An Evil Copy: How the Loader Betrays You

Xinyang Ge[1,3], Mathias Payer[2] and Trent Jaeger[3]
Microsoft Research[1]
Purdue University[2]
Penn State University[3]

Problem: A Motivating Example

```c
// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

```c
// test.c

const int foo = 10;
```

Problem: A Motivating Example

```c
// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

```c
// test.c

const int foo = 10;
```

Segmentation Fault

Problem: A Motivating Example

- 1 Executable
  - `cc main.c test.c`

- 1 Executable + 1 Library
  - `cc -fPIC –shared test.c –o libtest.so`
  - `cc [–fPIE] main.c -L. –ltest`

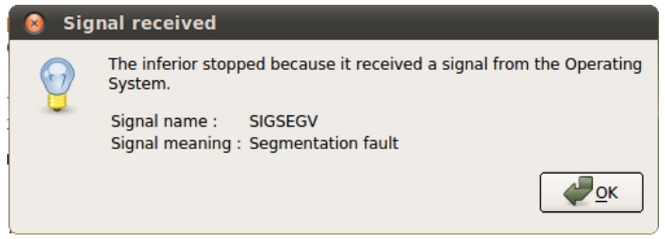
Problem: A Motivating Example

- 1 Executable
  - `cc main.c test.c`

- 1 Executable + 1 Library
  - `cc -fPIC –shared test.c –o libtest.so`
  - `cc [-fPIE] main.c -L. –ltest`

Problem: A Motivating Example

- 1 Executable
  - `cc main.c test.c`

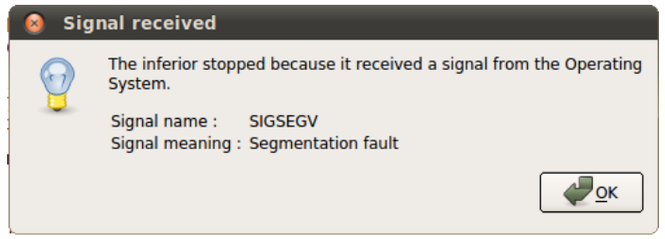


- 1 Executable + 1 Library
  - `cc -fPIC –shared test.c –o libtest.so`
  - `cc [-fPIE] main.c -L. –ltest`

…Nothing happened?

Problem: A Motivating Example

- 1 Executable
  - `cc main.c test.c`



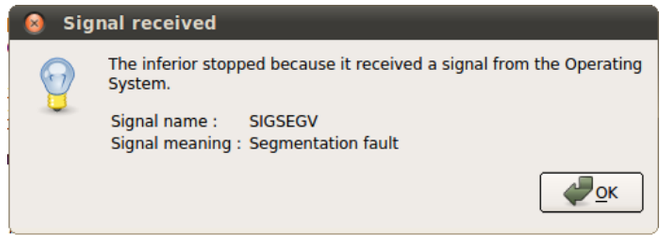
- 1 Executable + 1 Library
  - `cc -fPIC -shared test.c -o libtest.so`
  - `cc [-fPIE] main.c -L. -ltest`

…Nothing happened?

- 1 Executable + 1 Library
  - `cc -fPIC -shared test.c -o libtest.so`
  - `cc -fPIC main.c -L. -ltest`

Problem: A Motivating Example

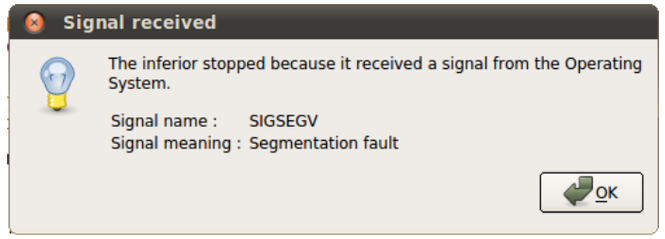
- 1 Executable
  - `cc main.c test.c`



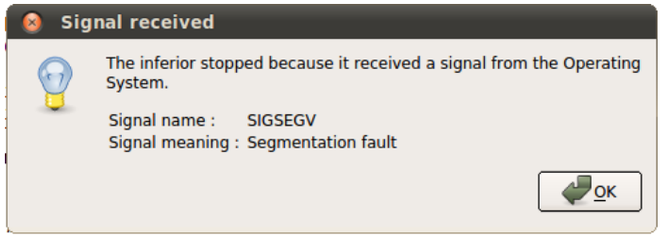
- 1 Executable + 1 Library
  - `cc -fPIC -shared test.c -o libtest.so`
  - `cc [-fPIE] main.c -L. -ltest`

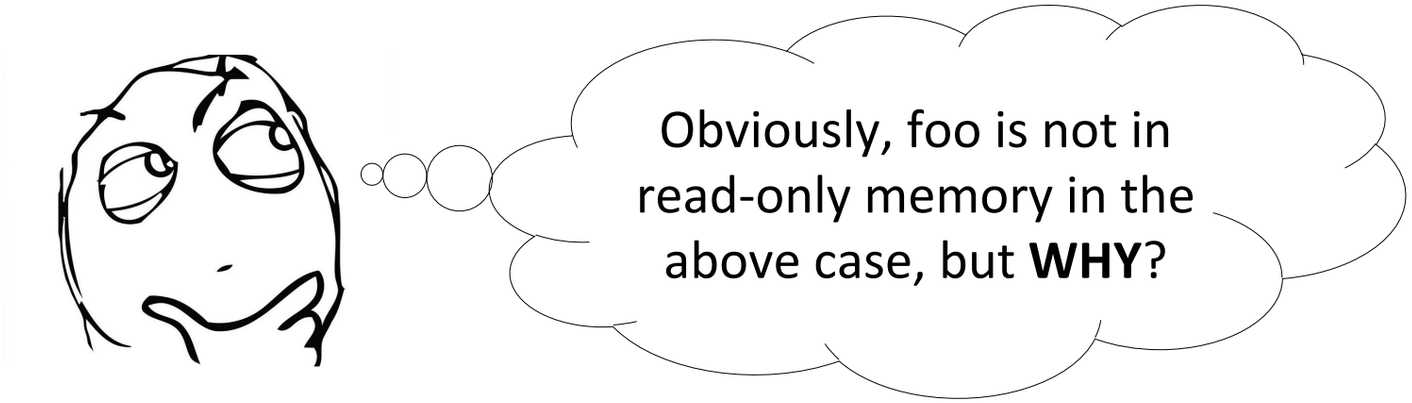
…Nothing happened?

- 1 Executable + 1 Library
  - `cc -fPIC -shared test.c -o libtest.so`
  - `cc -fPIC main.c -L. -ltest`

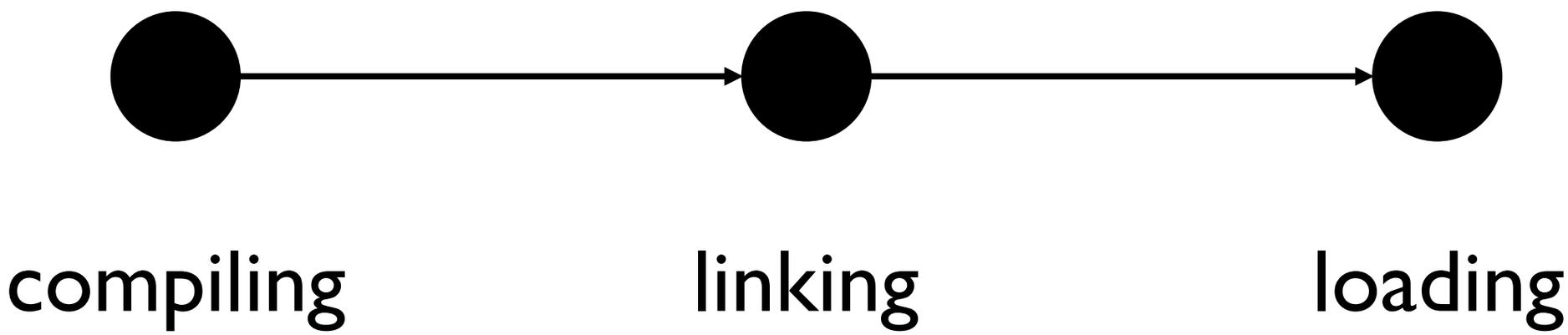


What happened so far...

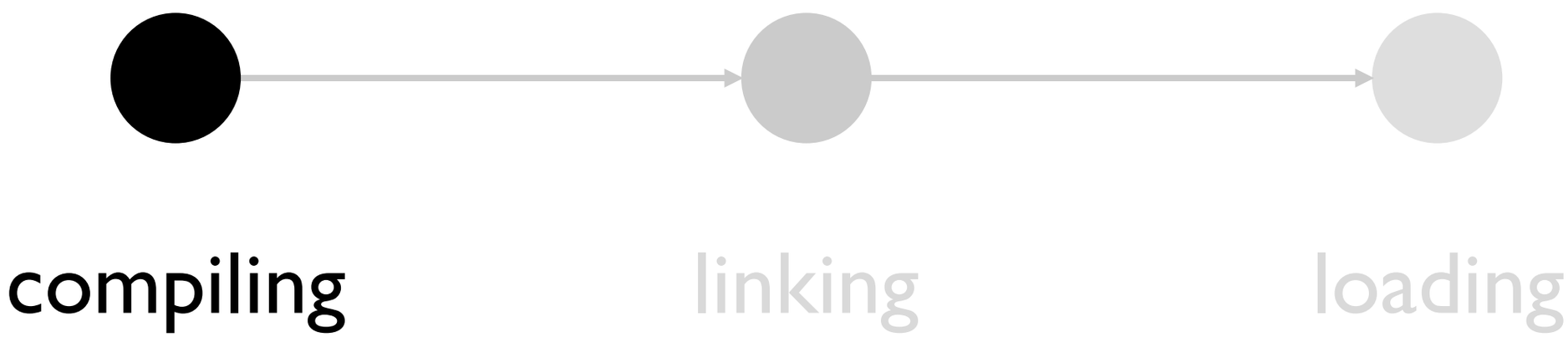
	non-PIC executable	PIC executable
local "foo"		
foreign "foo"	…Nothing happened?	



