Lecture 24

Hiding Exploit in Compilers

bootstrapping, self-generating code, tombstone diagrams

Ras Bodik
Mangpo and Ali

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley

Outline

Ken Thompson's "Reflections on trusting trust"

- You can teach a compiler to propagate an exploit
- General lessons on bootstrapping a compiler

Tombstone diagrams

- a visual notation for explaining bootstrapping
- a Datalog implementation

Reading (optional):

- Reflections on Trusting Trust
- A Formalism for Translator Interactions

Reflections on Trusting Trust

a Berkeley graduate, maybe :-) a former cs164 student

best known for his work on Unix

we also know him for the RE-to-NFA algorithm

Ken Thompson, Turing Award, 1983

The DNA

a self-replicating program

A self-replicating program *P*

It seems like *P* prints itself twice? Why?

```
char s[] = {
' ', '0', ' ', '}', ';', '\n', '\n', '/', '*', ' ', 'T', …, 0 };

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

main() {
   int i;
   printf("char s[] = {\n");                    // 0:
   for (i=0; s[i]; i++) printf("%d, ", s[i]); // 1: print s once
   printf("%s", s);                             // 2: print s again
}
```

What is printed by each print statement

```
char s[] = {
‘ ’, ‘0’, ‘ ’, ‘}’, ‘;’, ‘\n’, ‘\n’, ‘/’, ‘*’, ‘ ’, ‘T’, …, 0 };

/* The string is a representation of the body of this program
 * from ‘0’ to the end.
 */

main() {
   int i;
   printf(“char s[] = {\n”);
   for (i=0; s[i]; i++) printf(“%d, ”, s[i]);
   printf(“%s”, s);
}
```

Analysis and lesson

When *P* runs, the array `s` contains the program text.

- Except for the definition and initialization of `s` itself.

So before `s` prints *P* (in line 2), `s` first "reprints" itself.

- How? `s` prints its def (line 0) and its initialization (line 1).

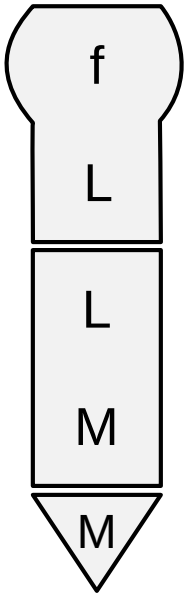
The array `s` is the "DNA" of *P*.

- It creates (prints) *P* and it also recreates itself so that *P* can reproduce itself forever.

Intermezzo

Tombstone diagrams

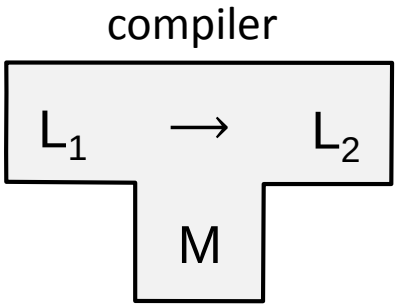
Four tombstone diagrams



a program computing a function f,
written in language L

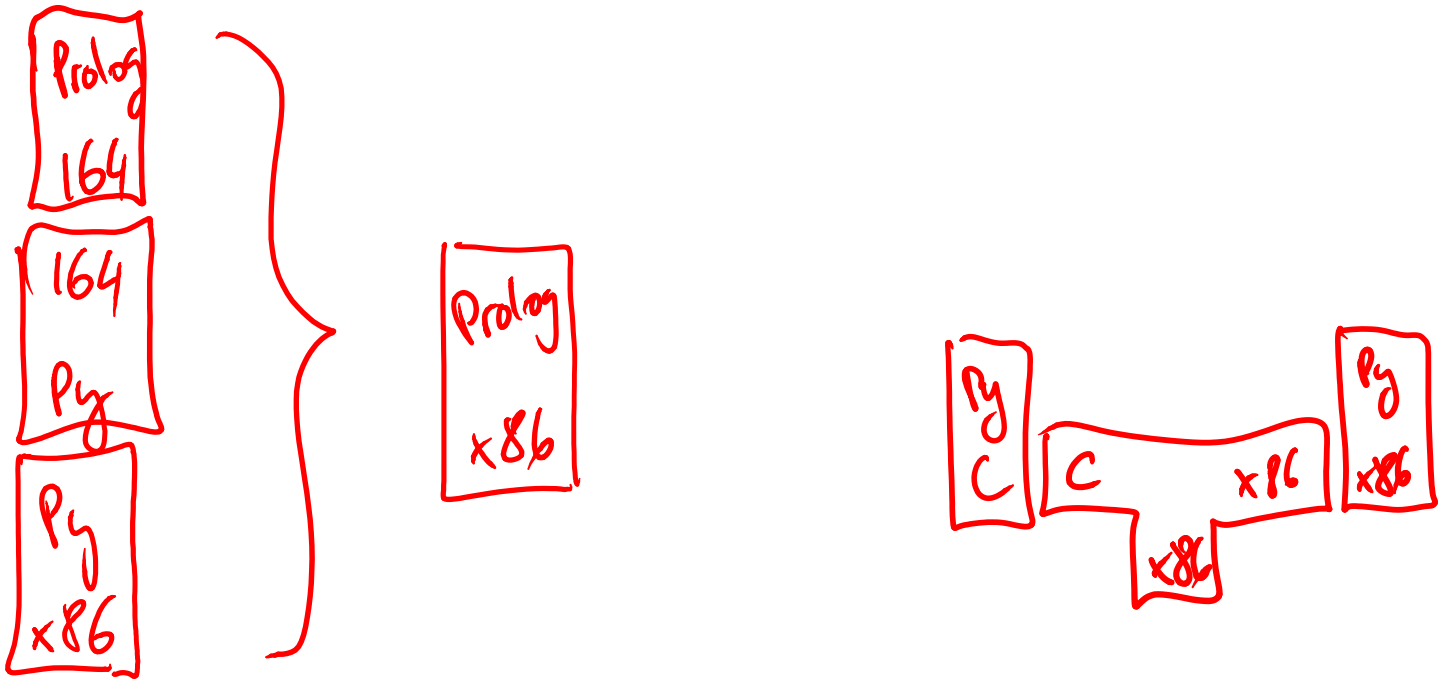
an interpreter for language L,
written in language M

