Software Security 1

Administrative

Kevin Borgolte kevin.borgolte@rub.de

Tentative Lecture Schedule / Deadlines

Lectures
Wednesday 10-12

1. Oct 9 – Lecture
2. Oct 16 – Flipped classroom
3. Oct 23 – Lecture ← First assignment due
4. Oct 30 – Flipped classroom
5. Nov 6 – Lecture ← Second assignment due
6. Nov 13 – Flipped classroom
7. Nov 20 – Lecture ← Third assignment due
8. Nov 27 – Flipped classroom
9. Dec 4 – Lecture ← Fourth assignment due
10. Dec 11 – Flipped classroom
11. Dec 18 – Guest? Lecture ← Fifth assignment due
12. Jan 8 – Lecture
13. Jan 15 – Flipped classroom
14. Jan 22 – Lecture ← Sixth assignment due
15. Jan 29 – Flipped classroom

Tentative

(will probably change)

Assignments

- Questions
  - Moodle or email us (softsec+teaching@rub.de)
- Assignment 2
  - 7 tasks (nuggets, dropped, coalmine, echo, echo2, over9000, peeky-blinders)
    - There seem to be some struggles with format strings?
  - Due: Midnight this evening! (November 7th, 0:00 Bochum time)
- Assignment 3
  - 4–5 tasks
    - In hindsight, 7 challenges seems too much, so next assignments will have fewer
  - Due: November 21st, 0:00 Bochum time

Exercise and Exam Room

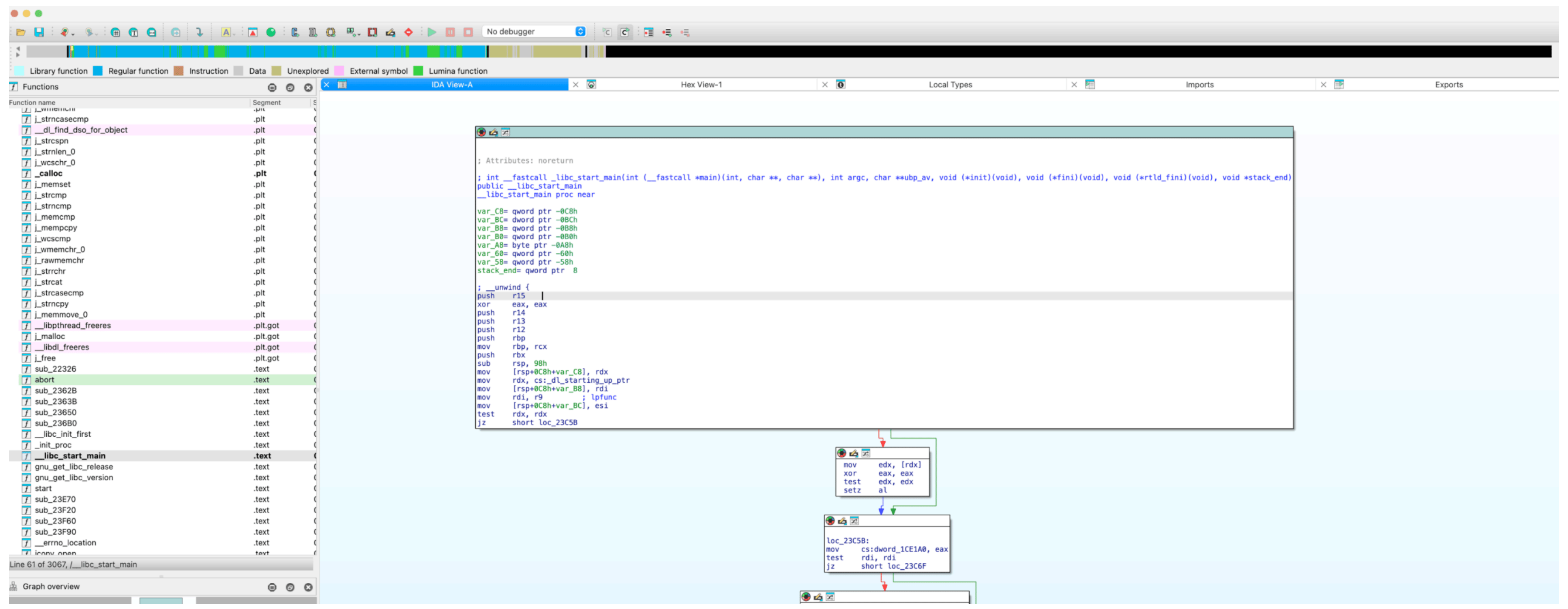
- MC 5/222 machines are now <u>finally</u> ready for us to set up
  - Will take a bit more time on our end, hopefully done week of Nov 18th
- Setup will be
  - Live image (we also be available to be run as a virtual machine)
  - Local public scratch space (wiped manually, but regularly)
    - Will survive a reboot
    - Won't be there tomorrow or after a few hours

Exam Setup

- Exam setup will be the same as MC 5/222, but without Internet
  - We will include any resources that you request and that we deem reasonable (GPT@RUB is not reasonable)
    - Assembly references? syscall tables? Lecture slides? Snippets of code? etc.
  - We will <u>try</u> to allow some (simple) dotfiles (this is difficult, TBD)
- Please use the image extensively and help us identify issues
  - There will be a dedicated test day in late January/early February with the final exam set up, exact date TBD
- Exam dates: Two days of 24.02–07.03
  - Some possible conflicts with "block courses", we know, complete Moodle poll!