Building Process



Building Process



What does “extern” mean

```c
// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

What does “extern” mean

```c
// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

foo is defined in a different file but still in the same image (**w/o -fPIC flag**)

foo is defined in a different file and potentially in a different image (**w/ -fPIC flag**)

What does “extern” mean

```c
// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

foo is defined in a different file but still in the same image (**w/o -fPIC flag**)

foo is defined in a different file and potentially in a different image (**w/ -fPIC flag**)

foo is defined in the same image

```
// main.o – assuming same image

<main>:
  push %rbp
  mov  %rsp,%rbp
  mov  $0x64,offset_to_foo(%rip)
  mov  $0x0,%rax
  pop  %rbp
  ret
```

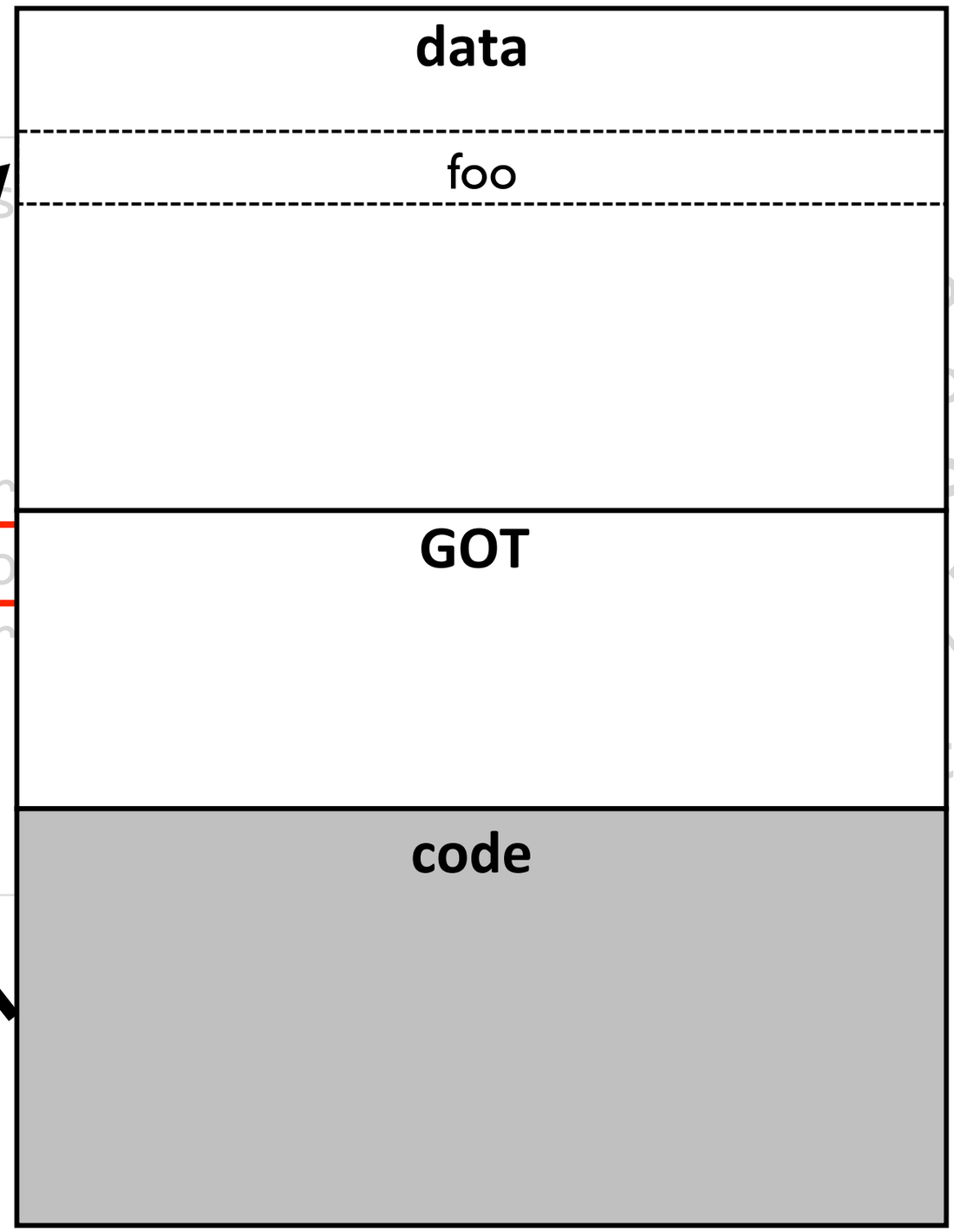
foo is defined in the same image

```
// main.o – assuming same image

<main>:
  push %rbp
  mov  %rsp,%rbp
  mov  $0x64,offset_to_foo(%rip)
  mov  $0x0,%rax
  pop  %rbp
  ret
```

The compiler assumes foo's location can be **statically** determined by the linker, and emits a single MOV instruction to write to foo.

foo is defined in the same image

```
// main.o - ms

<main>:
  push %rbp
  mov  %rsp,%r
  mov  $0x64,o
  mov  $0x0,%r
  pop  %rbp
  ret
```



ompiler assumes
ocation can be
tally determined by
ker, and emits a
MOV instruction to
o foo.

What does “extern” mean

```c
// main.c

extern const int foo;

int main()
{
    *(int *)&foo = 100;
    return 0;
}
```

foo is defined in a different file but still in the same image (**w/o -fPIC flag**)

foo is defined in a different file and potentially in a different image (**w/ -fPIC flag**)

foo is defined in a different image

```
// main.o – assuming same image

<main>:
  push %rbp
  mov  %rsp,%rbp
  mov  offset_to_foo_got(%rip),%rax
  mov  $0x64,(%rax)
  mov  $0x0,%rax
  pop  %rbp
  ret
```

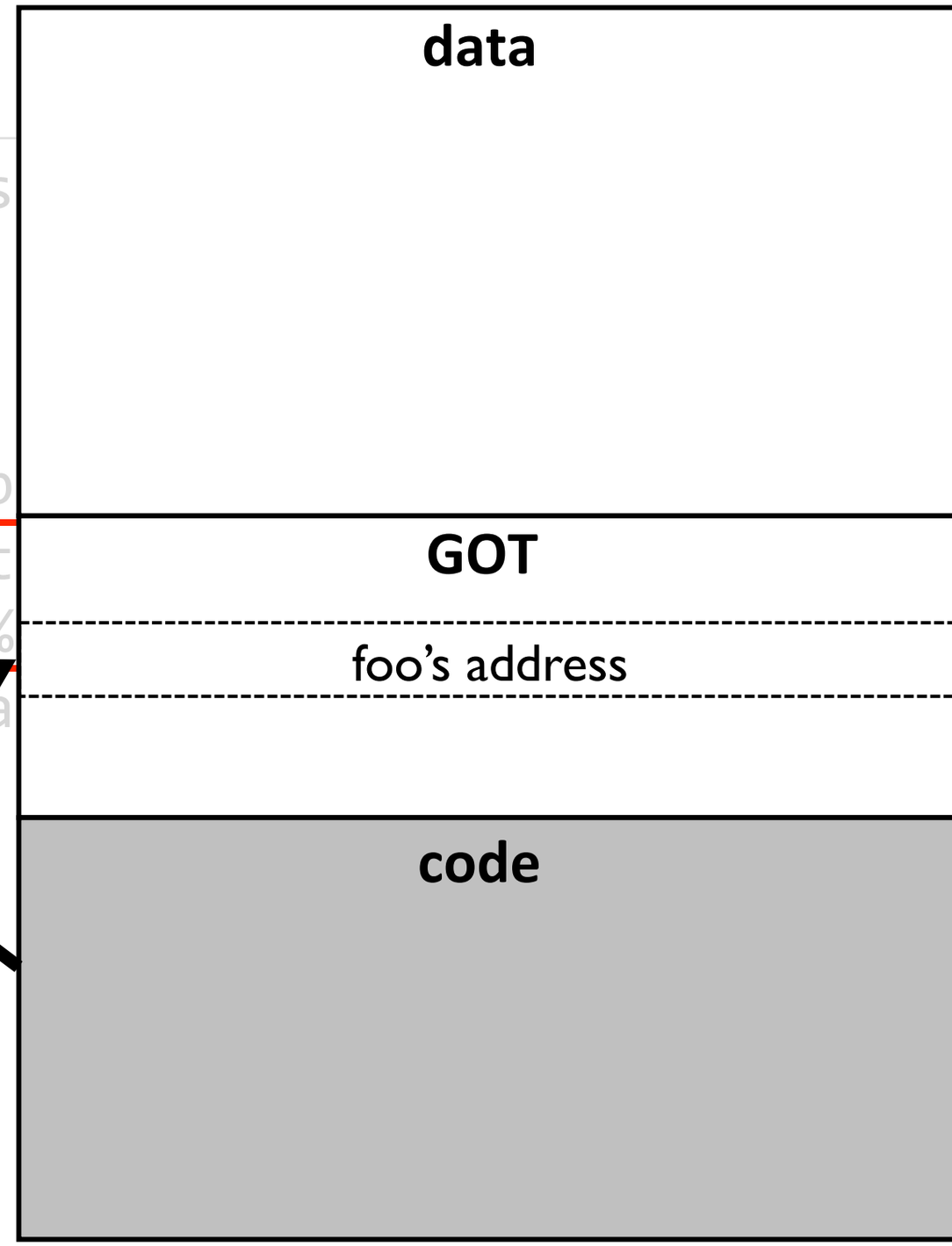
foo is defined in a different image

```
// main.o – assuming same image

<main>:
  push %rbp
  mov  %rsp,%rbp
  mov  offset_to_foo_got(%rip),%rax
  mov  $0x64,(%rax)
  mov  $0x0,%rax
  pop  %rbp
  ret
```

The compiler assumes foo's location cannot be statically determined and emits two MOV instructions: one to retrieve foo's address from its GOT slot, and the other to actually write to foo.

foo is defined in a different image

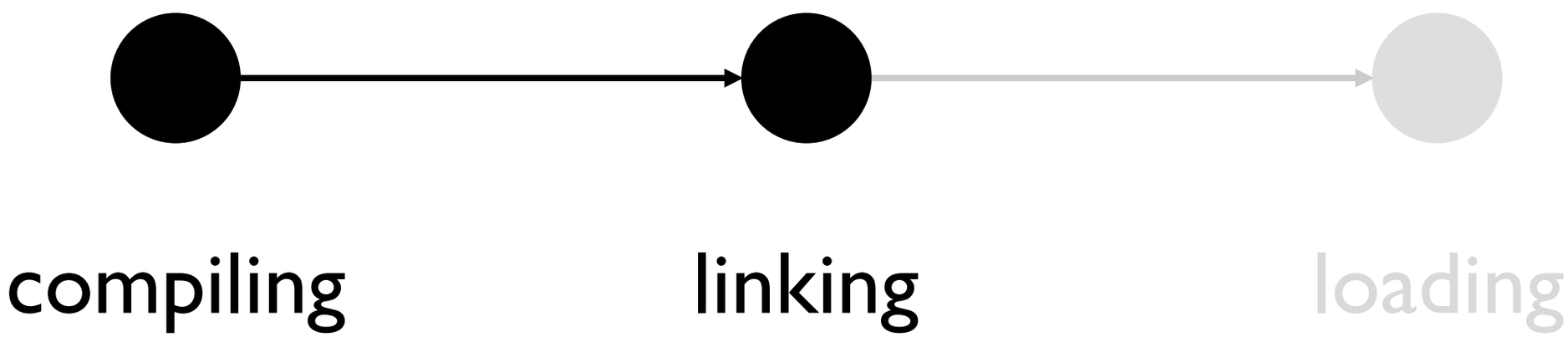
```
// main.o – ass

