Lec 04: On Trusting Trust

CSED415: Computer Security

Spring 2026

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Team project

- Welcome, enthusiasts!
  - 32 students
- Please form teams for the team project
  - Each with 4-5 students
  - Begin searching for teammates now!
    - Utilize the “Teammate Finding” board on PLMS
  - Finalize your teams by March 20
    - Submit team info - Refer to the “Team Forming” assignment on PLMS

Office hours

- TA office hours
  - Mondays, 11:00am-12:00pm at PIAI 441 or online
    - Jangseop jangseop@postech.ac.kr
  - Wednesdays, 1:30pm-2:30pm at PIAI 441 or online
    - Minki leeminki@postech.ac.kr
- My office hours
  - Thursdays, 1-2 PM in my office (PIAI 434)

If you plan to visit, please email us at least one day in advance!

Lab 01: Secure coding

POSTECH

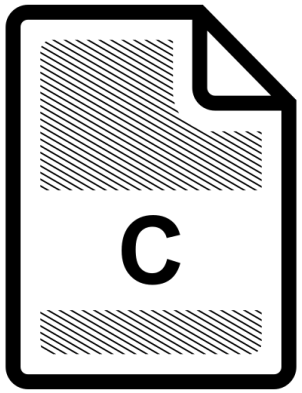
- Due Monday, March 9, 3:30 PM
- Make sure you report contains all items below:
  - Process memory map
  - Vulnerable code, analysis, and exploitation
  - Relevant secure coding guideline and a patch
- Attend office hours if you need help!

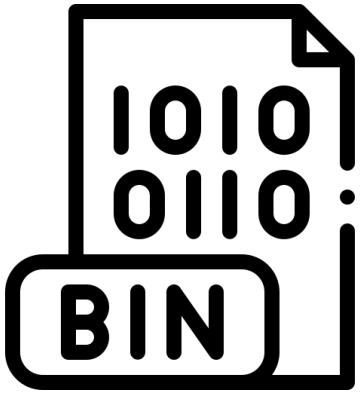
Recap

- Defensive programming and secure coding guidelines
  - Buggy code is the root of evil
    - Do you remember any of the secure coding rules?
    - Have you tried reviewing your own code?
  - From *Lecture 03*:
    - Conformance with the secure coding standard is necessary (but not sufficient) for reliable and secure software
    - Today's topic: Why is it not sufficient?

Source code vs Binary

- Q) Given both the source code and the binary of a program, which should you analyze to determine whether it is safe to execute the program?





Source code vs Binary

- Reasons to analyze the source code
  - **Clarity**: Source code is easier to read, understand, and review
  - **Availability of context**: It often contains comments, descriptive variable names, and meaningful structure
  - **Fixability**: Vulnerabilities found at the source level are typically easier to correct directly in the code

Source code vs Binary

- So, why bother with binaries??
  - Despite the advantages of source code analysis, security experts often analyze **binaries** to discover hidden vulnerabilities
  - Reason: Today's topic

Key question

- You have the complete source code of a program. Can you find all potential vulnerabilities in this program just by analyzing the given source code?

NO WAY!

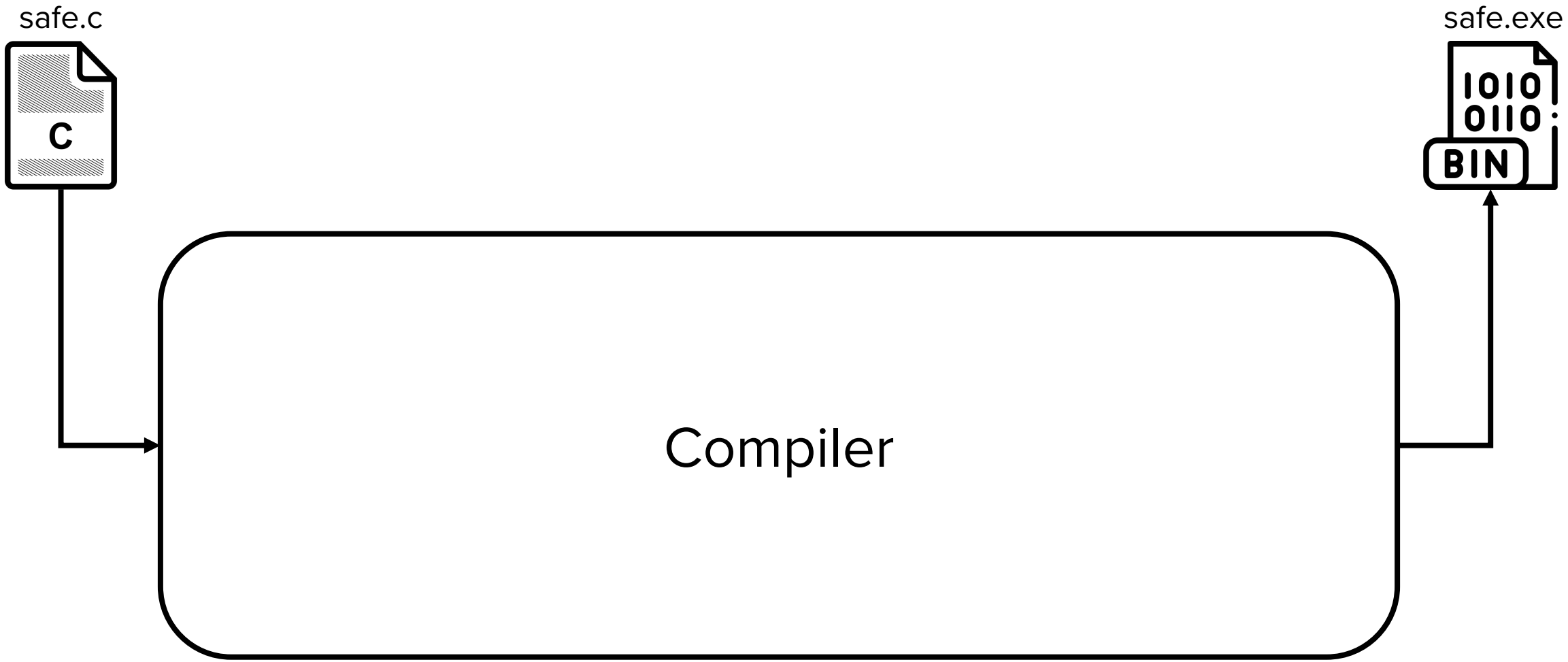
Ken Thompson

“Reflections on Trusting Trust”

