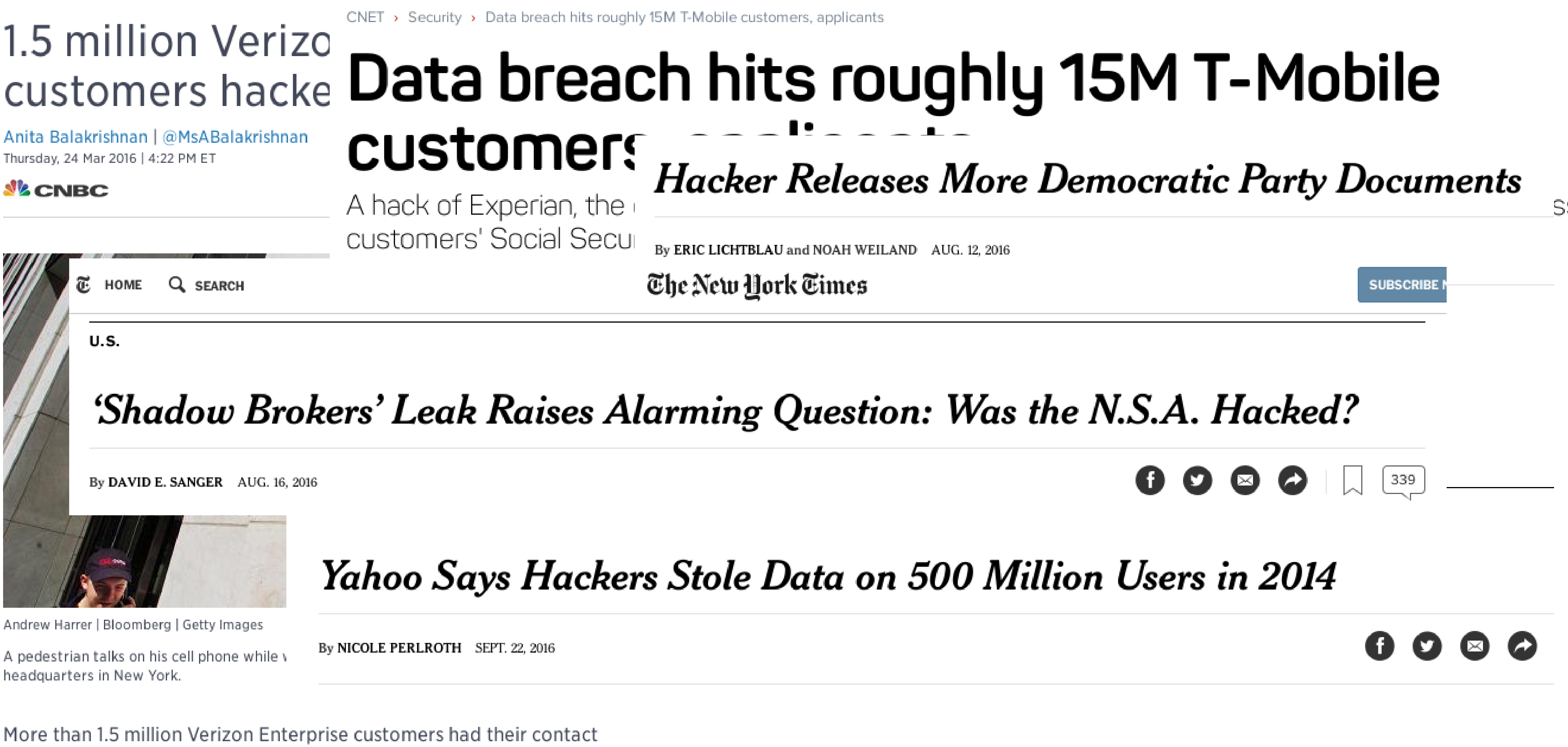
Shuffler: Fast and Deployable Continuous Code Re-Randomization

David Williams-King,
Graham Gobieski, Kent Williams-King, James P. Blake,
Xinhao Yuan, Patrick Colp, Michelle Zheng,
Vasileios P. Kemerlis, Junfeng Yang, William Aiello

OSDI 2016

Software Remains Vulnerable

- High-profile server breaches are commonplace



1.5 million Verizo
customers hacke
Anita Balakrishnan | @MsABalakrishnan
Thursday, 24 Mar 2016 | 4:22 PM ET
CNBC

Andrew Harrer | Bloomberg | Getty Images
A pedestrian talks on his cell phone while
headquarters in New York.

More than 1.5 million Verizon Enterprise customers had their contact

CNET › Security › Data breach hits roughly 15M T-Mobile customers, applicants

Data breach hits roughly 15M T-Mobile
customers

A hack of Experian, the
customers' Social Secu

Hacker Releases More Democratic Party Documents

By ERIC LICHTBLAU and NOAH WEILAND AUG. 12, 2016

HOME SEARCH The New York Times SUBSCRIBE

U.S.

'Shadow Brokers' Leak Raises Alarming Question: Was the N.S.A. Hacked?

By DAVID E. SANGER AUG. 16, 2016

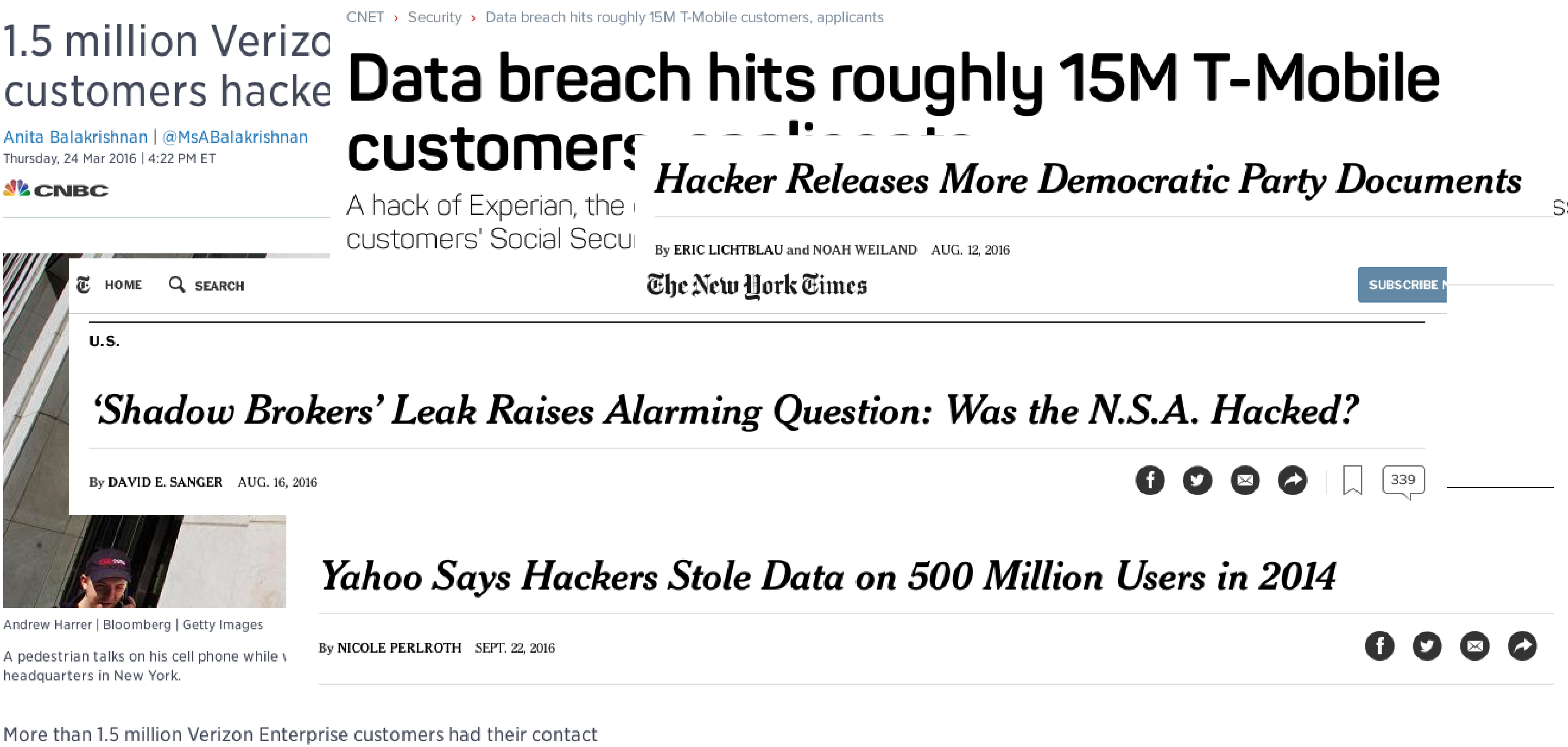
339

Yahoo Says Hackers Stole Data on 500 Million Users in 2014

By NICOLE PERLROTH SEPT. 22, 2016

Software Remains Vulnerable

- High-profile server breaches are commonplace
- 90% of today’s attacks utilize ROP [1]



CNET › Security › Data breach hits roughly 15M T-Mobile customers, applicants

1.5 million Verizo
customers hacke

Anita Balakrishnan | @MsABalakrishnan
Thursday, 24 Mar 2016 | 4:22 PM ET
CNBC

Data breach hits roughly 15M T-Mobile customers

A hack of Experian, the
customers' Social Secu

Hacker Releases More Democratic Party Documents

By ERIC LICHTBLAU and NOAH WEILAND AUG. 12, 2016

HOME SEARCH The New York Times SUBSCRIBE

U.S.

‘Shadow Brokers’ Leak Raises Alarming Question: Was the N.S.A. Hacked?

By DAVID E. SANGER AUG. 16, 2016

339

Yahoo Says Hackers Stole Data on 500 Million Users in 2014

By NICOLE PERLROTH SEPT. 22, 2016

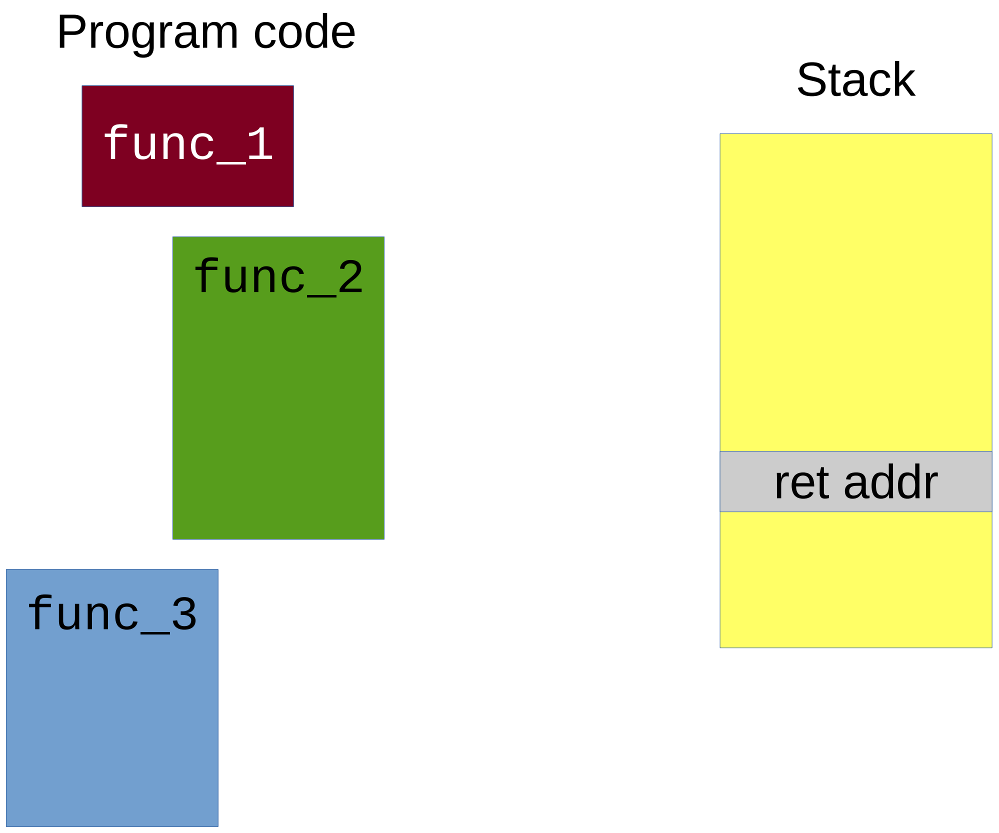
Andrew Harrer | Bloomberg | Getty Images

A pedestrian talks on his cell phone while
headquarters in New York.

More than 1.5 million Verizon Enterprise customers had their contact

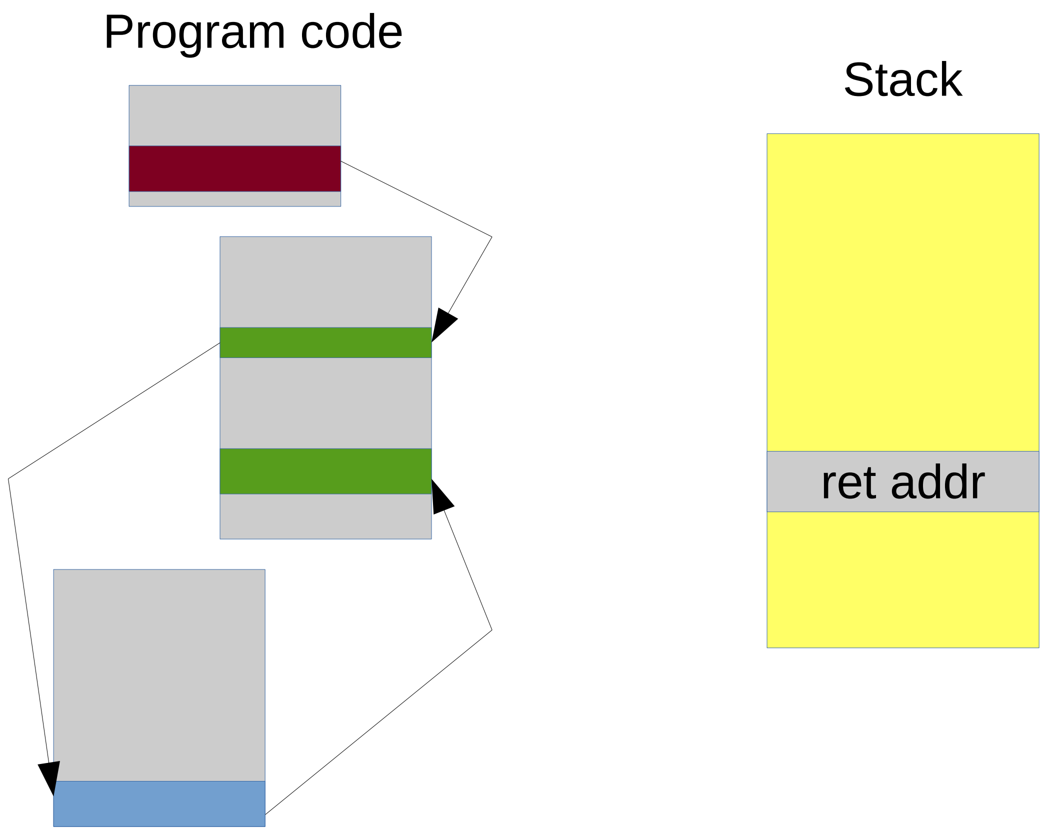
Return-Oriented Programming

- Reuse fragments of legitimate code (gadgets)



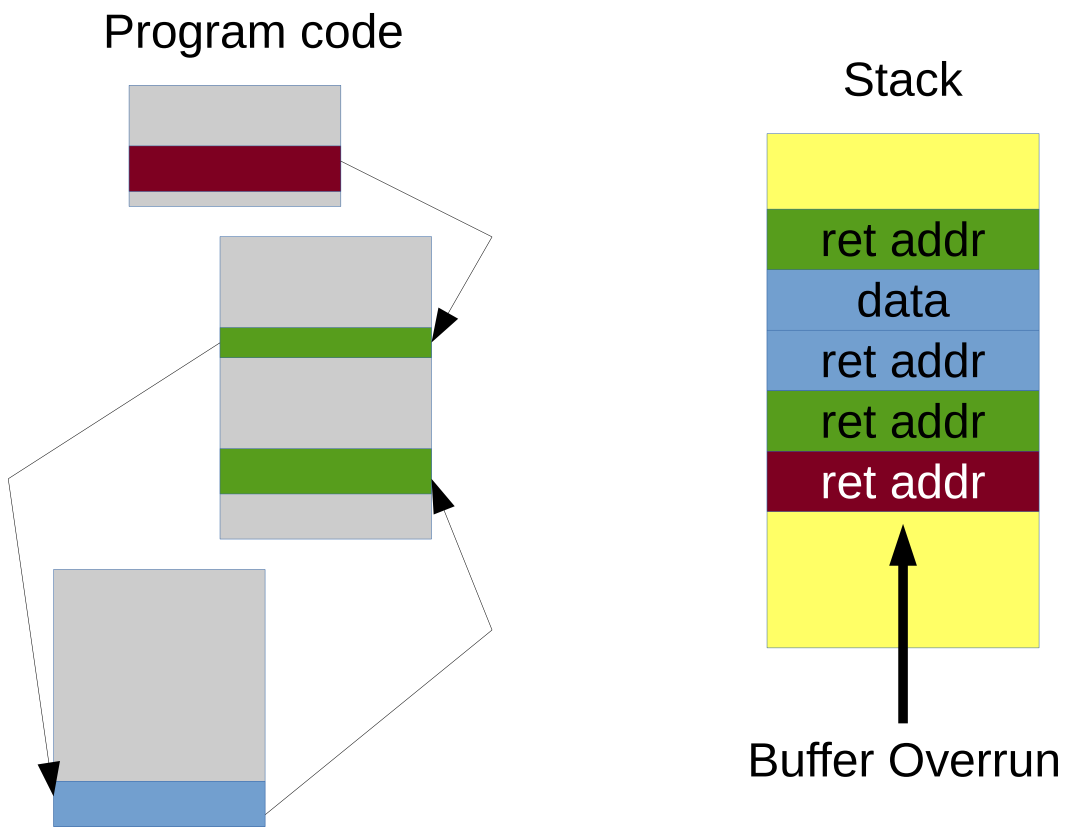
Return-Oriented Programming

- Reuse fragments of legitimate code (gadgets)



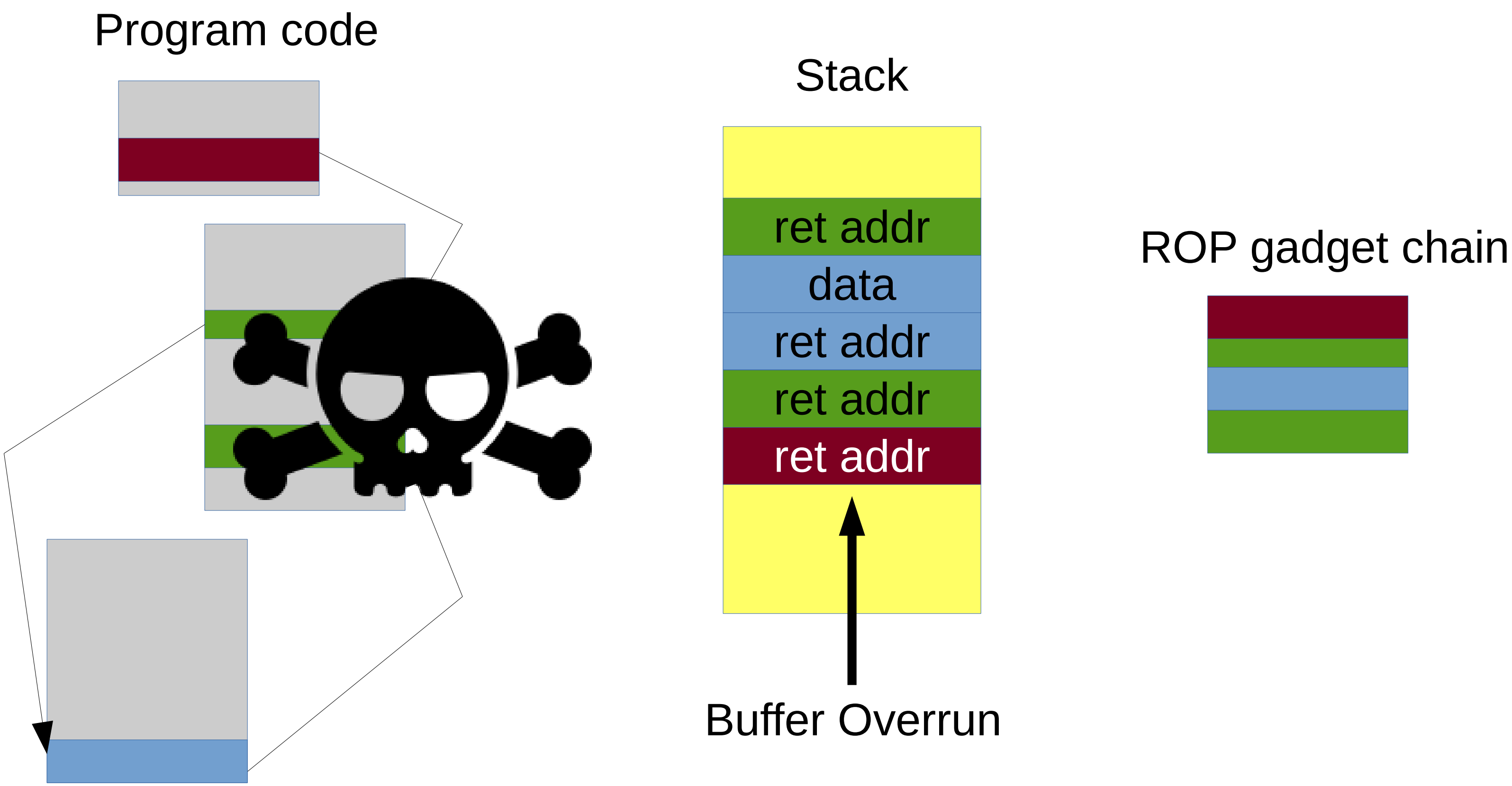
Return-Oriented Programming

- Reuse fragments of legitimate code (gadgets)