<main>:
  push %rbp
  mov  %rsp,%rb
  mov  offset_t
  mov  $0x64,(%
  mov  $0x0,%ra
  pop  %rbp
  ret
```



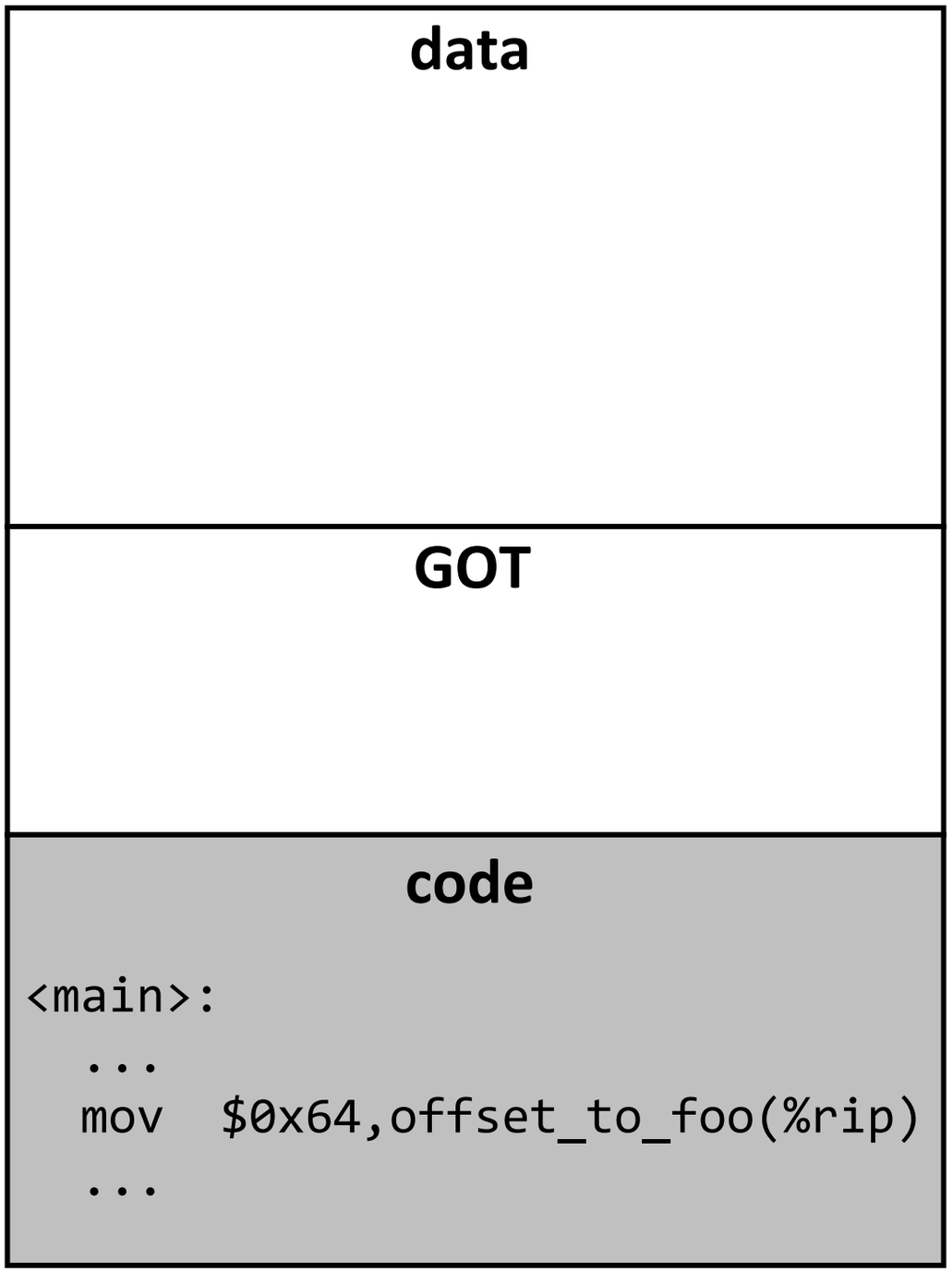
ompiler assumes
location cannot be
ally determined
mits two MOV
uctions: one to
ve foo's address
its GOT slot, and
ther to write to

Without –fPIC flag, GCC and Clang on Linux assumes foo is defined in the same image.

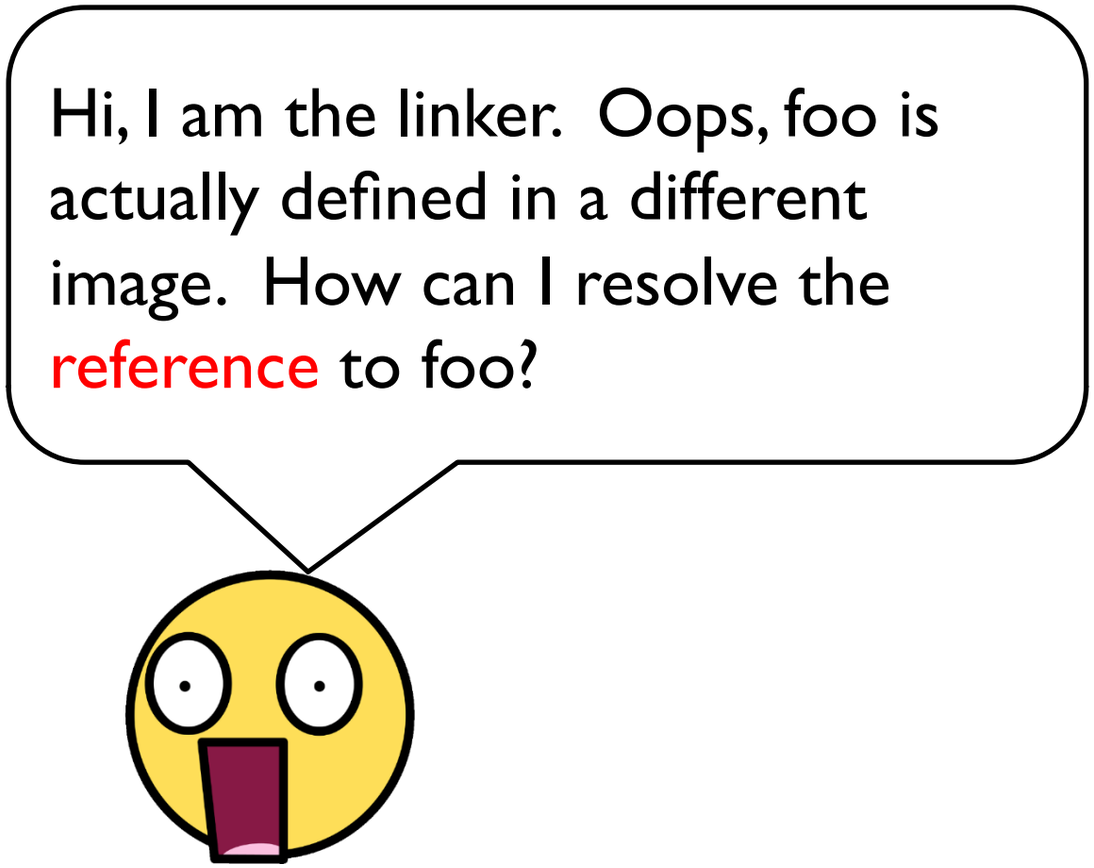
Building Process



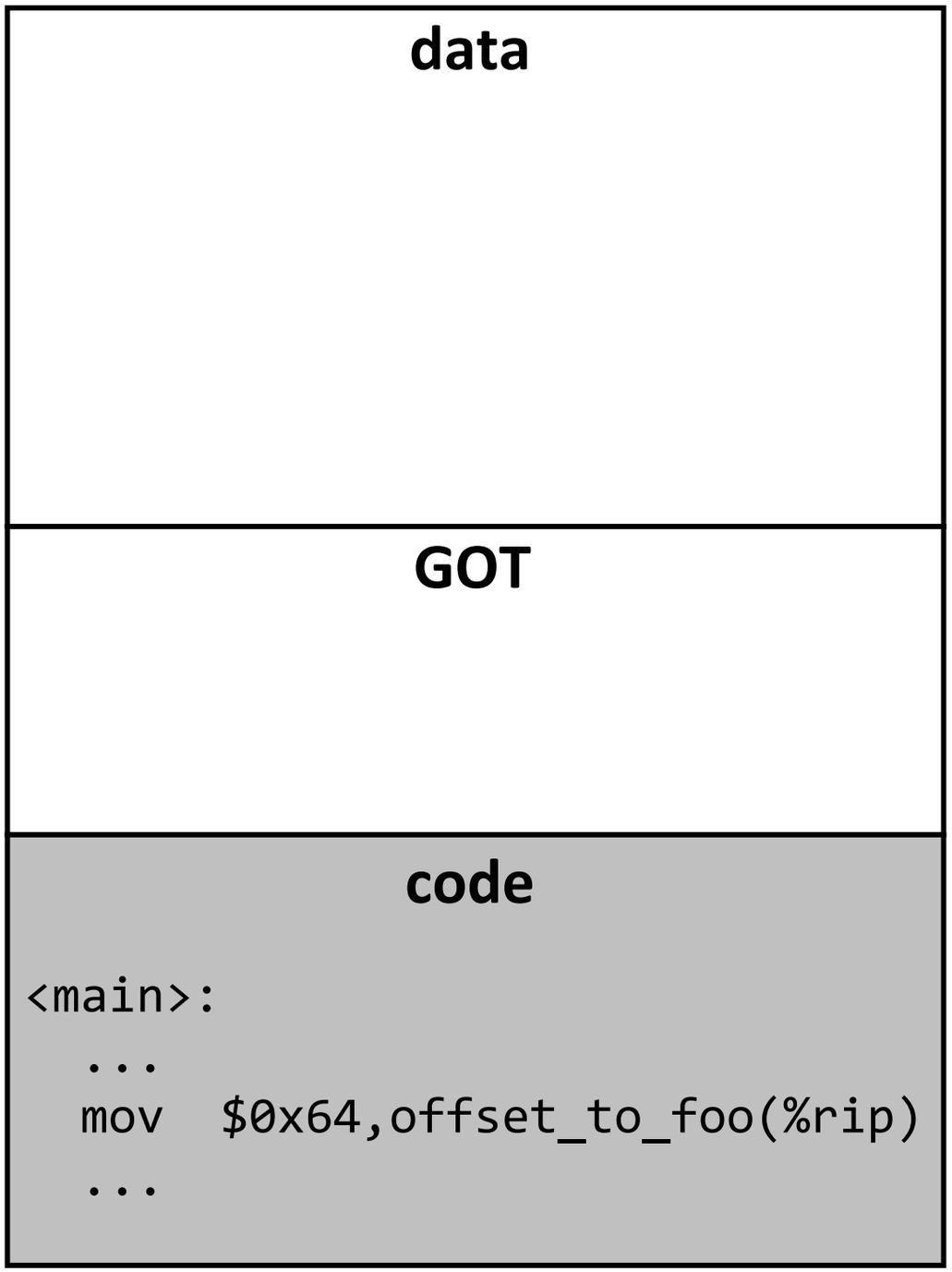
Copy Relocation



executable



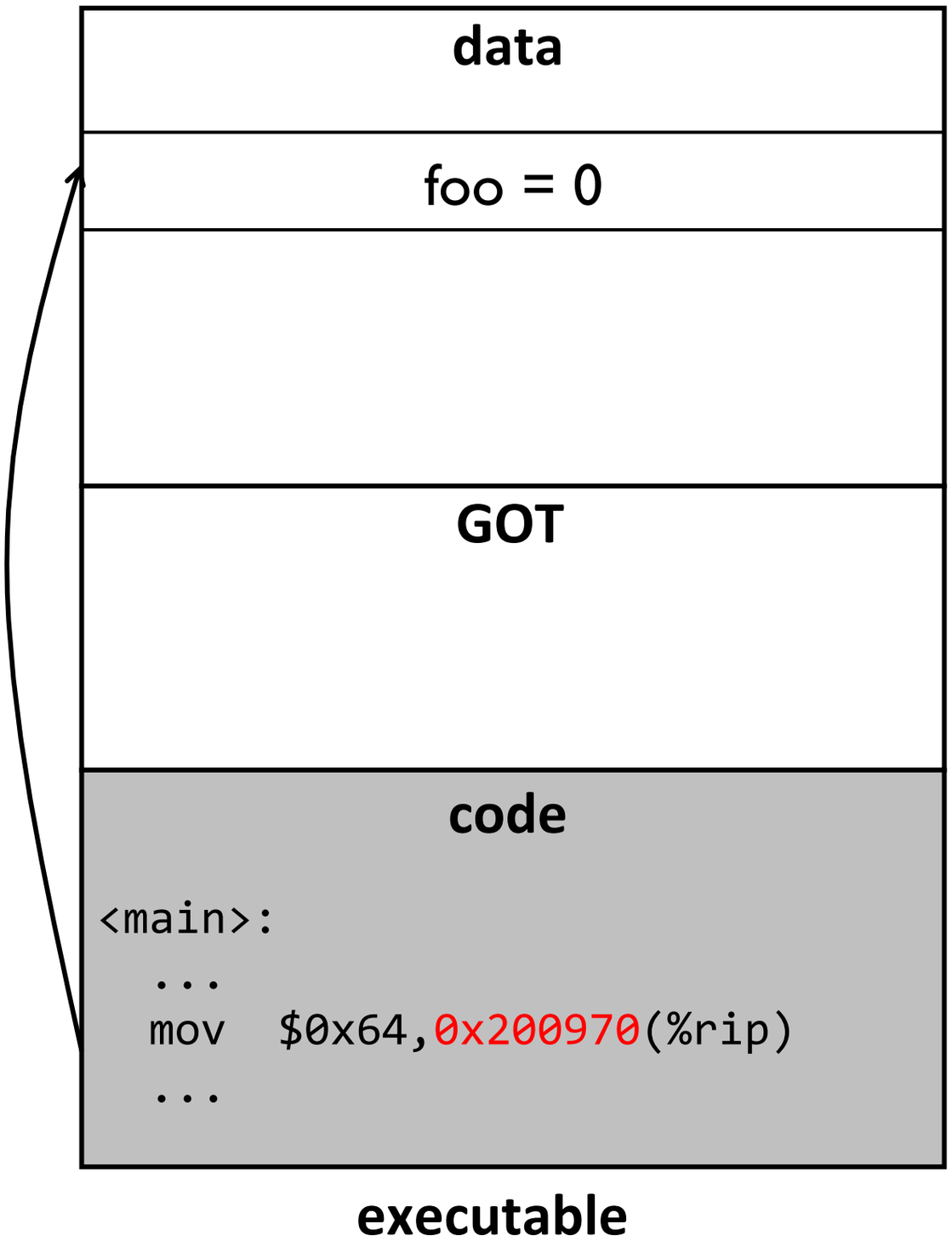
Copy Relocation

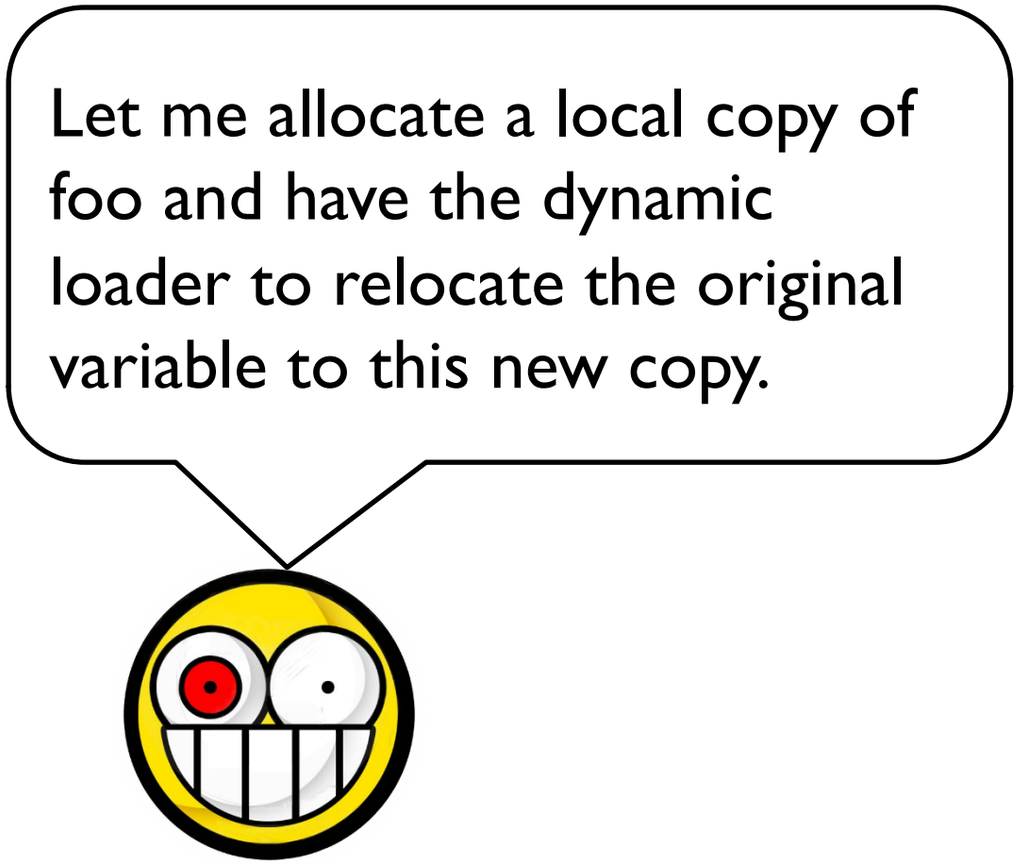


executable

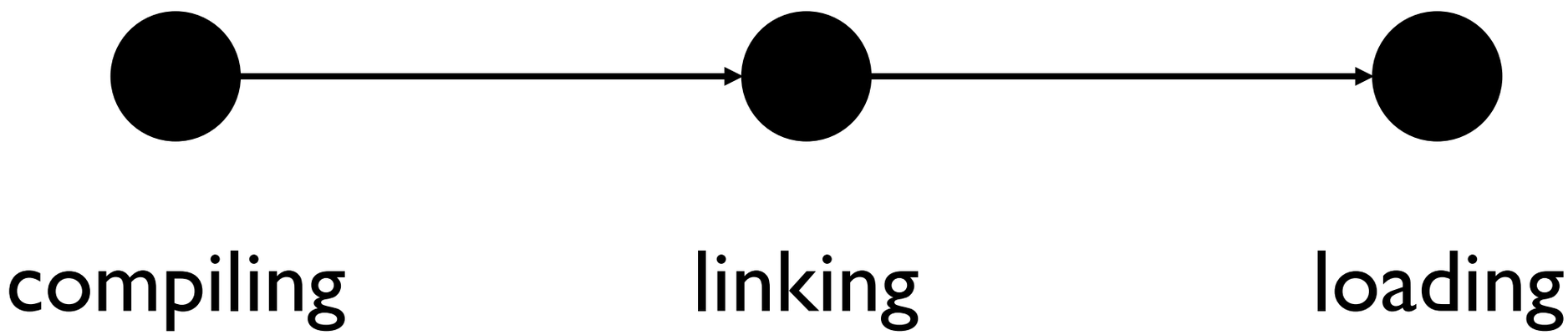
Let me allocate a local copy of foo and have the dynamic loader to relocate the original variable to this new copy.

Copy Relocation

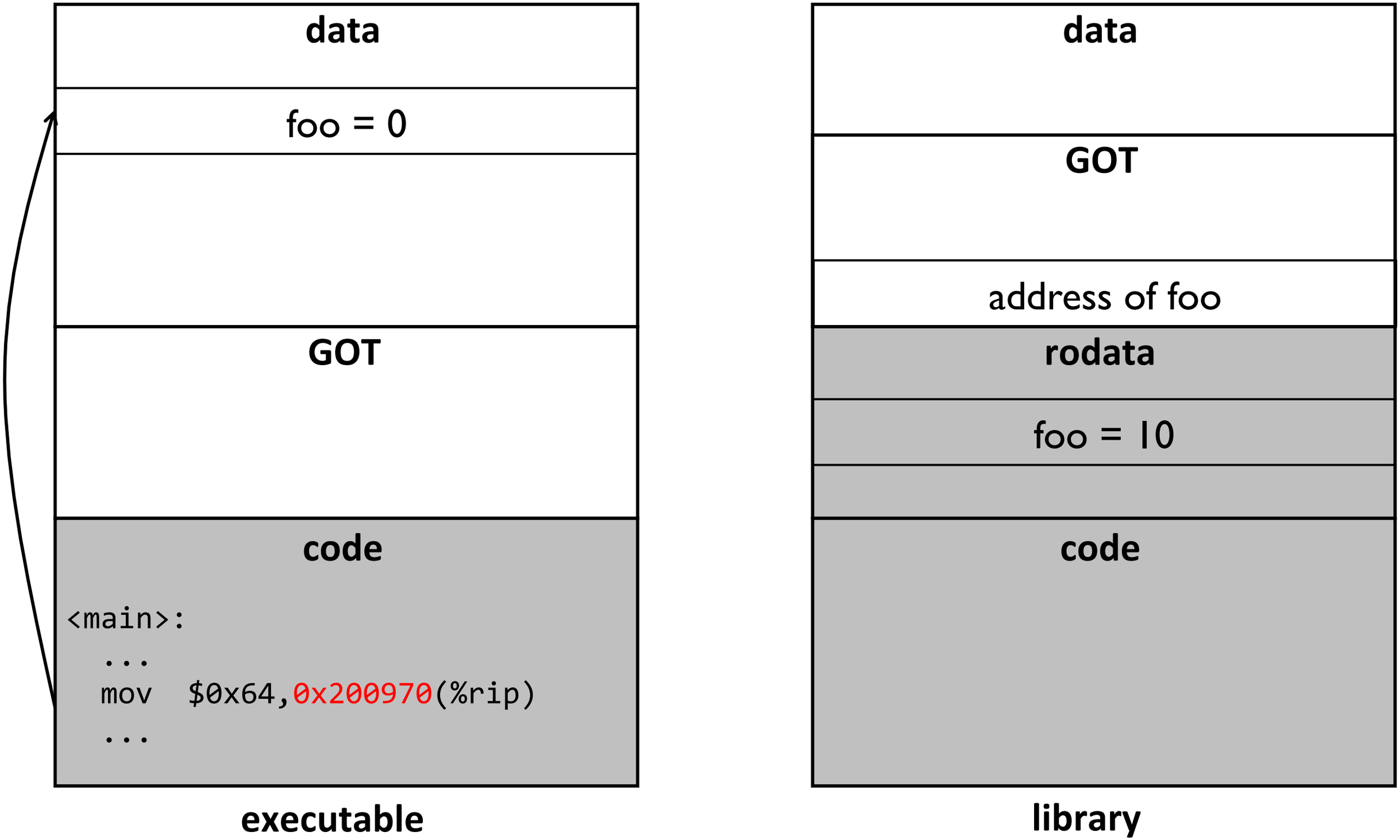




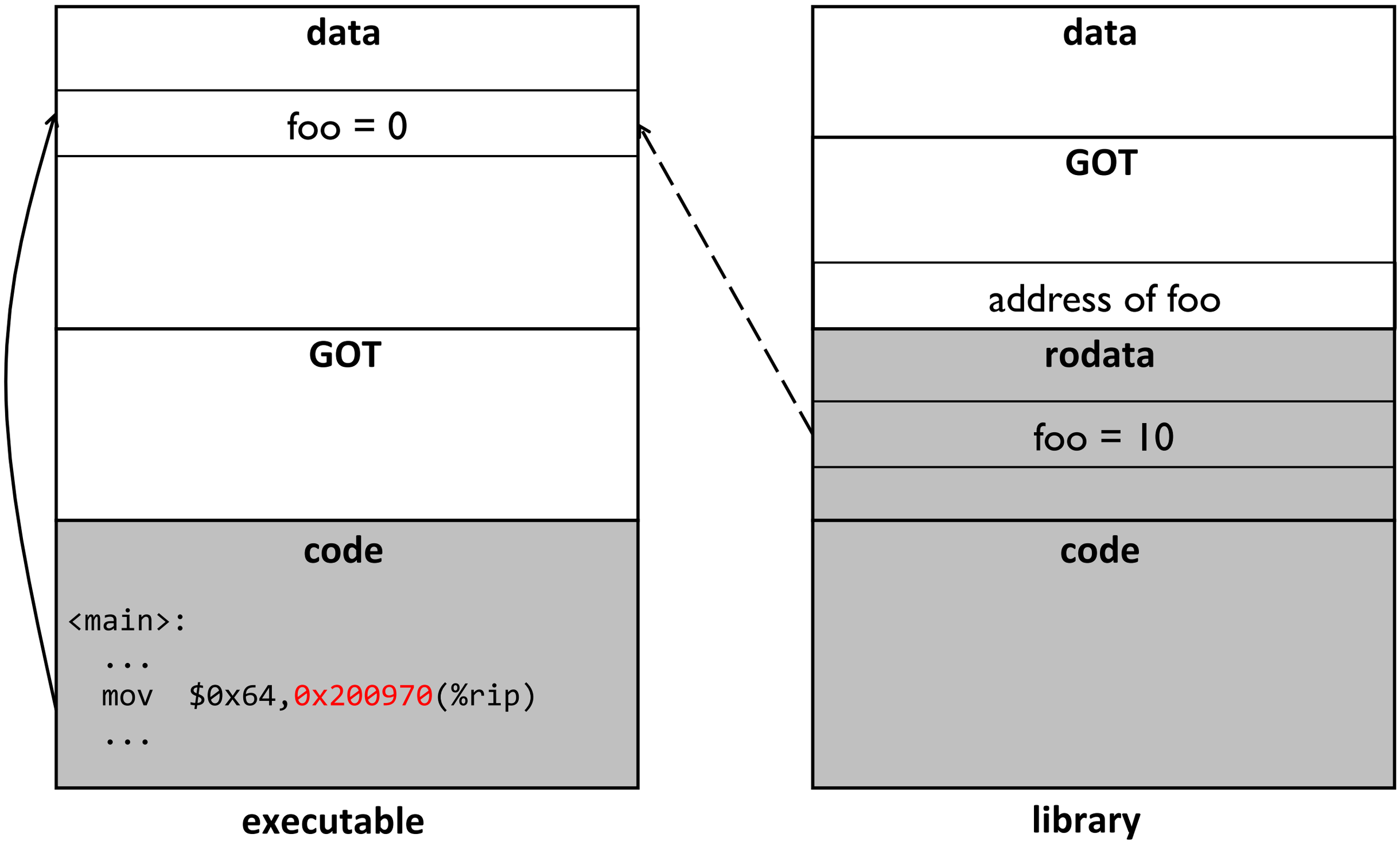
Building Process



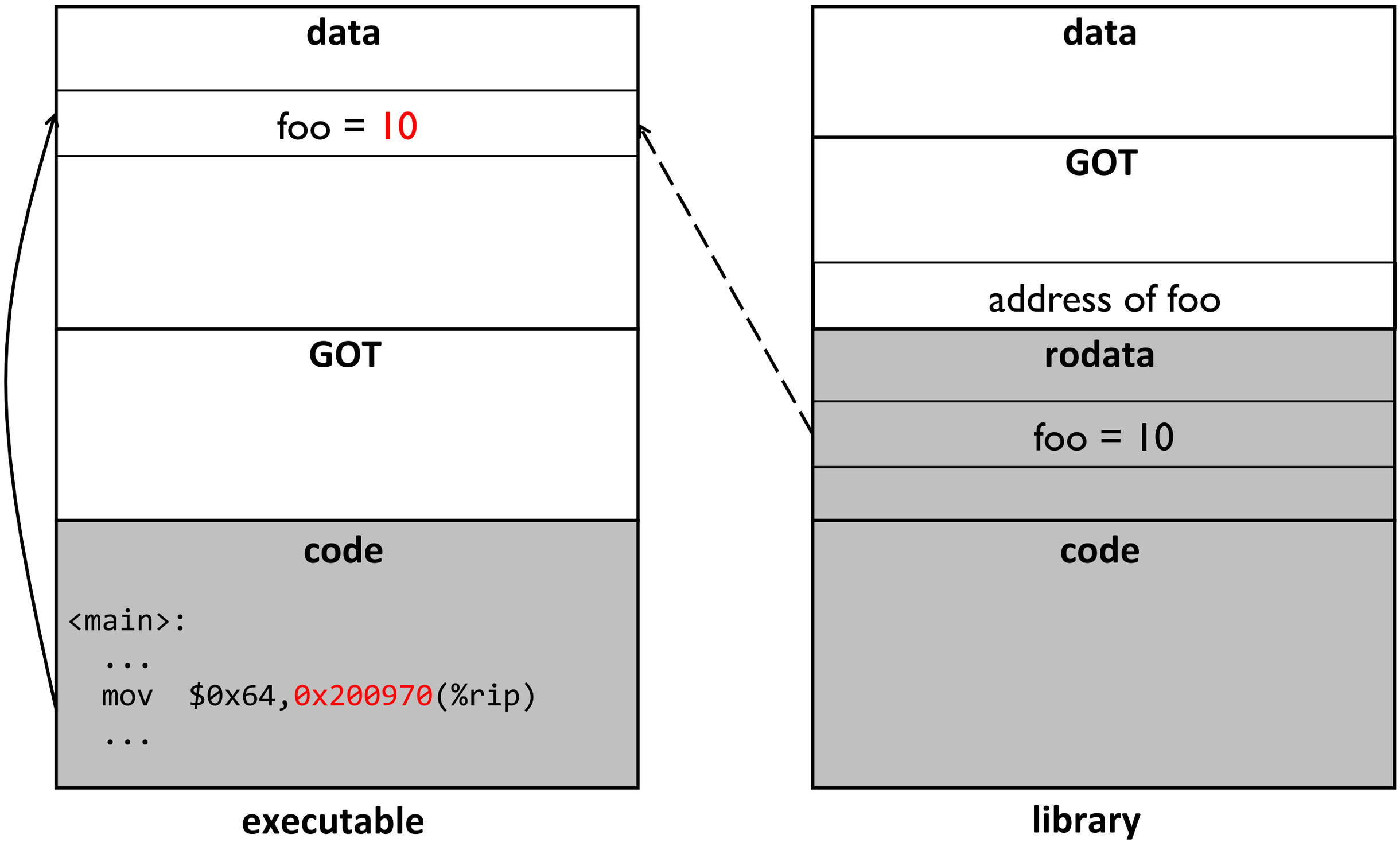
Copy Relocation



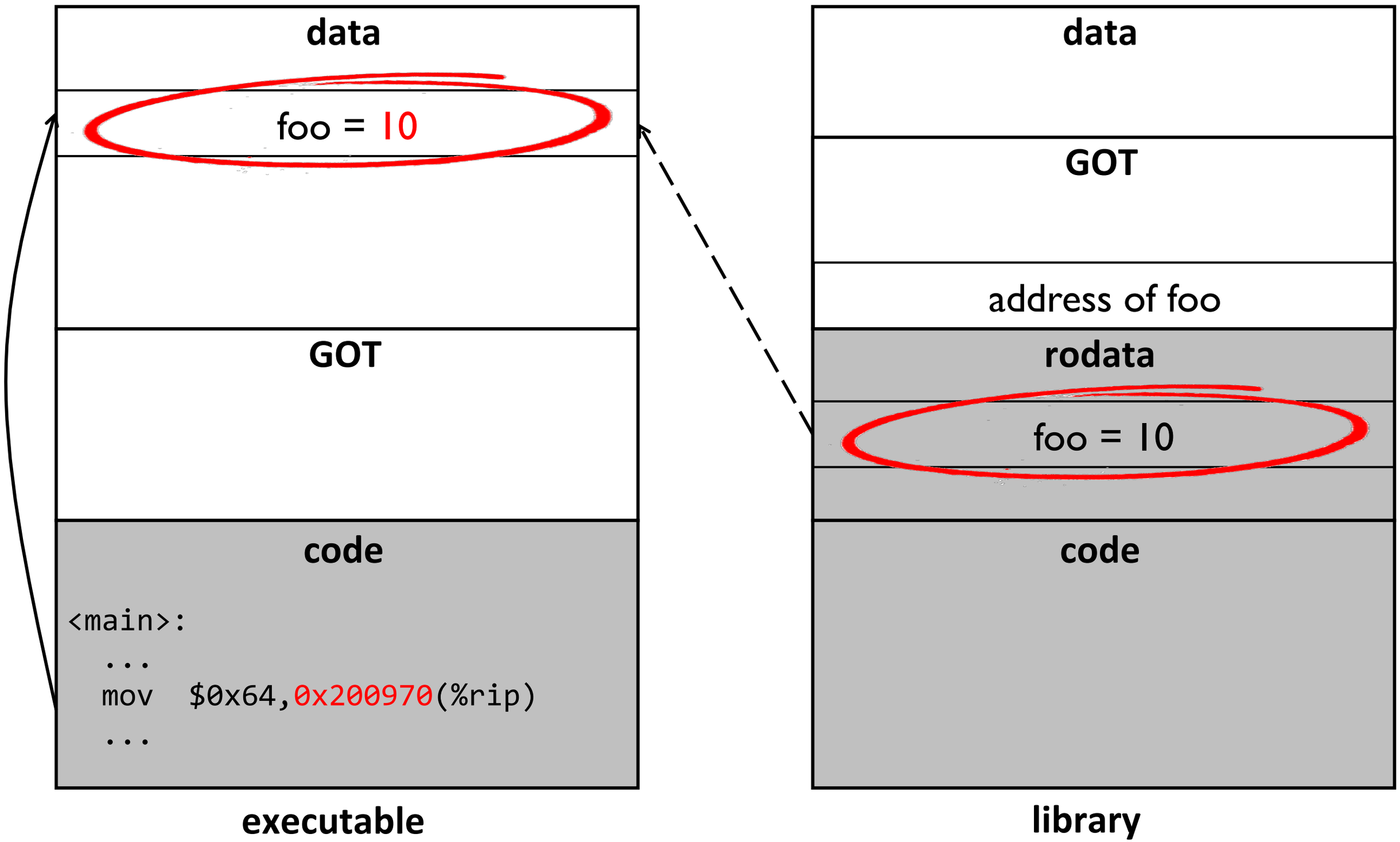
Copy Relocation



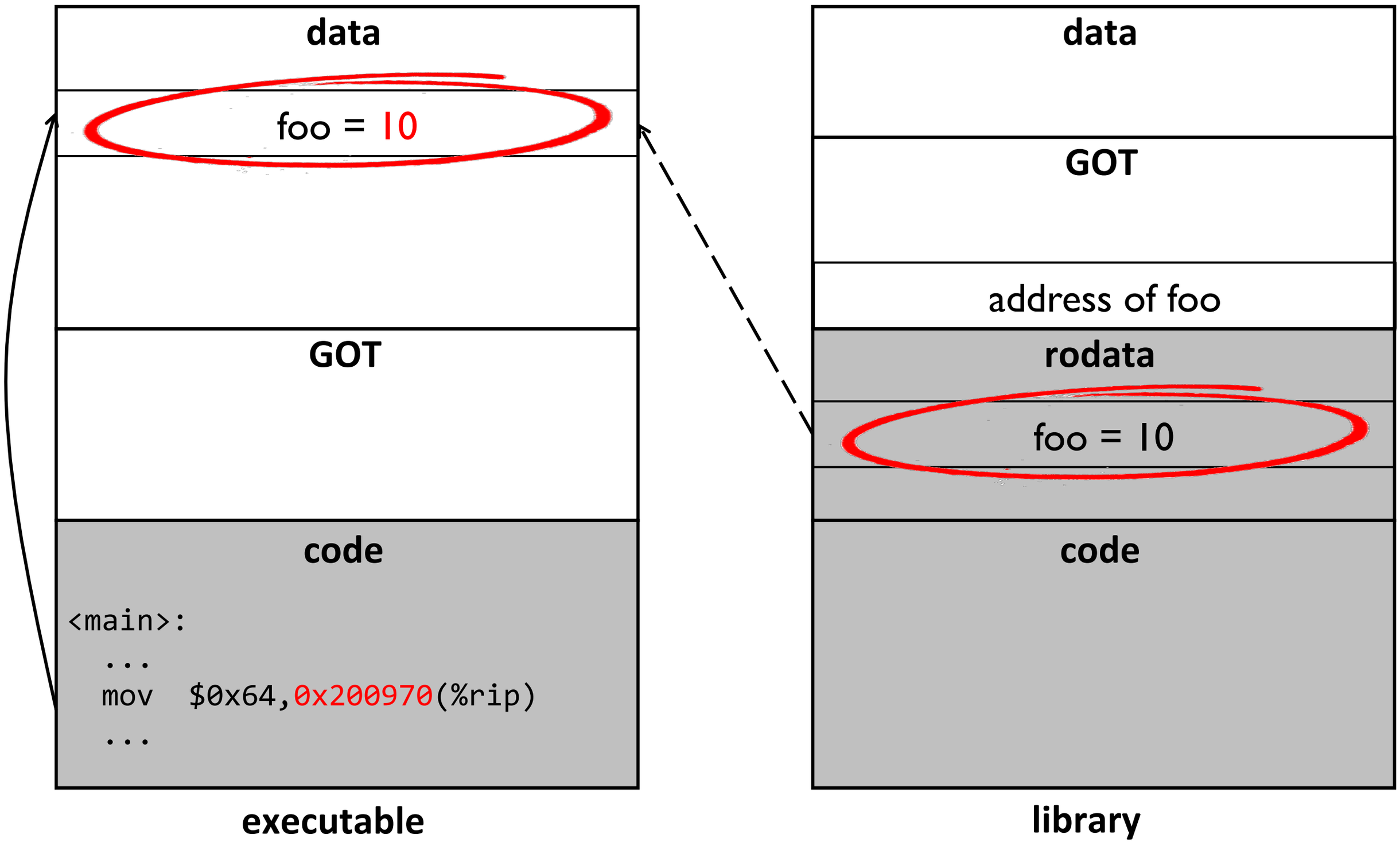
Copy Relocation



Copy Relocation



Copy Relocation Violation



Security Concerns