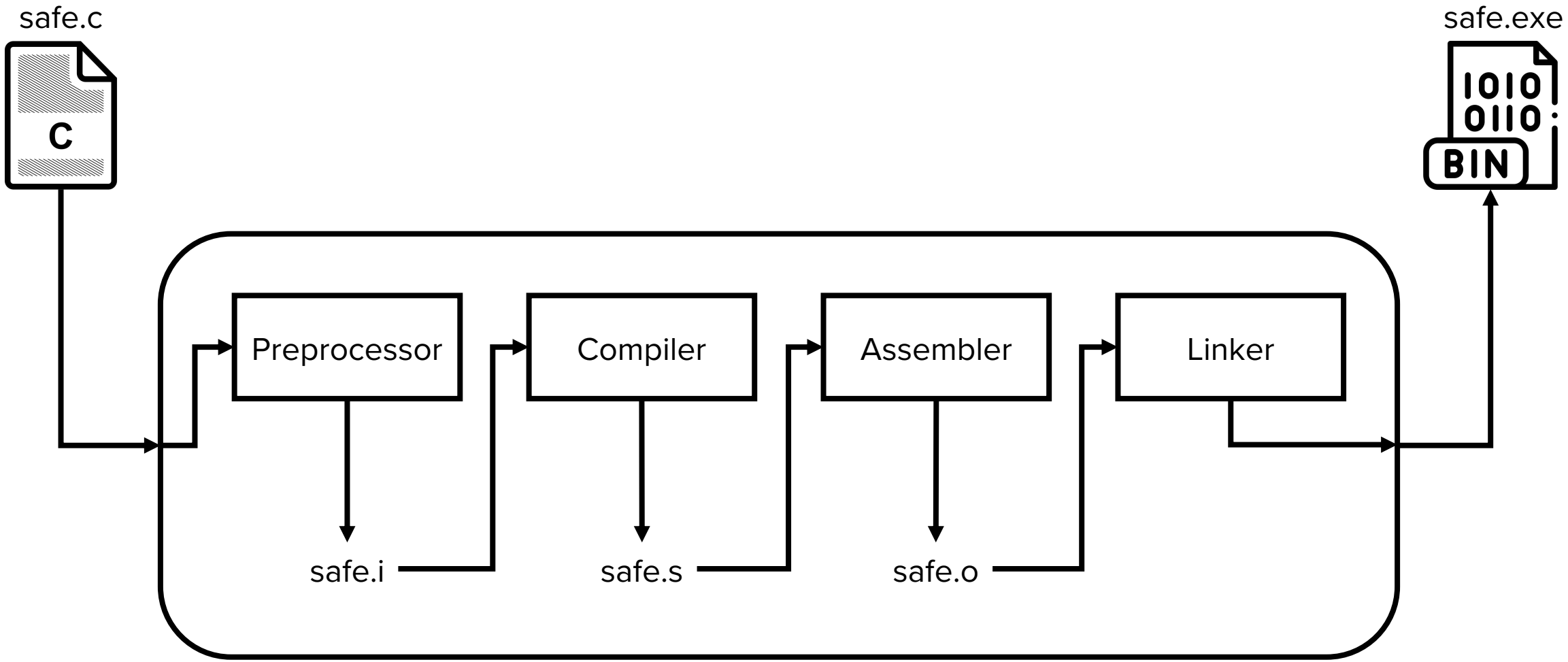
Communications of the ACM, 1984

On Trusting Trust

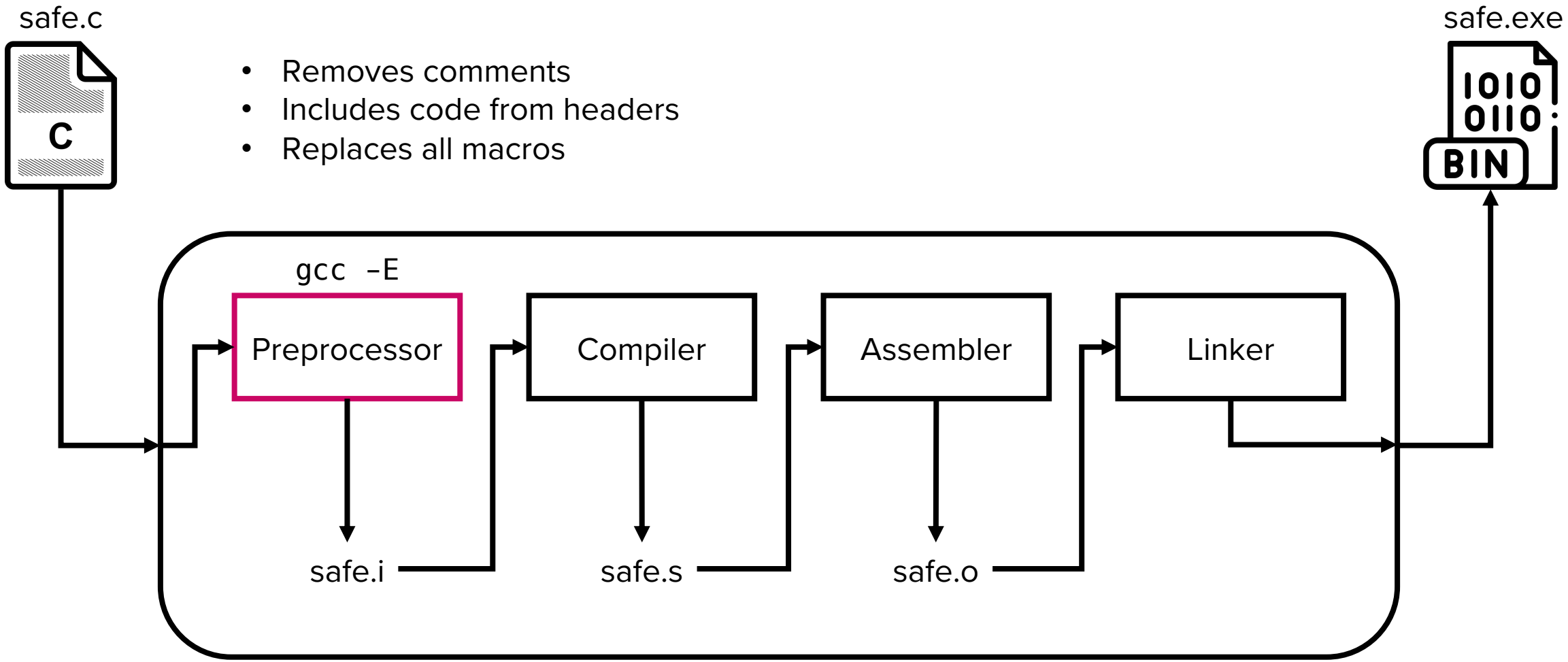
Compiler 101



Compiler 101

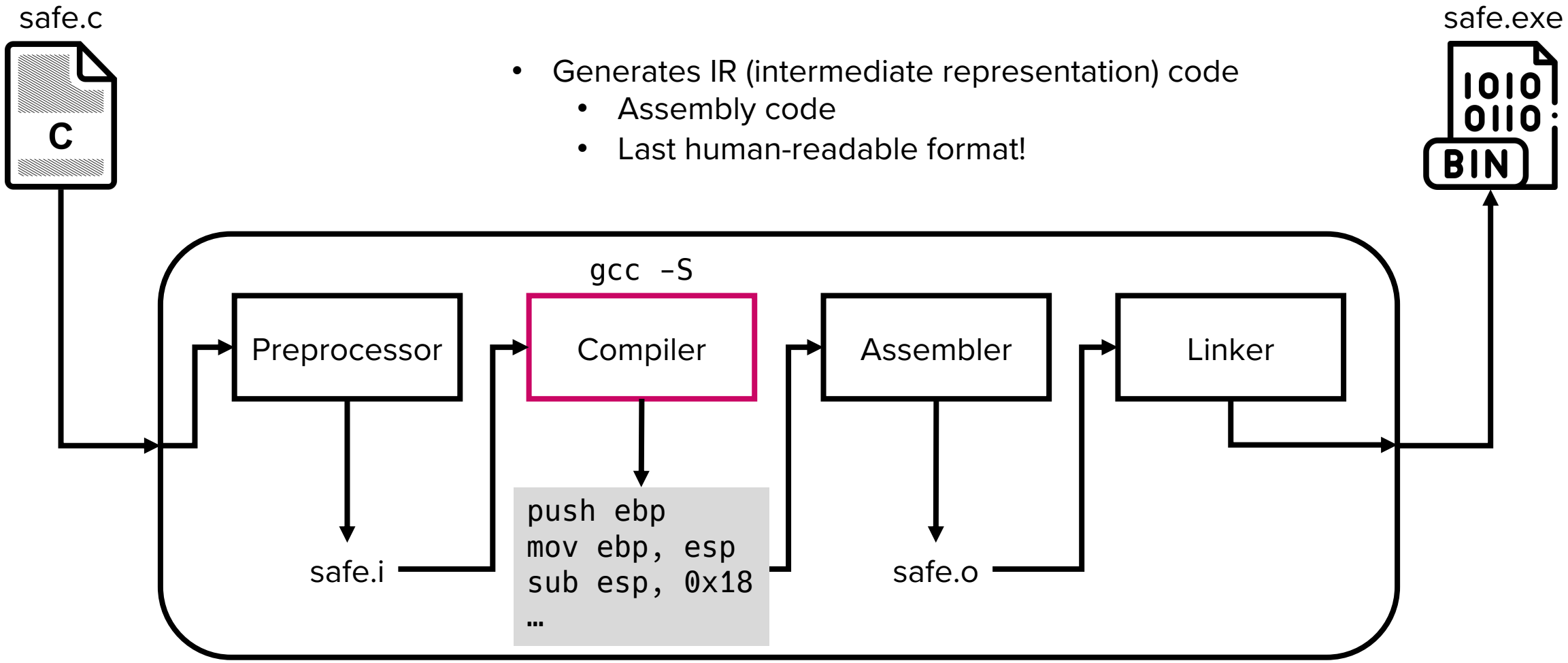


Compiler 101



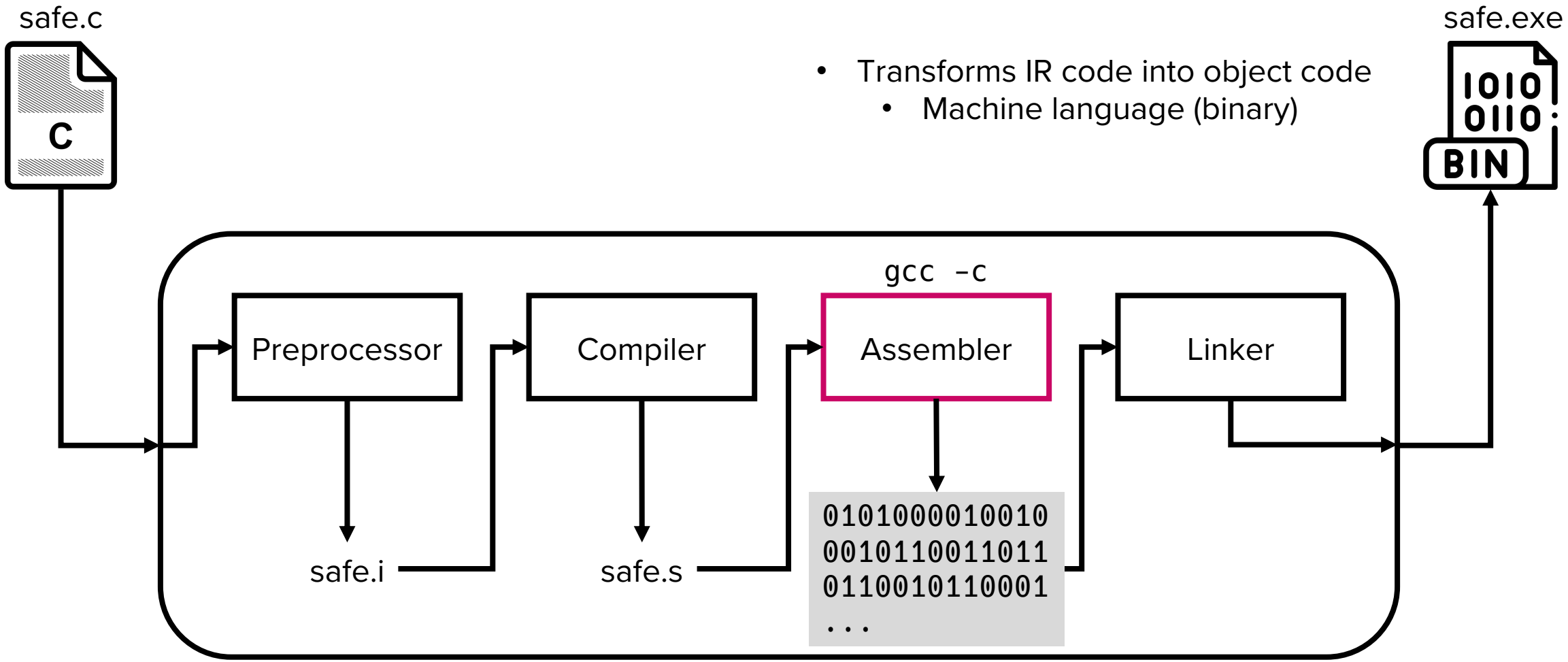
- Removes comments
- Includes code from headers
- Replaces all macros

Compiler 101



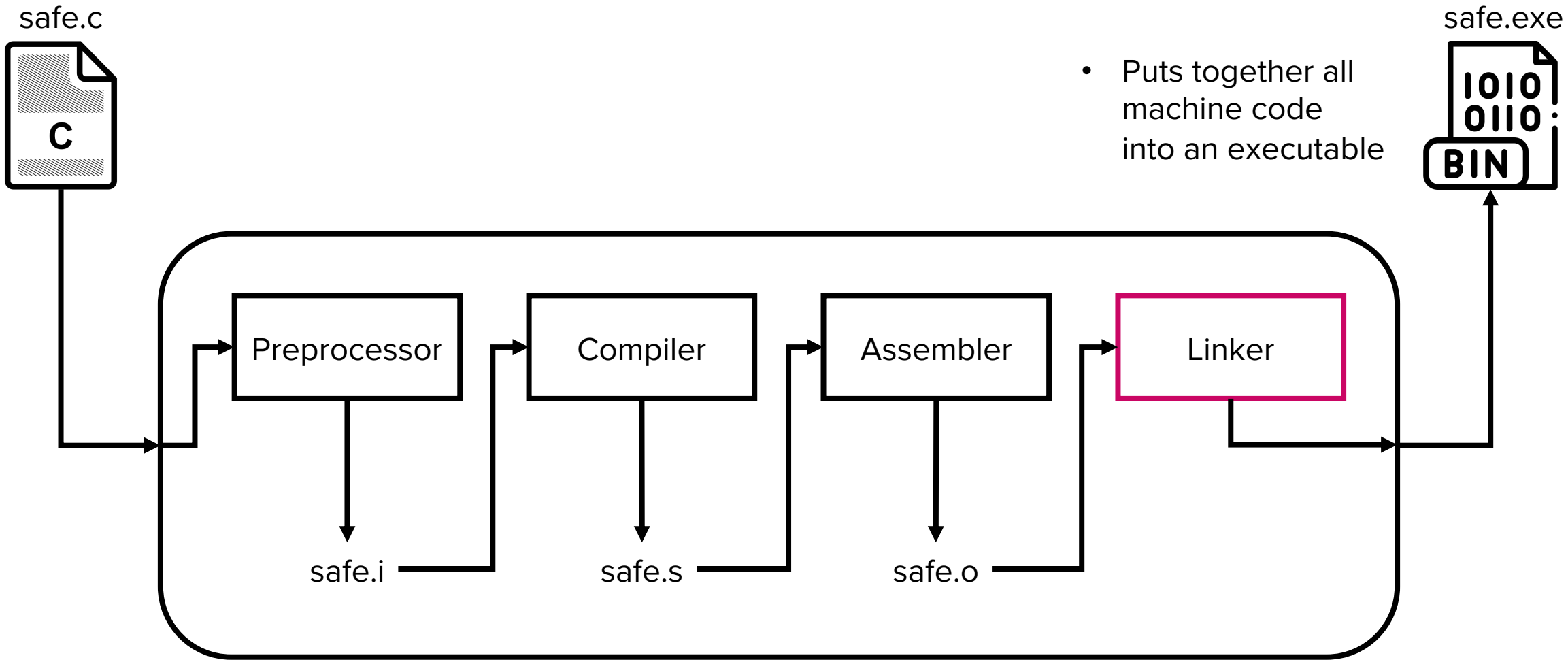
- Generates IR (intermediate representation) code
  - Assembly code
  - Last human-readable format!