Return-Oriented Programming

- Reuse fragments of legitimate code (gadgets)

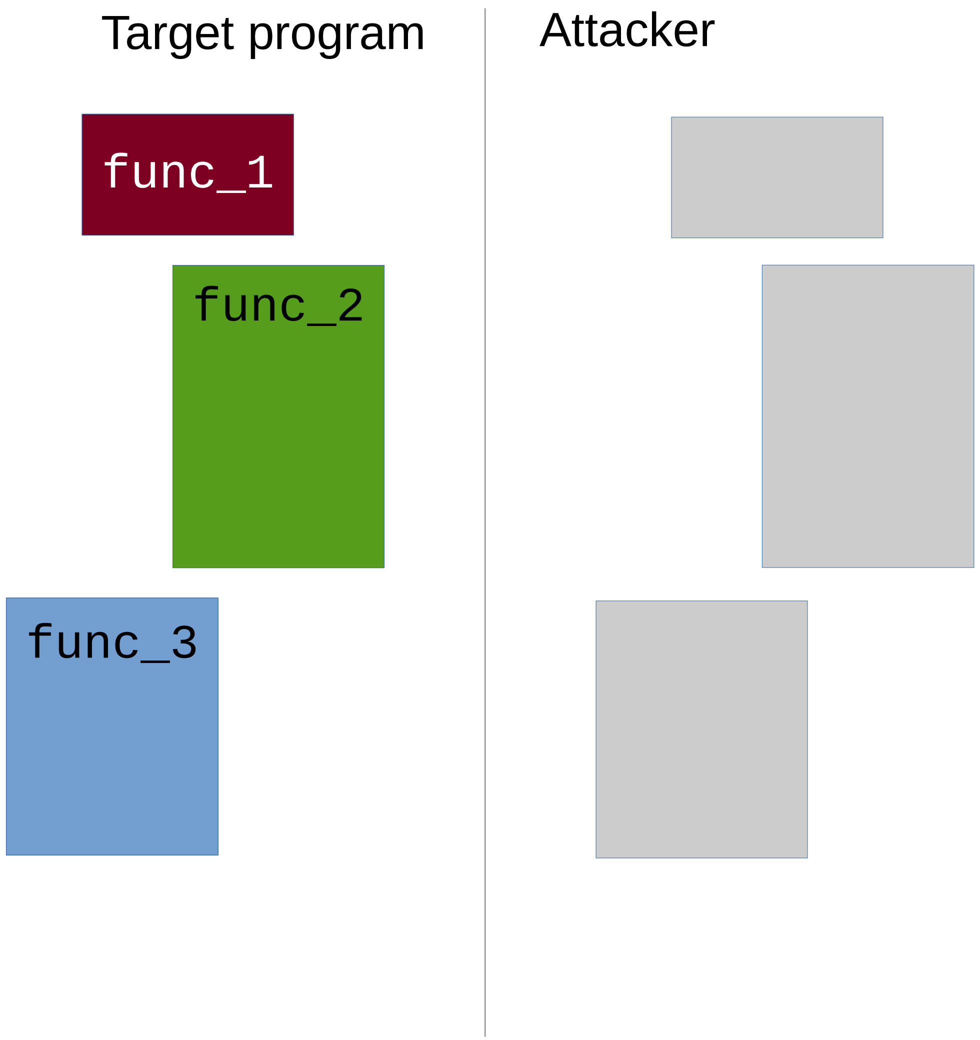


Modern ROP Attacks

- JIT-ROP [2]: iteratively read code at runtime

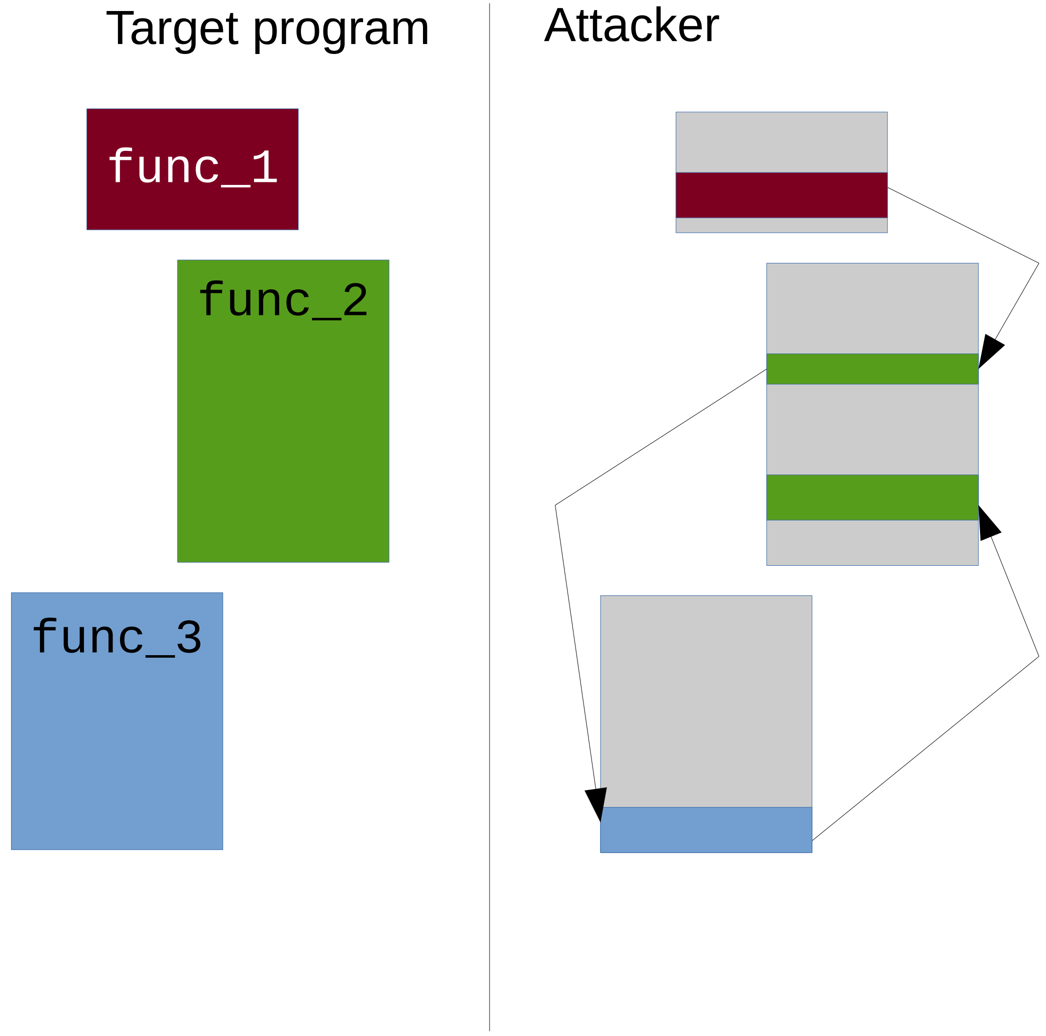
Modern ROP Attacks

- JIT-ROP [2]: iteratively read code at runtime



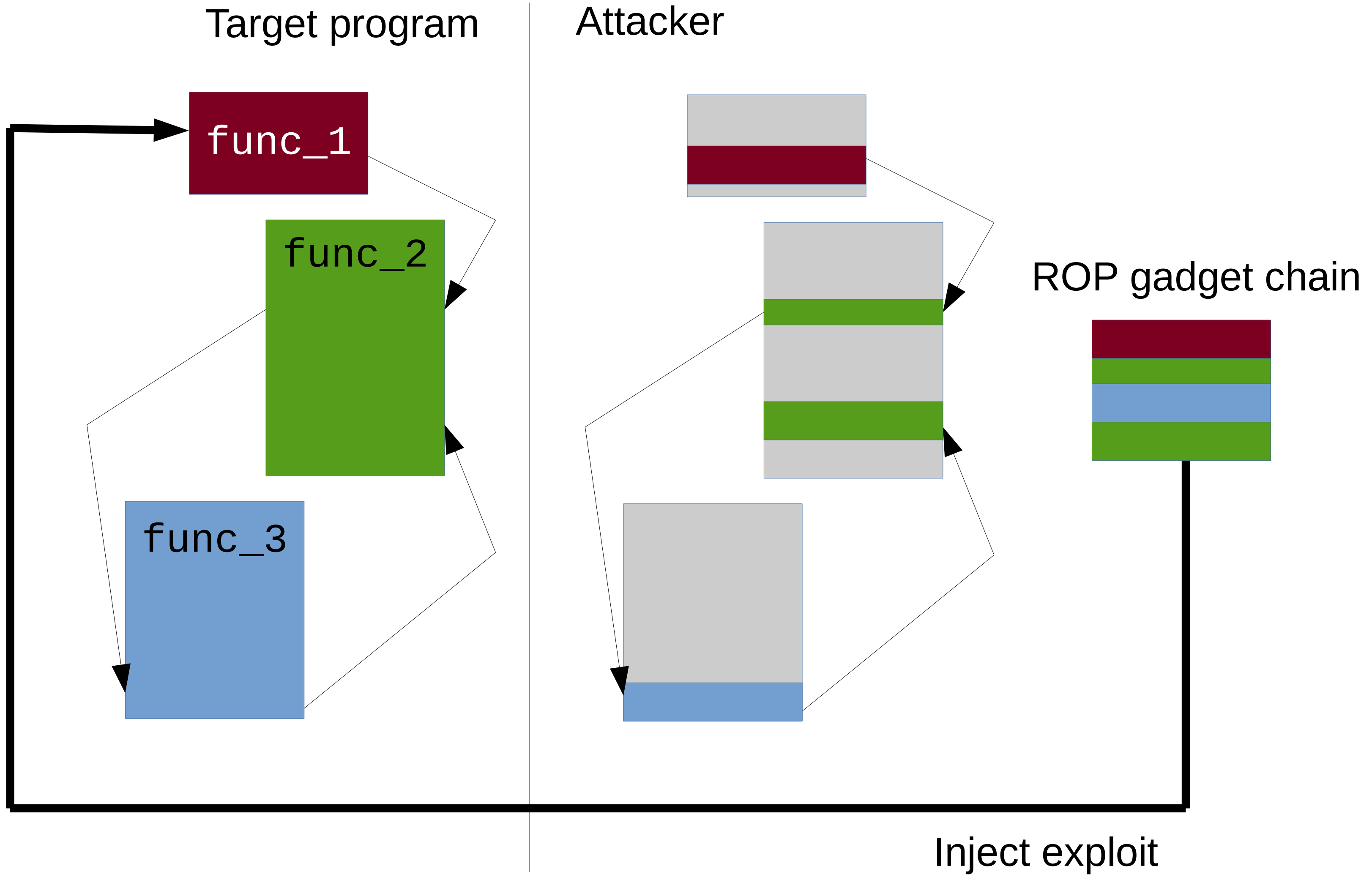
Modern ROP Attacks

- JIT-ROP [2]: iteratively read code at runtime



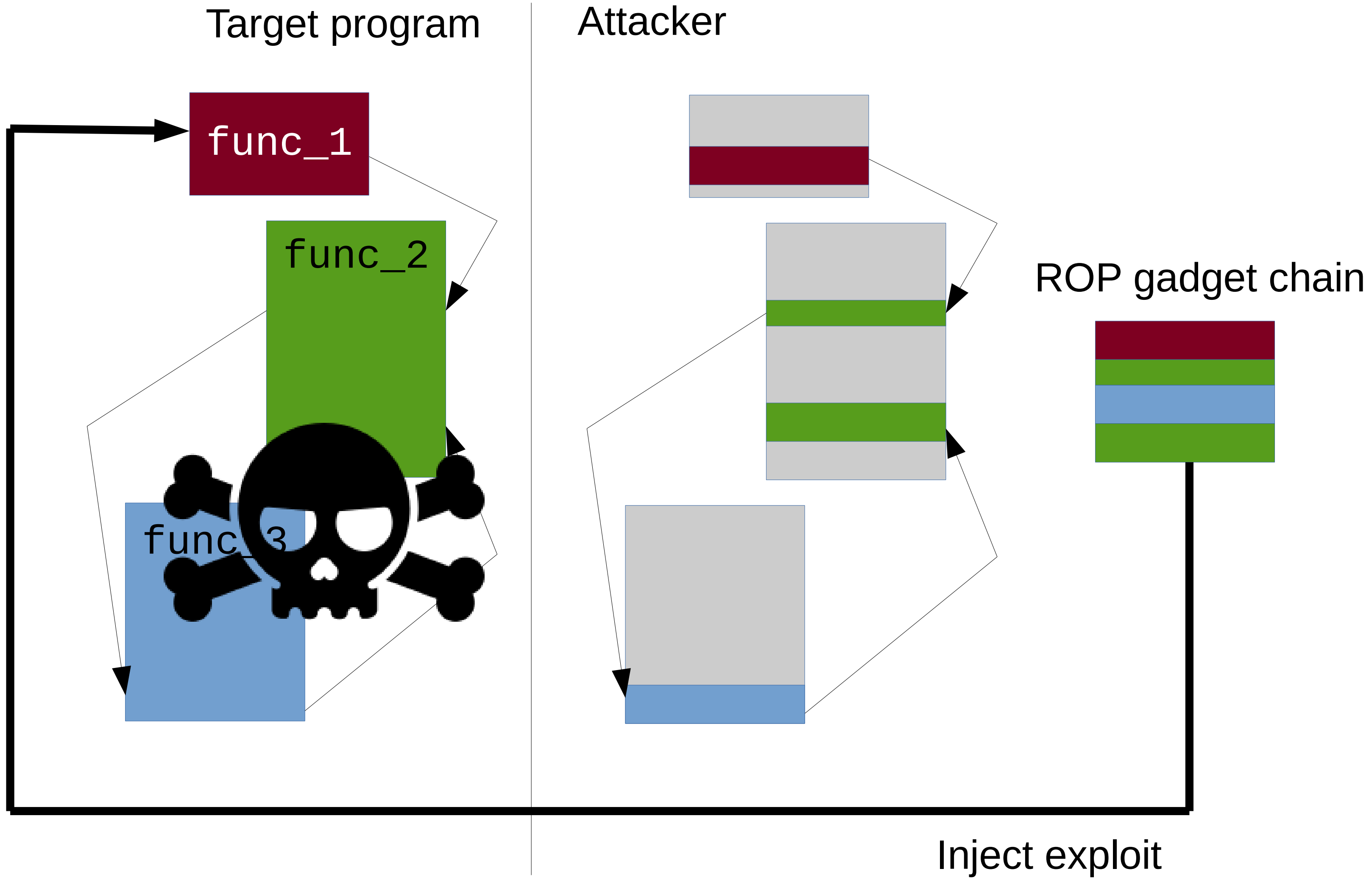
Modern ROP Attacks

- JIT-ROP [2]: iteratively read code at runtime



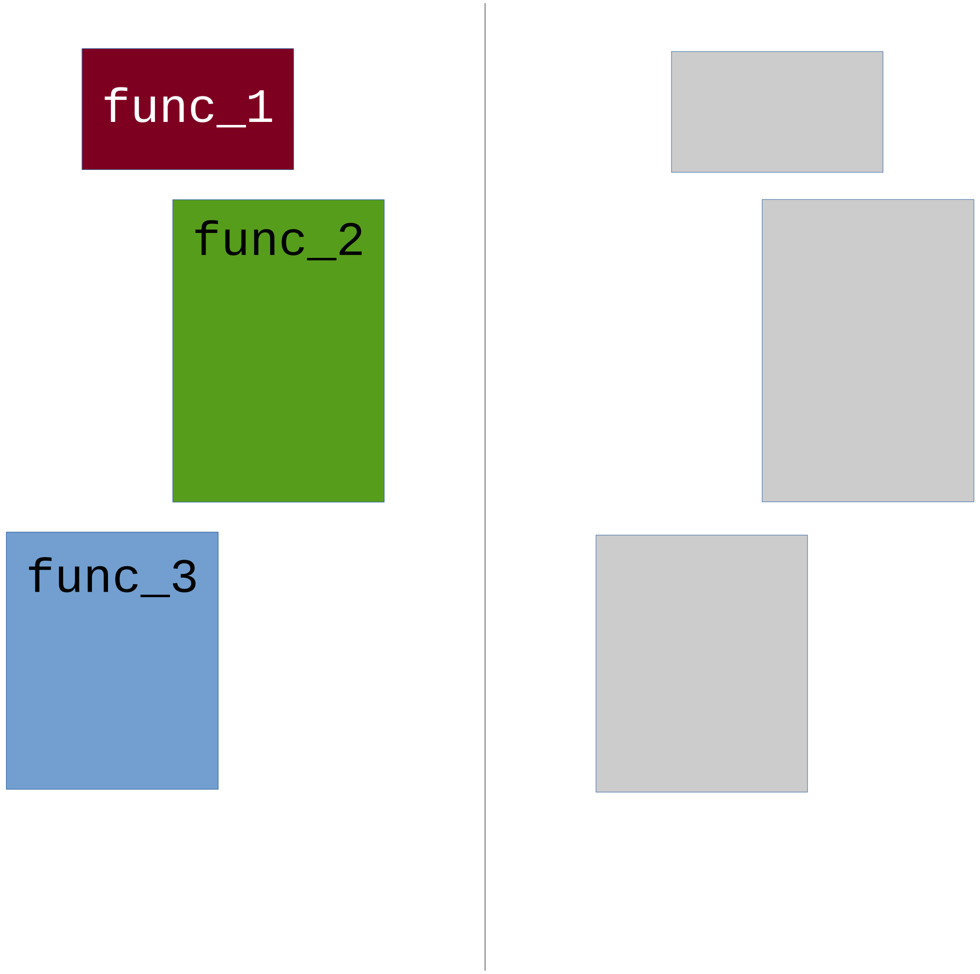
Modern ROP Attacks

- JIT-ROP [2]: iteratively read code at runtime



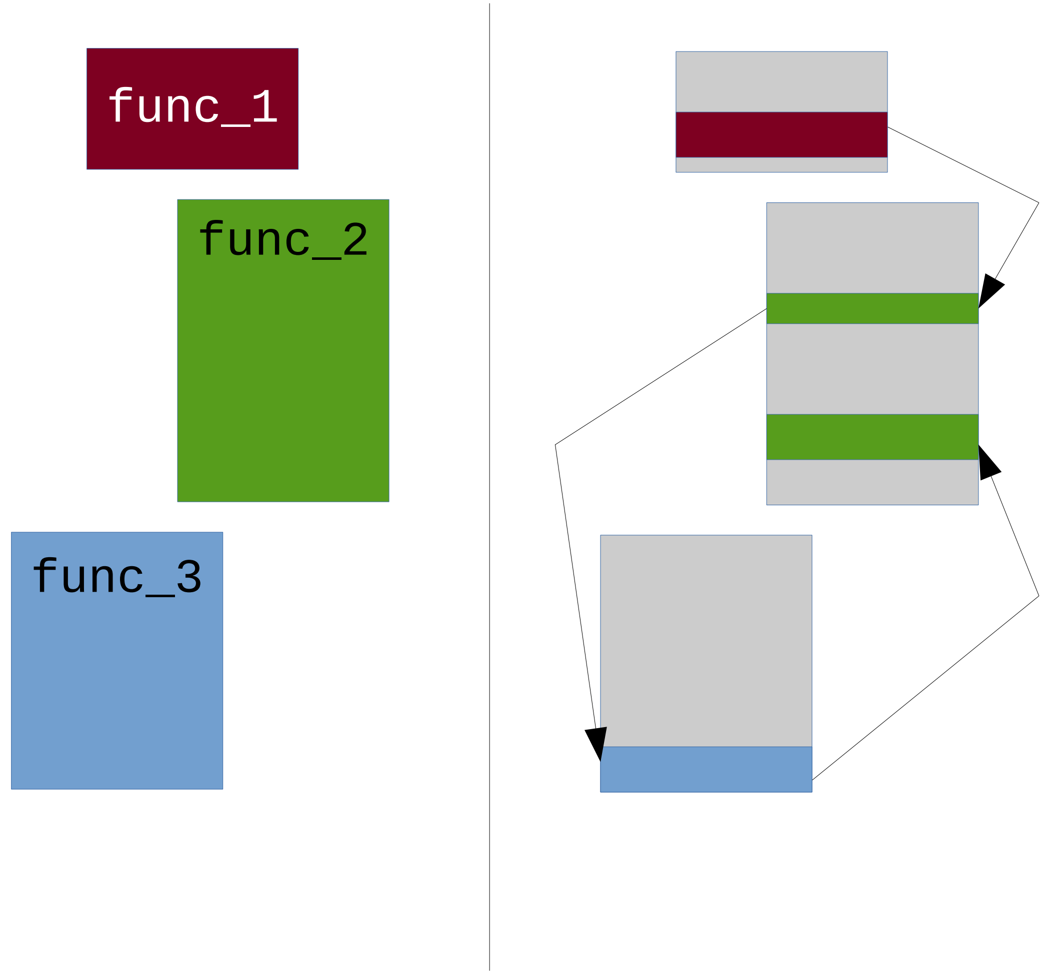
The Shuffler Idea

- What if we re-randomize code more rapidly than an attacker discovers gadgets?



