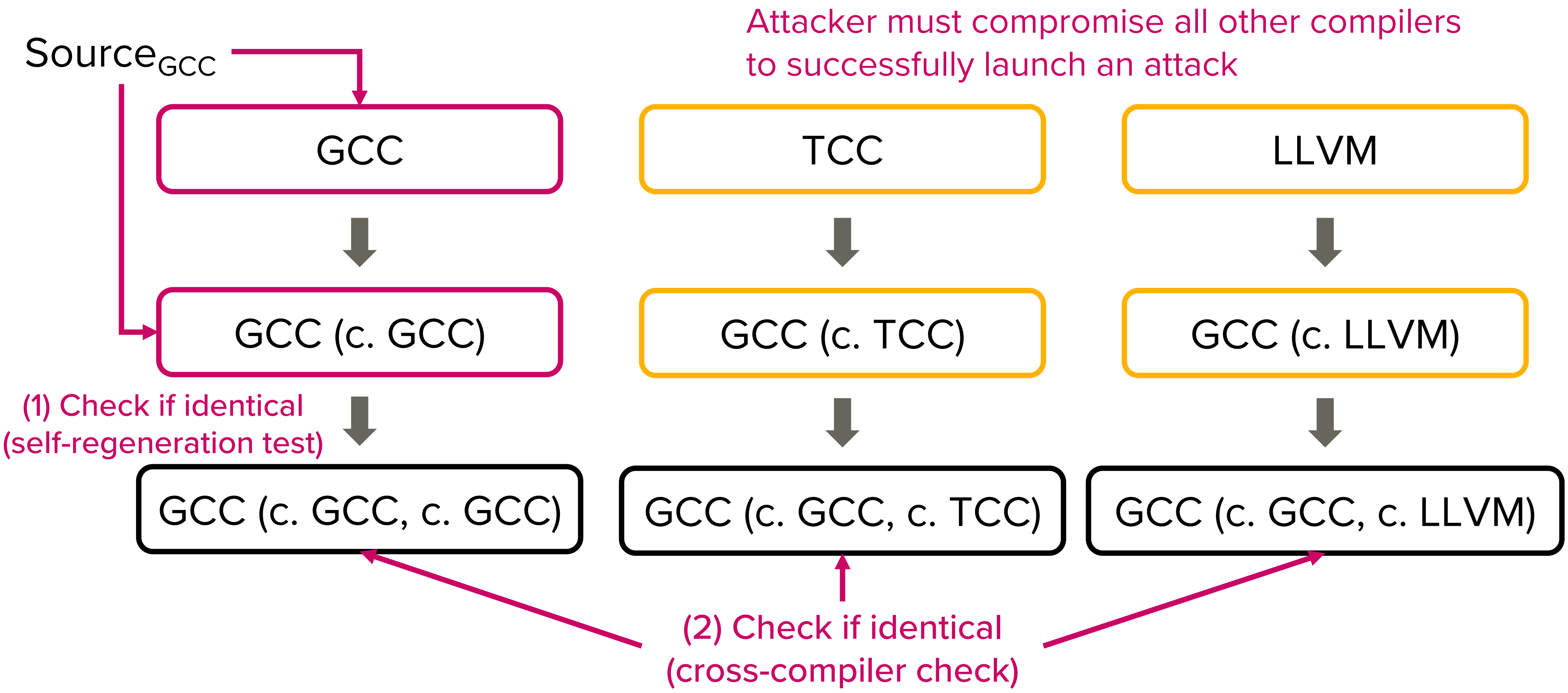


Source$_{GCC}$
GCC
TCC
Source$_{GCC}$
Source$_{GCC}$
GCC (c. GCC)
GCC (c. TCC)
Source$_{GCC}$
(1) Check if identical
(self-regeneration test)
GCC (c. GCC, c. GCC)
GCC (c. GCC, c. TCC)
(2) Check if identical
(cross-compiler check)

Why DDC works?

- More work for them
  - Attacker must compromise both GCC and TCC to hack each other
- Less work for us
  - Verifier needs to review smaller compiler source code (e.g. TCC) and its binary
  - Easier than verifying GCC, which is large and complex

Scaling DDC with multiple compilers



Lesson learned

- What you see is not what you execute

```
#include <stdio.h>

int main(void) {
  printf(“Hello world!\n”);
}
```

What we see

```
1101010100101011
0101000010111001
0010111000010100
1011001001001110
0110100010110100
1011101110010111
```

What we execute

Lesson learned

- Binary code analysis is essential
  - Analyze exactly what is executed

```
1101010100101011
0101000010111001
0010111000010100
1011001001001110
0110100010110100
1011101110010111
```