Compiler 101



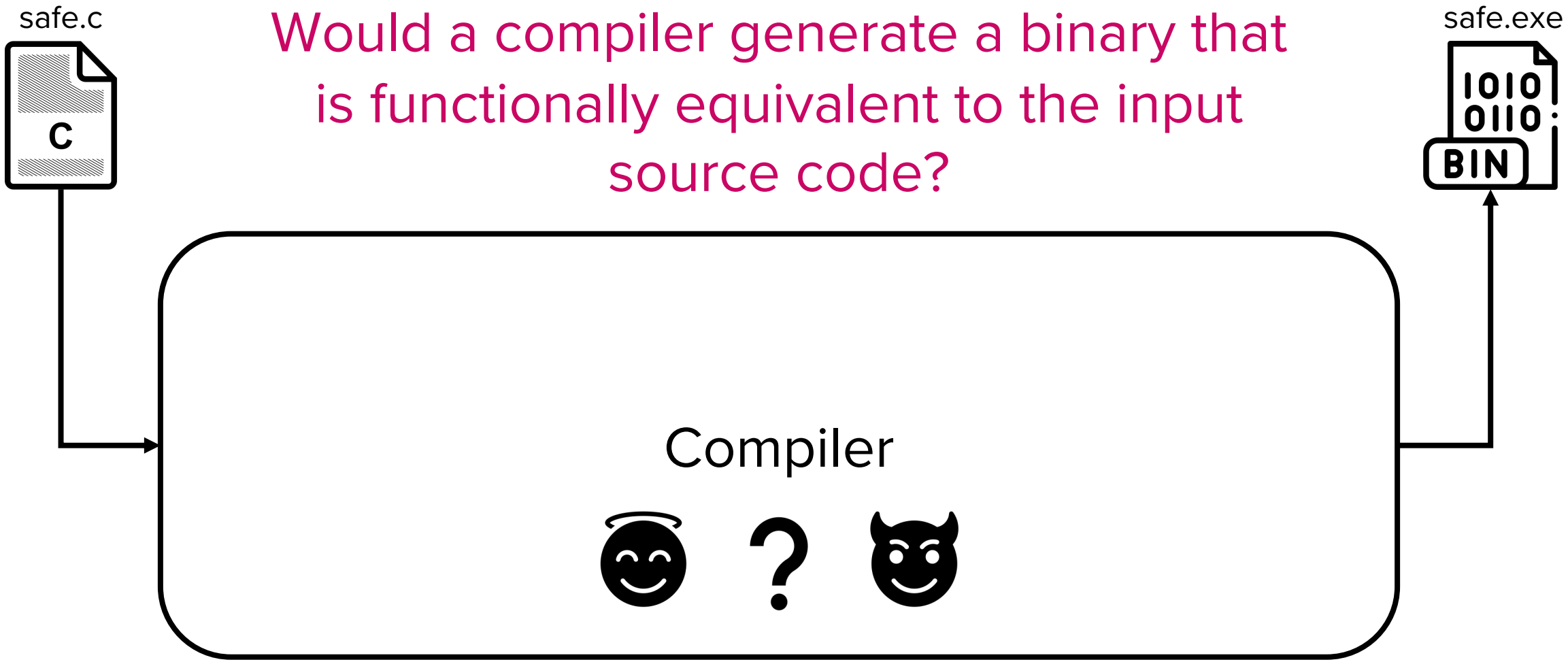
- Transforms IR code into object code
  - Machine language (binary)

Compiler 101



- Puts together all machine code into an executable

Question: Can we trust compilers?



Chasing down a rabbit hole

- To ensure a program is safe, we inspect its source code 😎
- But that code is compiled by another program (a compiler)
- So we inspect the compiler's source code 😐
- But that compiler was compiled by yet another compiler..
- and it goes on and on… infinitely 🤯

Reflections on Trusting Trust

- Ken Thompson (and Dennis Ritchie)
  - Received the Turing Award in 1983 for their work on Unix Operating System
  - In his acceptance speech, Thompson demonstrated how to build a backdoored compiler without leaving any trace in its source code

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.

"Reflections on Trusting Trust", Communications of the ACM, 1984

- https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf
- Full text also available on PLMS

Thompson Compiler – Stage 1

- Stage 1: Understanding the concept of “quine”
  - Quine is a self-reproducing code
    - A quine in Python 3:

```
c = 'c = %r; print(c %% c)'; print(c % c)
```

    - A quine in C

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){
printf(f,34,f,34,10);}
```

Thompson Compiler – Stage 1

POSTECH

- Thompson's quine in the paper:

```
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 initialization

// prints the rest (comments and the main function)

Thompson Compiler – Stage 2

- Stage 2: Understanding the concept of “training” a compiler

[Compiler v1’s code]

```
c = next();

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

compiler

- Compiler v1 knows how to parse \n (newline character):
  - If it reads '\', followed by 'n', it returns char '\n'
- However, it does not know how to parse \v (vertical tab) yet

Thompson Compiler – Stage 2

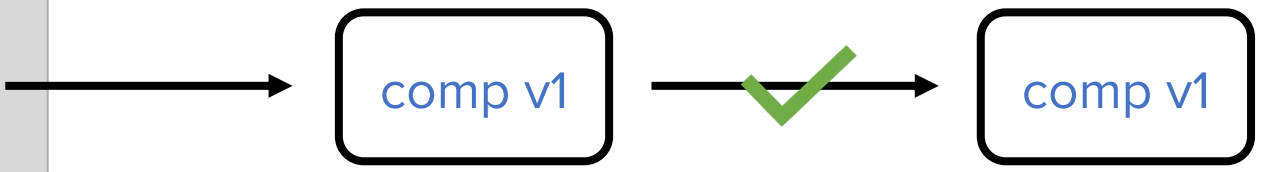
- Stage 2: Understanding the concept of “training” a compiler

[Compiler v1’s code]

```
c = next();

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

compiler



We can obtain comp v1 by compiling
Compiler v1’s code with comp v1

(self-reproduction!)

- Compiler v1 knows how to parse \n (newline character):
  - If it reads '\', followed by 'n', it returns char '\n'
- However, it does not know how to parse \v (vertical tab) yet

Thompson Compiler – Stage 2

- Stage 2: Understanding the concept of “training” a compiler

[Compiler v2’s code]

```
c = next();

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

compiler

- We want the complier to be able to compile code with vertical tabs
  - We can update the code to add a parsing logic for \v
  - If it reads '\', followed by 'v', it returns char '\v'

Thompson Compiler – Stage 2

POSTECH

- Stage 2: Understanding the concept of “training” a compiler

[Compiler v2’s code] compiler Result

```
c = next();

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

When we try to compile comp v2’s code using comp v1,
it fails because comp v1 cannot parse the \v in the new code

- We want the complier to be able to compile code with vertical tabs
  - We can update the code to add a parsing logic for \v
  - If it reads '\', followed by 'v', it returns char '\v'

Thompson Compiler – Stage 2

- Stage 2: Understanding the concept of “training” a compiler

[Compiler v2x’s code]

```
c = next();

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

compiler

comp v1

Result

- We update the compiler code to return 11 instead of \v
  - 11 is the ASCII code for char \v

Thompson Compiler – Stage 2

- Stage 2: Understanding the concept of “training” a compiler

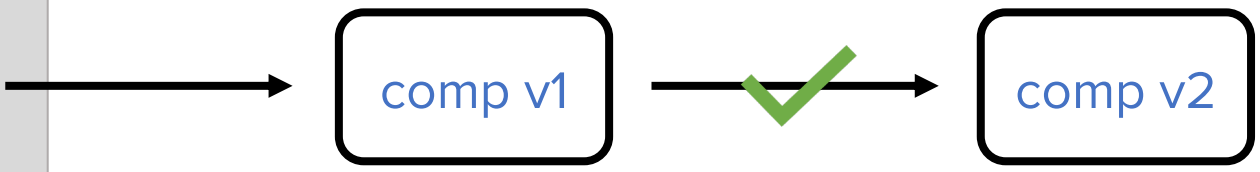
[Compiler v2x’s code]

```
c = next();

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

compiler

Result



It has “learned” what \v means

comp v1 is now able to compile the code and produce comp v2

- We update the compiler code to return 11 instead of \v
  - 11 is the ASCII code for char \v