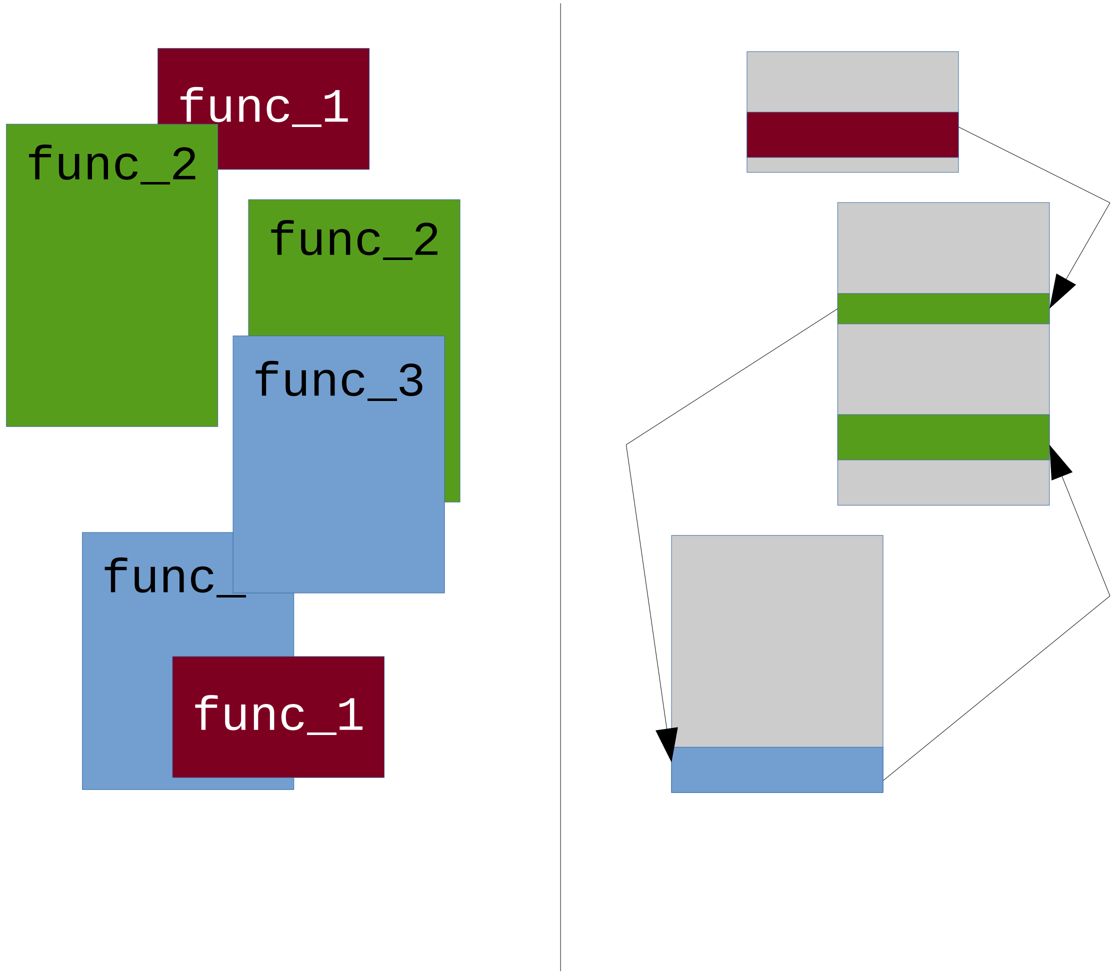
The Shuffler Idea

- What if we re-randomize code more rapidly than an attacker discovers gadgets?



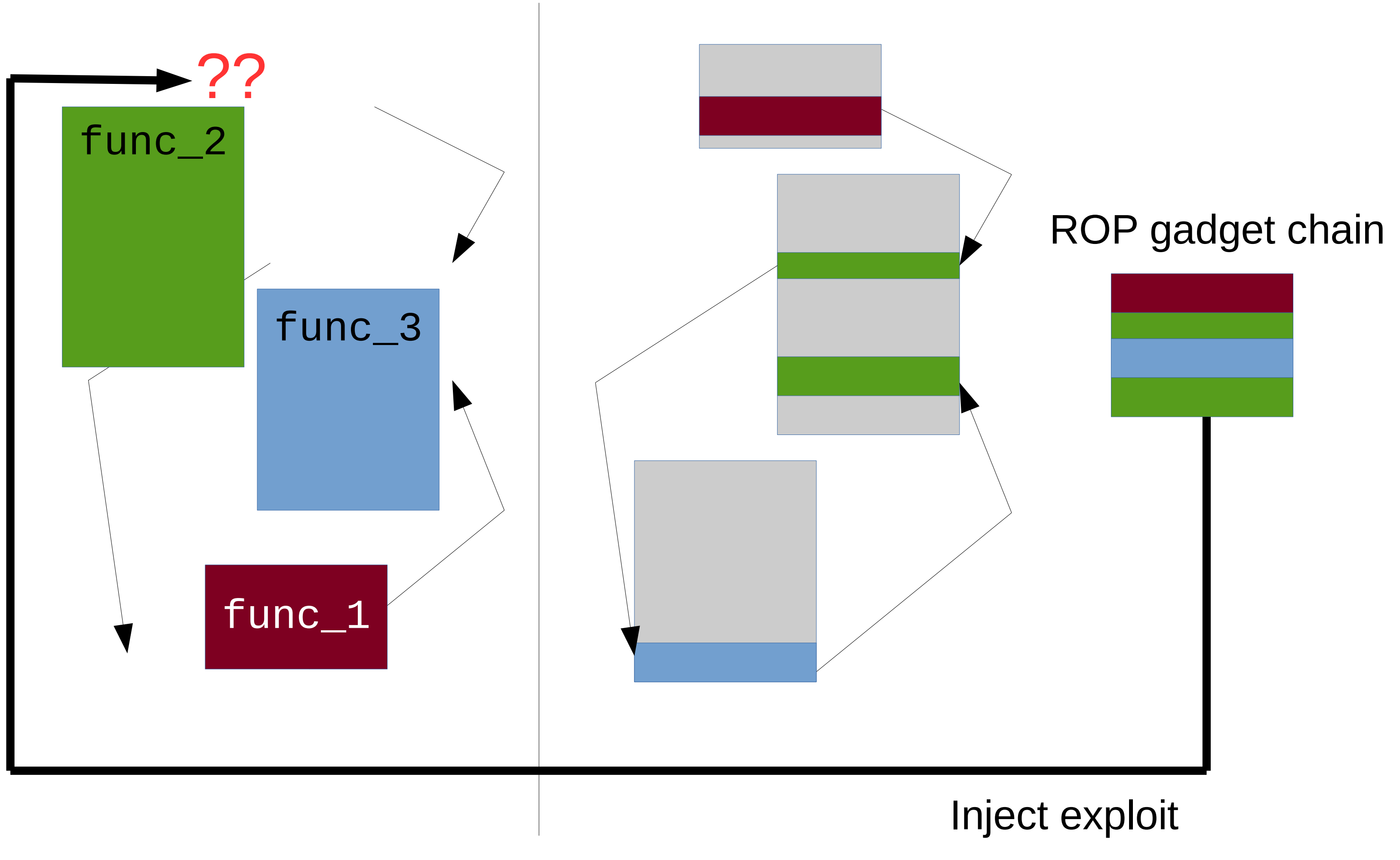
The Shuffler Idea

- What if we re-randomize code more rapidly than an attacker discovers gadgets?



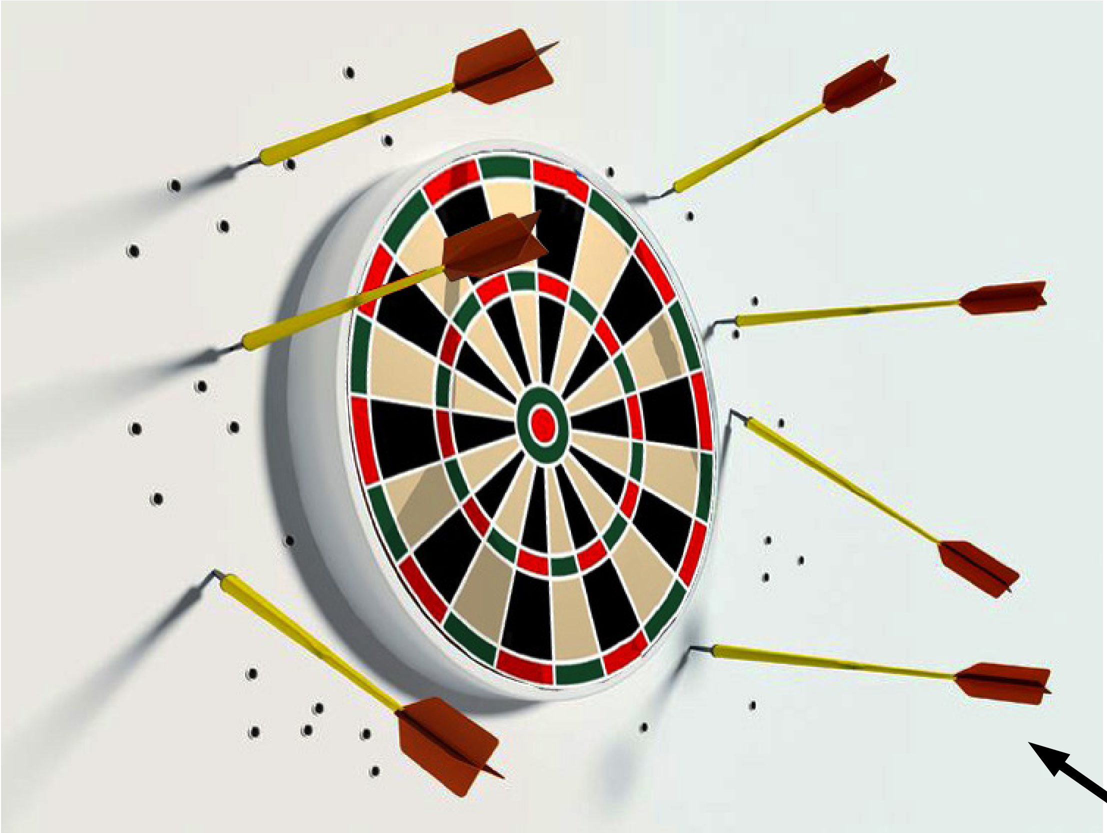
The Shuffler Idea

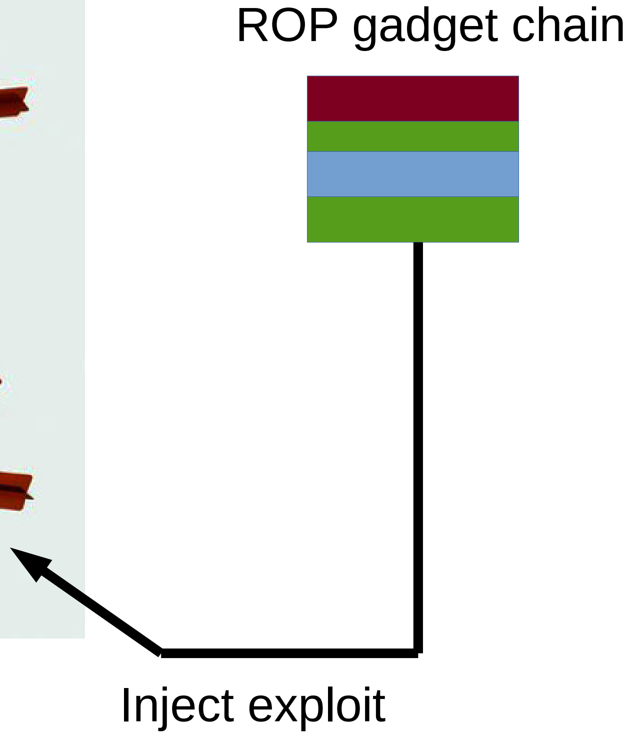
- What if we re-randomize code more rapidly than an attacker discovers gadgets?



The Shuffler Idea

- What if we re-randomize code more rapidly than an attacker discovers gadgets?





How Is This Possible?

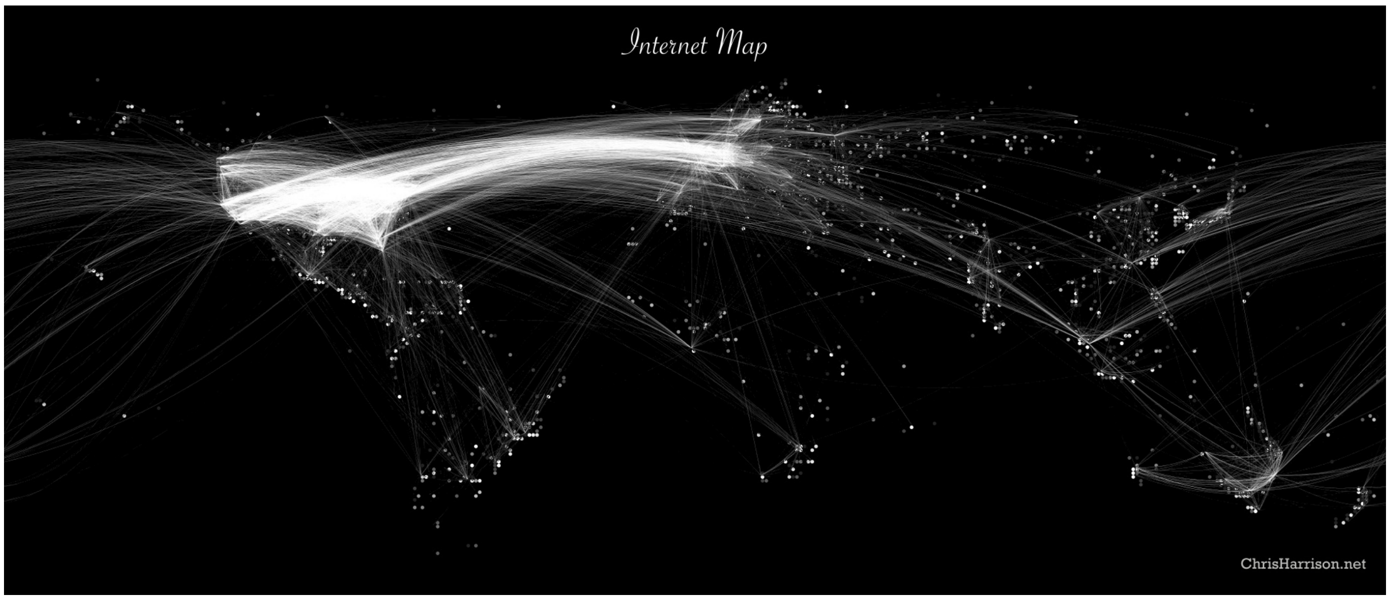
- Re-randomize code before an attacker uses it

How Is This Possible?

- Re-randomize code before an attacker uses it
  - faster than disclosure vulnerability execution time;
  - faster than gadget chain computation time;
  - or, faster than network communication time

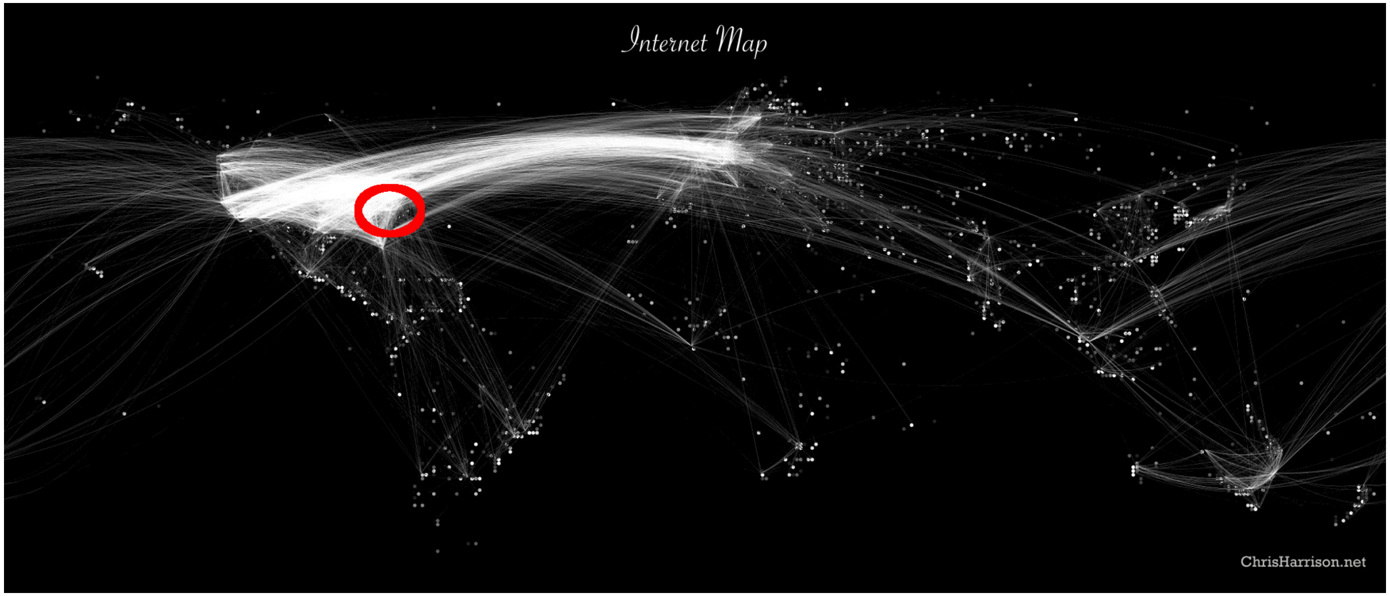
How Is This Possible?

- Re-randomize code before an attacker uses it
  - faster than disclosure vulnerability execution time;
  - faster than gadget chain computation time;
  - or, faster than network communication time



How Is This Possible?

- Re-randomize code before an attacker uses it
  - faster than disclosure vulnerability execution time;
  - faster than gadget chain computation time;
  - or, faster than network communication time
    - one memory disclosure can only travel 820 miles!



What Is Shuffler?