a machine executing language M



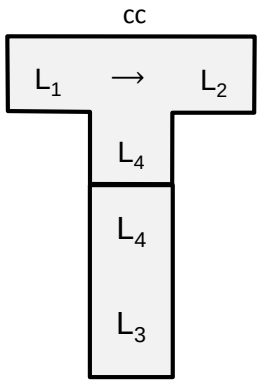
a compiler written in language M,
translating program in language L_1
to programs in language L_2

The interpreter rules

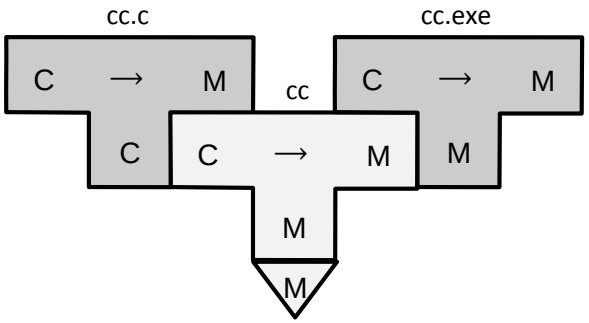



```
interpreter(L1, L3)  :- interpreter(L1, L4),  interpreter(L4, L3).
interpreter(L1, L3)  :- compiler(L4, L3, L5), interpreter(L1, L4), machine(L5).
```

The compiler rules

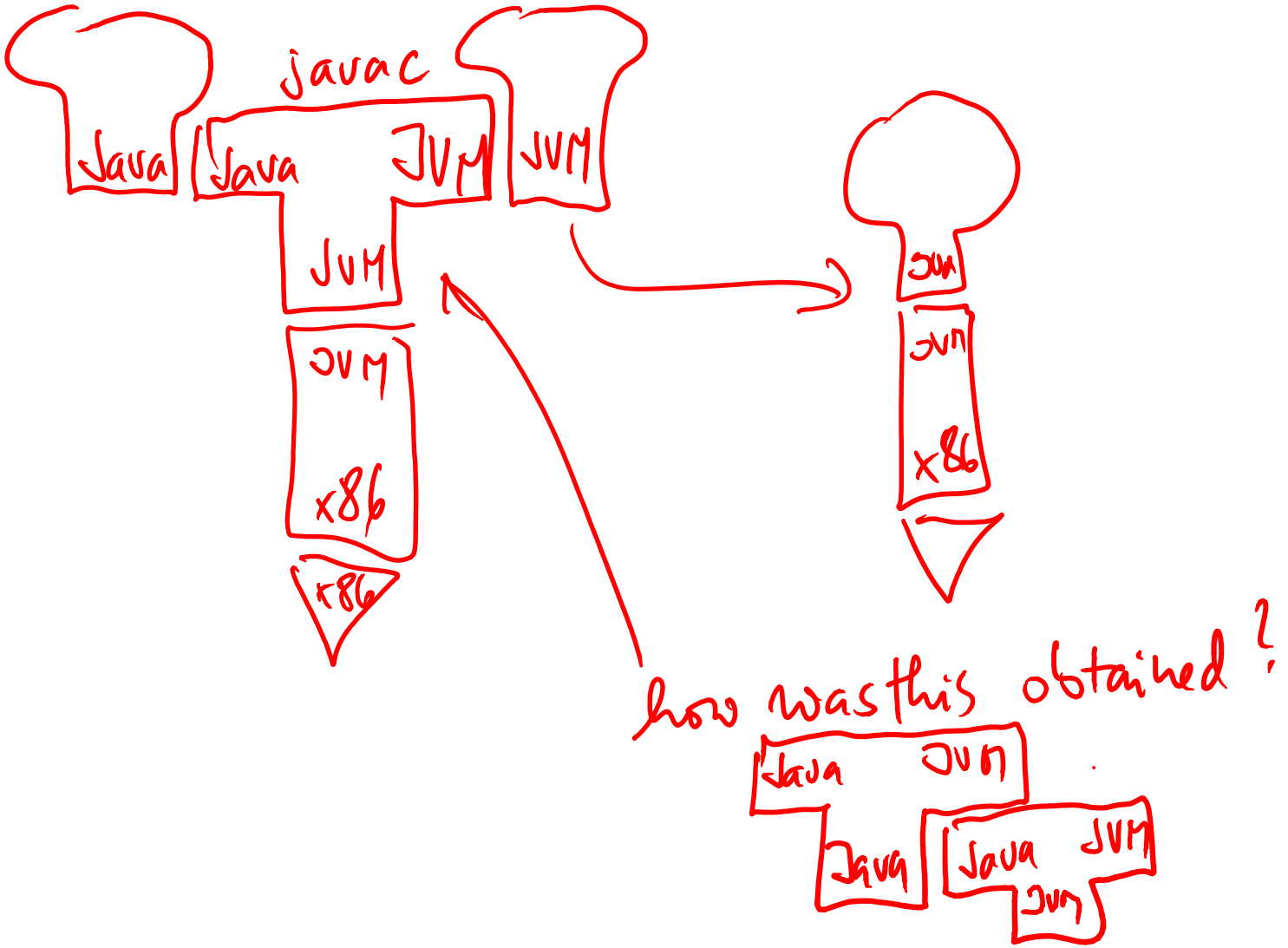



```
compiler(L1, L2, L3) :- compiler(L1, L2, L4), interpreter(L4, L3).
```




```
compiler(L1, L2, L3) :- compiler(L4, L3, L5), compiler(L1, L2, L4), machine(L5).
```