Hex Rays IDA Classroom Free

- So far, you got source code and binary executable
- For some future challenges, you will not get source code
  - You will need to analyze the disassembly or decompile it
- There are a variety of open source tools you can use, but Hex Rays also has given us ~50 licenses for IDA Classroom Free
  - They are named 1 year licenses and need to be requested individually
    - You need to have solved some assignments to request a license
  - We need to record who uses which license and you need to take care of not leaking your license (we may have to report it)



Topics Today

- Type Safety
- Control Flow Integrity
- Fundamentals of Data-only Attacks

Software Security 1

Type Safety

Kevin Borgolte kevin.borgolte@rub.de

Memory and Type Safety

- So far, we have seen mostly memory unsafe examples
  - Buffer overflows, (random) pointer derefs, uninitialized memory, etc.
- **Memory safety** adds checks to prevent these memory issues
  - For example, other languages, like Java and Python, throw a runtime error when you access memory that is out of bounds
- **Type safety** complements memory safety
  - Ensures that the operations you can do are well-defined

Types?

- Primitive data types (int, unsigned long, etc.),
- Composite types
  - structs/unions
  - classes
  - arrays
- More complex types (e.g., functions, curried functions, etc.)
- And generally any term (construct) of the programming language

Types

- Rooted in *type systems* and *type theory*
- Simplified:
  - A formal logical system that describes what you can do (in a programming language) (and where you can prove some properties)
- Less simple:
  - "Homotopy type theory is a new branch of mathematics" or "it is (among other things) a foundational language for mathematics, i.e., an alternative to Zermelo–Fraenkel set theory"
  - Type theory goes *way* beyond the scope of this course, but it is something you need to keep in the back of your mind

Type Safety

- The operations you can do are well-defined
  - A programming language is type safe if all programs are well-defined
    - A program is well-defined if no execution of it can exhibit undefined behavior (for all inputs and environments)
  - Well-defined depends on the formal definition of the language
    - 123 + "foobar" is OK in JavaScript because of implicit conversions (and it generally has other questionable conversions), but treating an integer as a function fails with a type error
- It helps to ensure program correctness
  - But it is undecidable (halting problem)

Undefined Behavior?

```c
#include <limits.h>
#include <stdio.h>

int main(void) {
  printf("%d\n", (INT_MAX+1) < 0);
  return EXIT_SUCCESS;
}
```

- Typical examples are
  - Out of bounds access (buffer overflow/read)
  - Integer overflow
- Undefined behavior at <u>any point</u> in a program's execution means that the <u>entire execution has no meaning</u>
  - Important: The execution is also not meaningful until the undefined behavior, but it is entirely meaningless
- Purpose: compiler's job gets easier
- Dependent on the language
  - Safe Rust has no undefined behavior, but once you use "unsafe" in Rust, you might

Undefined Behavior

```
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

Undefined Behavior

```
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

- f1 has no meaning whatsoever

Undefined Behavior

```
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

- f1 has no meaning whatsoever
  - y is not initialized but used

Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

- f1 has no meaning whatsoever
  - y is not initialized but used
- f2 has meaning for some y and z

Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

- f1 has no meaning whatsoever
  - y is not initialized but used
- f2 has meaning for some y and z
  - y = 0 is undefined

Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

- f1 has no meaning whatsoever
  - y is not initialized but used
- f2 has meaning for some y and z
  - y = 0 is undefined
  - y = -1 and z = INT_MIN is also undefined

Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f1(int z) {
  int y;
  printf("%d\n", z/y);
}

void f2(int y, int z) {
  printf("%d\n", z/y);
}
```

- f1 has no meaning whatsoever
  - y is not initialized but used
- f2 has meaning for some y and z
  - y = 0 is undefined
  - y = -1 and z = INT_MIN is also undefined
  - In those two cases, the compiler can do whatever it wants and omit them from consideration for its optimizations

No Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f3(int y, int z) {
  if(y == 0) {
    printf("div by 0\n");
    return;
  }

  if(y == -1 && z == INT_MIN) {
    printf("out of range\n");
    return;
  }

  printf("%d\n", z/y);
}
```

- f3 is well defined for all inputs
  - y = 0 return early
  - y = -1 and z = INT_MIN return early
  - in all other cases z/y is well-defined
- The two conditions are called pre-conditions to z/y

More Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f4(void) {
  char *s = "foobar";
  s[0] = 'F';
}

void f5(void) {
  int *x = NULL;
  int y = *x;
}

int f6(void) {
  int a = 0;
  int b = 0;
  return &a < & b;
}
```

- f4 is perfectly valid C code, but is actually undefined behavior for some C++ versions
  - "The effect of attempting to modify a string literal is undefined."
- f5 is undefined behavior because a null pointer was dereferenced
- f6 is undefined because the less than and greater than comparisons for pointers are only defined for members of the same element

And Even More Undefined Behavior

```c
#include <limits.h>
#include <stdio.h>

void f7(void) {
  int y = 0;
  int z = y++ + ++y;
}

void f8(void) {
  int a[] = {1,2,3};
  int i = 0;
  a[i] = i++;
  printf("%d++ = %d\n", i, ++i);
}

void f9(void) {
  int y = 1 << -1;
  int32_t z = 1 << 32;
}
```

- f7 is undefined behavior because we are modifying y more than once
- f8 is undefined behavior because we are modifying i and also accessing it
- f9 is undefined behavior for two reasons
  - negative number shift has no meaning
  - shifting by the width of the integer also has no meaning
- Many more variants of undefined behavior exist, the examples are not exhaustive!