Thompson Compiler – Stage 2

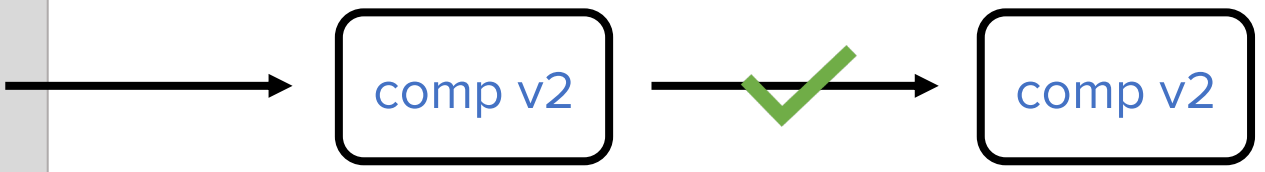
POSTECH

- Stage 2: Understanding the concept of “training” a compiler

[Compiler v2’s code] compiler Result

```
c = next();

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



Now that comp v2 has been trained, it can compile the original comp v2’s code and self-reproduce itself

- NOTE: The information it has learned, i.e., `'\v' == 11`, no longer appears in the code!
  - It is “baked” into the compiler’s binary

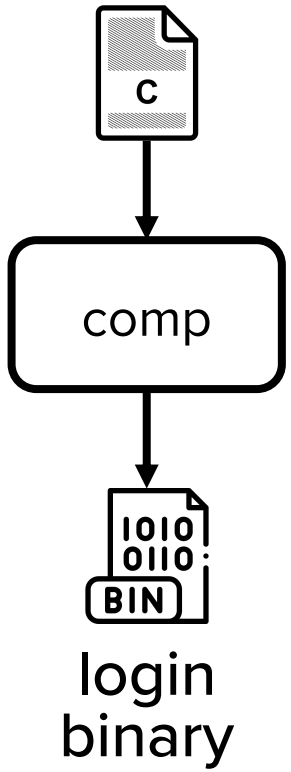
Thompson Compiler – Stage 3

- Stage 3: Injecting a backdoor

[Original compiler code]

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

login.c (checks if username and password matches)



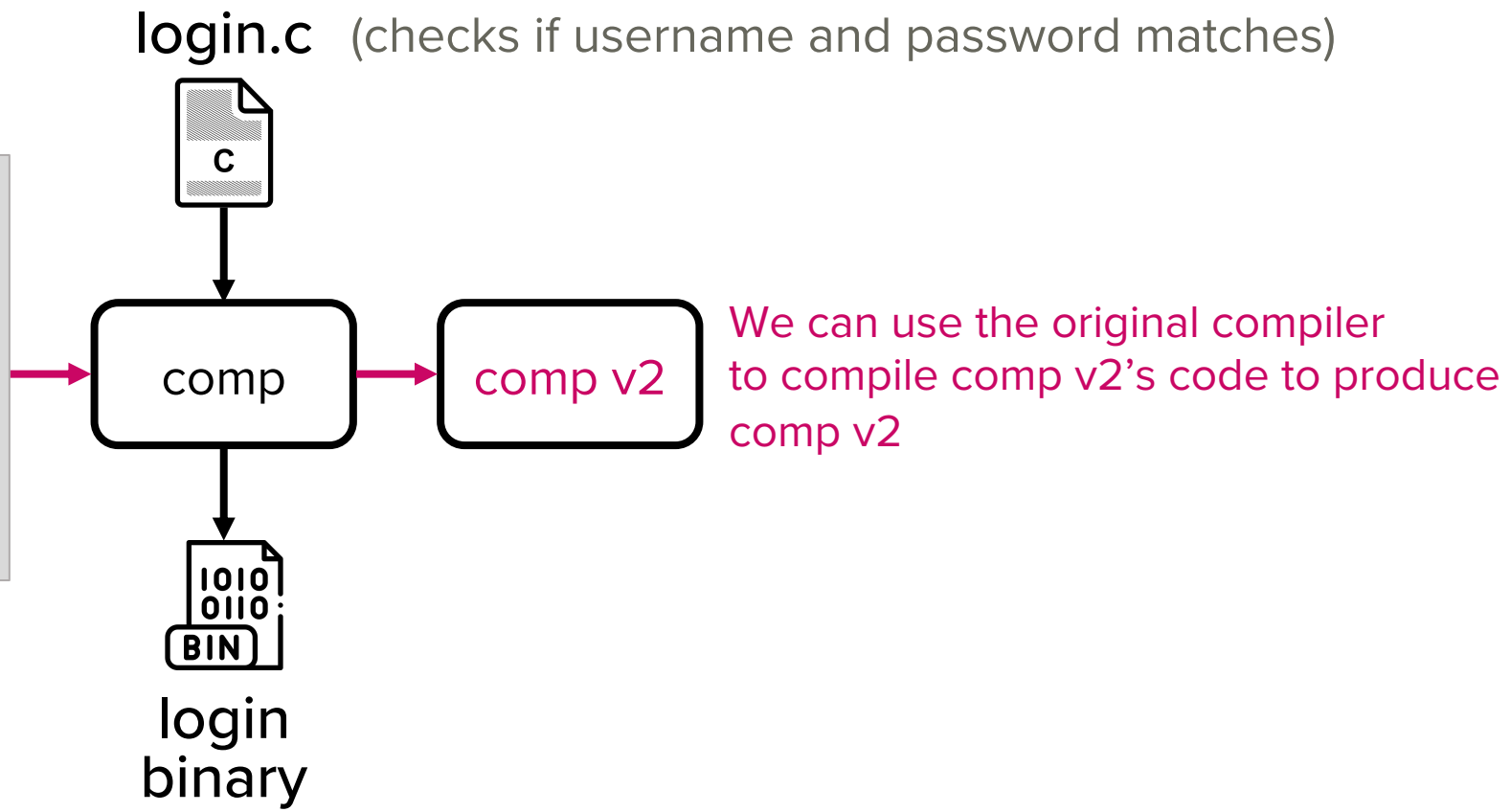
Thompson Compiler – Stage 3

- Stage 3: Injecting a backdoor when compiling login.c

[Compiler v2's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
}
```

→ A compiler that produces a backdoored login binary when compiling login.c



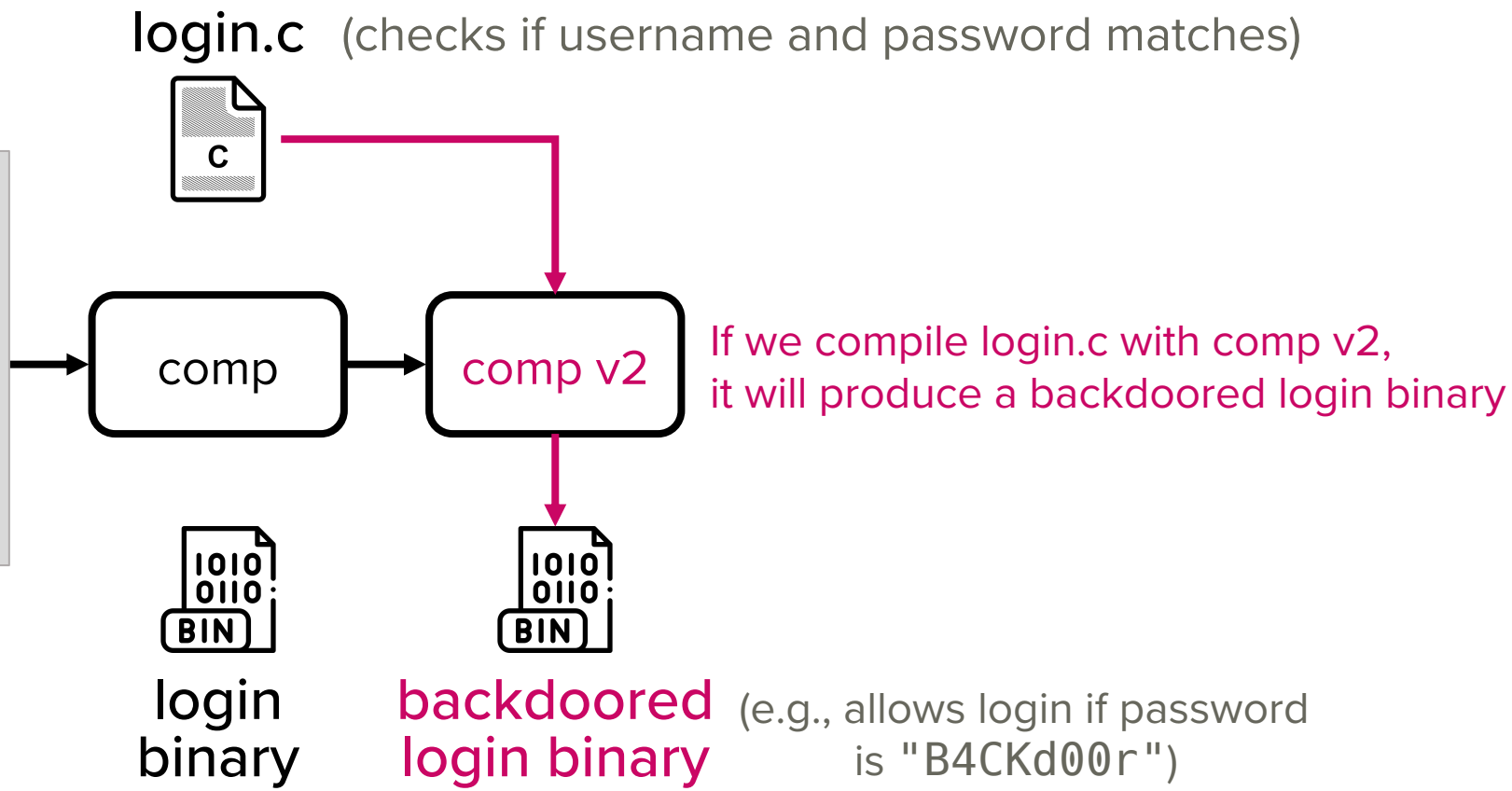
Thompson Compiler – Stage 3

- Stage 3: Injecting a backdoor when compiling login.c

[Compiler v2's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
}
```

→ A compiler that produces a backdoored login binary when asked to compile login



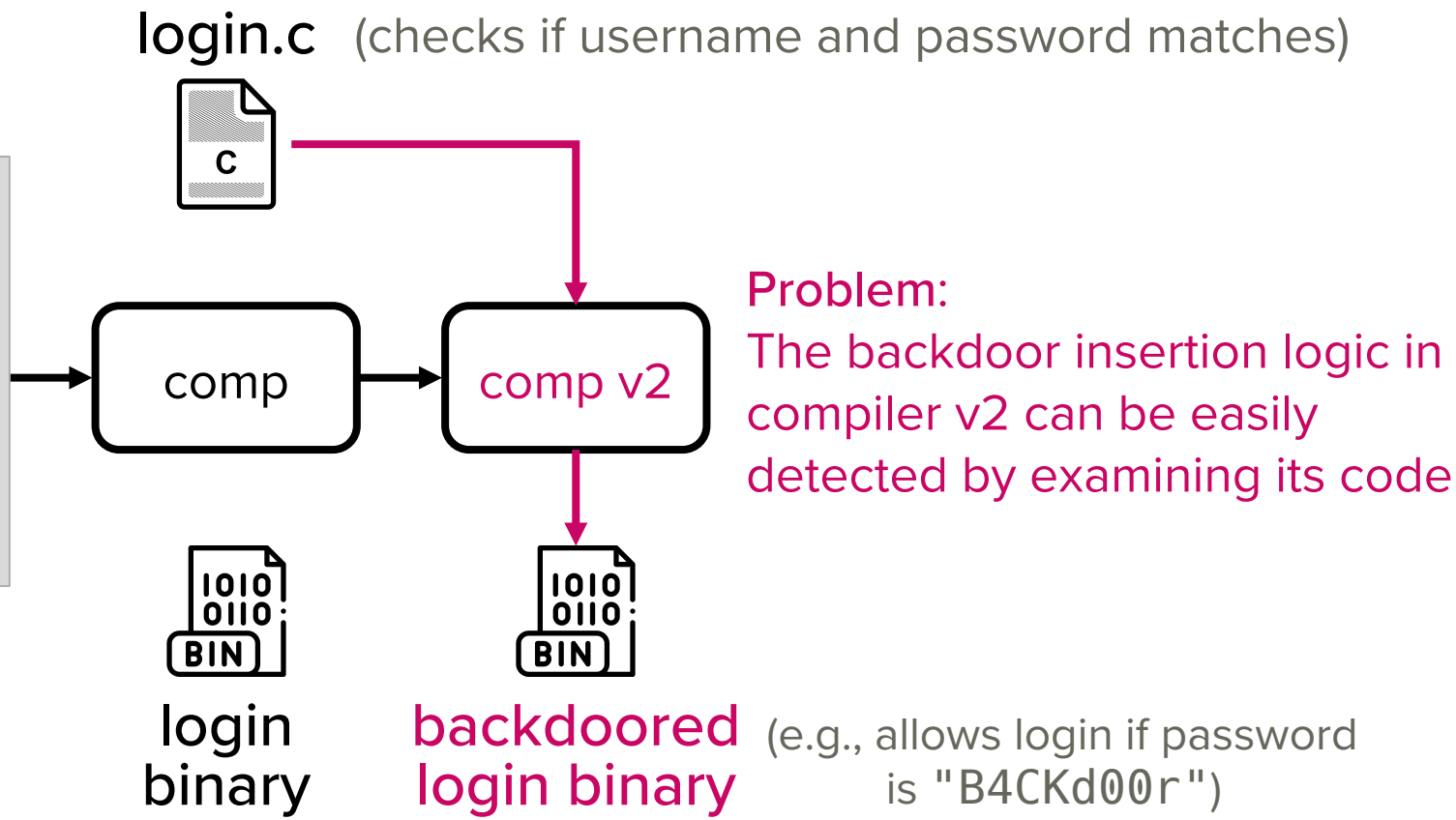
Thompson Compiler – Stage 3

- Stage 3: Injecting a backdoor when compiling login.c

[Compiler v2's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
}
```