- Defense based on continuous re-randomization
  - Defeats all known code reuse attacks
  - 20-50 millisecond shuffling, scales to 24 threads
- Fast: bounds attacker’s available time
  - Defeats even attackers with zero network latency
- Deployable:
  - Binary analysis w/o modifying kernel, compiler, ...
- Egalitarian:
  - Shuffler runs in same address space, defends itself

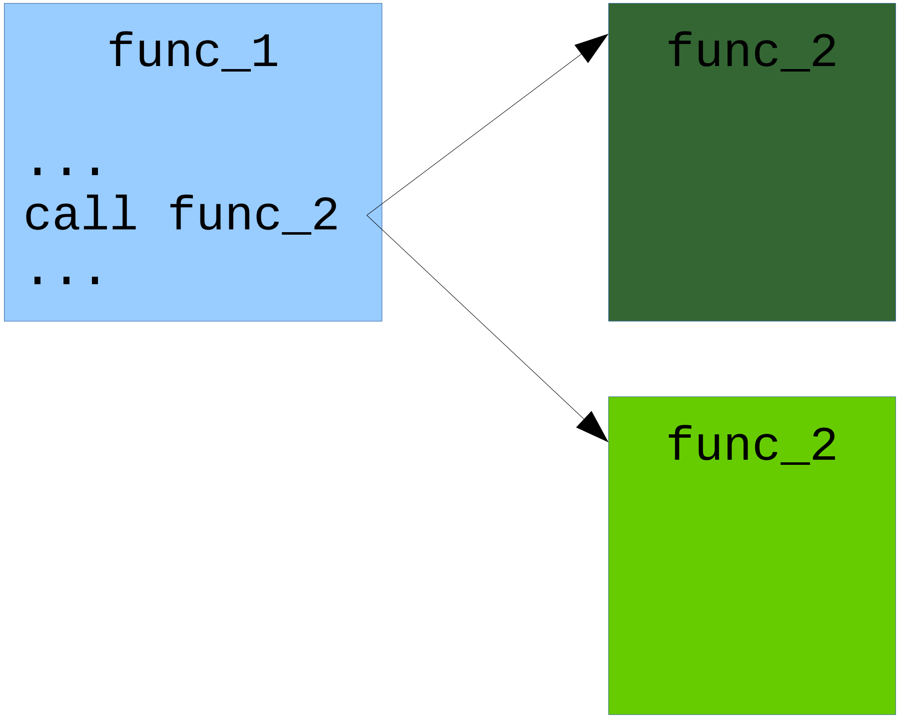
Outline

Outline

1. Continuous re-randomization
2. Accelerating our randomization
3. Binary analysis and egalitarianism
4. Results and Demo

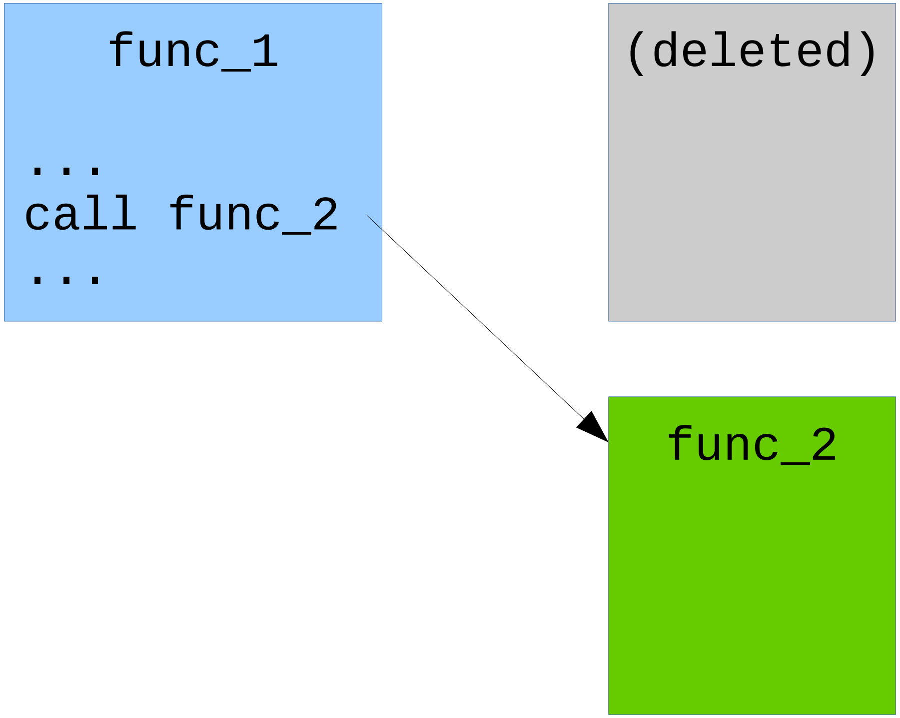
Continuous Re-Randomization

- Easy to copy code & fix direct references



Continuous Re-Randomization

- Easy to copy code & fix direct references

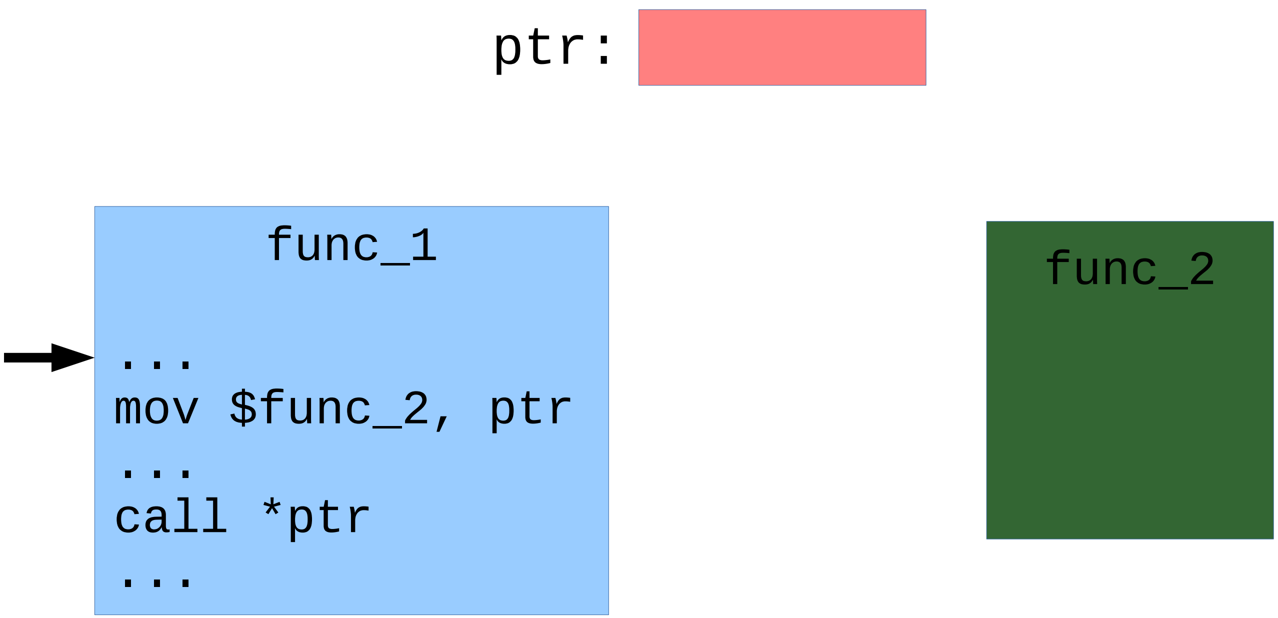


Continuous Re-Randomization

- Easy to copy code & fix direct references
- What about code pointers?

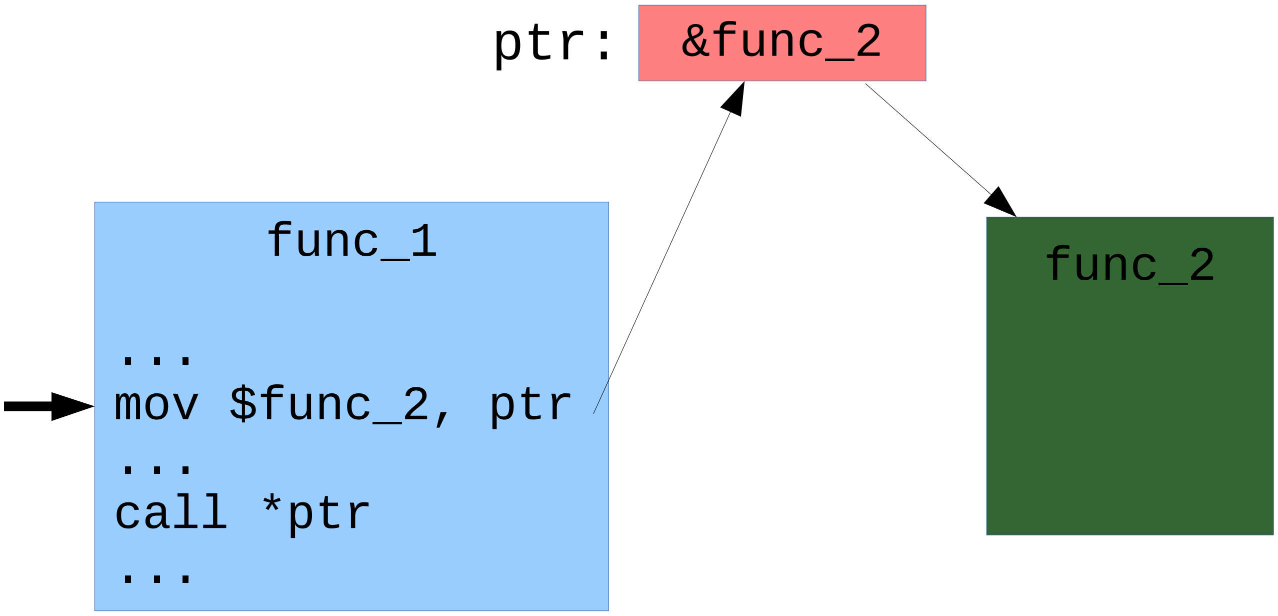
Continuous Re-Randomization

- Easy to copy code & fix direct references
- What about code pointers?



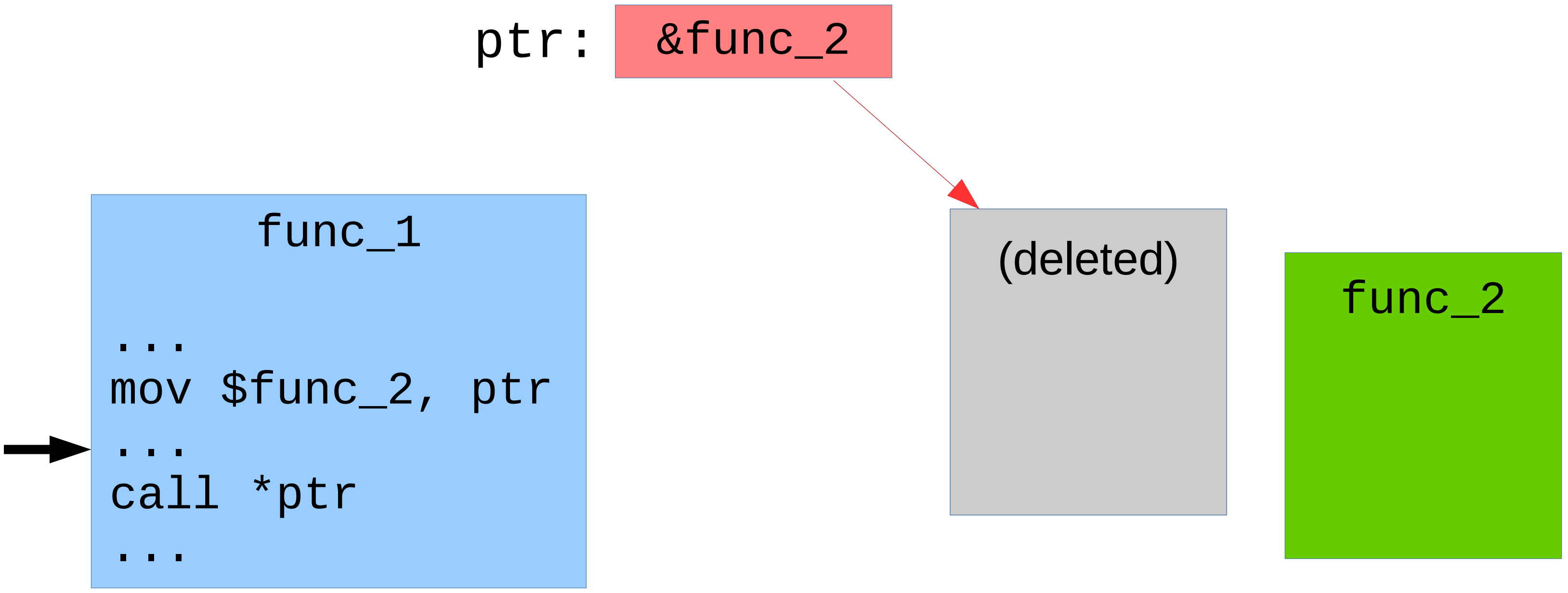
Continuous Re-Randomization

- Easy to copy code & fix direct references
- What about code pointers?



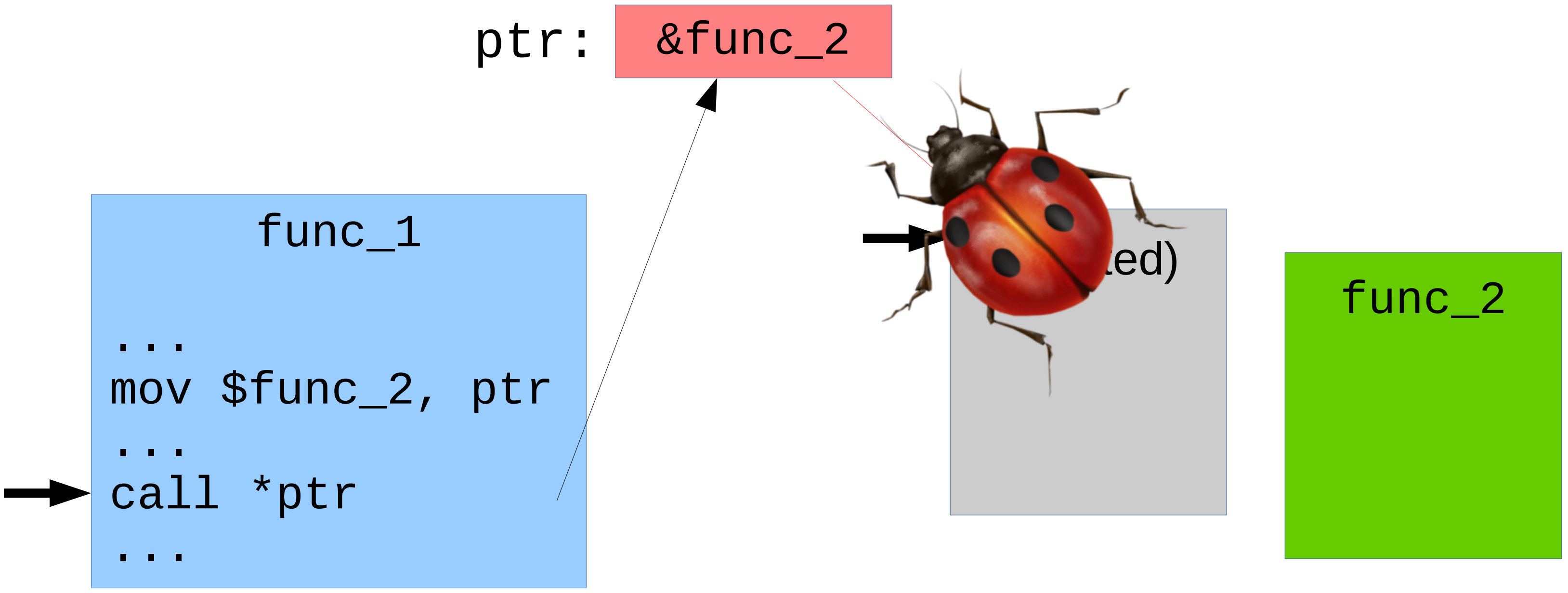
Continuous Re-Randomization

- Easy to copy code & fix direct references
- What about code pointers?



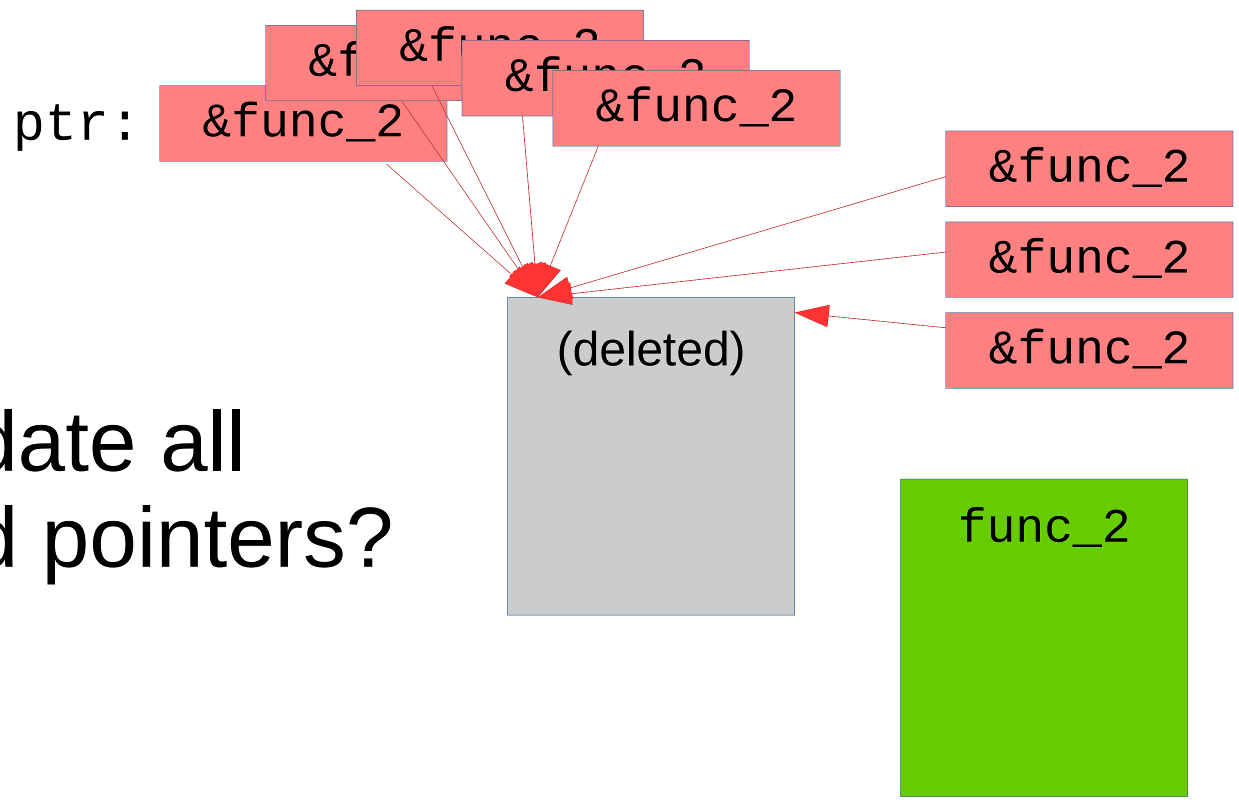
Continuous Re-Randomization

- Easy to copy code & fix direct references
- What about code pointers?



Continuous Re-Randomization

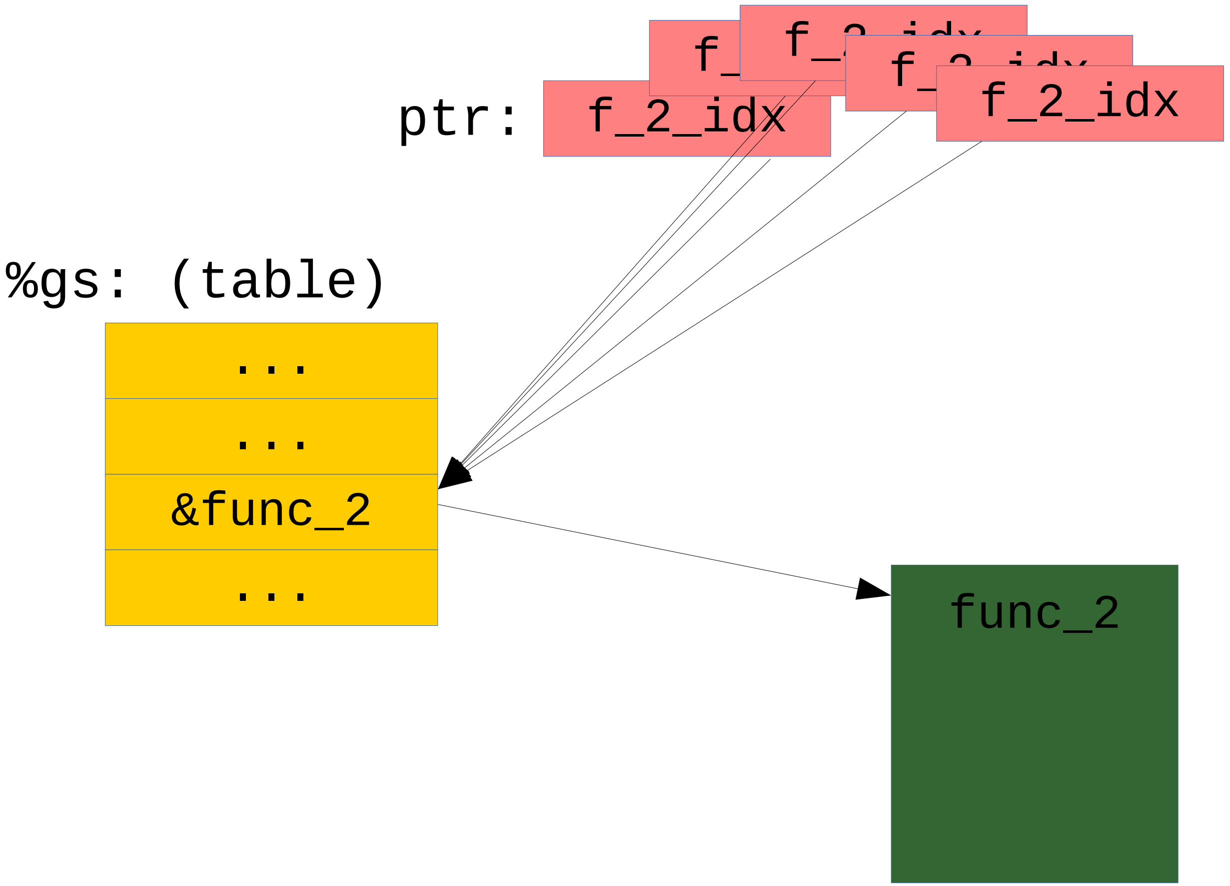
- Easy to copy code & fix direct references
- What about code pointers?



- How to update all propagated pointers?