Undefined Behavior in Practice

```
static void __devexit foobar_pci_remove (struct pci_dev *pdev) {
  struct foobar_hw *dev = pci_get_drvdata(pdev);
  struct foobar_priv *priv = dev->priv;

  if (!dev) {
    return;
  }
  /* ... do stuff using dev ... */
}
```

- What is the problem with this code snippet?

Undefined Behavior in Practice

```
static void __devexit foobar_pci_remove (struct pci_dev *pdev) {
  struct foobar_hw *dev = pci_get_drvdata(pdev);
  struct foobar_priv *priv = dev->priv;

  if (!dev) {
    return;
  }
  /* ... do stuff using dev ... */
}
```

- What is the problem with this code snippet?
  - The null pointer check is considered dead code by the compiler!
    If dev is null, then the dereference dev->priv is undefined behavior, and that means the compiler does not need to consider the case that dev is ever null. But then, the comparison is always false, so it can remove it, creating a potential vulnerability

Unaligned Return Addresses

During a routine refactoring, code that once read

```
aligned_tramp_ret = tramp_ret & ~(nap->align_boundary - 1);
```

was changed to read

```
return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);
```

Besides the variable renames (which were intentional and correct), a shift was introduced, treating nap->align_boundary as the log2 of bundle size.

We didn't notice this because NaCl on x86 uses a 32-byte bundle size. On x86 with gcc, (1 << 32) == 1. (I believe the standard leaves this behavior undefined, but I'm rusty.) Thus, the entire sandboxing sequence became a no-op.

This change had four listed reviewers and was explicitly LGTM'd by two. Nobody appears to have noticed the change.

This happened in Google's Sandbox for native code in Google Chrome.

"Issue 245: NaCl/x86 appears to leave return addresses unaligned when returning through the springboard.", https://bugs.chromium.org/p/nativeclient/issues/detail?id=245

Unaligned Return Addresses

During a routine refactoring, code that once read

```
aligned_tramp_ret = tramp_ret & ~(nap->align_boundary - 1);
```

was changed to read

```
return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);
```

Besides the variable renames (which were intentional and correct), a shift was introduced, treating nap->align_boundary as the log2 of bundle size.

We didn't notice this because NaCl on x86 uses a 32-byte bundle size. On x86 with gcc, (1 << 32) == 1. (I believe the standard leaves this behavior undefined, but I'm rusty.) Thus, the entire sandboxing sequence became a no-op.

This change had four listed reviewers and was explicitly LGTM'd by two. Nobody appears to have noticed the change.

This happened in Google's Sandbox for native code in Google Chrome.

Impact

There is a potential for untrusted code on 32-bit x86 to unalign its instruction stream by constructing a return address and making a syscall. This could subvert the validator. A similar vulnerability may affect x86-64.

"Issue 245: NaCl/x86 appears to leave return addresses unaligned when returning through the springboard.", https://bugs.chromium.org/p/nativeclient/issues/detail?id=245

Undefined Behavior and Type Safety

- Undefined behavior can be useful, but it also often leads to bad things because it is difficult to <u>fully</u> understand
- If the program is <u>truly type safe</u>, then all executions are well-defined and there is <u>no undefined behavior</u>
  - Important: well-defined does <u>not</u> mean that there are no bugs that an attacker can misuse, it just means <u>internally consistent</u>
- Type safety is not trivial to do
  - Static vs. dynamic type checking
  - Strong vs. weak typing

Static Type Checking

- At compile-time, check if the types are "compatible"
  - C uses static type checking (but it is not type safe)
  - Python has not static type checking by default, but supports it via PEP 484
  - Also possible if you do not declare types directly (type inference also allows statically typed languages that do not require explicit typing, think auto in C++)
- Practically (for Turing-complete languages), static type checking is somewhat problematic (basically, pick two out of three):
  - Soundness (detects all incorrect programs statically)
  - Decidability (a program can decide whether a program is well typed)
  - Completeness (no correct programs are claimed to be incorrect)

```
#include <stdio.h>

int main(void) {
    void *x = NULL;
    char *y = "foobar";
    printf("%s\n", x + y);
}

error: invalid operands to binary
       expression ('void *' and 'char *')
    printf("%s\n", x + y);
                   ~ ^ ~
1 error generated.
```

Dynamic Type Checking

- At runtime, verify type safety by carrying information about the type of data (type tag)
  - Introduces performance overhead
  - Can sometimes be disabled by the compiler to regain the overhead at the expense of possibly having operations that are not well-defined (or disabling language features)
  - For C++, this is called runtime type information (RTTI)
- Type information can also be used for other language features, including dynamic dispatch (OOP), reflection, etc.
- Operations on types that are invalid and detected dynamically lead to the programs being stopped with an exception/error

```
Python Example:

>>> x = 1
>>> y = "foobar"
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s)
   for +: 'int' and 'str'
```

Practical Type Safety

- Trade-off question between static and dynamic type checking
  - Static type checking to detect type errors early on and to help the compiler/JIT/interpreter to optimize your code
  - Dynamic type checking to detect type errors that the compiler might not be able to detect because they were too complicated to check for or unsupported (and either accepts or rejects as valid code)
- Combination of both possible
  - Advantage: Safety benefits of both
  - Disadvantage: Performance cost of both