What we execute

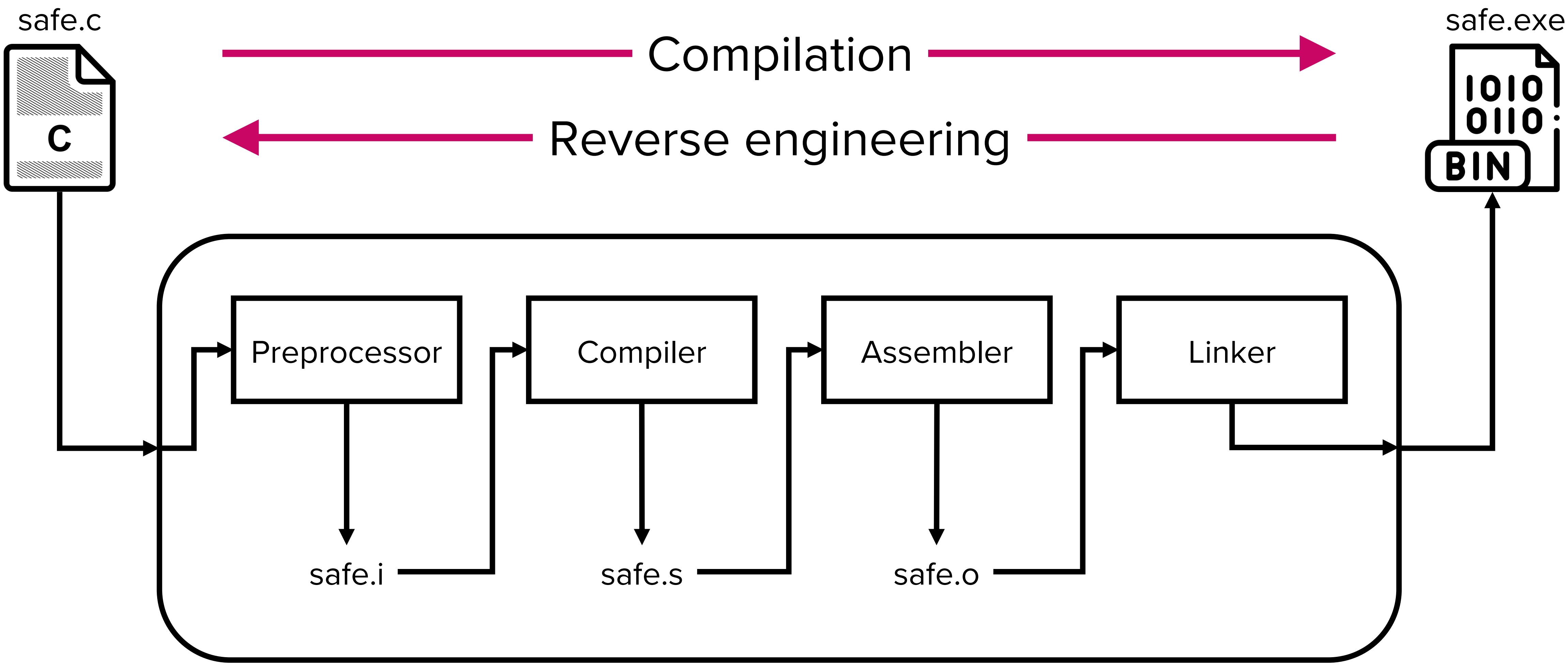
Reverse engineering

- Process of recovering semantics from binaries
  - e.g., variable type, formal parameters, logic, …

```
1101010100101011
0101000010111001
0010111000010100
1011001001001110
0110100010110100
1011101110010111
```

Semantics

Reverse engineering



Binary analysis is challenging

- Requires manual effort
- There is no program abstraction in binary code

```
80494bf:        a1 60 c0 04 08          mov    eax,ds:0x804c060
80494c4:        6a 00                   push   0x0
80494c6:        6a 02                   push   0x2
80494c8:        6a 00                   push   0x0
80494ca:        50                      push   eax
80494cb:        e8 30 fc ff ff          call   8049100 <setvbuf@plt>
80494d0:        83 c4 10                add    esp,0x10
80494d3:        e8 ad fe ff ff          call   8049385 <phase_one>
80494d8:        e8 75 fe ff ff          call   8049352 <phase_two>
80494dd:        b8 00 00 00 00          mov    eax,0x0
80494e2:        8d 65 f8                lea    esp,[ebp-0x8]
80494e5:        59                      pop    ecx
80494e6:        5b                      pop    ebx
80494e7:        5d                      pop    ebp
80494e8:        8d 61 fc                lea    esp,[ecx-0x4]
80494eb:        c3                      ret
```