PENNSTATE 1855

- Expose "read-only" data to memory corruption attacks
  - Making C++ vtables mutable can break existing defenses
    - VTV, Interleaving, SafeDispatch
  - Making format string writable can enable printf-oriented programming
    - Printf-oriented programming requires mutable format string to implement branching
  - File names
  - IP addresses
  - ...

Security Concerns

PENNSTATE 1855

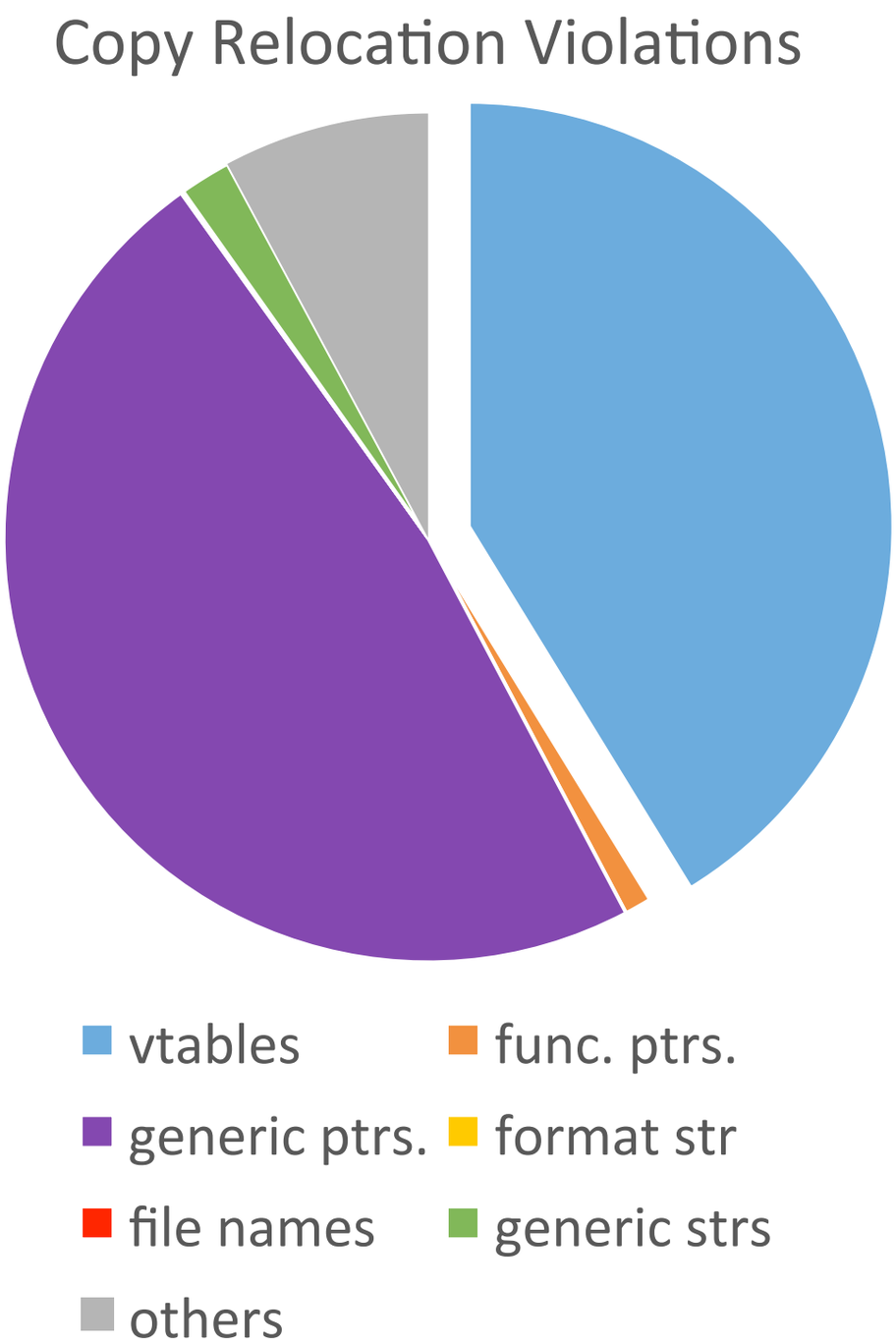
- Copy Relocation Violation does not directly lead to exploitation
- Defenses depending on read-only data being immutable can be bypassed
  - vtables
  - format strings
  - file names
  - IP addresses
  - ...

Evaluations

PENNSTATE 1855

- Do Copy Relocation Violations commonly exist?
  - Analyze 54,045 packages in Ubuntu 16.04 LTS
    - 34,291 executables + 58,862 dynamic libraries
- Do Copy Relocation Violations weaken security mitigations?
  - Evaluate a set of CFI defenses in face of copy relocation violations
- Implications on other platforms?
  - Windows and macOS

Real-world Copy Relocation Violations



- 69,098 copy relocation violations in 6,449 (out of 34,291) executables