Weak vs. Strong Typing

- Be careful about terminology
  - Weak/strong typing sometimes means dynamic/static typing
- For us, weak/strong concerns itself with <u>implicit</u> conversion
  - Strongly typed: no attempt to find a type that fits the operation
  - Weakly typed: will try to find a compatible type for the operation

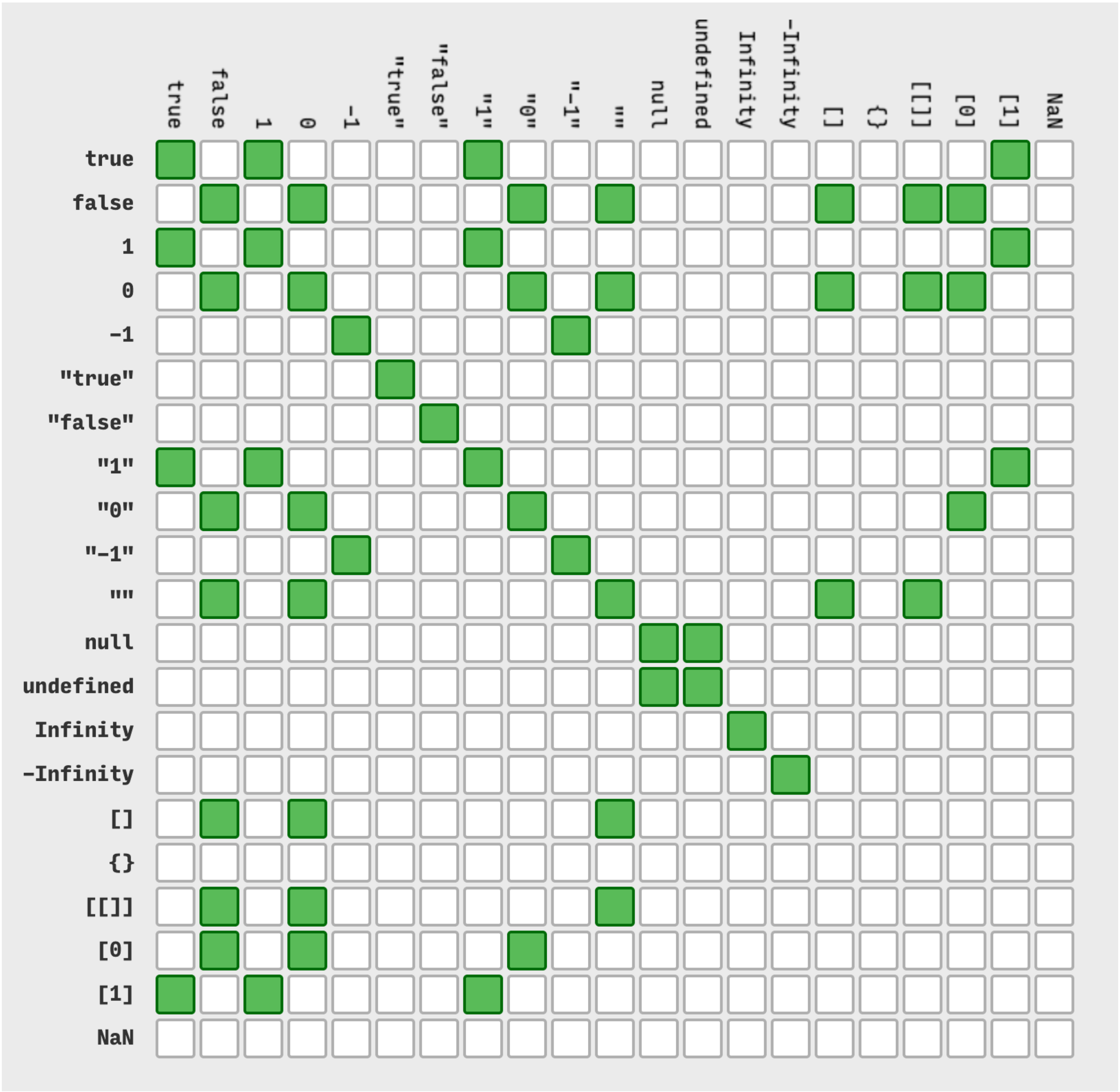
```
Python Example (strongly typed):

>>> x = 1
>>> y = "foobar"
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s)
   for +: 'int' and 'str'
```

```
JavaScript Example (weakly typed)

> x = 1
1
> y = "foobar"
"foobar"
> x + y
"1foobar"
```

Issues and Why It Matters

- Type confusion vulnerabilities occur because developers do not understand the type guarantees of their programming language correctly
- JavaScript is a prime example
  - Strict and weak comparison operators because a lot of mistakes were made that led to security vulnerabilities

```javascript
let a = "test"
console.log(a == "test")
console.log(a === "test")

let b = ["test"]
console.log(b == "test")
console.log(b === "test")
```



https://dorey.github.io/JavaScript-Equality-Table/

Type Aliasing

- You can alias types for clarity, but it doesn't create new types
  - A function might take a Distance argument, but you could supply a variable of type Balance, and compilers would not warn you (and there is implicit conversion)

```
# C++
using Distance = int;
using Balance = int;


# Rust
type Distance = usize;
type Balance = usize;
```