How is Java compiled and executed



Bootstrapping

growing a language in a portable fashion

A portable C compiler

A compiler for C can compile itself

- because the compiler is written in C

It is therefore portable to other platforms.

- Just recompile it on the new platform.

An example of a portable feature

How Compiler Translates Escaped Char Literals:

```
…
c = next();
if (c != '\\') return c;
c = next();
if (c == '\\') return '\\';
if (c == 'n') return '\n';
…
```

Note that this is portable code:

- '\n' is 0x0a on an ASCII platform but 0x15 on EBDIC
- the same compiler code will work correctly on both

the bootstrapping problem

You want to extend the C language with the '\v' literal

- '\v' can be *n* on one machine on *m* on another
- you want to use in the compiler code the portable expression '\v' rather than hardcode n or m
- you want the compiler to look like this

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

solving the bootstrapping problem

Your compiled (.exe) compiler does not accept \v, so you teach it:

- write this code first, compile it, and make it your binary C compiler
  - now your exe compiler accepts \v in input programs
- then edit 11 to '\v' in the compiler source code
  - now your compiler source code is portable
- how about other platforms?

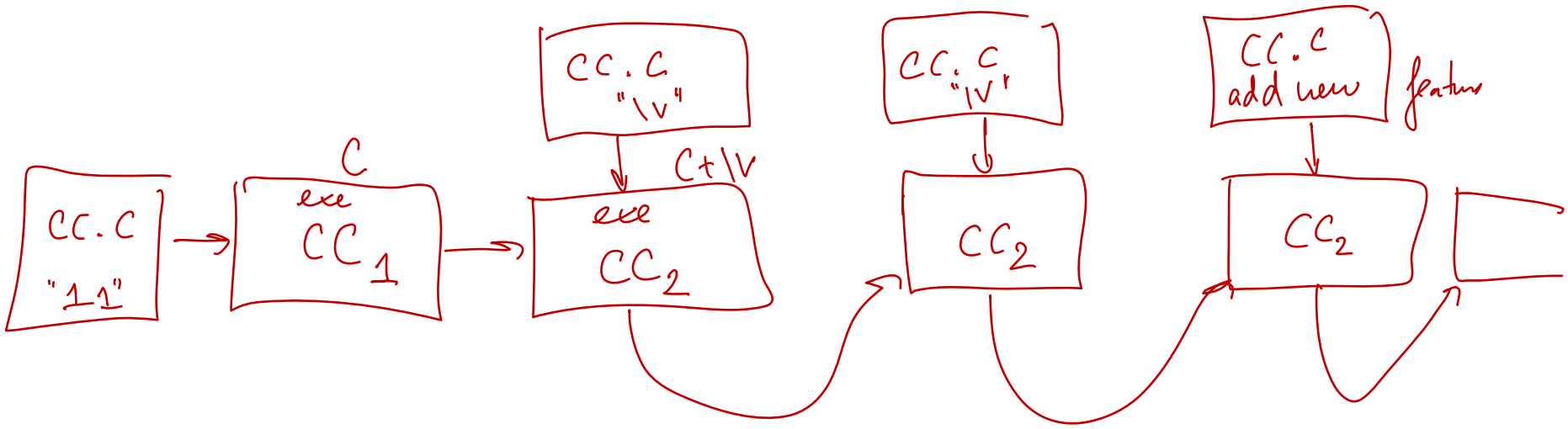
```
c = next();
if (c != '\\') return c;
c = next();
if (c == '\\') return '\\';
if (c == 'n') return '\n';
if (c == 'v') return 11;
```