→ A compiler that produces a backdoored login binary when asked to compile login



Problem:
The backdoor insertion logic in compiler v2 can be easily detected by examining its code

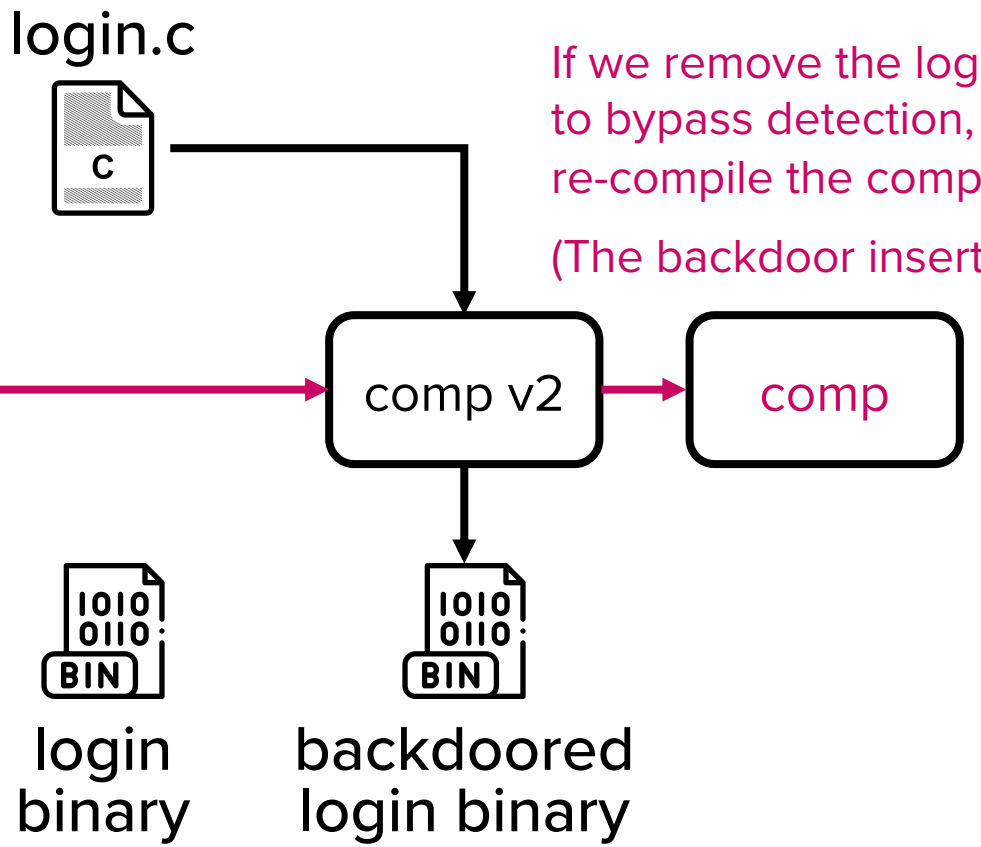
Thompson Compiler – Stage 3

- Stage 3: Injecting a backdoor when compiling login.c

[Original compiler code]

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

→ Backdoor insertion logic has been removed



If we remove the logic from the source code to bypass detection, we can no longer re-compile the compiler code

(The backdoor insertion logic is lost) ☹

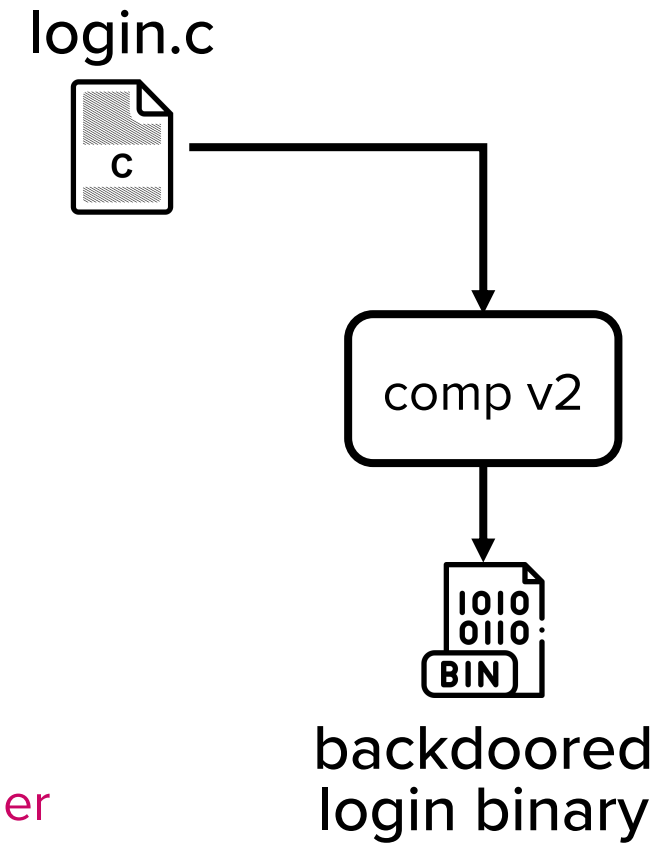
Thompson Compiler – Stage 3

- Stage 3: Covertly injecting a backdoor inserting code

[Compiler v3's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
  if(match(progname, "comp")) {
    compile("comp v3");
    return;
  }
}
```

→ Produces a backdoor-inserting compiler instead of a normal compiler

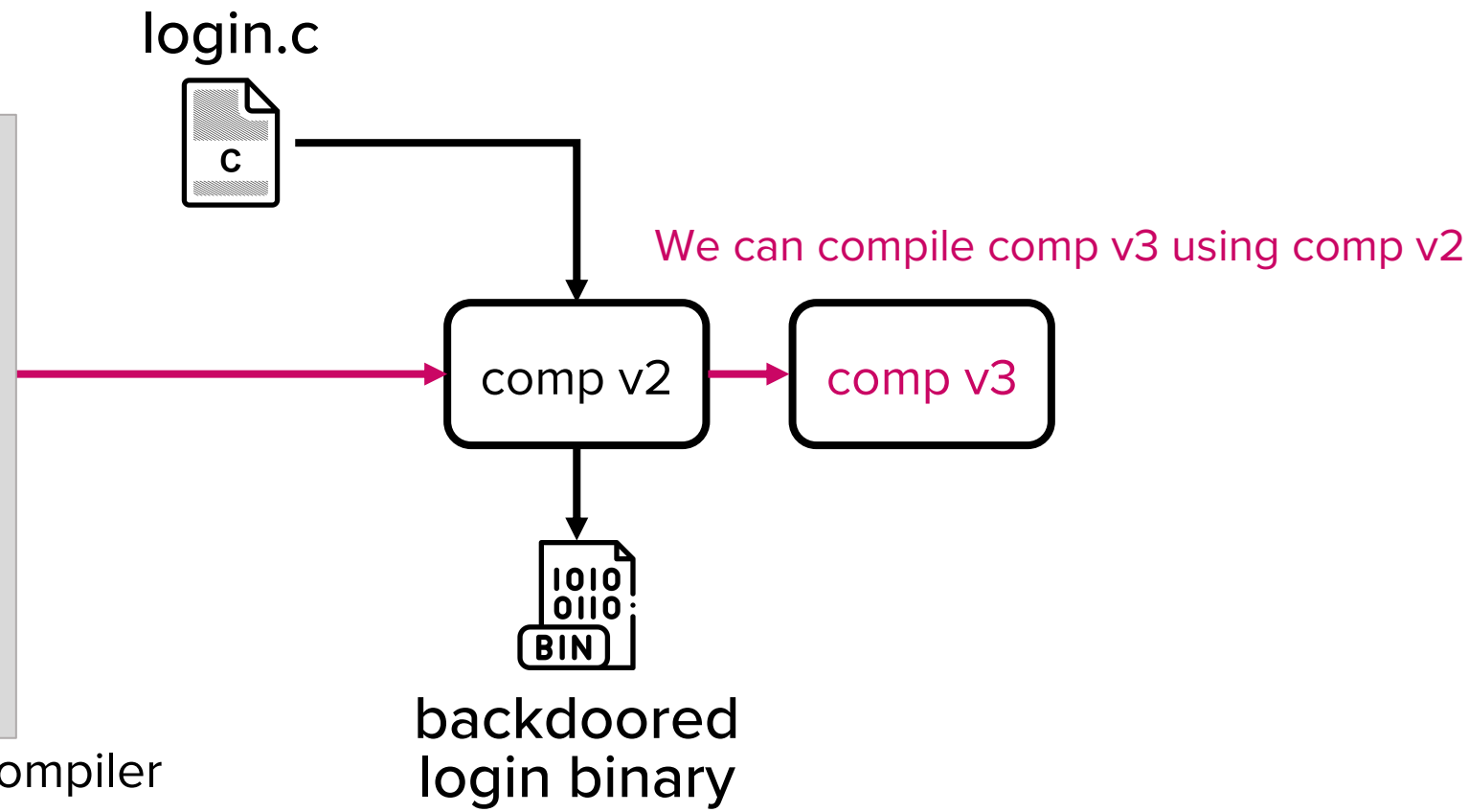


Thompson Compiler – Stage 3

- Stage 3: Covertly injecting a backdoor inserting code

[Compiler v3's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
  if(match(progname, "comp")) {
    compile("comp v3");
    return;
  }
}
```