- 28,497 vtables copied to writable memory in 4,291 executables
- Among the top 10 most common copy relocation violations, 8 of them are vtables from libstdc++.so

Security Evaluation

- Developed a small C++ program that has an intentional vtable corruption vulnerability
- Evaluate the program under 7 CFI defenses

Defenses	Check Func Ptr	Check VTable	Bypassable
VTrust	✓	✓	✗
VTV	✗	✓	✓
vfGuard	✓	✗	✗
Interleaving	✗	✓	✓
SafeDispatch	✓	✗	✗
SafeDispatch2	✗	✓	✓
RockJIT	✓	✗	✗

Other Platforms

PENNSTATE 1855

- Windows
  - MSVC requires explicit annotation to differentiate "intra-module extern" from "inter-module extern"
  - The example program cannot be built on Windows
- macOS
  - The compiler conservatively assumes "extern" is from a different image
  - The linker uses GOT to serve those references
  - Copy relocations do not exist on macOS

macOS issue

- macOS has its own issue that results in the same consequence
  - macOS's compiler allocates data that **potentially** requires runtime patching in __DATA__.__const section
  - However, the loader does not reprotect it as read-only after runtime patching
  - Read-only data (e.g., vtable) remains writable

Copy relocation violations seem prevalent in current Linux systems. Then, how can we get rid of them?

Mitigations

PENNSTATE 1855

- Eliminate copy relocations entirely
  - Recompile executable using -fPIC flag, -fPIE not enough
  - -fPIC flag forces the compiler to treat non-static global variables as defined in a different image
- Respect the memory protection while performing copy relocations
  - Determine the memory protection permission at link time
  - Allocate the variable copy from a section protected by RELRO
  - Both GNU Binutils and LLVM are adopting this approach

Mitigations

PENNSTATE 1855

- es

D28272 ELF: Reserve

reviews.llvm.org/D28272

Phabricator

Search

Differential › D28272

ELF: Reserve space for copy relocations of read-only symbols in relro.

Closed Public

Authored by **pcc** on Jan 3 2017, 6:14 PM.

Details

Reviewers rafael, davide, ruiu, tstellarAMD

Commits rL291524: ELF: Reserve space for copy relocations of read-only symbols in relro.

SUMMARY