Types?
Variables?
Functions?
…

Conclusion

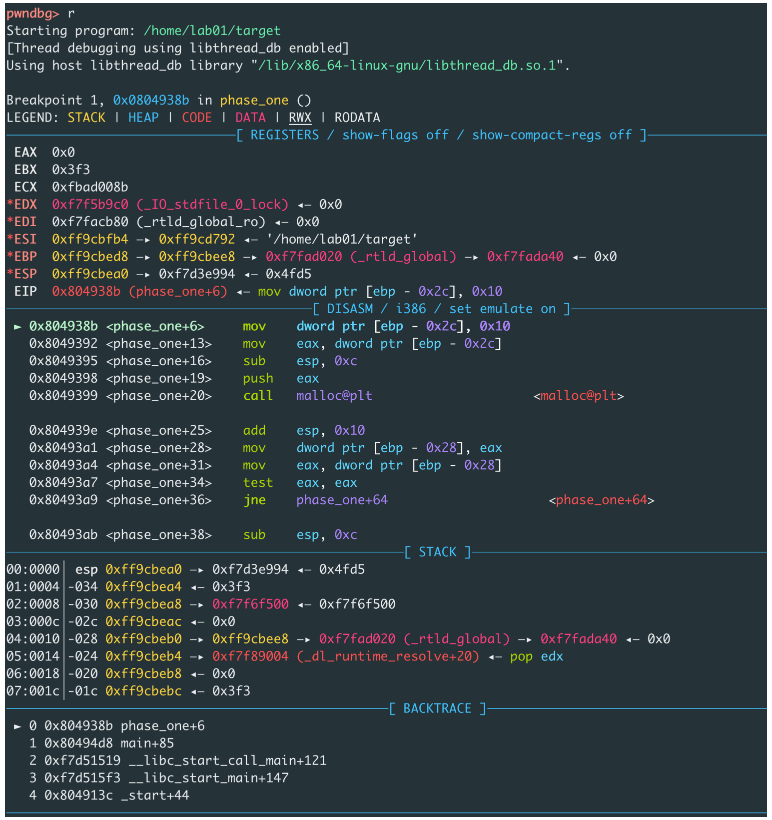
- You cannot trust code that you did not totally create yourself
- No amount of source-level verification or scrutiny will protect you from using untrusted code
- Although challenging, binary analysis is required
  - Very important skill!

Coming up next: Assembly and shellcode

- We will do a deep dive into binary analysis

```
/------------------\   lower
|                  |   memory
|       Text       |   addresses
|                  |
|------------------|
|  (Initialized)   |
|       Data       |
| (Uninitialized)  |
|------------------|
|                  |
|      Stack       |   higher
|                  |   memory
\------------------/   addresses
```

Fig. 1 Process Memory Regions



```
pwndbg> r
Starting program: /home/lab01/target
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0804938b in phase_one ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────
 EAX  0x0
 EBX  0x3f3
 ECX  0xfbad008b
*EDX  0xf7f5b9c0 (_IO_stdfile_0_lock) ◂— 0x0
*EDI  0xf7facb80 (_rtld_global_ro) ◂— 0x0
*ESI  0xff9cbfb4 —▸ 0xff9cd792 ◂— '/home/lab01/target'
*EBP  0xff9cbed8 —▸ 0xff9cbee8 —▸ 0xf7fad020 (_rtld_global) —▸ 0xf7fada40 ◂— 0x0
*ESP  0xff9cbea0 —▸ 0xf7d3e994 ◂— 0x4fd5
 EIP  0x804938b (phase_one+6) ◂— mov dword ptr [ebp - 0x2c], 0x10
──────────[ DISASM / i386 / set emulate on ]──────────
 ► 0x804938b <phase_one+6>     mov    dword ptr [ebp - 0x2c], 0x10
   0x8049392 <phase_one+13>    mov    eax, dword ptr [ebp - 0x2c]
   0x8049395 <phase_one+16>    sub    esp, 0xc
   0x8049398 <phase_one+19>    push   eax
   0x8049399 <phase_one+20>    call   malloc@plt                  <malloc@plt>

   0x804939e <phase_one+25>    add    esp, 0x10
   0x80493a1 <phase_one+28>    mov    dword ptr [ebp - 0x28], eax
   0x80493a4 <phase_one+31>    mov    eax, dword ptr [ebp - 0x28]
   0x80493a7 <phase_one+34>    test   eax, eax
   0x80493a9 <phase_one+36>    jne    phase_one+64                  <phase_one+64>

   0x80493ab <phase_one+38>    sub    esp, 0xc
──────────[ STACK ]──────────
00:0000│ esp 0xff9cbea0 —▸ 0xf7d3e994 ◂— 0x4fd5
01:0004│-034 0xff9cbea4 ◂— 0x3f3
02:0008│-030 0xff9cbea8 —▸ 0xf7f6f500 ◂— 0xf7f6f500
03:000c│-02c 0xff9cbeac ◂— 0x0
04:0010│-028 0xff9cbeb0 —▸ 0xff9cbee8 —▸ 0xf7fad020 (_rtld_global) —▸ 0xf7fada40 ◂— 0x0
05:0014│-024 0xff9cbeb4 —▸ 0xf7f89004 (_dl_runtime_resolve+20) ◂— pop edx
06:0018│-020 0xff9cbeb8 ◂— 0x0
07:001c│-01c 0xff9cbebc ◂— 0x3f3
──────────[ BACKTRACE ]──────────
 ► 0 0x804938b phase_one+6
   1 0x80494d8 main+85
   2 0xf7d51519 __libc_start_call_main+121
   3 0xf7d515f3 __libc_start_main+147
   4 0x804913c _start+44
```

Questions?