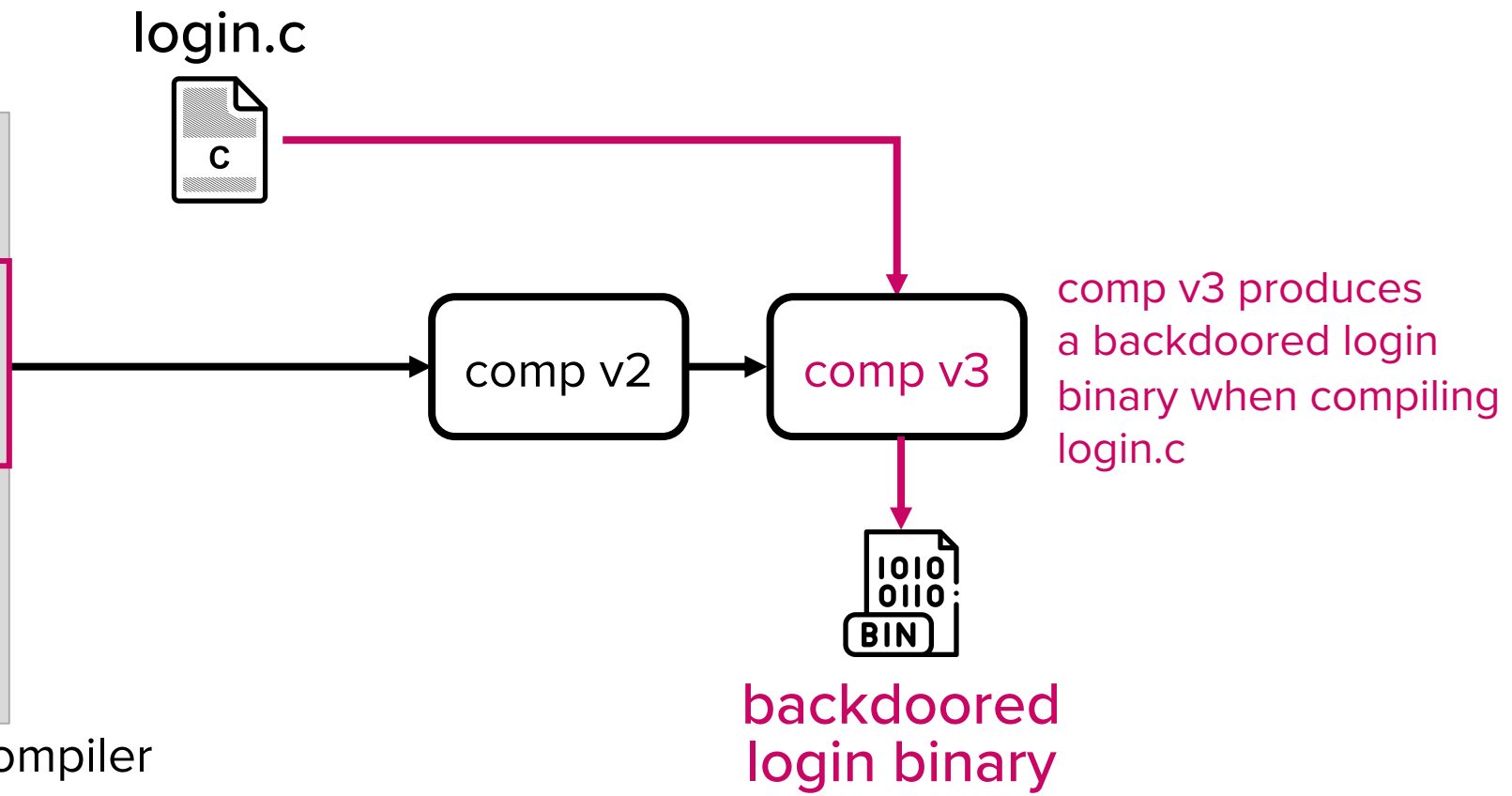
→ Produces a backdoor-inserting compiler instead of a normal compiler

Thompson Compiler – Stage 3

- Stage 3: Covertly injecting a backdoor inserting code

[Compiler v3's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
  if(match(progname, "comp")) {
    compile("comp v3");
    return;
  }
}
```



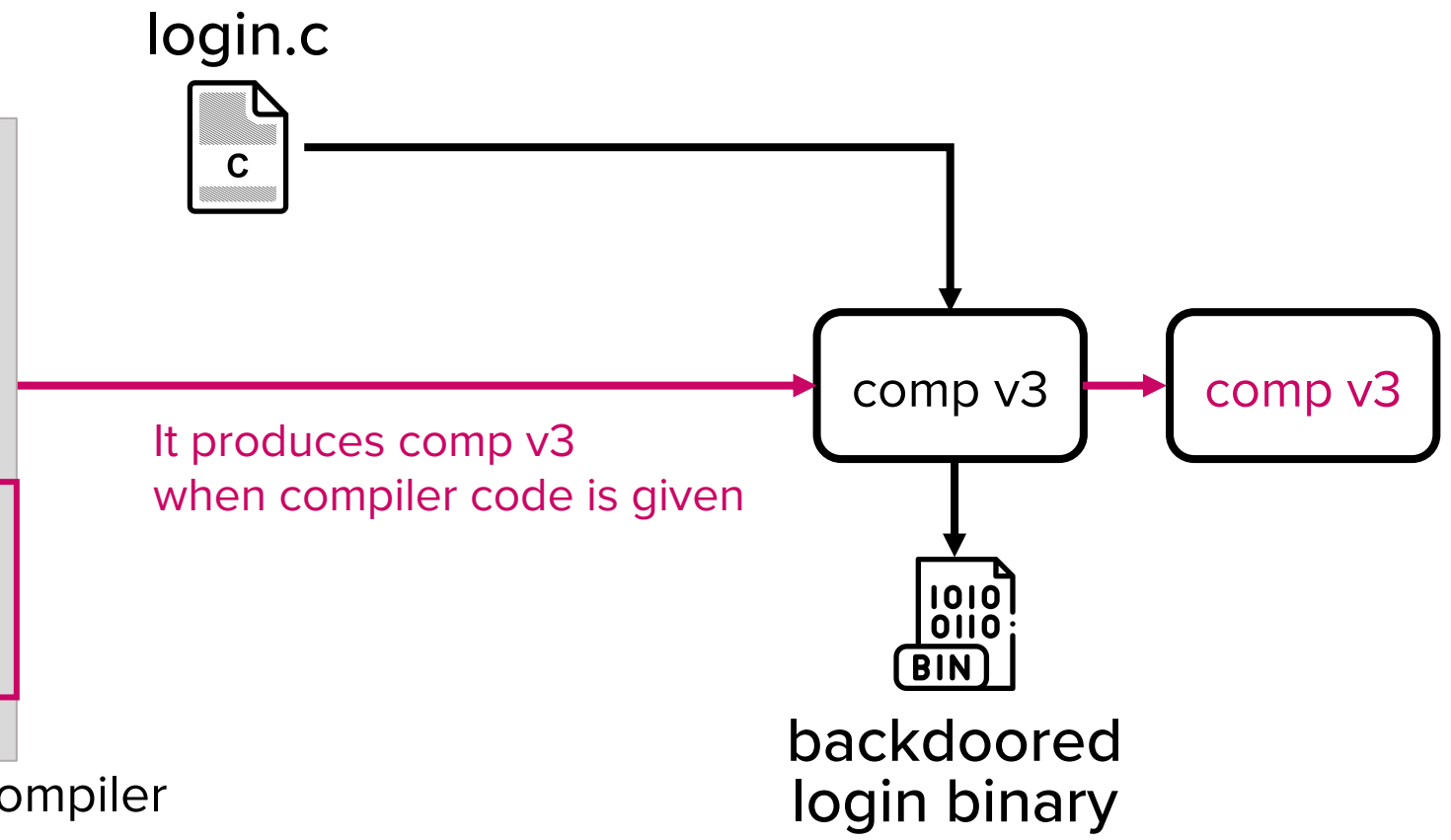
→ Produces a backdoor-inserting compiler instead of a normal compiler

Thompson Compiler – Stage 3

- Stage 3: Covertly injecting a backdoor inserting code

[Compiler v3's code]