Type Aliasing

- You can alias types for clarity, but it doesn't create new types
  - A function might take a Distance argument, but you could supply a variable of type Balance, and compilers would not warn you (and there is implicit conversion)
- An alternative are newtype idioms, which create a new type
  - Depending on the programming language, you can provide a Distance/Balance for a usize, but not a usize for Distance/Balance, or Balance for Distance etc.

```
# C++
using Distance = int;
using Balance = int;


# Rust
type Distance = usize;
type Balance = usize;


# Rust, newtype
struct Distance(usize);
struct Balance(usize);
```

Type Safety

- Type systems formalize and enforce programming constructs (and are not limited to the standard data types)
- Helps to develop correct software
- Active area of research in compilers, reliability, and security
  - e.g., algebraic data types let you statically type check eval functions

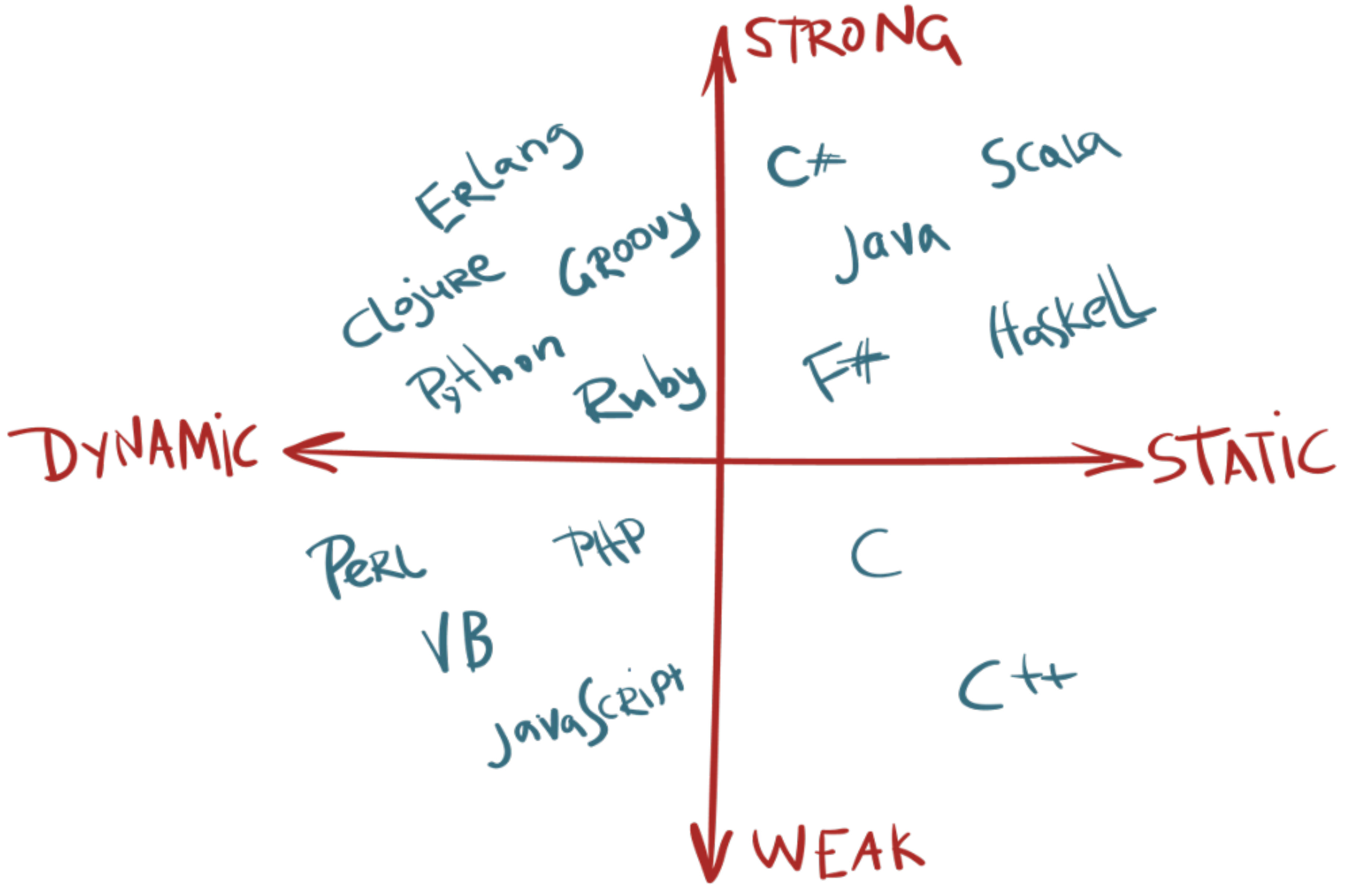


Figure by Mayank Bhatnagar

Software Security 1

Control Flow Integrity

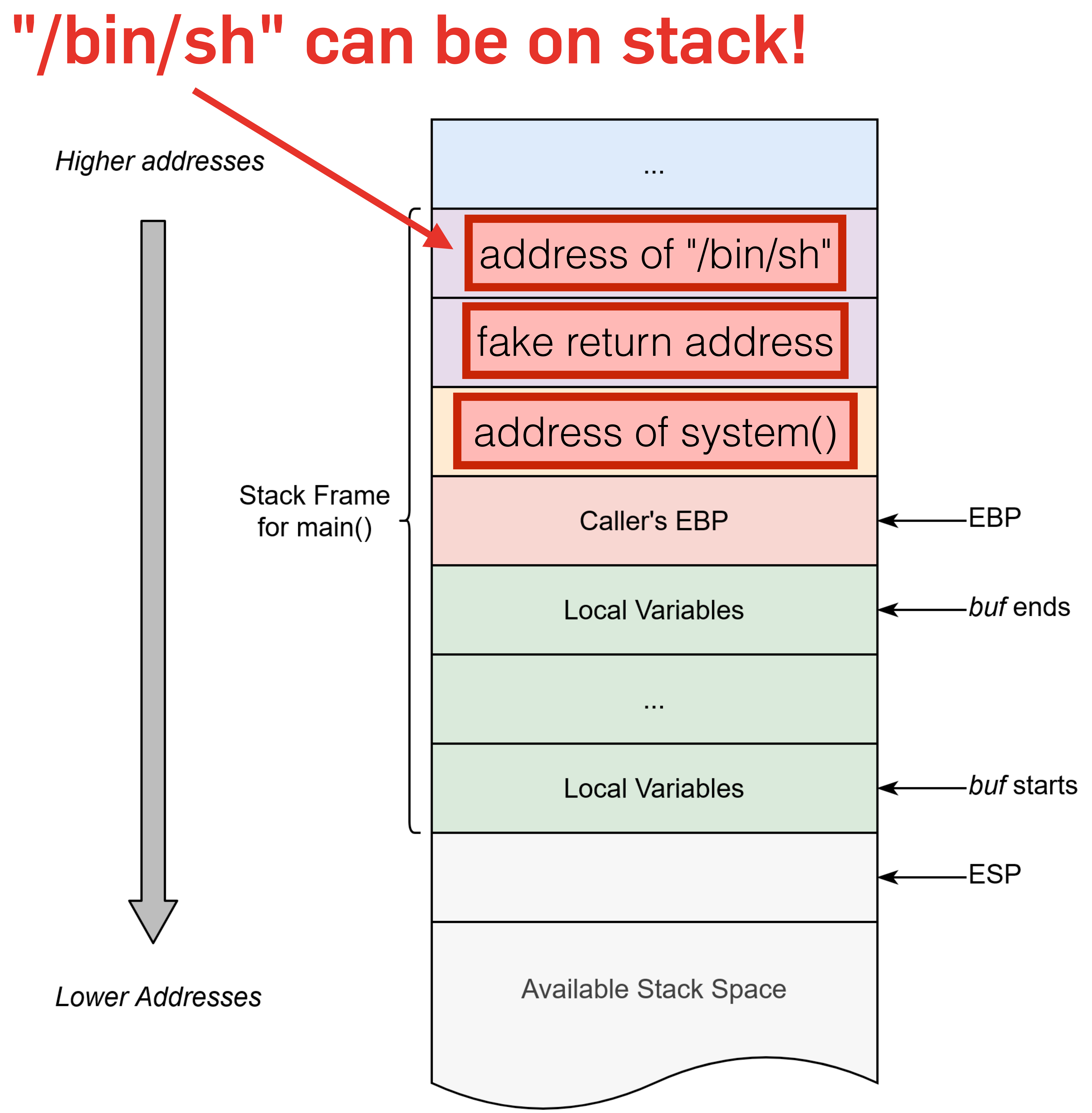
Kevin Borgolte kevin.borgolte@rub.de

Recap: Code Reuse

- ~~Code injection~~
  - ~~Basic idea: inject some new code into the application~~
  - Doesn't work anymore ☹️
- Instead: reuse code that is already there
  - return-to-libc
  - more generally, return oriented programming (ROP)
    - ROP chains

Recap: return-to-libc

- If we know the address of a function, say system(), we can misuse the vulnerable program to call it and get a shell
  - Overwrite return address with the address of system()
  - Add a fake return address to the stack from which we (supposedly) called system() from
  - Add the arguments to system() to the stack



Recap: Return-oriented Programming and ROP Chains

- ROP steps
  1. Write to the stack
  2. Overwrite the first return address
  3. Overwrite the second return address
  4. …
  5. Overwrite the n-th return address
- Not much different from writing shellcode
  - Different "instruction" set (ROP gadgets)
  - "Instructions" are just longer
  - "Instructions" have more (side-)effects
  - = Programming a weird machine, that is often accidentally Turing complete (fun read: https://beza1e1.tuxen.de/articles/accidentally_turing_complete.html)