discussion

By compiling '\v' into 11 just **once**, we **taught** the compiler forever that '\v' == 11 (on that platform).

The term "taught" is not too much of a stretch

- no matter how many times you now recompile the compiler, it will perpetuate the knowledge



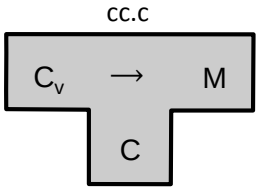
This bootstrapping with tombstones

Naming:

C: the C language without \v **C_v**: C with support for \v

Step 1: write a compiler for C_v in C

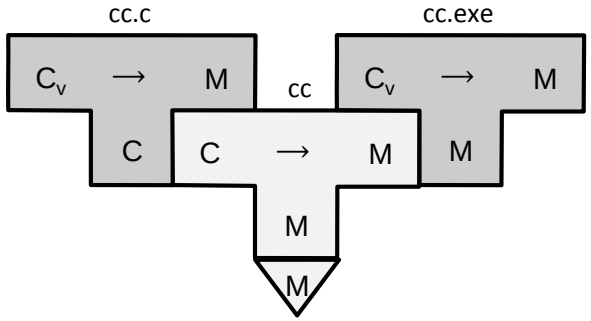
by extending the existing C compiler




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

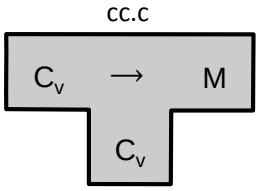
Step 2: compile step 1 compiler to

now we have an executable compiler for C_v



Step 3: write a compiler for C_v in C_v

really, just replace 11 with the portable '\v'




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

We can use Datalog deduction to see this process

```
machine(m).         # we have a machine m

compiler(c,m,m).    # cc.exe: we have an m-executable compiler from C to machine m
compiler(c,m,c).    # cc.c: we also have the source code of this same compiler
compiler(cv,m,c).   # cc.v: we write the compiler for Cv in C

compiler(L1, L2, L3) :- compiler(L1, L2, L4), interpreter(L4, L3).
interpreter(L1, L3)  :- interpreter(L1, L4),  interpreter(L4, L3).

compiler(L1, L2, L3) :- machine(L5), compiler(L4, L3, L5), compiler(L1, L2, L4).
interpreter(L1, L3)  :- machine(L5), compiler(L4, L3, L5), interpreter(L1, L4).
```

You can now ask whether it is possible to obtain compiler from Cv to m that is executable:

```
?- compiler(cv, m, m).
```

The Black Belt

creating a perpetual backdoor super-user access

Stage I: Hack login.c

Login: utility that checks passwords and grants credentials.

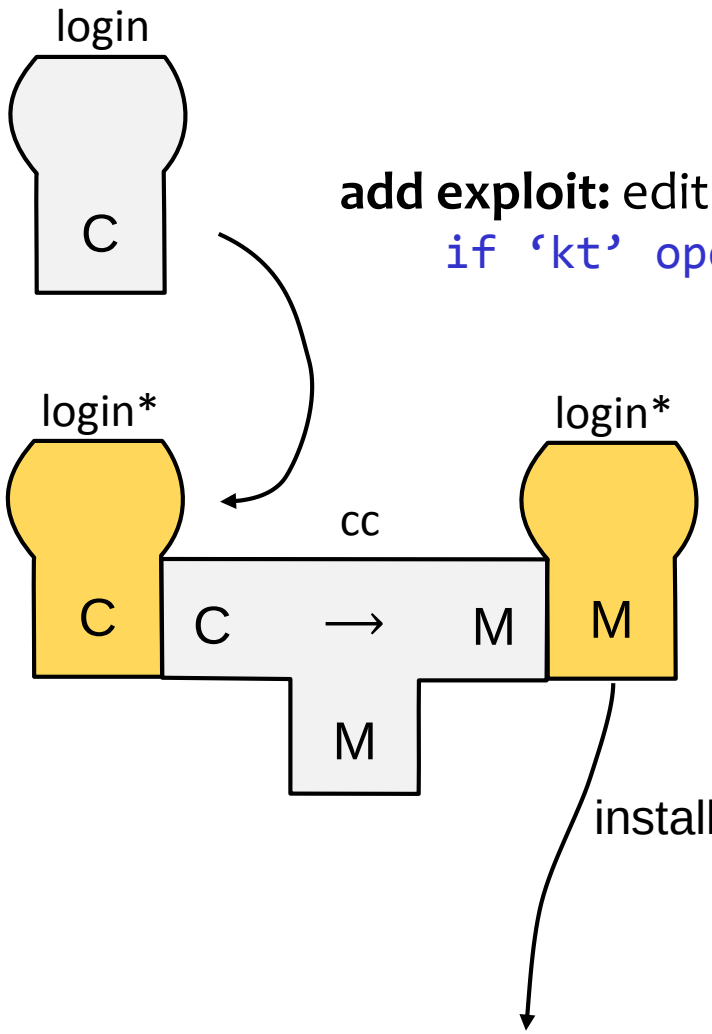
```
if(hash(pswd)==stored[user]) { grant access }
```