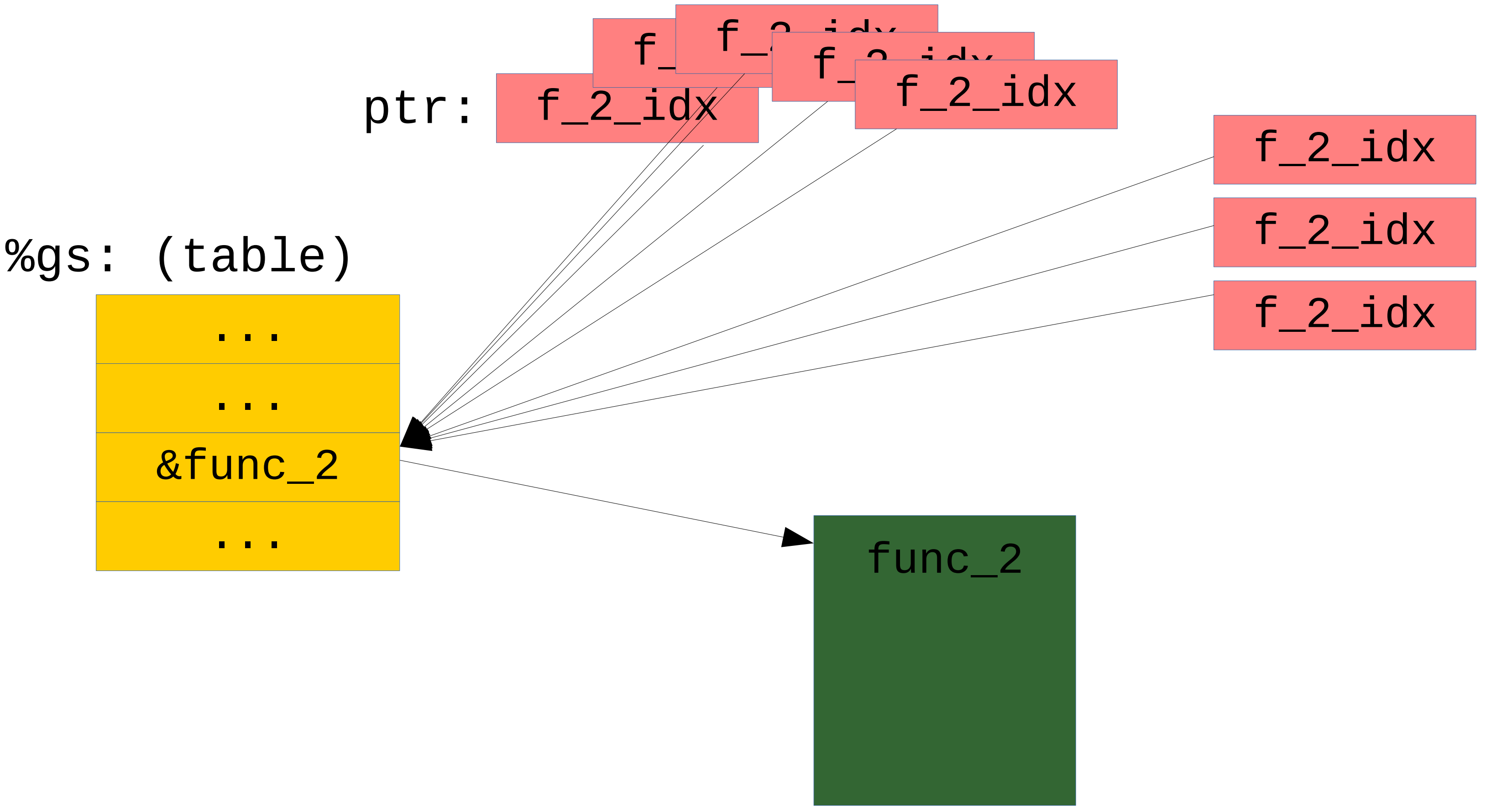
Continuous Re-Randomization

- Solution: add extra level of indirection



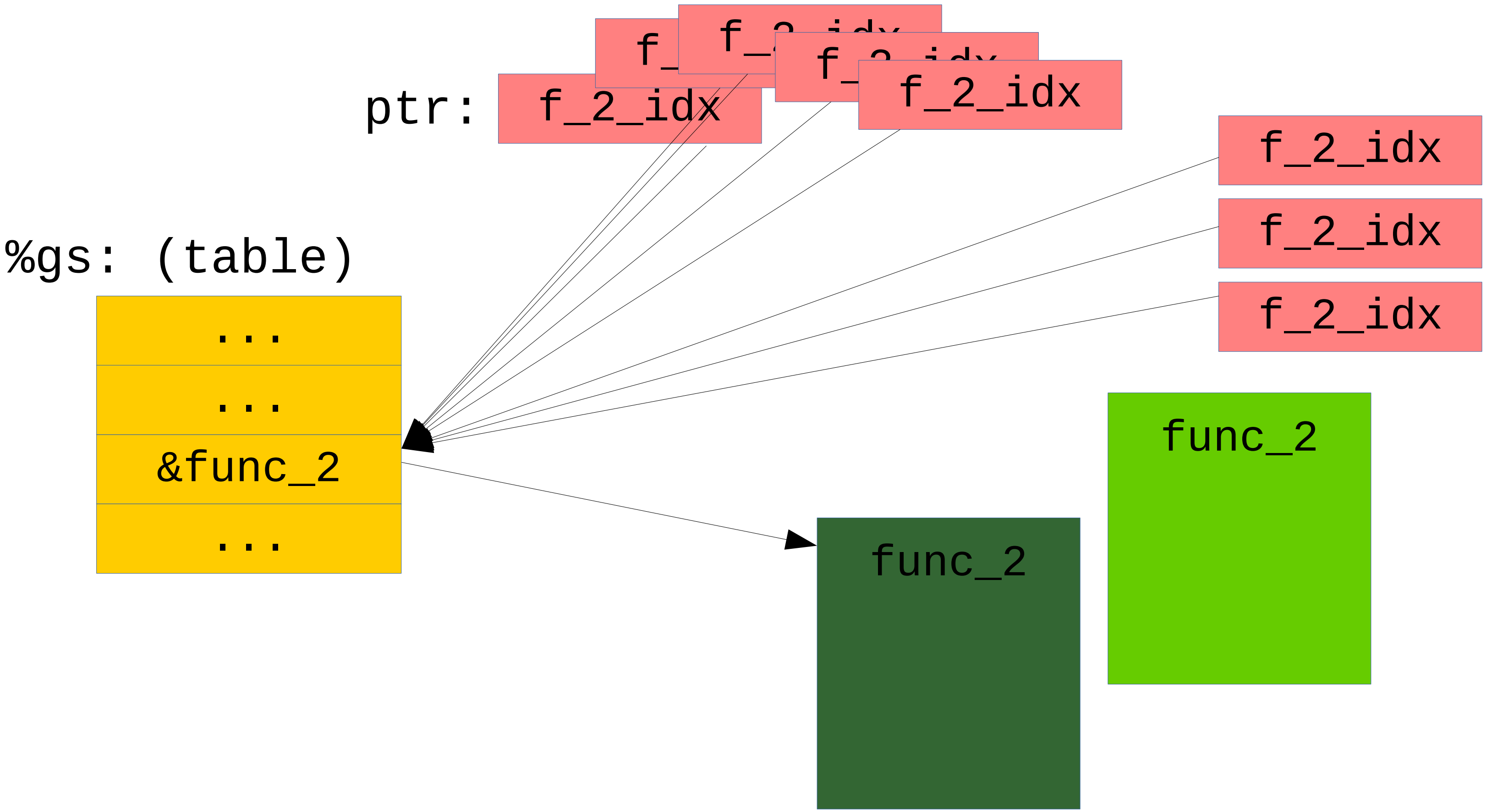
Continuous Re-Randomization

- Solution: add extra level of indirection



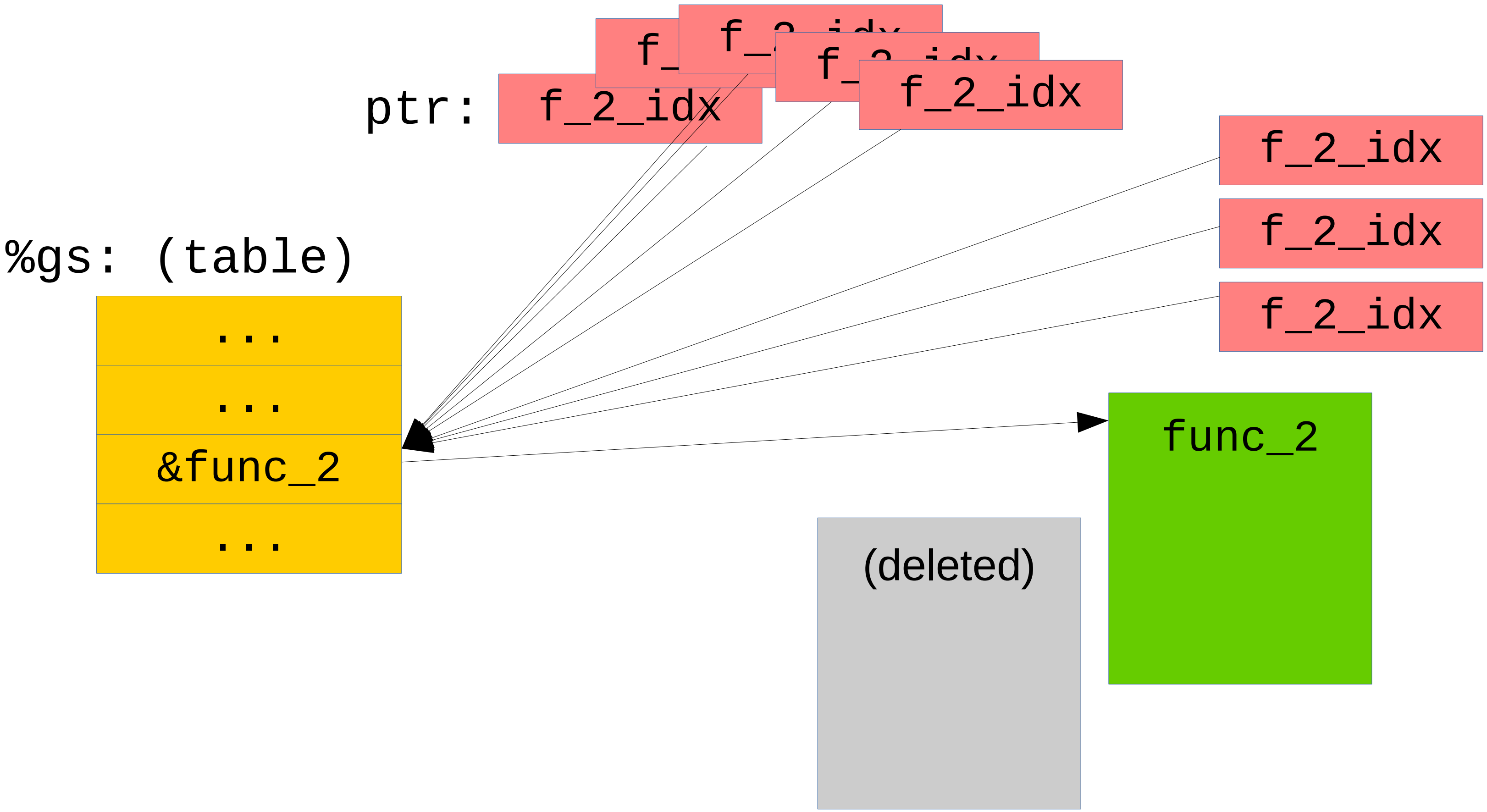
Continuous Re-Randomization

- Solution: add extra level of indirection



Continuous Re-Randomization

- Solution: add extra level of indirection

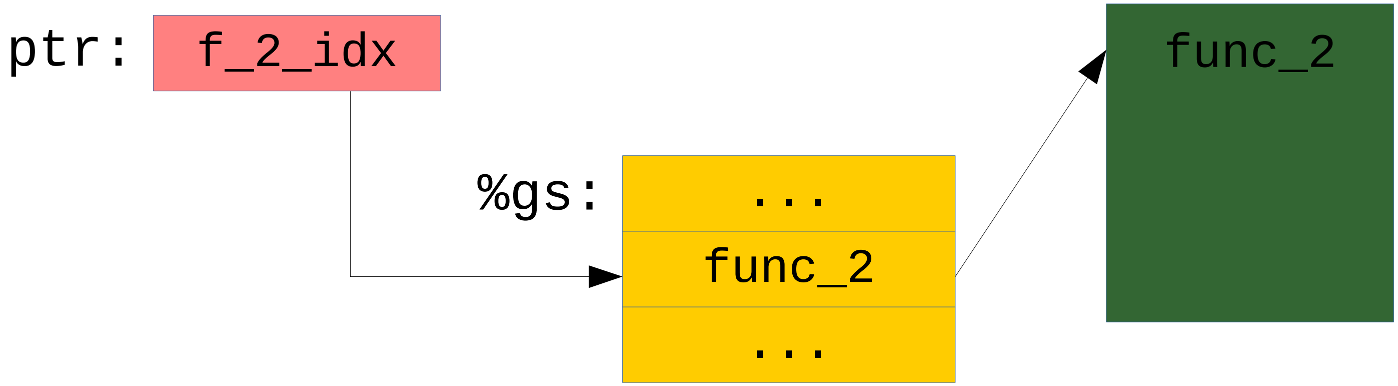


Code Pointer Abstraction

- Transforming *code_ptr into **code_ptr
  - **Correctness**: pointer updates sound & precise
  - **Disclosure-resilience**: code ptr table is hidden

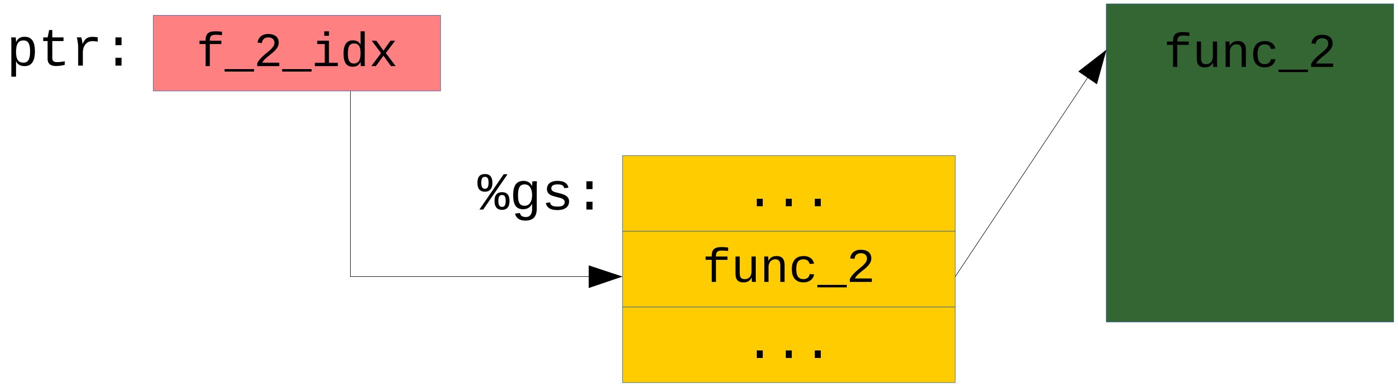
Code Pointer Abstraction

- Transforming *code_ptr into **code_ptr
  - **Correctness**: pointer updates sound & precise
  - **Disclosure-resilience**: code ptr table is hidden



Code Pointer Abstraction

- Transforming *code_ptr into **code_ptr
  - **Correctness**: pointer updates sound & precise
  - **Disclosure-resilience**: code ptr table is hidden



Rewrite call sites

```
    callq *%rax

=> callq *%gs:(%rax)
```

Rewrite initialization points

```
    mov  $0x40054d, %rax

=> mov  $0x20, %rax
```

Outline

1. Continuous re-randomization
2. Accelerating our randomization
3. Binary analysis and egalitarianism
4. Results and Demo

Return Address Encryption

- Return addresses are code pointers too
- Could use code pointer table, but inefficient
  - call/ret instructions highly optimized

Return Address Encryption

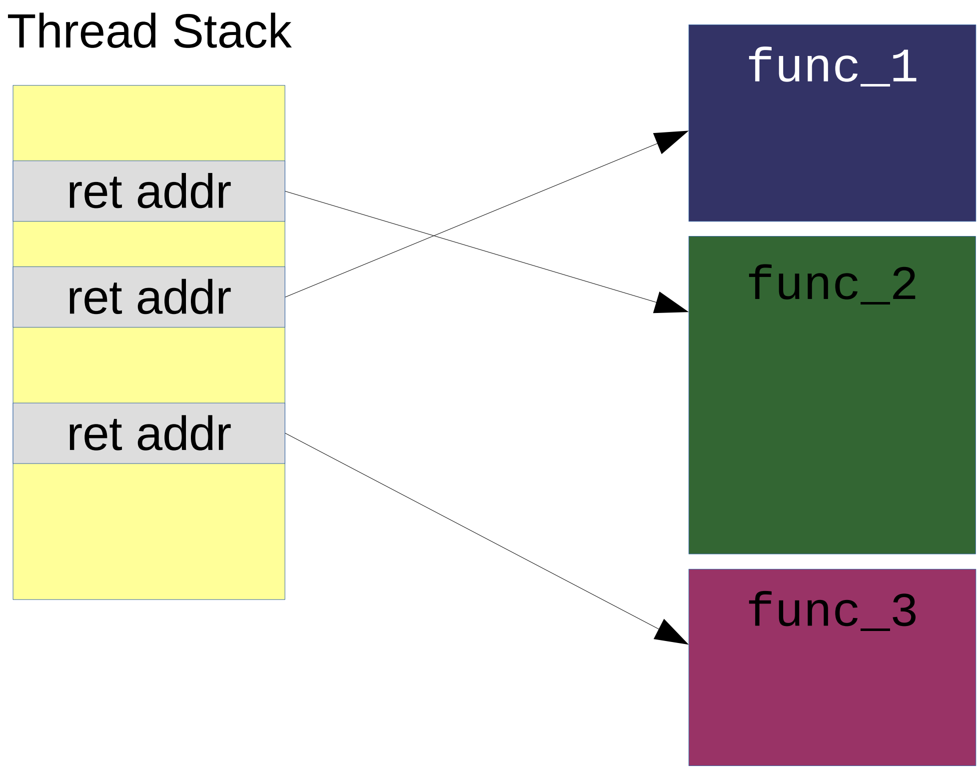
- Return addresses are code pointers too
- Could use code pointer table, but inefficient
  - call/ret instructions highly optimized
- Alternative mechanism – **correct** and **hidden**
  - Use normal call instructions
  - Encrypt return addresses with XOR key