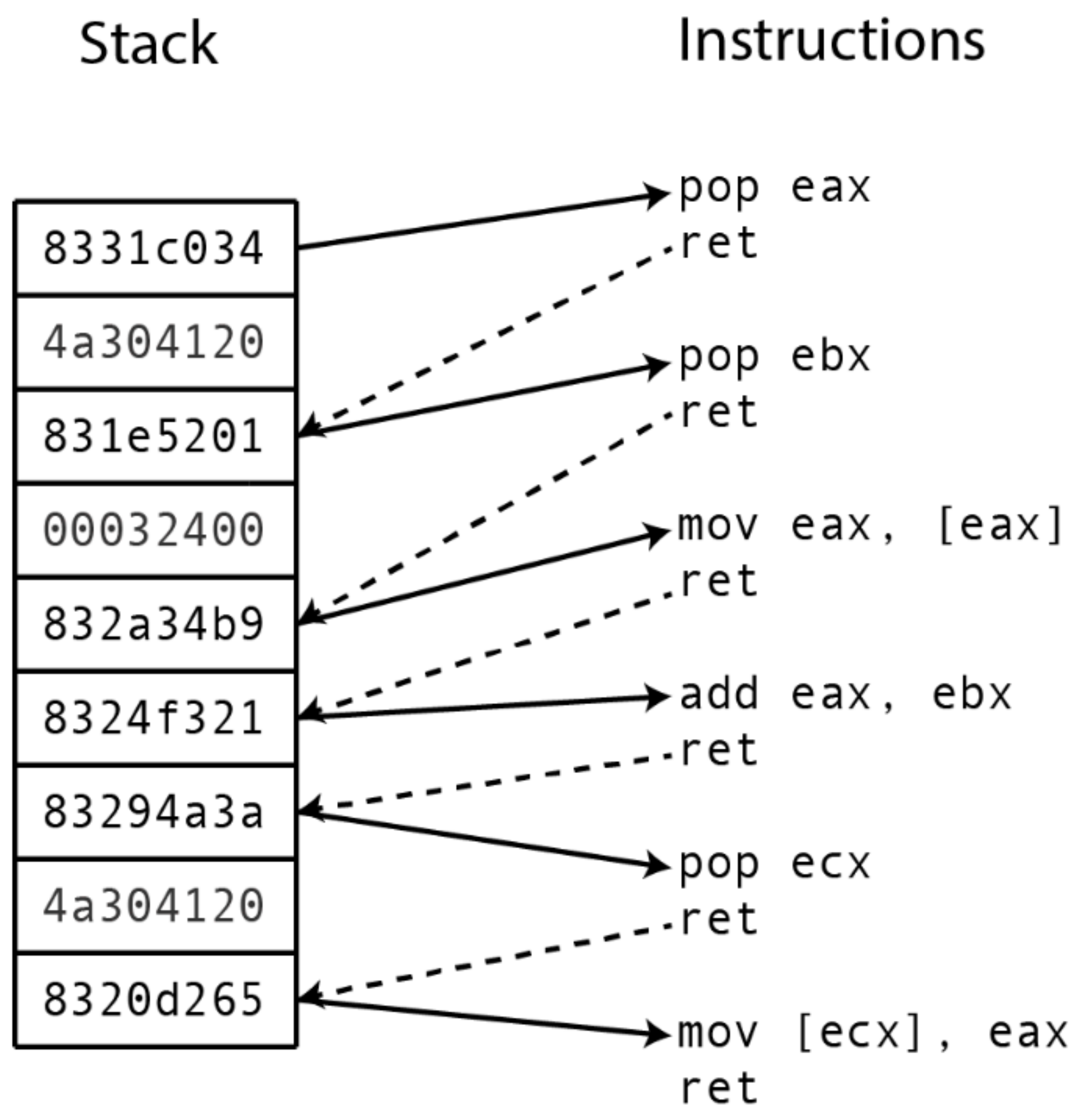


Image source: "ROP is Still Dangerous: Breaking Modern Defenses", https://people.eecs.berkeley.edu/~daw/papers/rop-usenix14.pdf