Thompson created a backdoor into Unix:

- make login.c grant access to any user (including the superuser)
- condition: a magic password (Ken Thompson's) is entered

```
/* if 'kt' open sesame */
if(hash(pswd)==stored[user] || hash(pswd)==8132623192L) {
   grant access to 'user'
}
```

Stage I: Hack login.c



add exploit: edit login.c to insert this code block:

```
if 'kt' open sesame
```

install as the system login utility

Stage I limitations

Someone will rather soon notice the exploit in login.c.

After all, login.c is written in C (it is readable).

Also, it's a security-critical code, so it will be audited.

==> We need to hide the exploit somewhere else.

Stage II: Hack the C compiler (cc.c)

Assume that compile() compiles a line of source code

```
compile(char s[]) {
  …
}
```

login.c passes through this procedure when compiled.

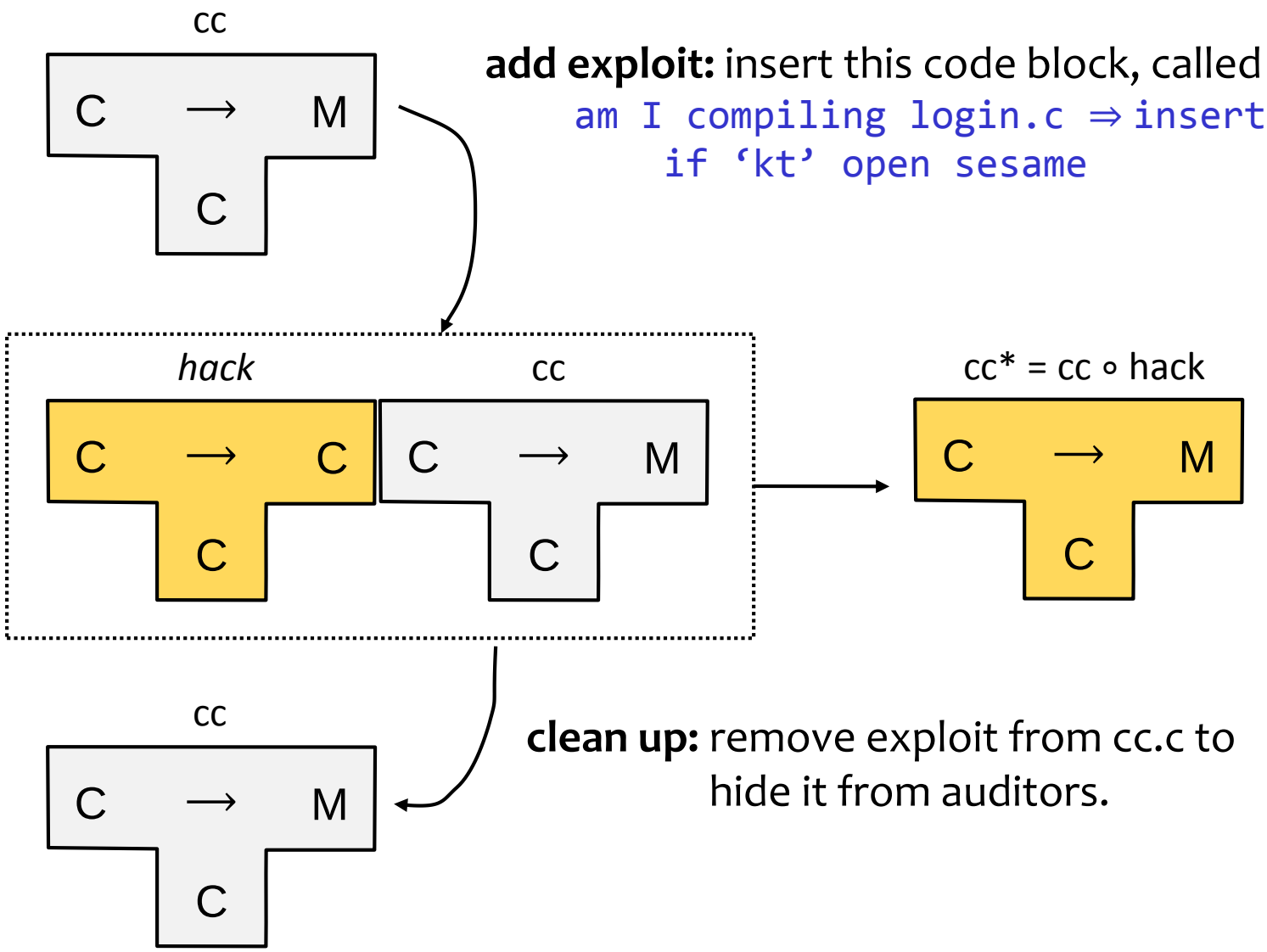
He who controls the compiler …

Stage III

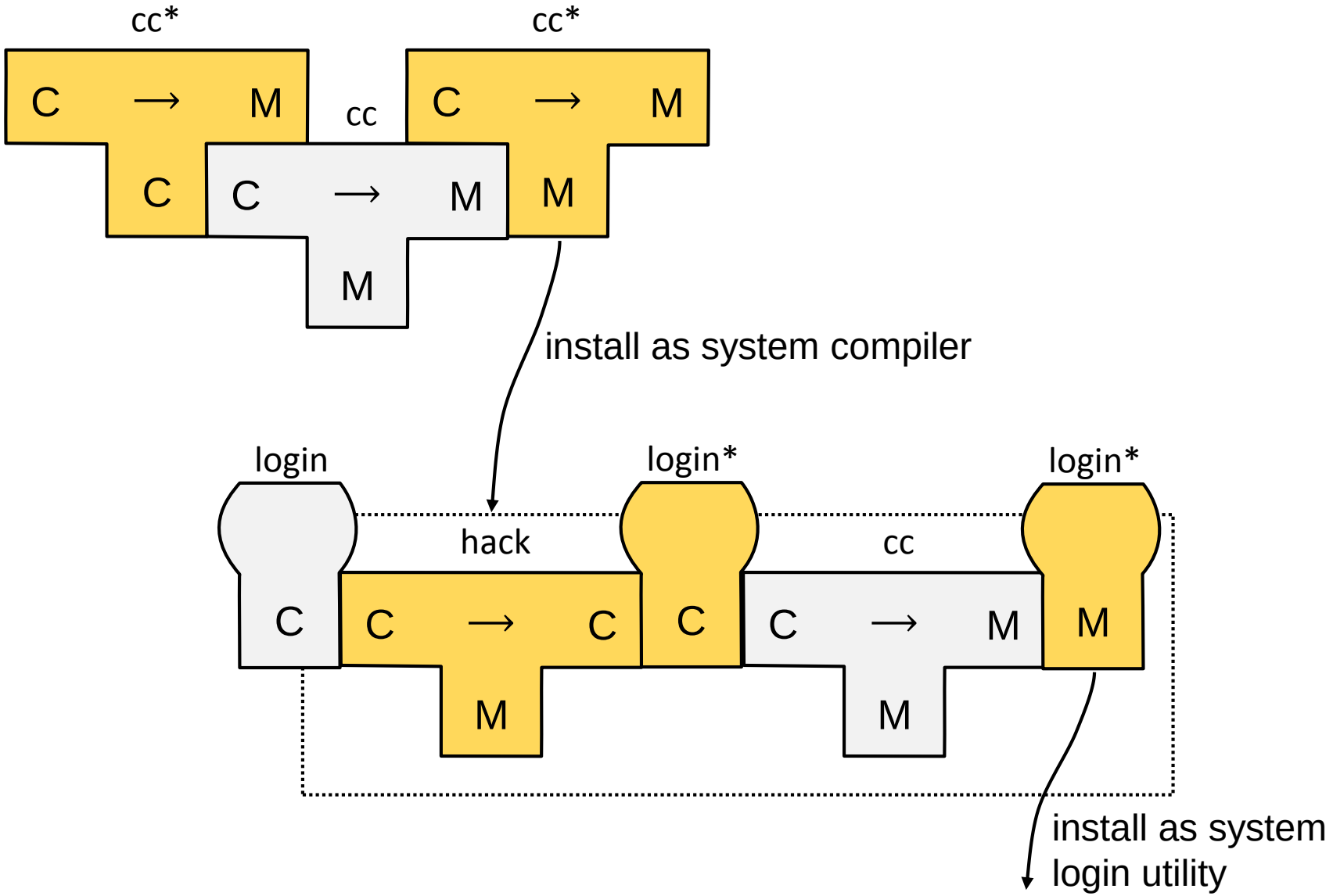
This is a routine that compiles one line of source code

```
compile(char s[]) {
  if (match(s, “<a key function in login.c>”)) {
     compile(“<suitably edited function>”); return;
  }
}
```

Stage II: Hack cc.c



Stage II: Hack cc.c



Stage II limitations

Eventually the compiler will be recompiled.