When reserving copy relocation space for a shared symbol, scan the DSO's program headers to see if the symbol is in a read-only segment. If so, reserve space for that symbol in a new synthetic section named .bss.rel.ro which will be covered by the relro program header.

This fixes the security issue disclosed on the binutils mailing list at: https://sourceware.org/ml/libc-alpha/2016-12/msg00914.html

Diff Detail

Repository rL LLVM

pcc retitled this revision from to *ELF: Reserve space for copy relocations of read-only symbols in relro.*. Jan 3 2017, 6:14 PM

pcc updated this object.

pcc added reviewers: **rafael**, **ruiu**, **davide**.

pcc added subscribers: **llvm-commits**, **tmsriram**, **eugenis**.

Edit Revision
Update Diff
Download Raw Diff
Edit Related Revisions...
Edit Related Objects...
Subscribe
Award Token
Flag For Later

Tags

None

Subscribers

nhaehnle, emaste, eugenis and 2 others

Conclusions

PENNSTATE 1855

- Identified a design flaw in the compiler toolchain on Linux
  - Copy relocation can strip the "const" attribute specified by the programmer
- Proposed mitigations
  - Eliminate copy relocations entirely
  - Preserve the memory protection of the relocated variables
- Evaluated copy relocation violations in real world
  - Studied 54,045 packages in Ubuntu 16.04 LTS
  - Copy relocation violations occur commonly in many programs
  - Copy relocation violations can subvert existing defenses

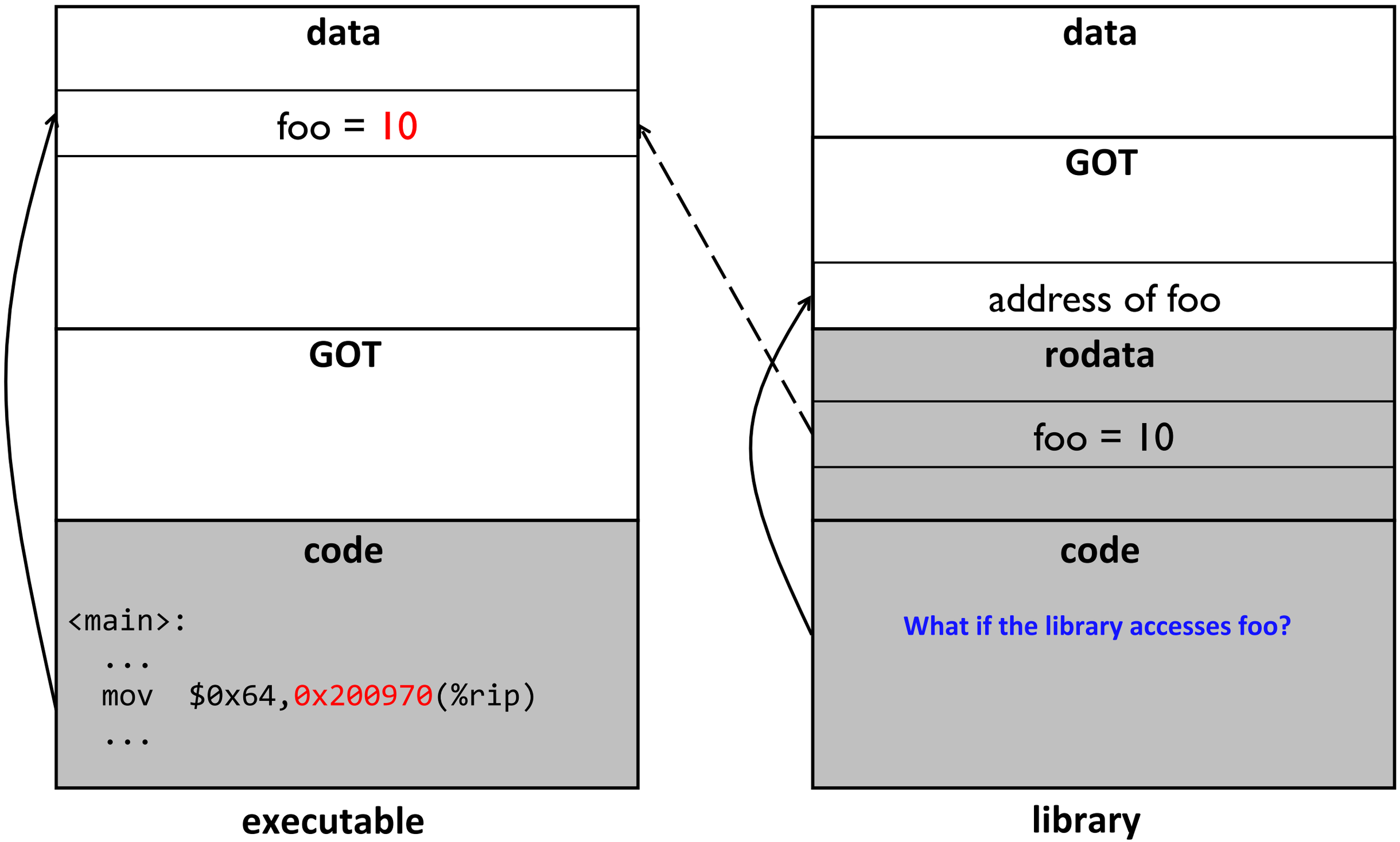
Questions

Variable Type Inference

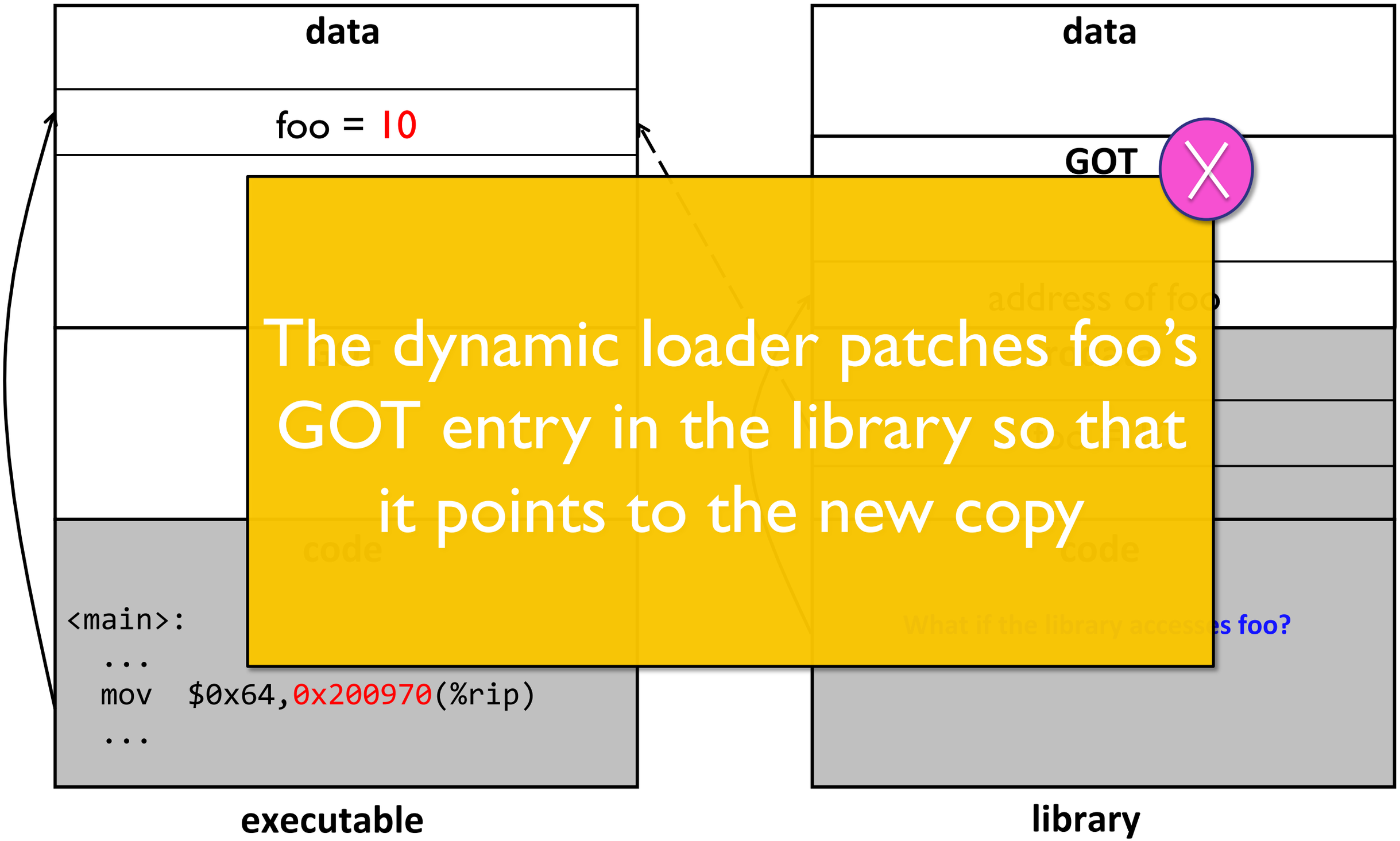
- Requirements
  - No source code
  - No debug information

- Heuristics
  - Pointers:
    - Use relocation information to identify pointers in general
    - Use pointer value to determine code pointer vs data pointer
  - Strings:
    - All bytes are ASCII characters
    - Use '/' to determine file paths and '%' to determine format strings

Copy Relocation

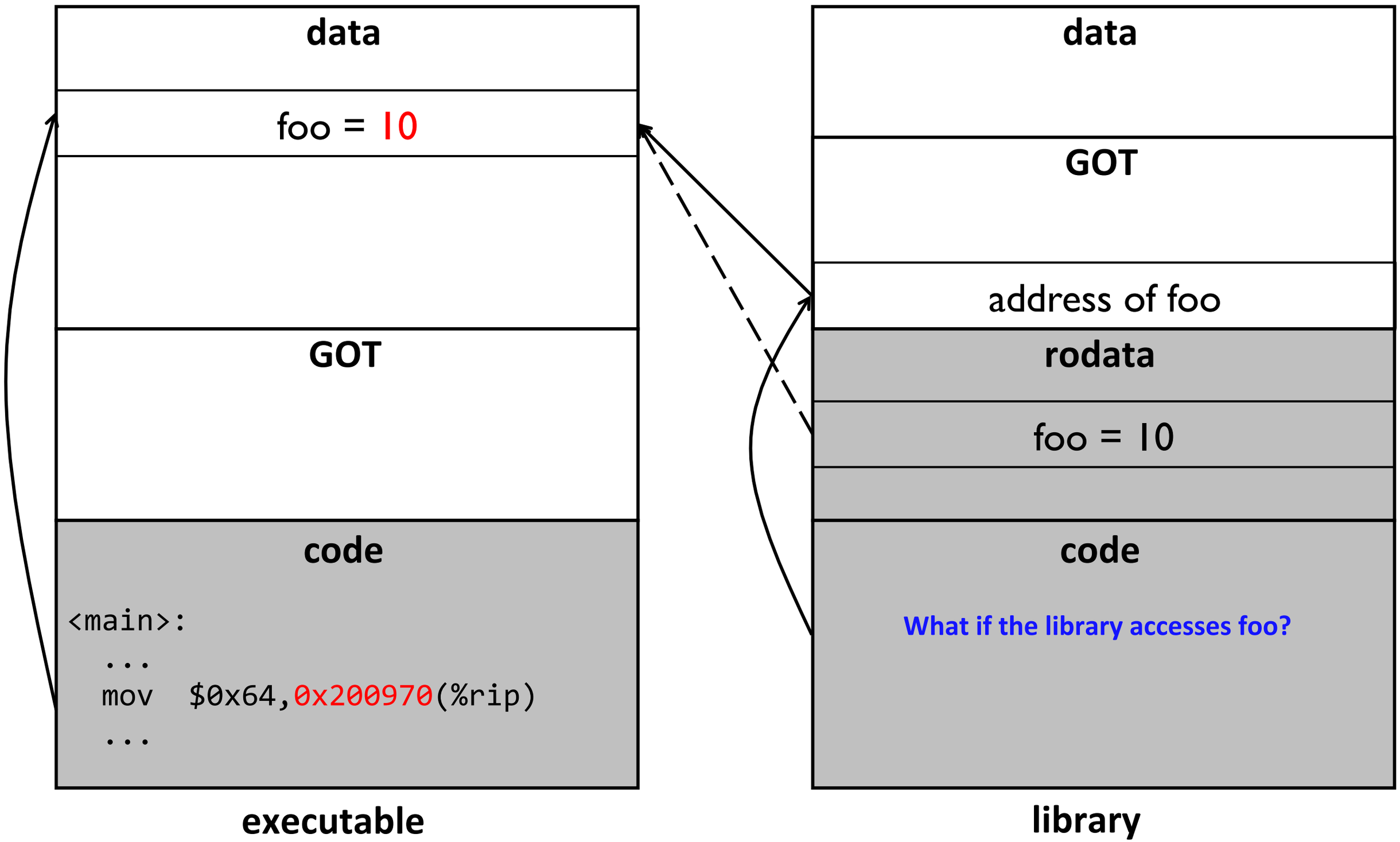


Copy Relocation

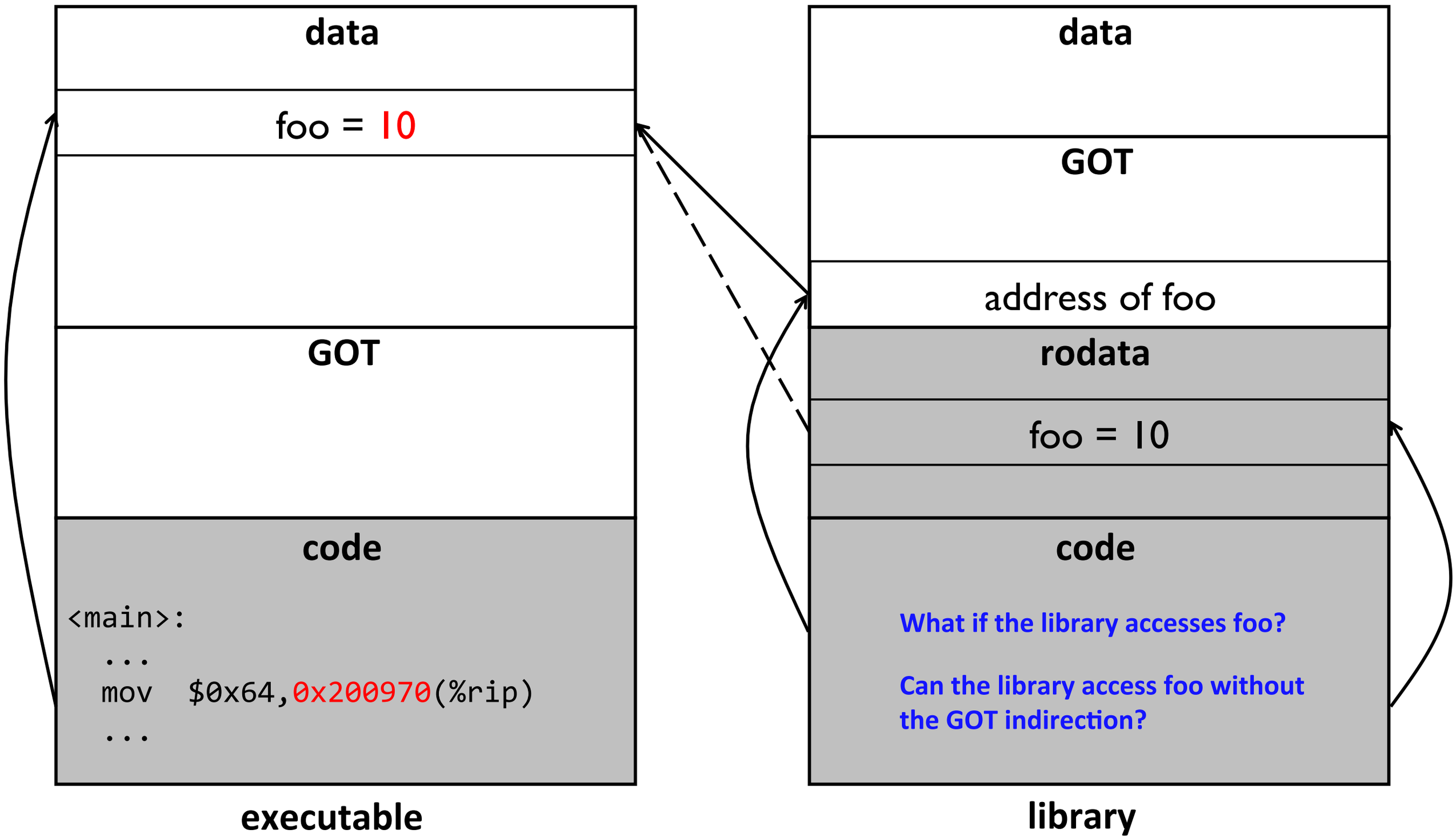


The dynamic loader patches foo's GOT entry in the library so that it points to the new copy

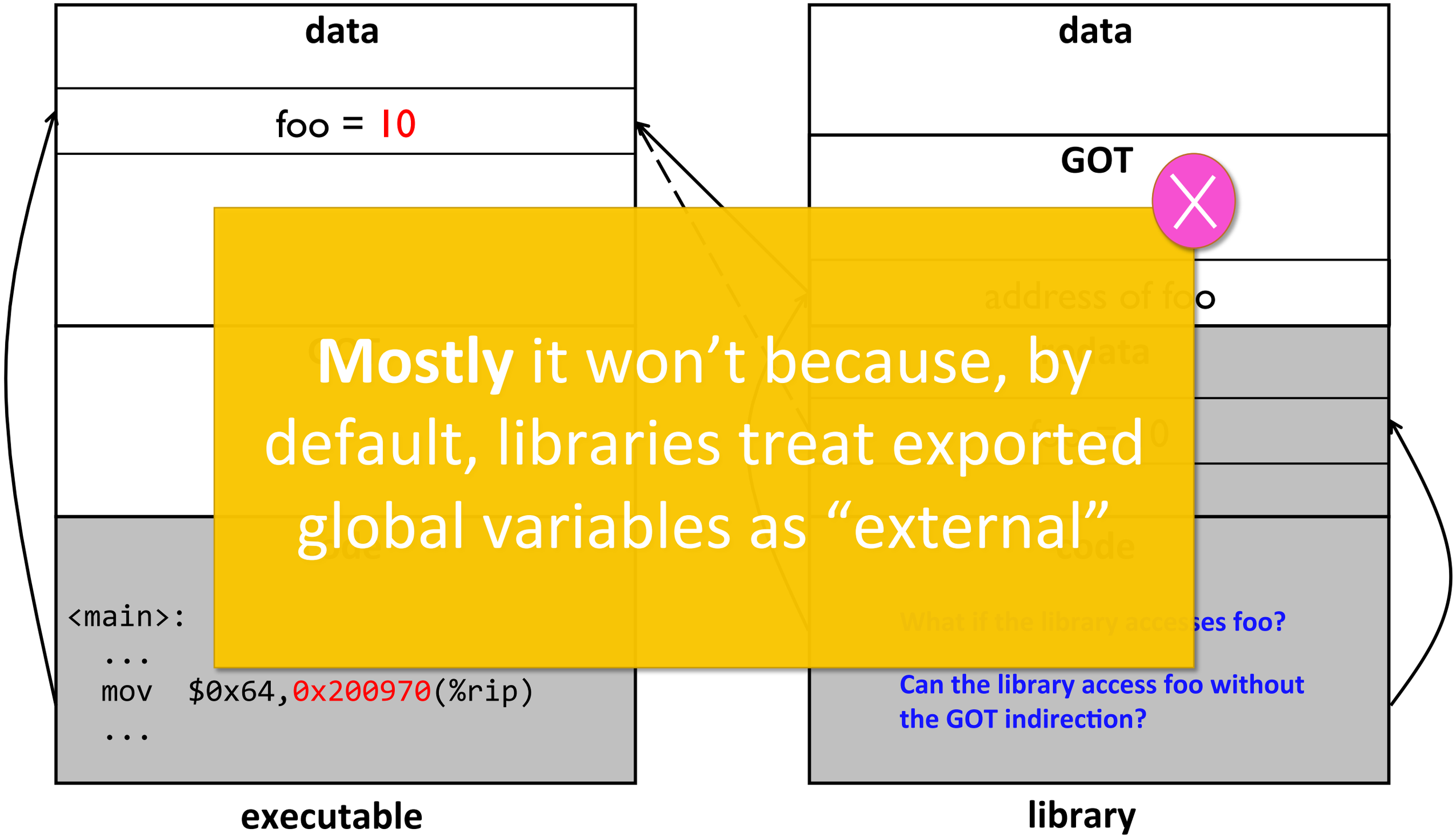
Copy Relocation



Copy Relocation

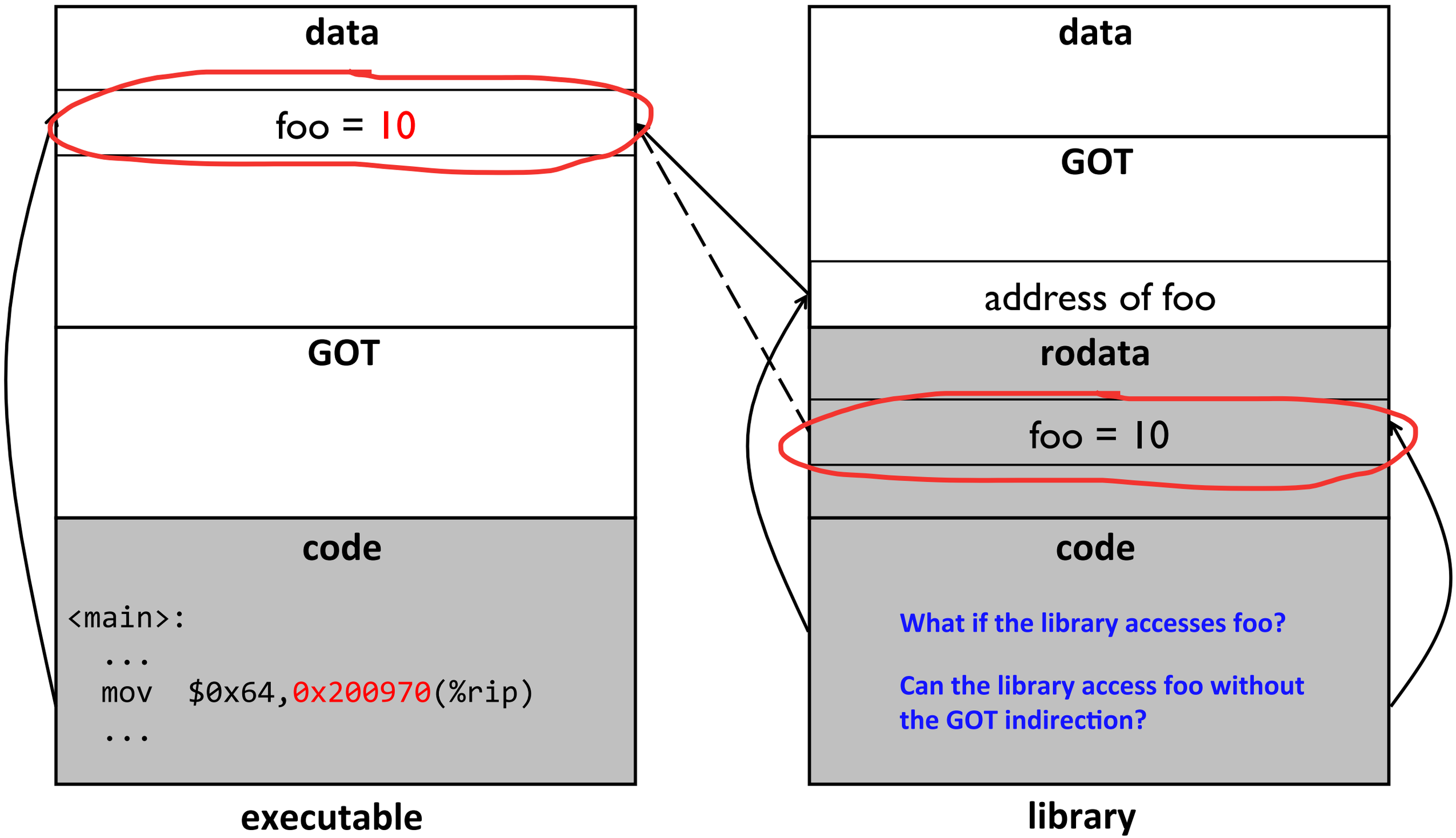


Copy Relocation



Mostly it won't because, by default, libraries treat exported global variables as "external"

Copy Relocation



Copy Relocation Violation