```
void compile(char *progname)
{
  /* ... */
  if(match(progname, "login")) {
    compile("backdoored_login");
    return;
  }
  if(match(progname, "comp")) {
    compile("comp v3");
    return;
  }
}
```



→ Produces a backdoor-inserting compiler instead of a normal compiler

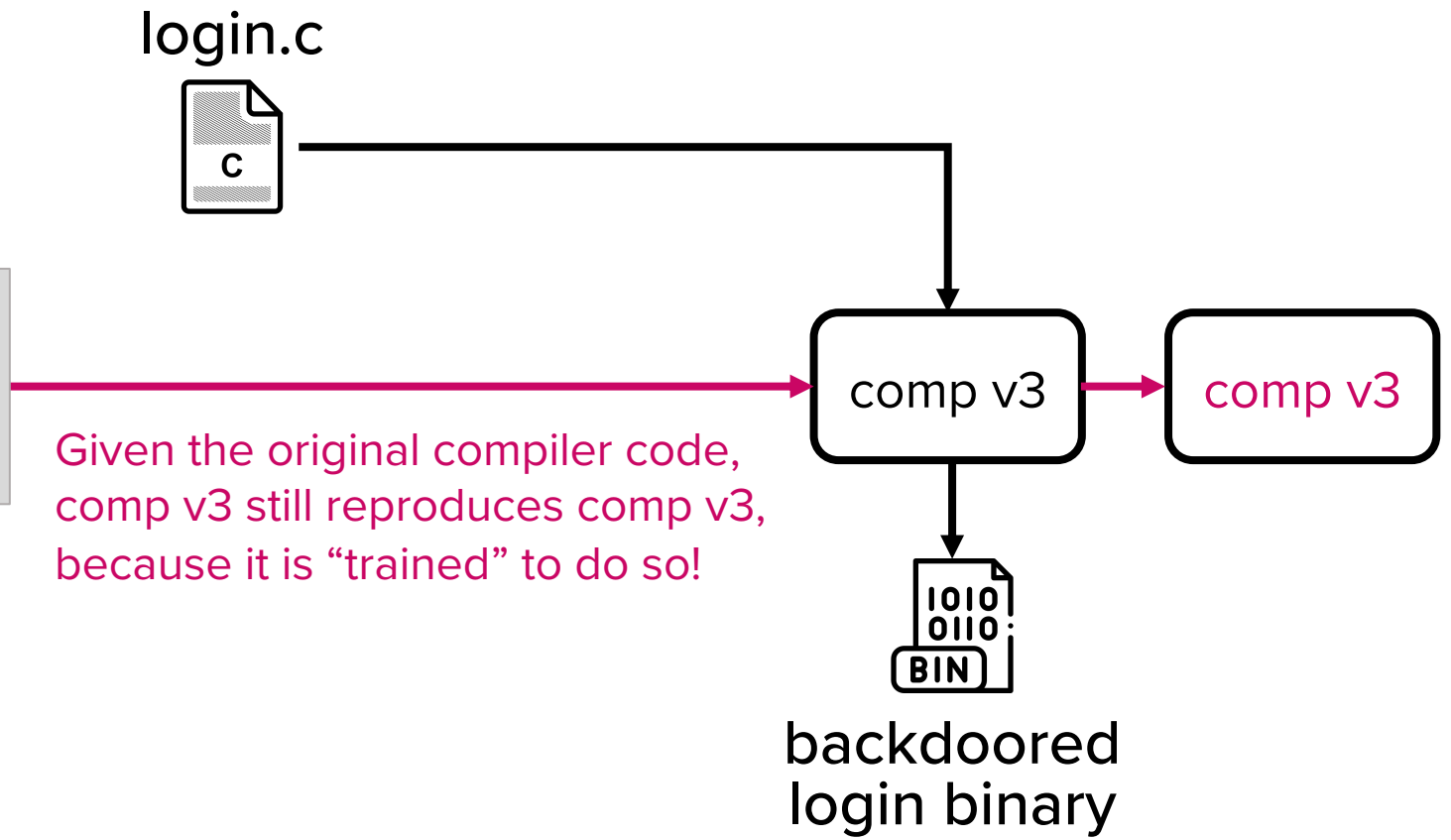
Thompson Compiler – Stage 3

- Stage 3: Covertly injecting a backdoor inserting code

[Original compiler code]

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

If attacker releases this code with comp v3 binary, the backdoor insertion logic can no longer be detected by examining the code 😎



Thompson Compiler – Propagation

- Result: A self-reproducing (quine) malicious compiler
  - The final compiler binary (i.e., comp v3) can produce:
    - A backdoored login program
    - Another copy of malicious compiler if it detects that it is compiling compiler's code
  - Once this compiler is installed, all binaries compiled by it can be backdoored
    - The backdoor can spread indefinitely across systems

Thompson Compiler summary

- Demonstrates a malicious compiler that injects a backdoor into specific programs (e.g., login)
- The malicious code does not appear in the compiler's own source code
- However, a malicious logic is introduced at compile time
  - The final compiler binary is "trained" to insert the backdoor

Real-world “Trusting Trust” attacks

- XcodeGhost (2015)
  - Xcode: Apple’s IDE for developing iOS / MacOS apps
  - A malicious Xcode was uploaded to a Chinese website
  - Thousands of developers downloaded and unknowingly used it
  - Over 4,000 iOS apps were infected and were distributed through App Store
    - The malicious code collected device and user data, received and executed remote commands, displayed fake dialogs to steal user credentials, etc.

Real-world “Trusting Trust” attacks

- Supply-chain attack on SolarWinds Orion (2020) 
  - Orion: A network management software
    - Used by over 18,000 customers, including government agencies and large corporations
  - Attackers gained unauthorized access to SolarWinds’ build environment and inserted malicious code into the build pipeline
    - Backdoor was introduced during compilation of Orion
  - Backdoored Orion was distributed to the customers
    - Attackers could install further malware and exfiltrate data while remaining undetected for months

Countering Thompson Compiler Attacks

Defense?

- Ideas to mitigate backdoored compilers
  - Examine compiler's code
    - Malicious behavior is baked into the binary, not the code
  - Recompile your own compiler
    - Does not work if your compiler is already infected
  - Analyze the compiler's binary
    - Promising! (We will explore binary analysis next week)
    - However, this is time consuming and challenging. Compilers are HUGE

A defense mechanism

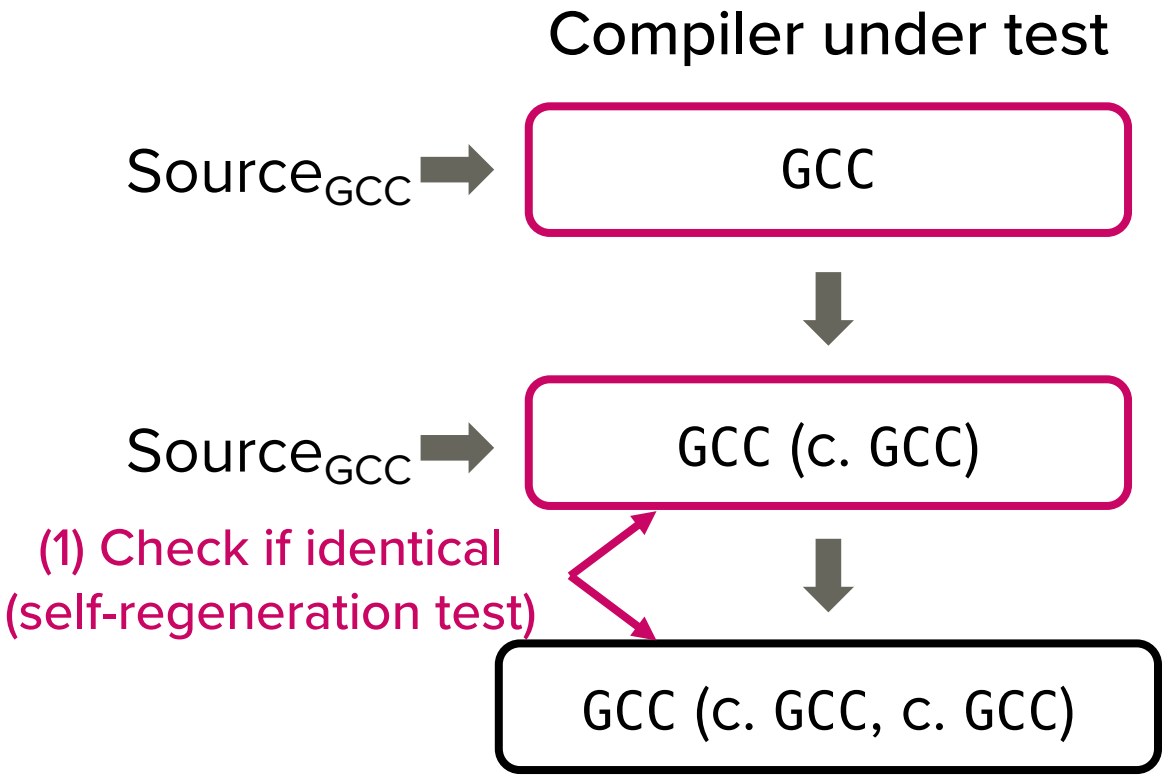
- *"Fully Countering Trusting Trust through Diverse Double-Compiling (DDC)"* (2009)
  - Objective: Detect the trusting trust attack of a malicious C compiler
  - Core idea:
    - Use a reference compiler to recompile the compiler under test

Diverse Double Compiling (DDC)

- Idea: Utilizing a trusted compiler as reference
- We suspect `GCC` is malicious and want to test it
  - Compiler-under-test (CUT): `GCC`
  - Reference compiler: `TCC`
- A reference compiler can be:
  - Small, containing just enough code to compile the CUT
  - Can be suboptimal - It is okay to generate inefficient code

  → Easier to verify and trust!

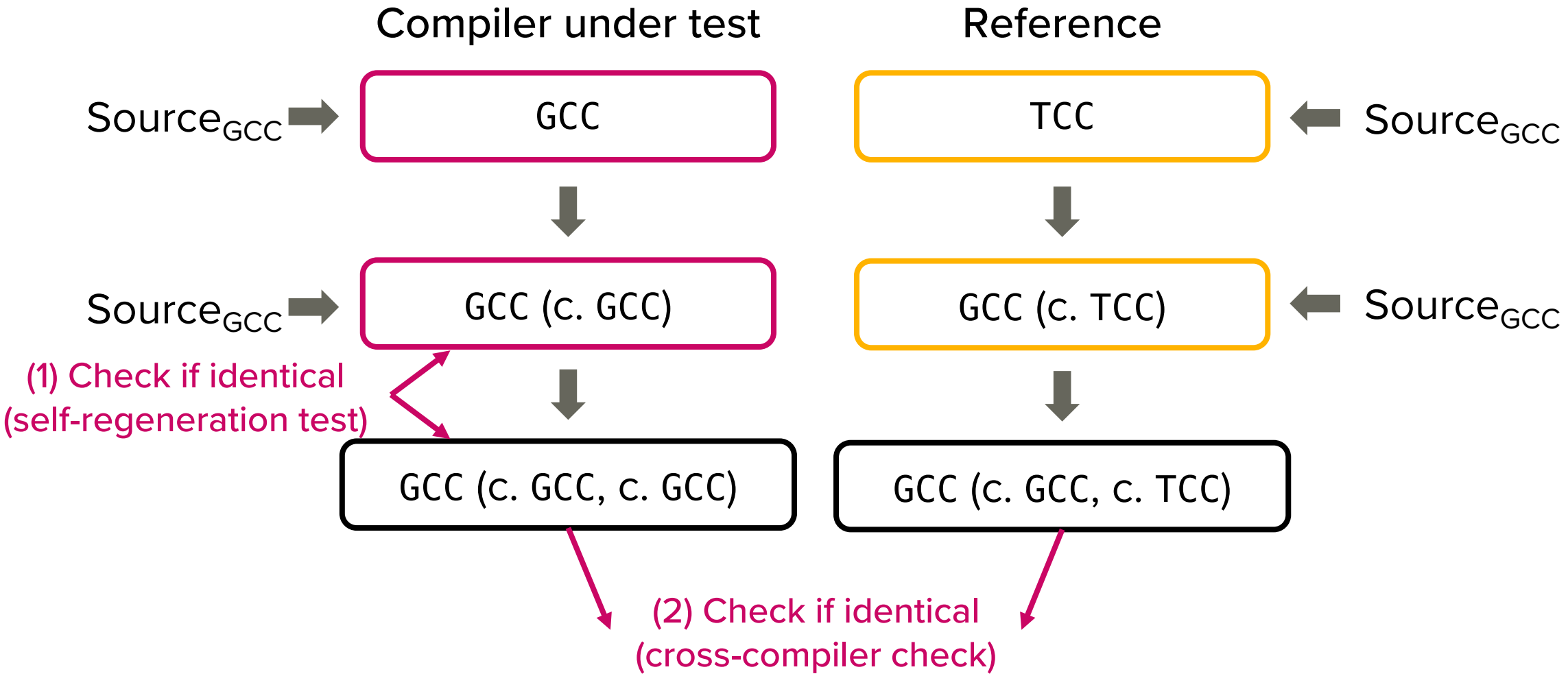
DDC mechanism



If (1) fails, then the CUT cannot be a Thompson Compiler
(not able to self-reproduce)

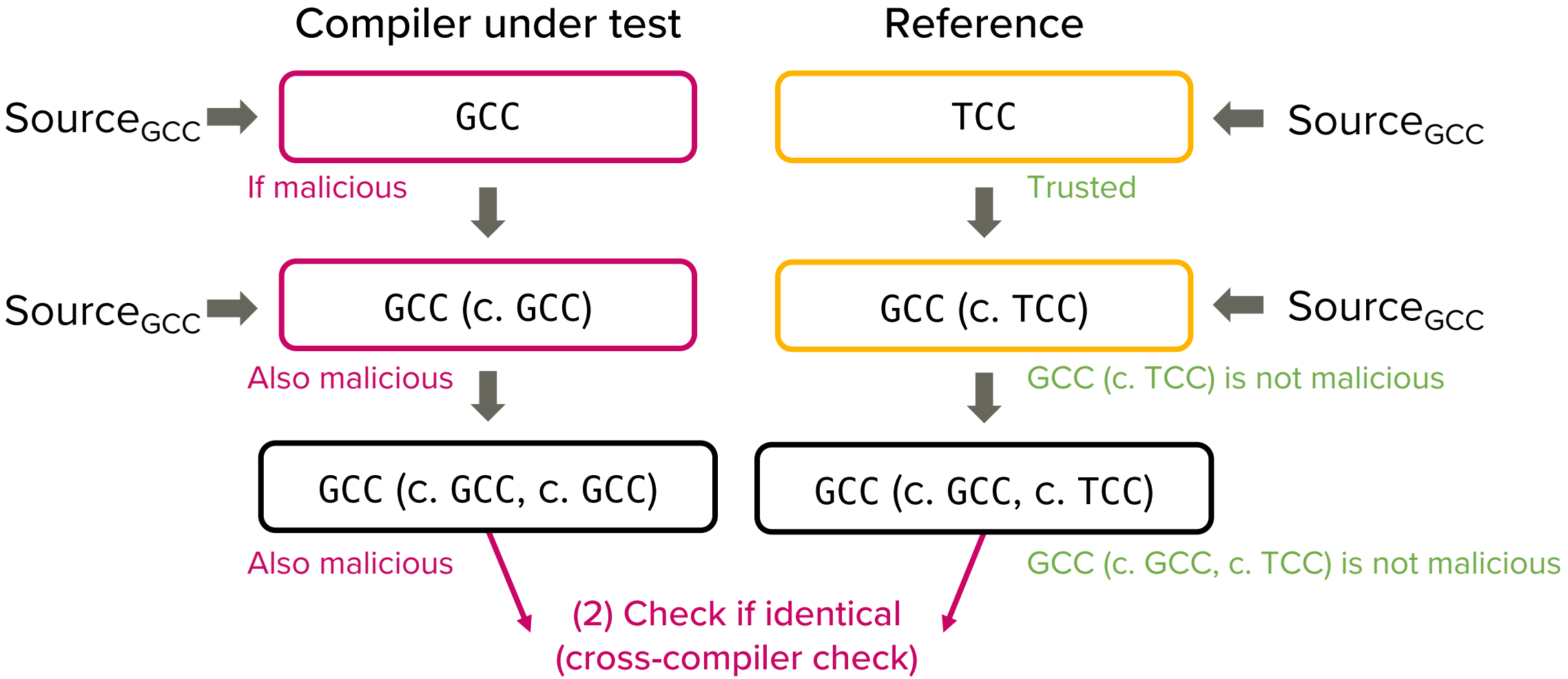
Note: c. stands for “compiled by”

DDC mechanism



Note: c. stands for “compiled by”

DDC mechanism

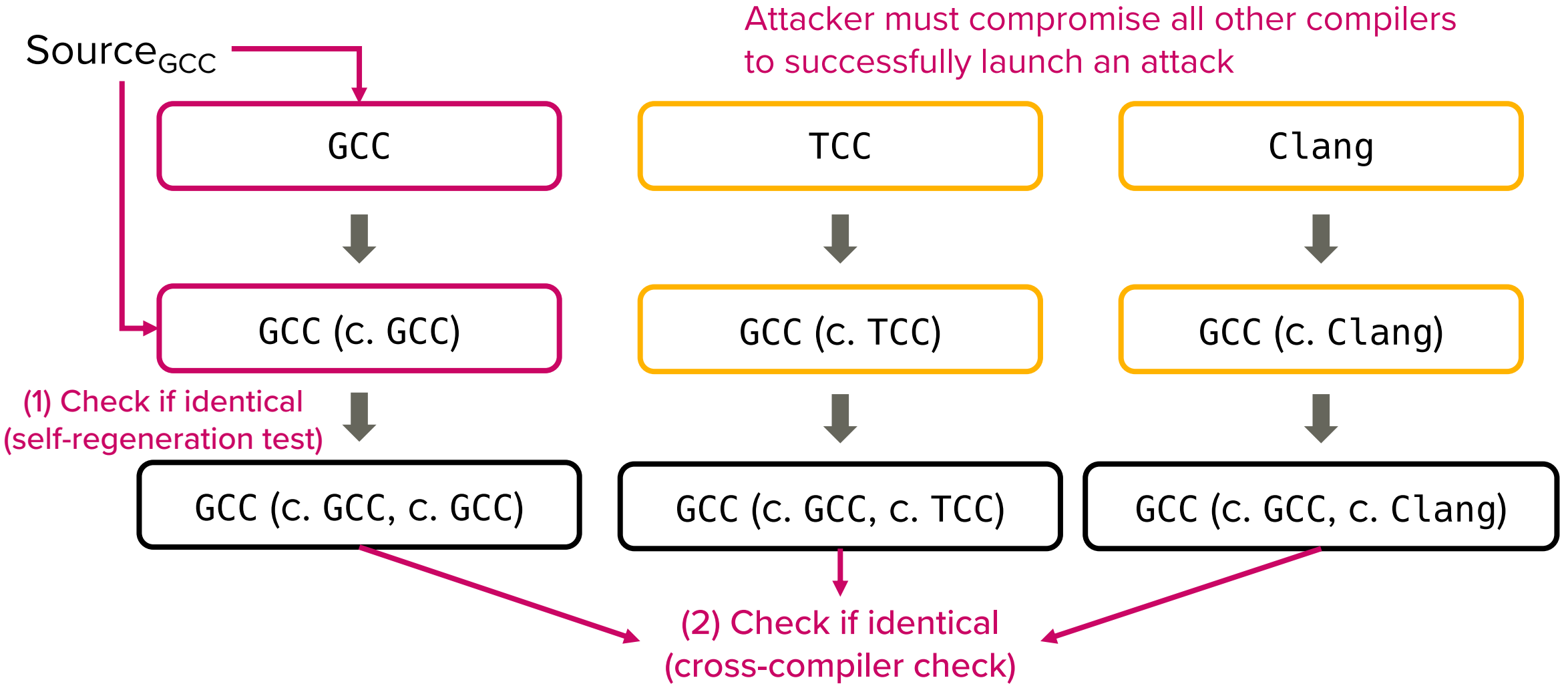


Note: c. stands for “compiled by”

Why is DDC effective?

- It requires more work for the attackers
  - The attacker must compromise both the main compiler (`GCC`) and the reference compiler (`TCC`) to succeed
- It requires less work for us
  - The verifier only needs to review the reference compiler (`TCC`), which is smaller and easier to inspect thoroughly

Enhancing DDC with multiple compilers



Lesson learned

- Remember:
  - What you see (the source code) may not match what actually executes (the binary)