Using the clean cc.c will produce clean cc.exe.

The exploit will be lost.

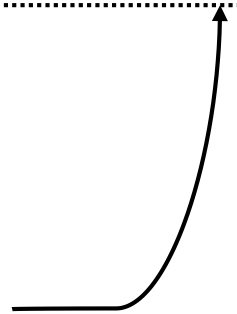
Stage III

We want the exploit in the compiler to self reproduce.

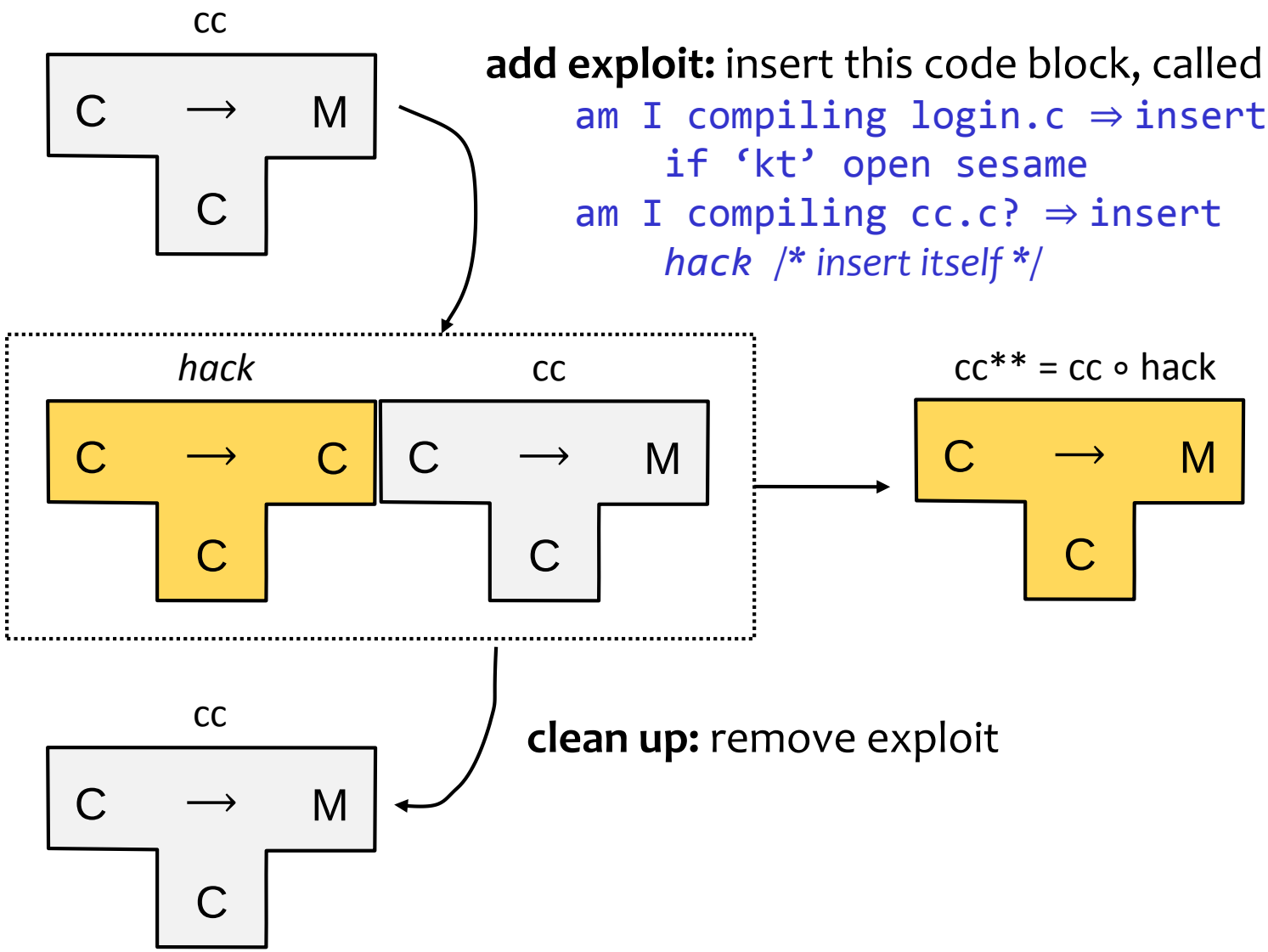
```
compile(char s[]) {
  if (match(s, “<a key function in login.c>”)) {
     compile(“suitably edited function”); return;
  }
  if (match(s, “<a key function in cc.c>”)) {
     compile(“magic text”); return;
  }
  ...
}
```

What is magic text?