Return Address Encryption

- Prevent return address disclosure

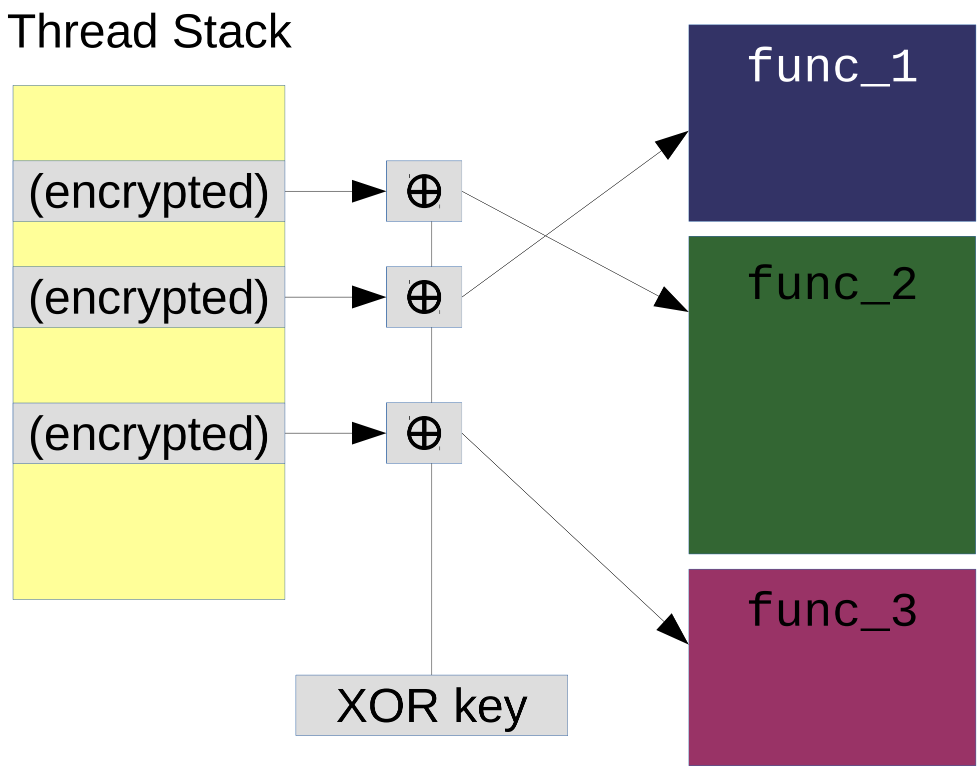
Return Address Encryption

- Prevent return address disclosure



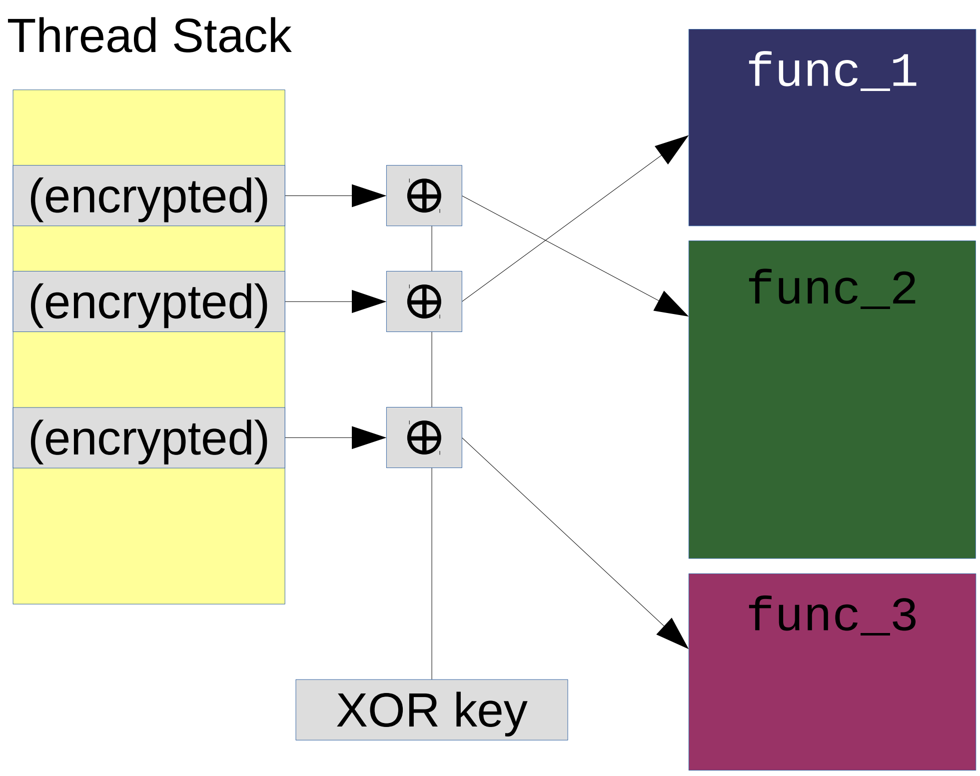
Return Address Encryption

- Prevent return address disclosure



Return Address Encryption

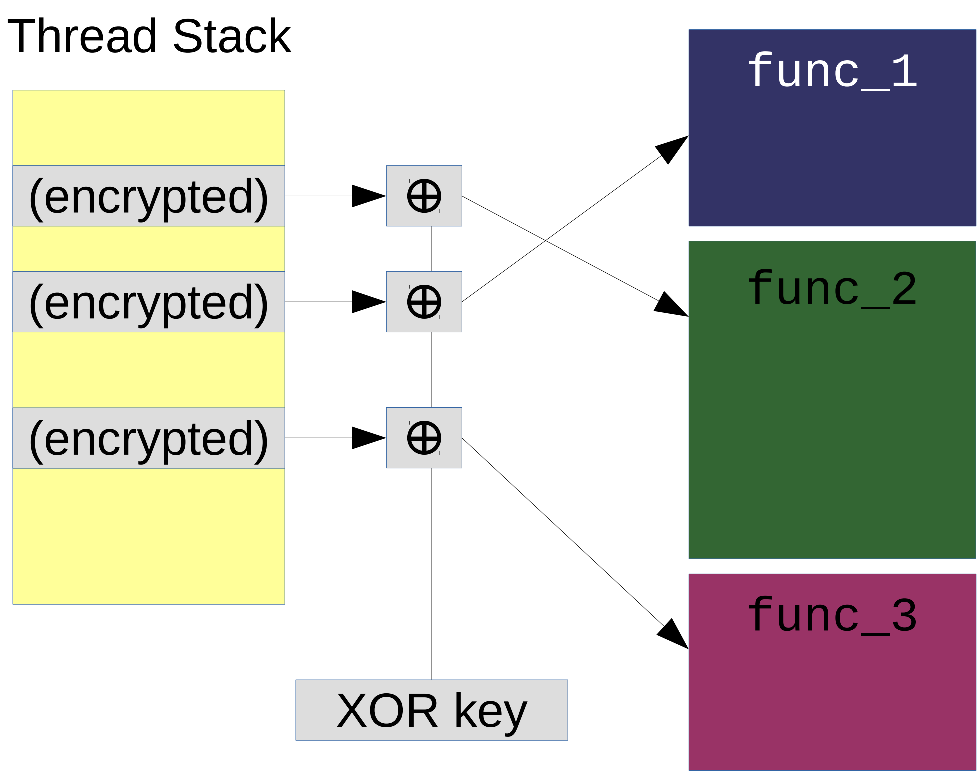
- Prevent return address disclosure




```
func:
    ; original code
    ret
```

Return Address Encryption

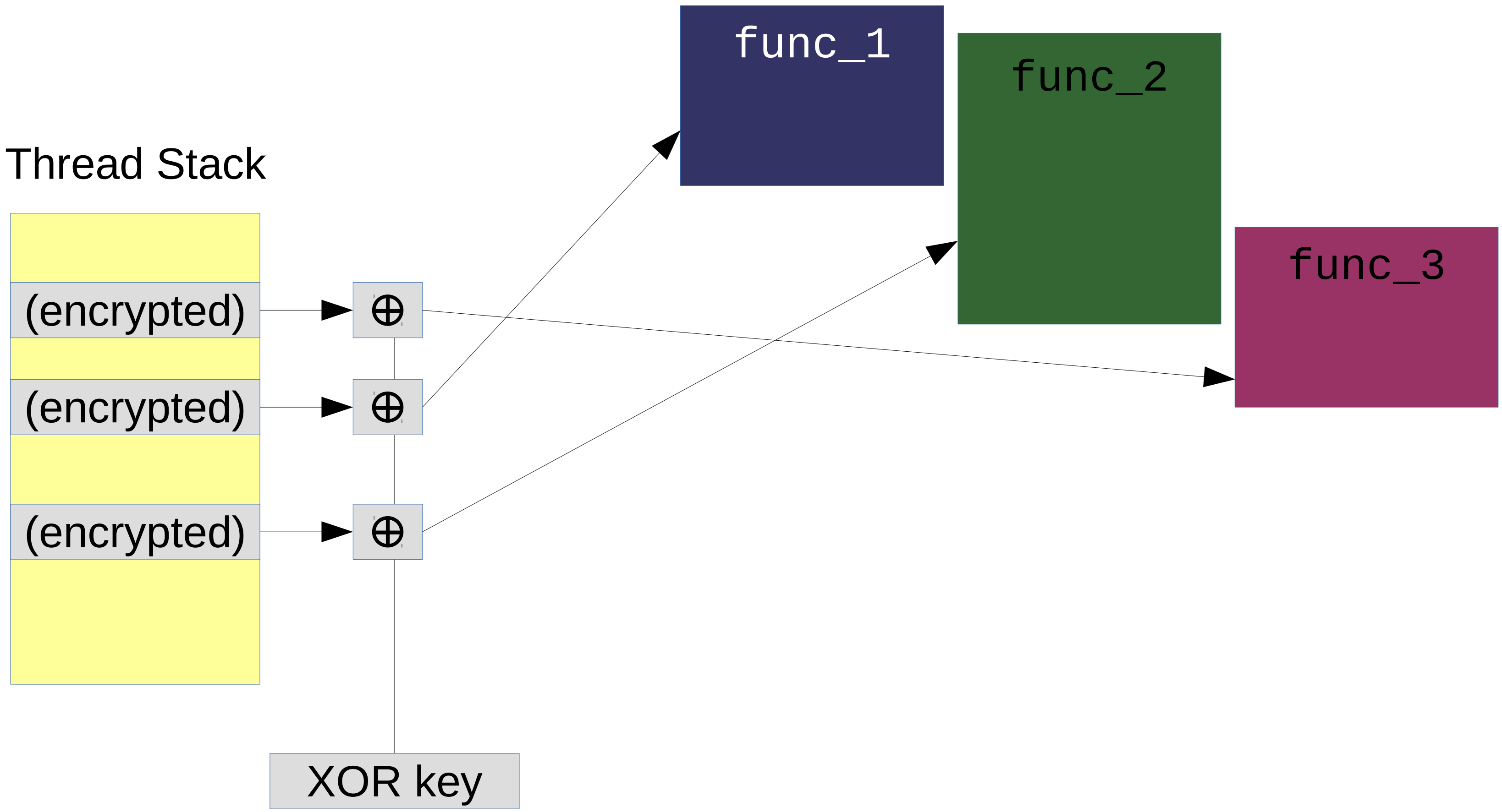
- Prevent return address disclosure
- We use binary rewriting (expand basic blocks)




```
func:
    mov    %fs:0x28,%r11
    xor    %r11,(%rsp)
    ; original code
    mov    %fs:0x28,%r11
    xor    %r11,(%rsp)
    ret
```

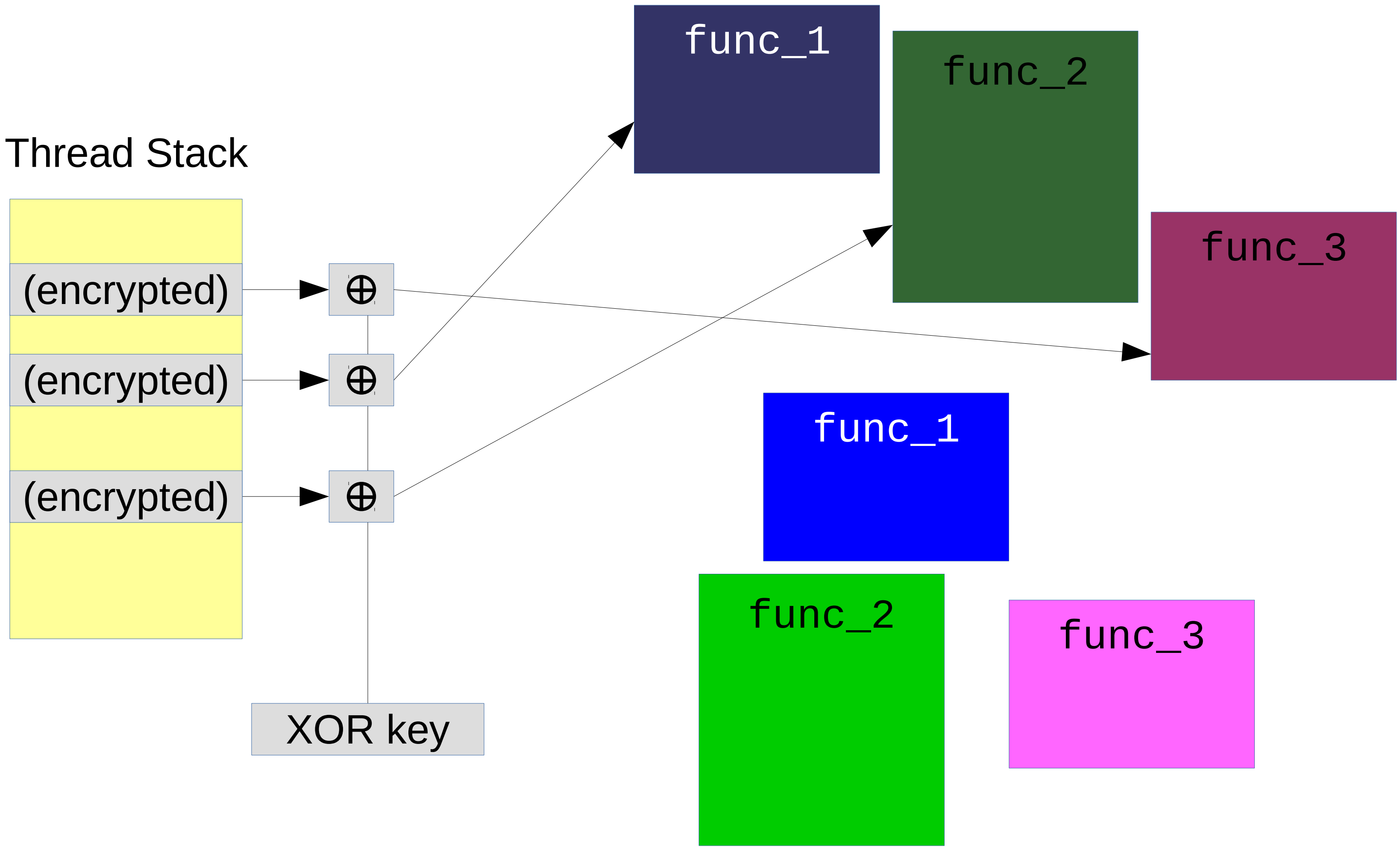
Return Address Migration

- Unwind stack and re-encrypt new addresses



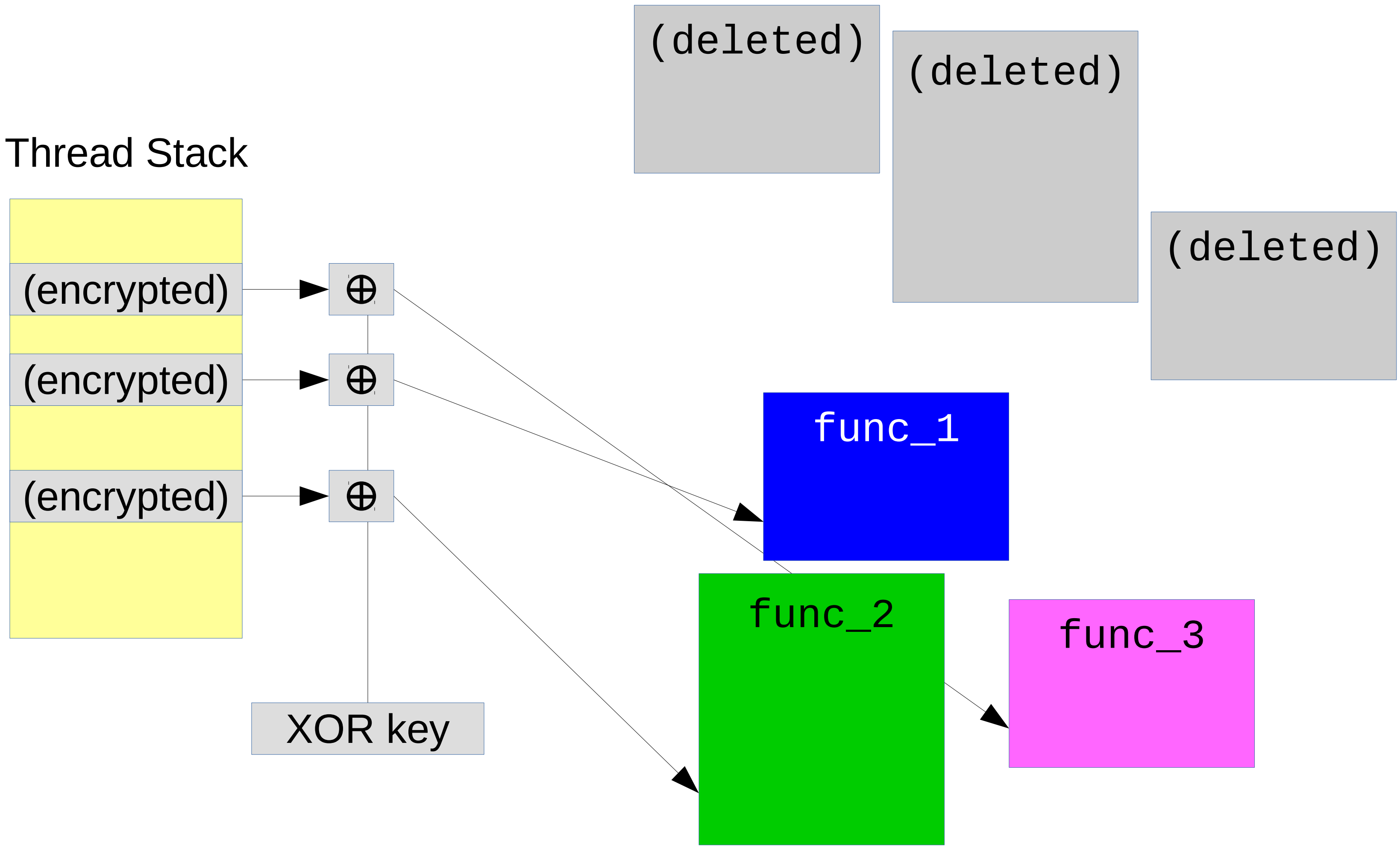
Return Address Migration

- Unwind stack and re-encrypt new addresses



Return Address Migration

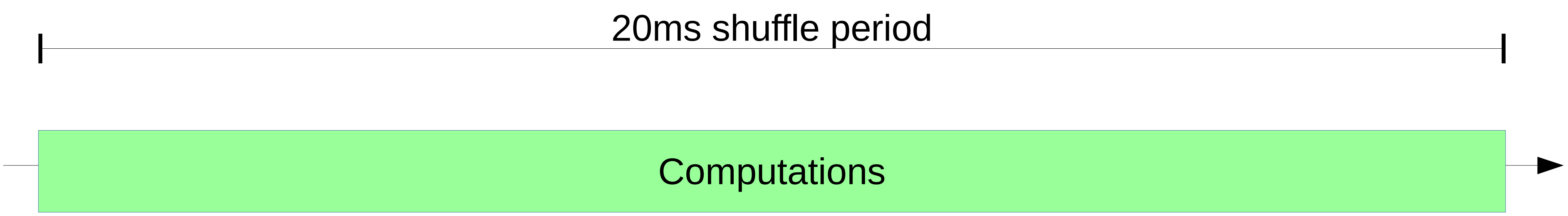
- Unwind stack and re-encrypt new addresses



Asynchronous Randomization

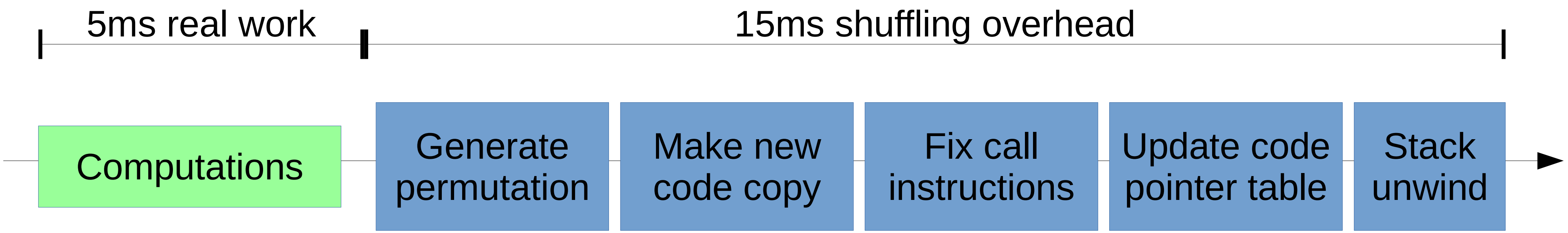
Asynchronous Randomization

- Creating new code copies takes time



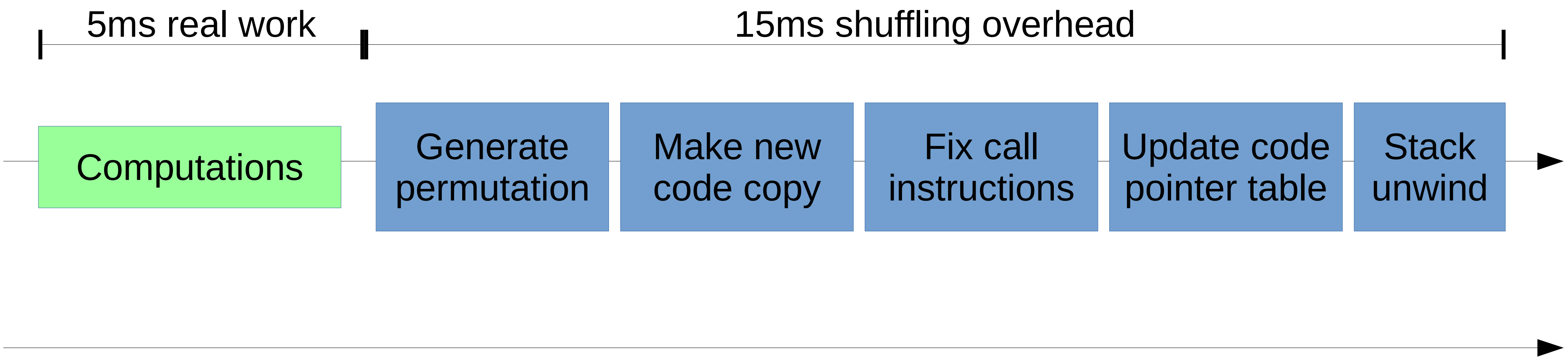
Asynchronous Randomization

- Creating new code copies takes time



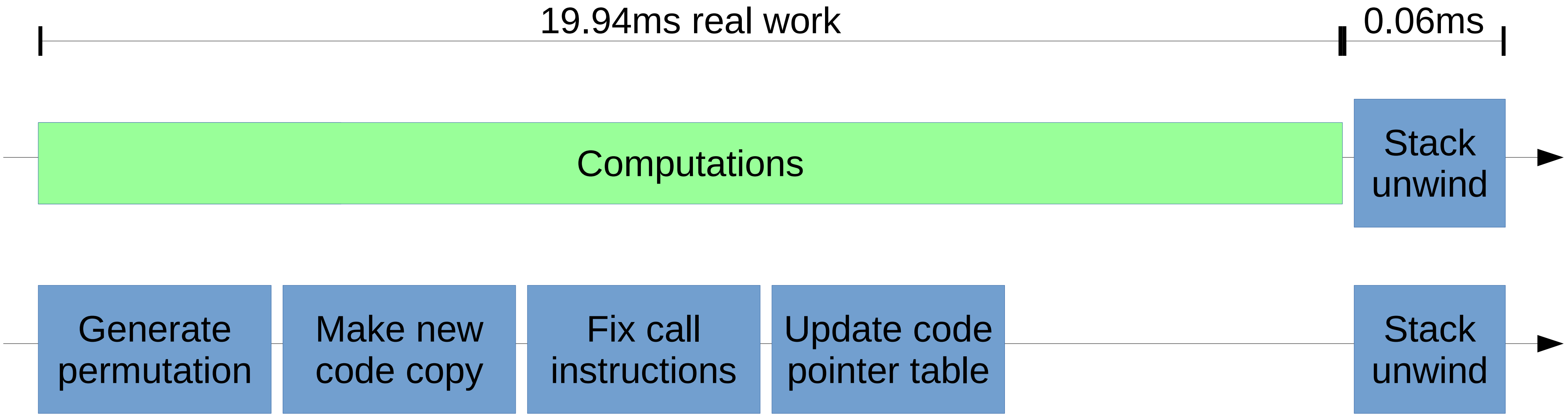
Asynchronous Randomization

- Creating new code copies takes time
- Shuffler prepares new code asynchronously