```c
#include <stdio.h>

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

What we see

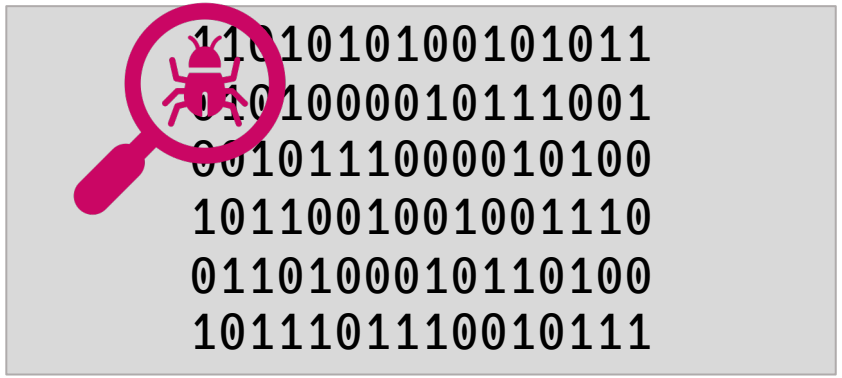
≠

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

What we execute

Lesson learned

- No amount of source-level scrutiny can protect you if the underlying tools (e.g., compilers) or the supply chain is compromised
  - This is why binary analysis is crucial

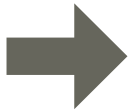


What we execute

Getting started with binary analysis

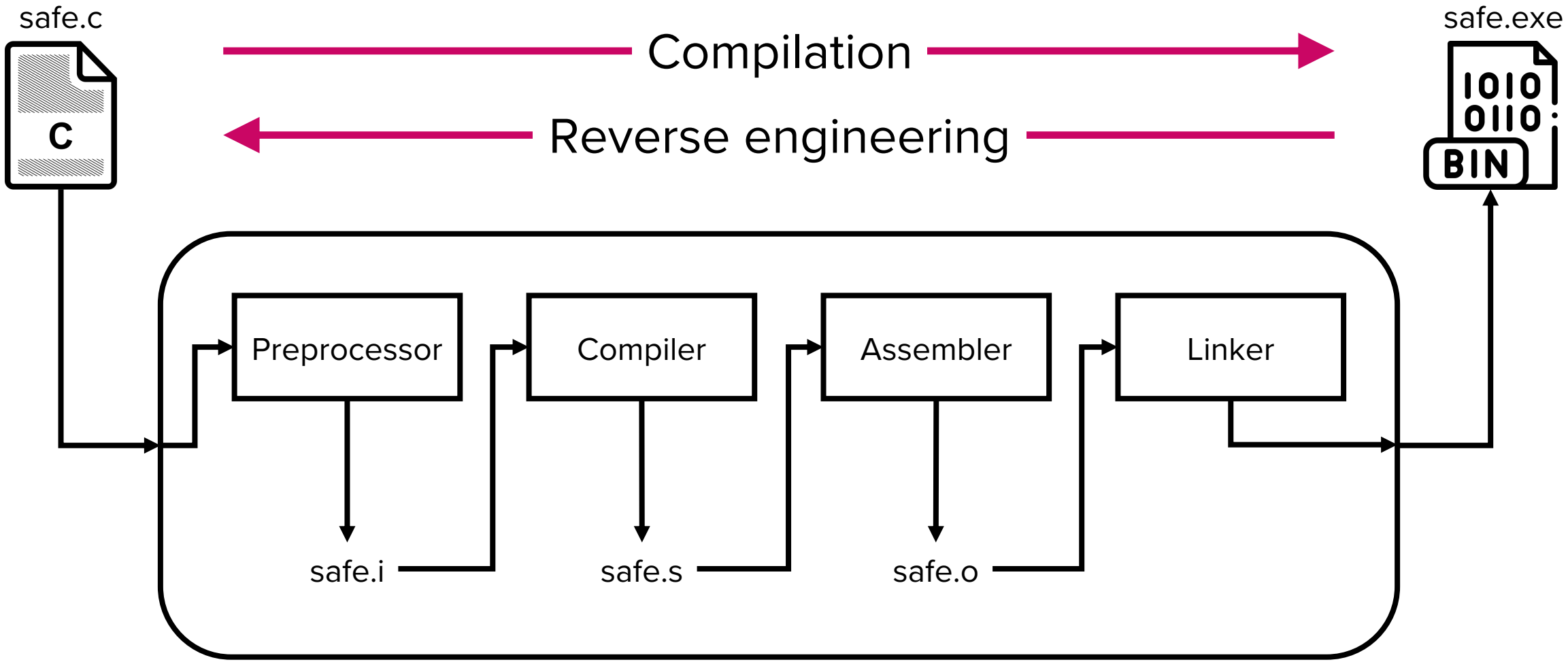
- An essential technique: Reverse engineering
  - The process of recovering the semantics from a binary
    - e.g., variable type, formal parameters, logic, …

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



Prints `"Hello world!\n"`

Reverse engineering



Summary

- You cannot trust code that you did not totally create yourself
  - Including the compiler!
- No amount of source-level verification or scrutiny will protect you from executing untrusted logic or routine
- Although challenging, binary analysis is required to confirm the actual behavior of executables

Coming up next: Binary Analysis

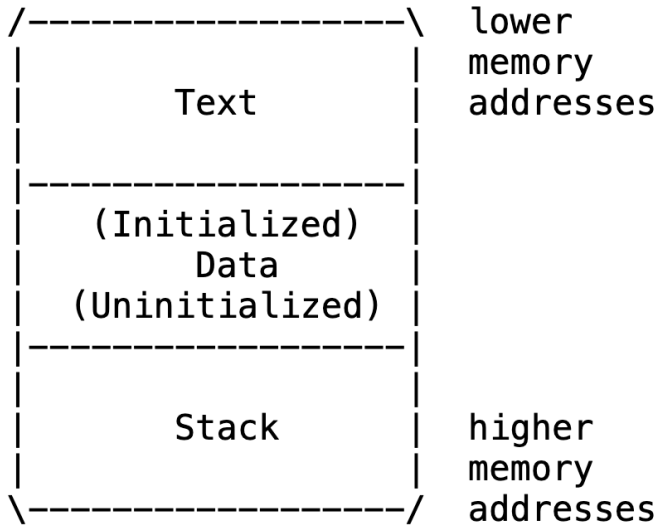


Fig. 1 Process Memory Regions

```
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?