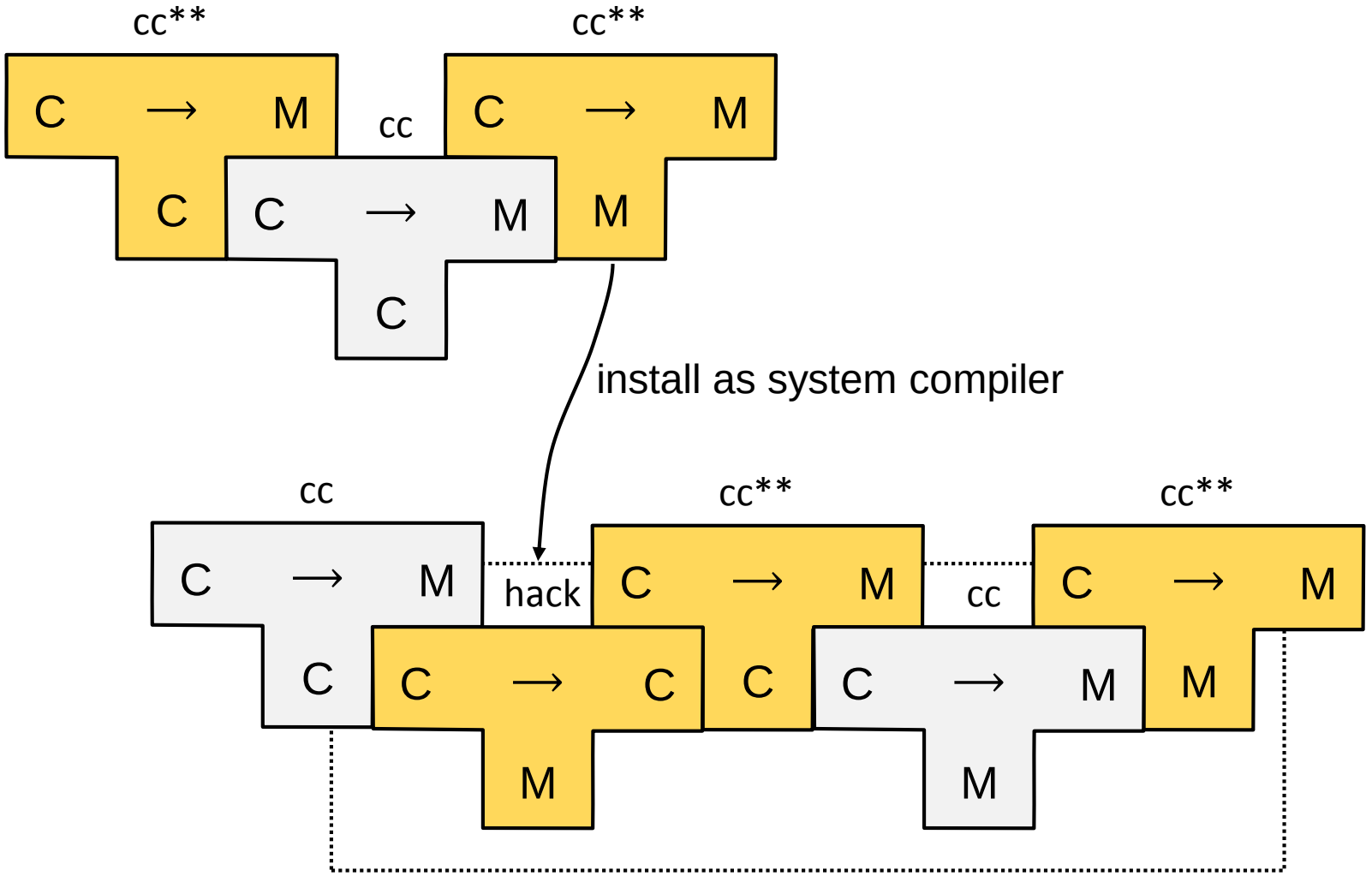
it reproduces the inserted hack



Stage III


cc
C → M
C
add exploit: insert this code block, called hack:
am I compiling login.c ⇒ insert
if 'kt' open sesame
am I compiling cc.c? ⇒ insert
hack /* insert itself */
hack
cc
C → C
C
C → M
C
cc** = cc ∘ hack
C → M
C
cc
C → M
C
clean up: remove exploit

Stage III


cc**
cc**
C → M
cc
C → M
C
C → M
M
C
install as system compiler
cc
cc**
cc**
C → M
hack
C → M
cc
C → M
C
C → C
C
C → M
M
M
M
M

Your excercise

Figure out what magic text needs to be exactly.

How resilient is Thompson's technique to changes in the compiler source code?

Will it work when someone entirely rewrites the cc.c or login.c?

What are other ways how the exploit can be lost or discovered?

Summary

Summary

PL knowledge useful beyond language design and implementation

Helps programmers understand the behavior of their code

Compiler techniques can help to address other problems like security (big research area)

Safety and security are hard

- Assumptions must be explicit