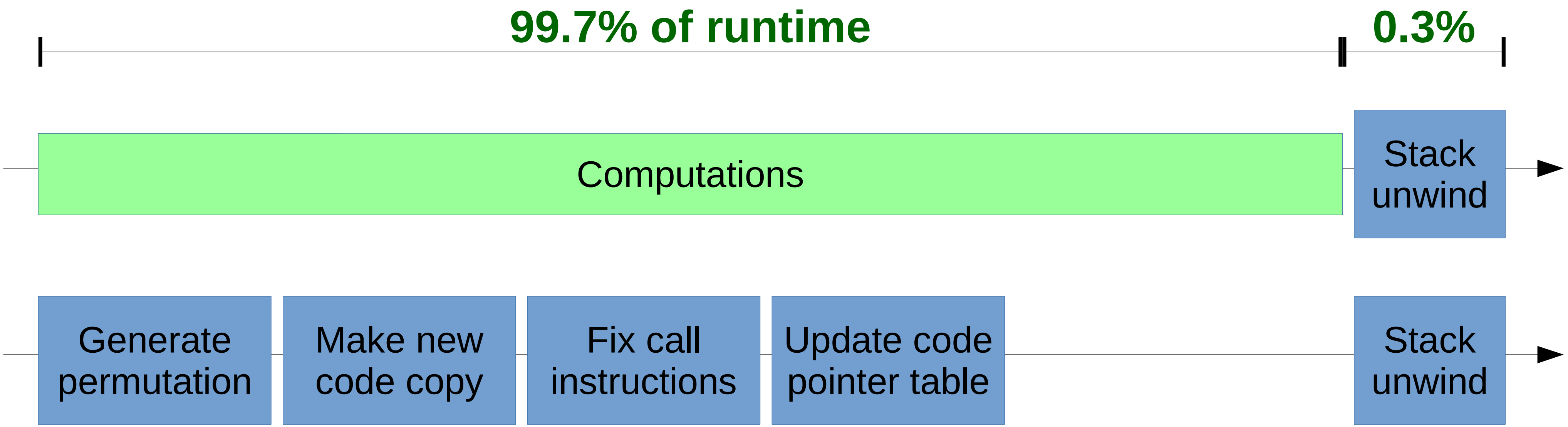
Asynchronous Randomization

- Creating new code copies takes time
- Shuffler prepares new code asynchronously



Asynchronous Randomization

- Creating new code copies takes time
- Shuffler prepares new code asynchronously
- Each thread unwinds its own stack in parallel



Outline

1. Continuous re-randomization
2. Accelerating our randomization
3. Binary analysis and egalitarianism
4. Results and Demo

Augmented Binary Analysis

- Use additional info from unmodified compilers
  - Symbols, to distinguish code and data (no -s)
  - Relocations, to find all code pointers (--emit-relocs)

Augmented Binary Analysis

- Use additional info from unmodified compilers
  - Symbols, to distinguish code and data (no -s)
  - Relocations, to find all code pointers (--emit-relocs)

Code pointer, or integer?

```
.section .rodata:
    .quad  0x400620

.section .text:
    mov    $0x400620, %rax
```

Augmented Binary Analysis

- Use additional info from unmodified compilers
  - Symbols, to distinguish code and data (no -s)
  - Relocations, to find all code pointers (--emit-relocs)

Code pointer, or integer?

```
.section .rodata:
    .quad  0x400620

.section .text:
    mov    $0x400620, %rax
```

```
.section .rodata:
    .quad  4195872

.section .text:
    mov    $4195872, %rax
```

Augmented Binary Analysis

- Use additional info from unmodified compilers
  - Symbols, to distinguish code and data (no -s)
  - Relocations, to find all code pointers (--emit-relocs)

Code pointer, or integer?

```
.section .rodata:
    .quad  0x400620

.section .text:
    mov    $0x400620, %rax
```

```
.section .rodata:
    .quad  4195872

.section .text:
    mov    $4195872, %rax
```

Relocations (meta-data)

Augmented Binary Analysis

- Use additional info from unmodified compilers
  - Symbols, to distinguish code and data (no -s)
  - Relocations, to find all code pointers (--emit-relocs)
    - ask linker to preserve relocations

Code pointer, or integer?

```
.section .rodata:
    .quad  0x400620

.section .text:
    mov    $0x400620, %rax
```

```
.section .rodata:
    .quad  4195872

.section .text:
    mov    $4195872, %rax
```

Relocations (meta-data)

Augmented Binary Analysis

- Allows accurate and complete disassembly

Augmented Binary Analysis

- Allows accurate and complete disassembly
- Many special cases, but we handle them

Issue	Description	How to handle
Missing symbol sizes	Internal GCC functions have a symbol size of zero.	Hard-code sizes; `_start` is 42 bytes.
Fall-through symbols	Functions implicitly fall through to the following function.	Attach a copy of the following code.
Overlapping symbols	Some functions are a strict subset of an enclosing function.	Binary search for targets very carefully.
Symbol aliases	Symbol tables have many names for the same function.	Pick one representative name.
Ambiguous names	One LOCAL name, multiple versions (`bsloww` in libm).	Look up address resolved by the loader.
Pointers to static functions	For pointers to functions within the same module, the offset is known, and object files contain no relevant relocations.	Determine if `lea` instructions target a known symbol (not completely sound).
`noreturn` function calls	GCC always generates a NOP after calls to `noreturn` functions like `longjmp`, but omits unwind information.	Detect when at a NOP following a call and use unwind info from at the call.
COPY relocations	Object initialized in one library, then `memcpy`'d to another.	Track data symbols, not just code.
IFUNC symbols	Return pointer to actual function to call (cached in PLT).	Statically evaluate from `lea` refs.
Conditional tail recursion	Does not appear in normal GCC-generated code. Used in hand-coded assembly by glibc (`lowlevellock.h`).	Can do XOR'ing both before and after, works whether or not the jump is taken.
Indirect tail rec.	Difficult to tell apart from jump-table jumps.	Use a function epilogue heuristic.
Finding jump tables	Jump tables are not clearly delineated.	See the text for a discussion on this.

Where to Re-Randomize From

- Most defenses operate at higher privilege level
  - i.e. kernel, hypervisor, hardware
  - Or else declare their own code “trusted”

Where to Re-Randomize From

- Most defenses operate at higher privilege level
  - i.e. kernel, hypervisor, hardware
  - Or else declare their own code “trusted”
- Shuffler is *egalitarian*
  - Same level of privilege, no system modifications
  - Defends itself from attack

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers

memcpy's code

```
mov     0x400620(,%rax,8),%rax
jmpq    *%rax


0x400620:  0x400508  0x400514
0x400630:  0x400520  0x40052c
0x400640:  0x400538  0x400544
```

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers

`memcpy`'s code

```
mov     0x400620(,%rax,8),%rax
jmpq    *%rax


0x400620:  0x400508  0x400514
0x400630:  0x400520  0x40052c
0x400640:  0x400538  0x400544
```

Rewrite `main`, `printf`, `...`, `memcpy`, `...`

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers

memcpy's code

```
mov     0x400620(,%rax,8),%rax
jmpq    *%rax
```

New memcpy code

```
mov     0x400620(,%rax,8),%rax
jmpq    *%gs:(%rax)
```

```
0x400620:    0x20     0x28
0x400630:    0x30     0x88
0x400640:    0x40     0x48
```

Rewrite `main`, `printf`, `...`, `memcpy`, `...`

Invalidates `memcpy` jump table

But rewrite process uses (old) `memcpy`

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. `memcpy` uses code pointers

`memcpy`'s code

```
mov     0x400620(,%rax,8),%rax
jmpq    *%rax
```

New `memcpy` code

```
mov     0x400620(,%rax,8),%rax
jmpq    *%gs:(%rax)
```

```
0x400620:    0x20    0x28
0x400630:    0x30    0x88
0x400640:    0x40    0x48
```

??

Rewrite `main`, `printf`, `. . .`, `memcpy`.

Invalidates `memcpy` jump table

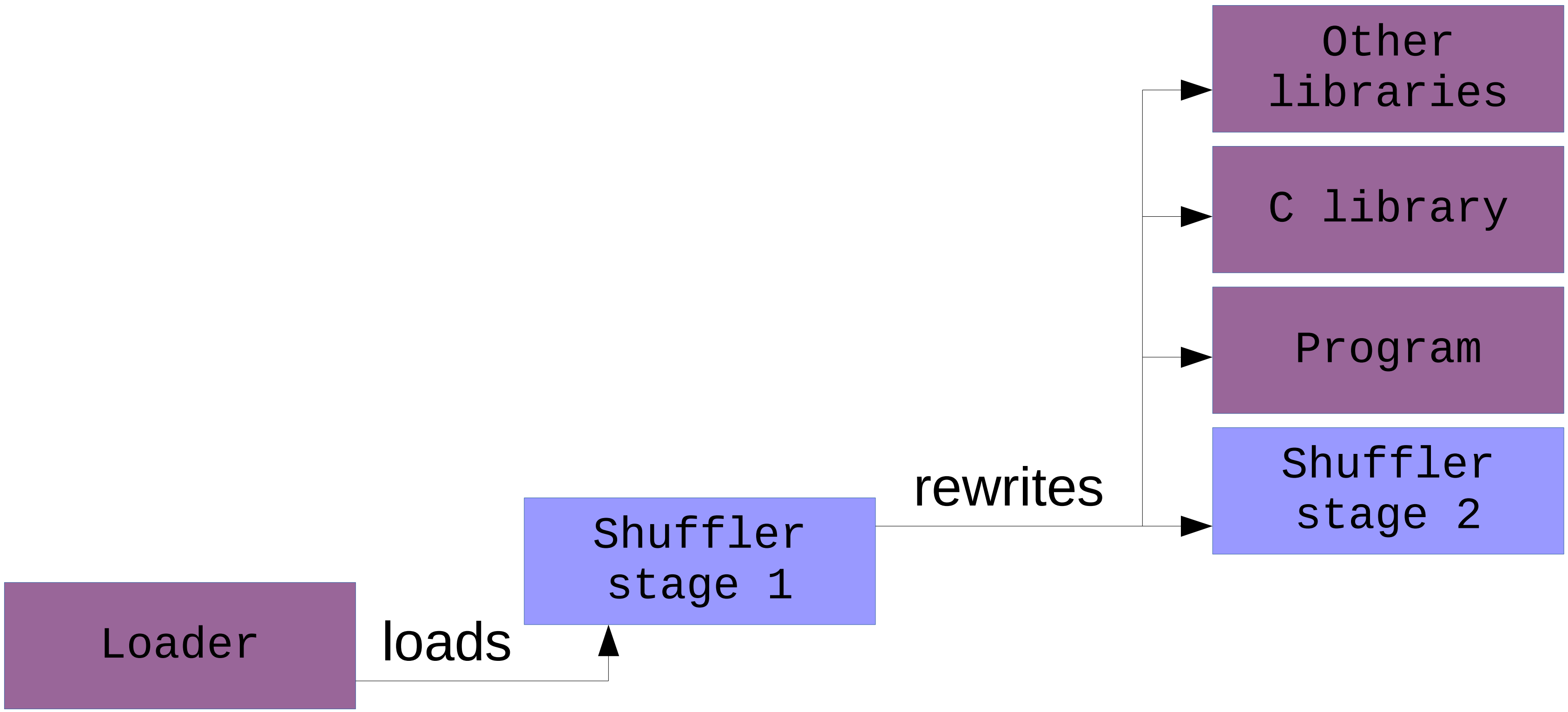
But rewrite process uses (old) `memcpy`

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler

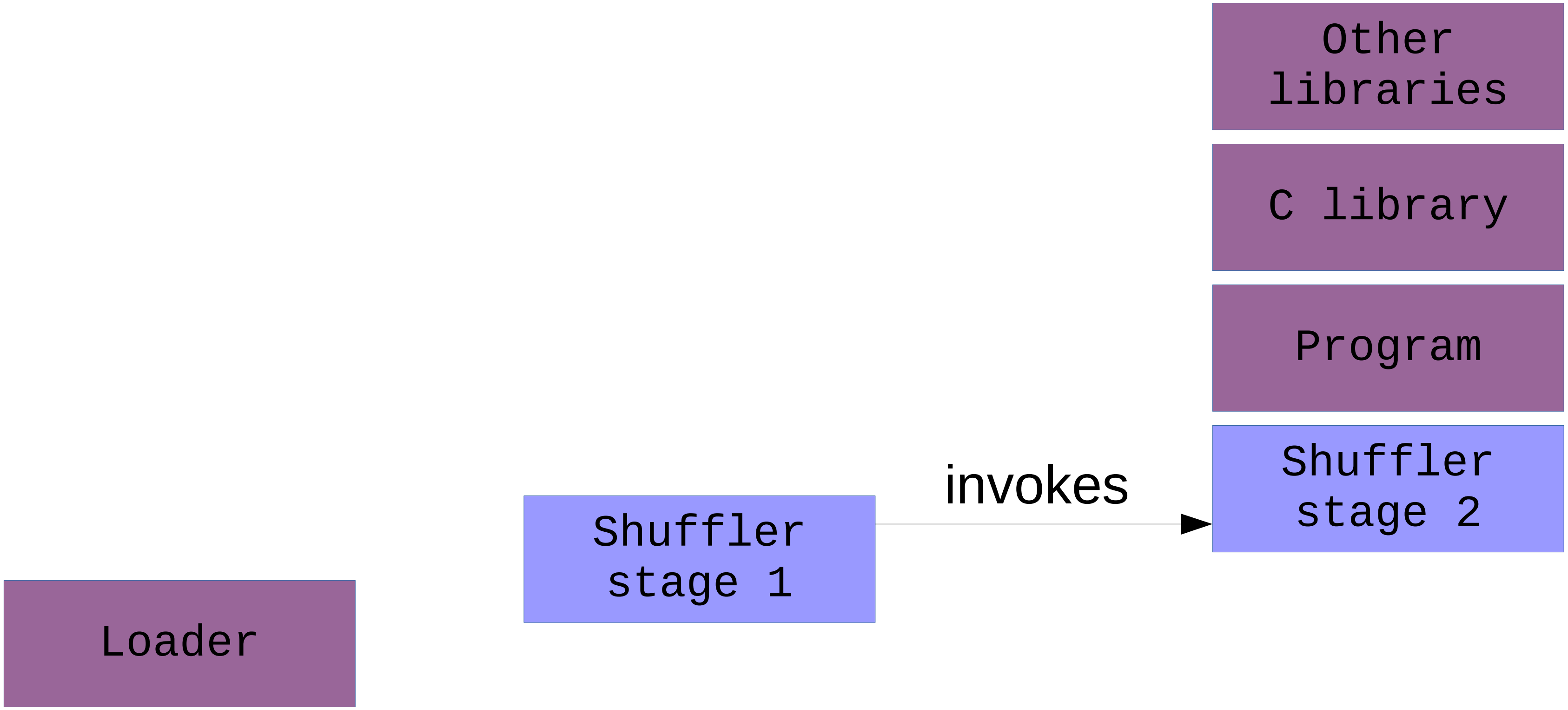
Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler



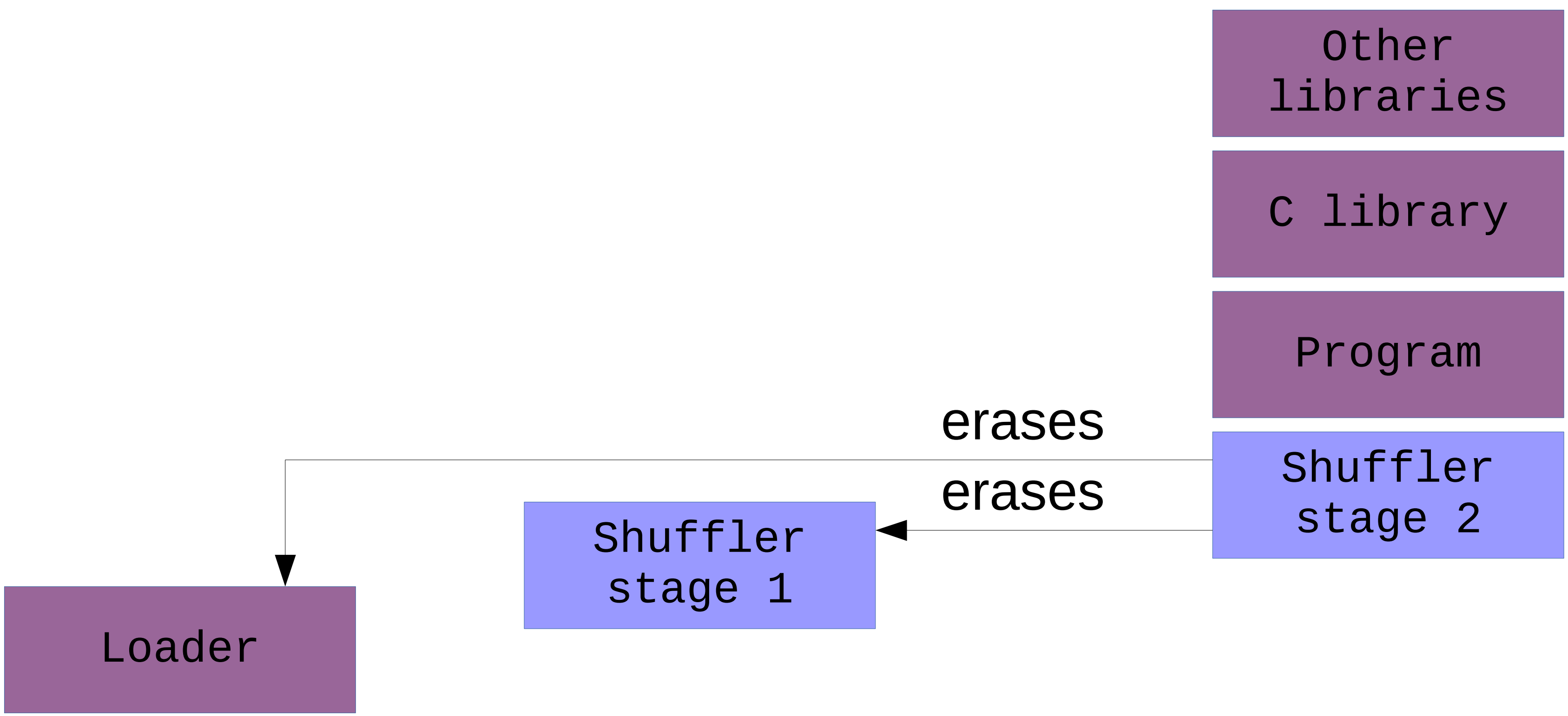
Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler



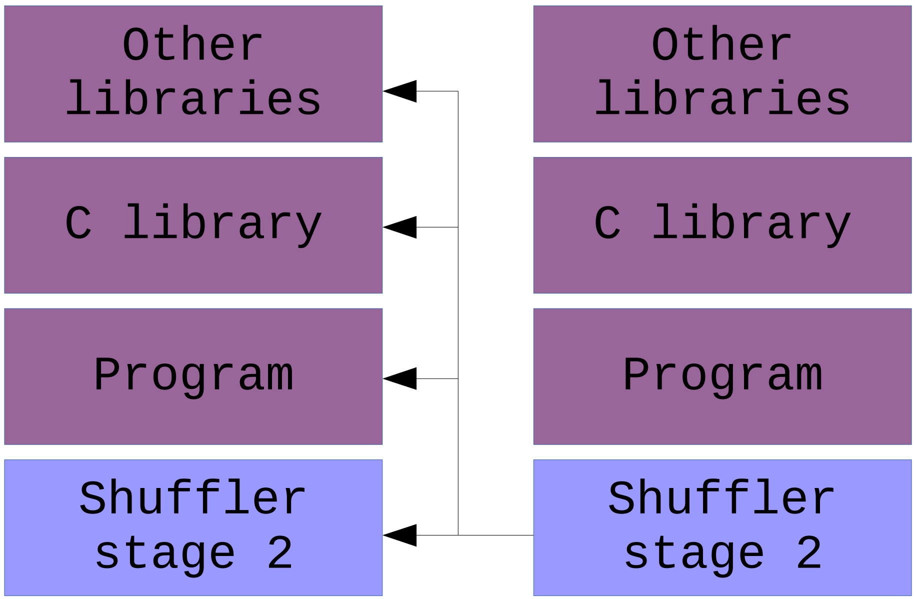
Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler



Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler
  - Make new copies

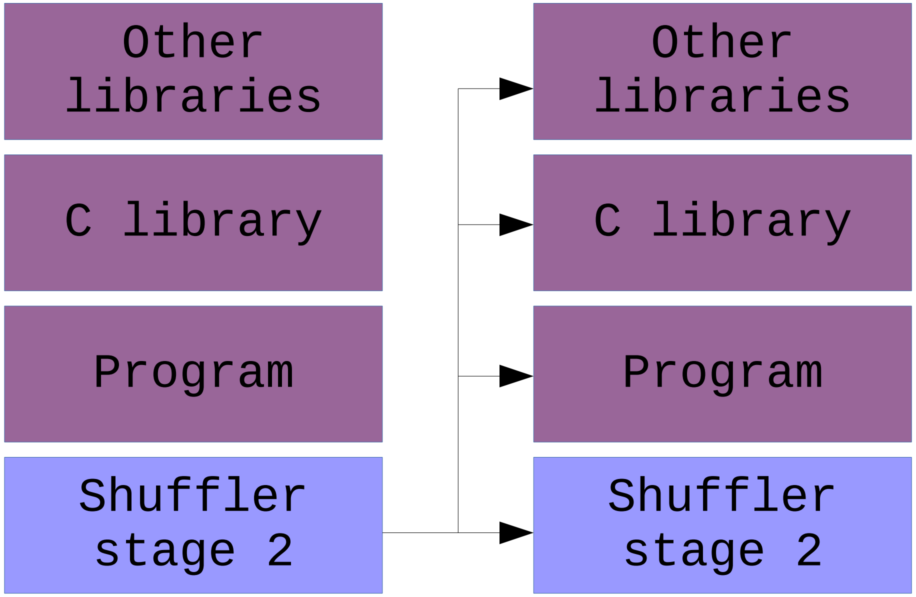


Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler
  - Make new copies

Egalitarian Bootstrapping

- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler
  - Make new copies



Egalitarian Bootstrapping

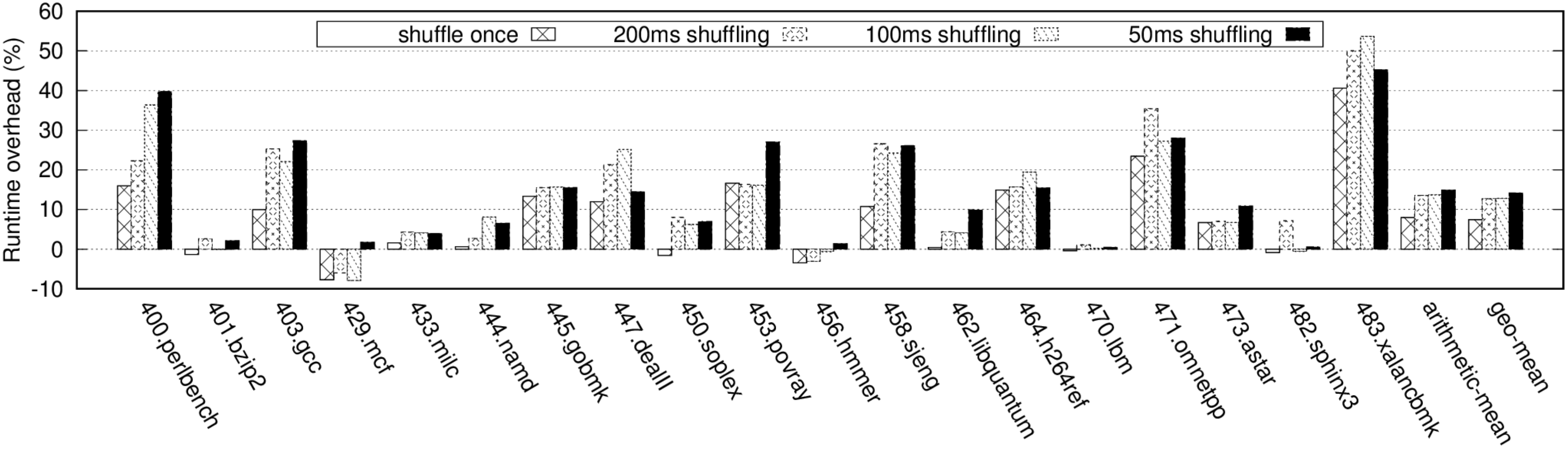
- Problem: transformations break original code
  - e.g. memcpy uses code pointers
- Solution: use two copies of Shuffler
  - Make new copies

Outline

1. Continuous re-randomization
2. Accelerating our randomization
3. Binary analysis and egalitarianism
4. Results and Demo

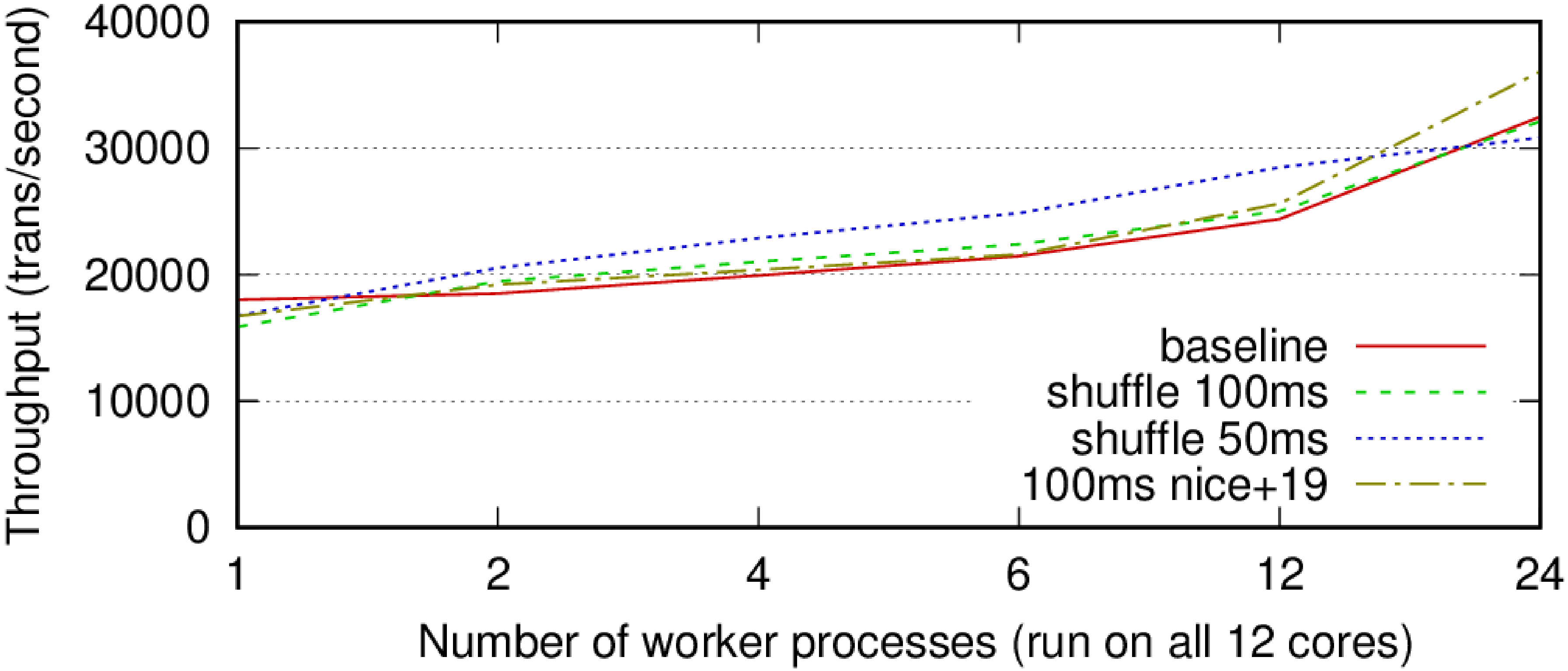
Performance Evaluation

- SPEC CPU overhead at 50ms = 14.9%



Performance Evaluation

- SPEC CPU overhead at 50ms = 14.9%
- Multiprocess Nginx up to 24 workers



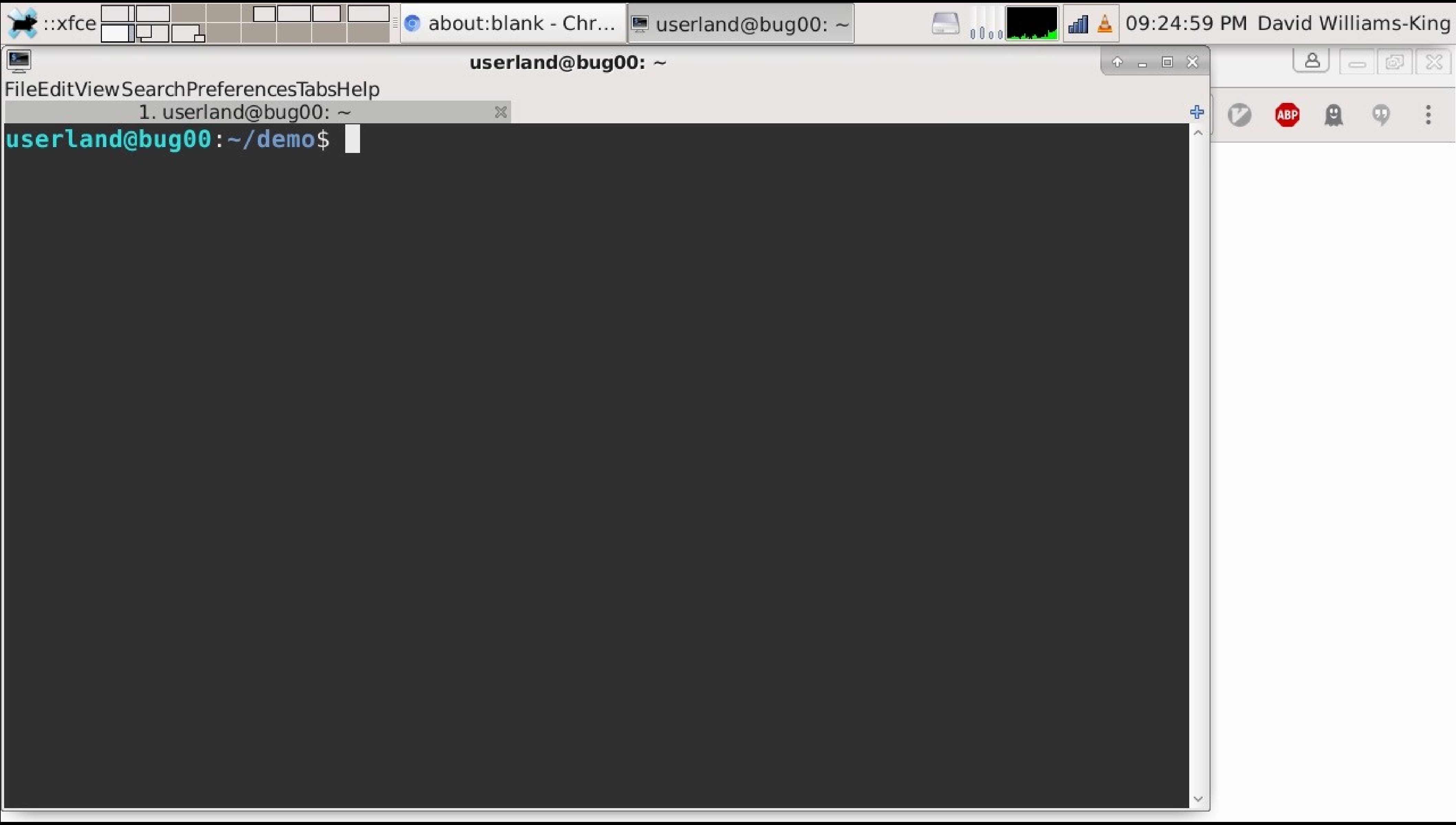
Security Evaluation

- Two disclosure-based attack methodologies:
  - Scan many pages for the desired gadgets
    - impacted by disclosure time, network latency
  - Explore gadget space in small number of pages
    - impacted by ROP chain computation time (> 40 seconds)

Security Evaluation

- Two disclosure-based attack methodologies:
  - Scan many pages for the desired gadgets
    - impacted by disclosure time, network latency
  - Explore gadget space in small number of pages
    - impacted by ROP chain computation time (> 40 seconds)
- Published JIT-ROP takes 2300-378000 ms
- We can re-randomize typically every 20-50 ms

Demo



::xfce
about:blank - Chr...
userland@bug00: ~
09:24:59 PM
David Williams-King
userland@bug00: ~
FileEditViewSearchPreferencesTabsHelp
1. userland@bug00: ~
ABP
userland@bug00:~/demo$

Conclusion

- Continuous re-randomization every 20-50 ms

Conclusion

- Continuous re-randomization every 20-50 ms
- Fast:
  - Defeats all known code reuse attacks
  - Asynchronous shuffling offloads overhead
- Deployable:
  - Binary analysis w/o modifying kernel, compiler, ...
- Egalitarian:
  - No additional privileges required
  - Shuffler defends its own code

Questions?

Demo website: `http://shuffled.elfery.net:8000`

Related Work

- JIT-ROP, SOSP 2013
- Oxymoron, Usenix Sec 2014
- Code Pointer Integrity, OSDI 2014
- Stabilizer, SIGARCH 2013
- Remix, CODASPY 2016
- TASR, CCS 2015
- ...more related work in our paper

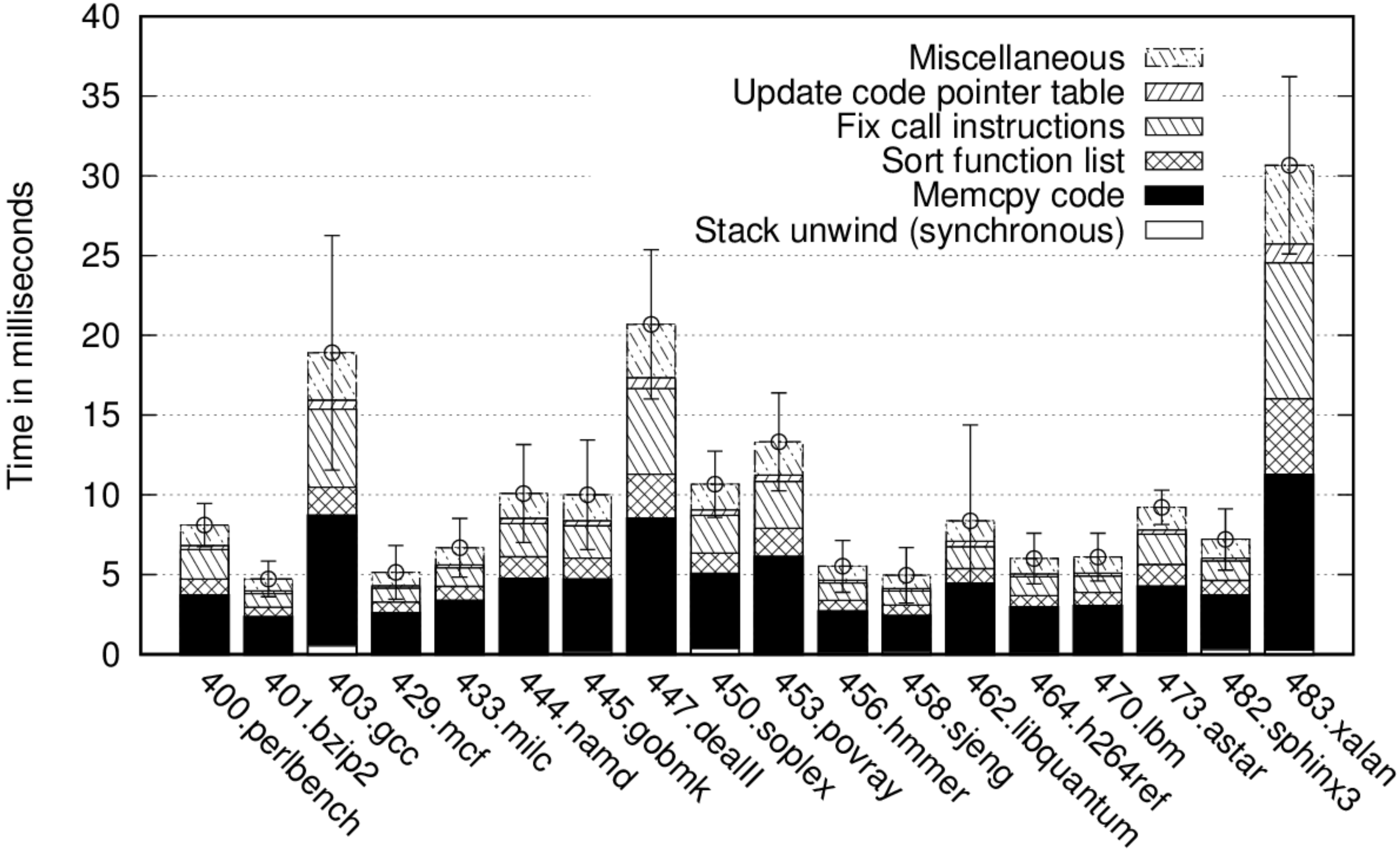
[1] https://securityintelligence.com/anti-rop-a-moving-target-defense/
[2] http://www.ieee-security.org/TC/SP2013/papers/4977a574.pdf

Future Work

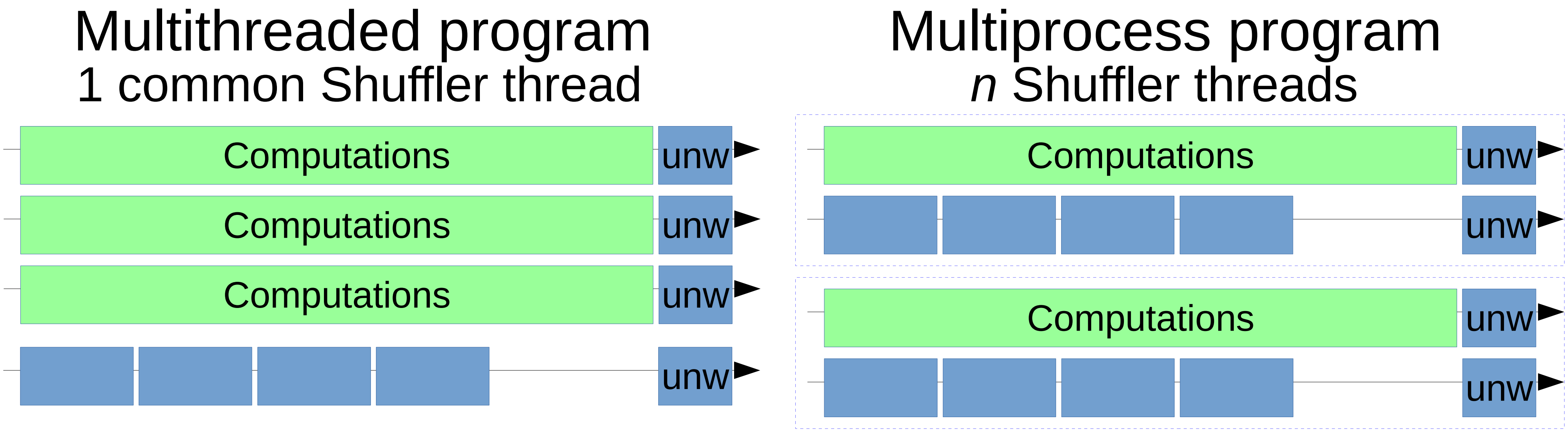
- Translating stack unwind information
  – Breaks C++ exceptions, pthread_cancel, etc.
- Cannot shuffle the loader currently
  – Breaks dlopen
- If shuffling takes too long, no mechanism to pause target program

Shuffler Thread Performance

- Asynchronous shuffling runs quickly
- Synchronous runtime is 0.3% of total runtime



Scalability



- Tradeoff for server workers
  - Multithreaded => better performance overhead
  - Multiprocess => no disclosures across workers
- Both techniques scale well in practice (up to 24x)