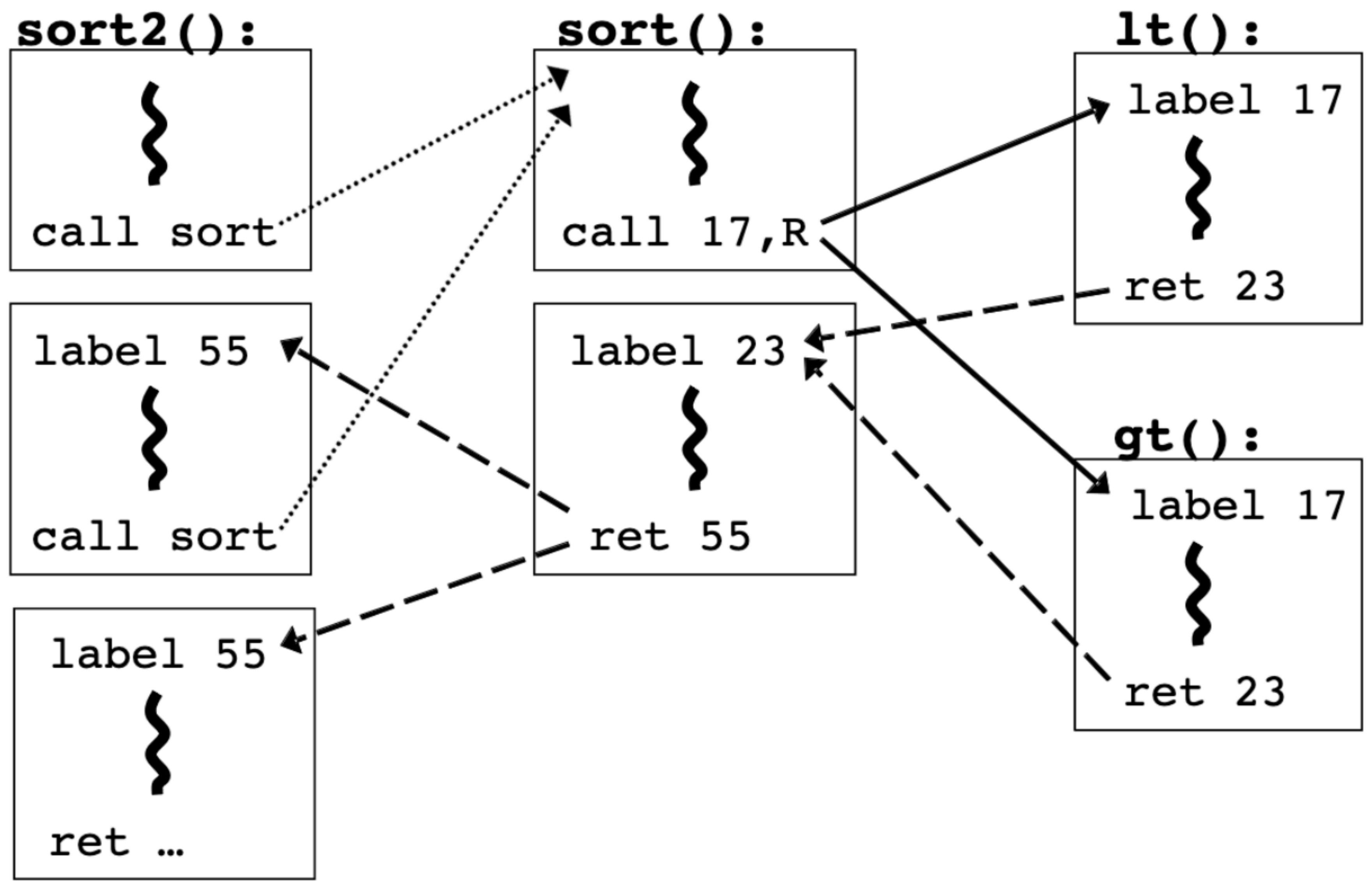
Recap: ROP Defenses

- Removing ROP gadgets (too onerous, slow, inefficient)
  - "G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries" (https://dl.acm.org/doi/abs/10.1145/1920261.1920269)
- Detecting ROP attacks in progress (bypassable):
  - "kBouncer: Efficient and Transparent ROP Mitigation" (https://people.csail.mit.edu/hes/ROP/Readings/kbouncer.pdf)
  - "ROPecker: A Generic and Pracical Approach for Defending Against ROP Attacks" (https://www.ndss-symposium.org/wp-content/uploads/2017/09/02_1_1.pdf)
- **Control Flow Integrity**

Control Flow Integrity

- Idea: Whenever a control flow transfer occurs that can be hijacked (e.g., return), check the target is one that the control is supposed to return to.

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Control Flow Integrity

- Put differently (and slightly weakened): During program execution, whenever an instruction transfers control, it targets a valid destination, as determined by a Control Flow Graph (CFG) created ahead of time.
  - Slightly weaker because a CFG might have multiple destinations, but only one destination is the supposed destination.
- CFI attacker is powerful
  - Arbitrary read-write without directly setting EIP/RIP or reserved registers
  - W∧X memory: Code is read-exec, data is read-write

Recap: Control Flow Graph

- Control Flow Graph
  - Describes a function/procedure
  - Directed graph of basic blocks
  - Usually intra-procedural
    - For CFI, sometimes inter-procedural!
- Basic Block
  - Maximal sequence of consecutive instructions with
    - Control flow entering only at the first instruction
    - Control flow only changing with the last instruction

```
__int64 __fastcall start(int a1, __int64 a2)
{
  unsigned int v2; // r14d
  int v3; // r15d
  char *v4; // r12
  char *v5; // rax
  const char *v6; // rdi
  __int64 v7; // r12
  const char *v8; // r14
  unsigned int v9; // eax
  int v10; // eax
  unsigned int v12; // eax

  v2 = 1;
  if ( a1 )
  {
    v3 = a1;
    v4 = *(char **)a2;
    v5 = strrchr(*(char **)a2, 47);
    v6 = v5 + 1;
    if ( !v5 )
      v6 = v4;
    if ( !strcmp(v6, "[") )
    {
      v7 = v3;
      if ( strcmp(*(const char **)(a2 + 8LL * v3 - 8), "]") )
        sub_1000010B5("missing ]", (char)"]");
      --v3;
      *(_QWORD *)(a2 + 8 * (v7 - 1)) = 0LL;
    }
    if ( v3 >= 2 )
    {
      setlocale(2, &byte_100001F12);
      dword_100003000 = v3 - 1;
      qword_100003008 = a2 + 8;
      dword_100003010 = 0;
      v8 = *(const char **)(a2 + 8);
      if ( v3 == 5 && !strcmp(v8, "!") )
      {
        dword_100003000 = 3;
        qword_100003008 = a2 + 16;
        v12 = sub_1000011D5(*(_QWORD *)(a2 + 16));
        v2 = sub_100001132(v12);
      }
      else
      {
        v9 = sub_1000011D5(v8);
        v2 = sub_100001132(v9) == 0;
      }
      v10 = dword_100003000--;
      if ( v10 >= 2 )
        sub_100001339(*(_QWORD *)qword_100003008, (char)"unexpected operator");
    }
  }
  return v2;
}
```

Control Flow Integrity

```
A:
  if(*func != nop IMM_1) exit
  call *func
  nop IMM_2
```

```
B:
  nop IMM_1
  [...]
  if(**rsp != nop IMM_2) exit
  ret
```

- nop_IMM1 and nop_IMM2 are our destination IDs
- Careful: Anything that starts with this bit pattern is a valid target/return, so we need to ensure it does not show up

Control Flow Integrity

- If A can also call C, then C needs to also start with nop_IMM1

```
A:
  [...]
  if(*func != nop IMM_1) exit
  call *func
  nop IMM_2
  [...]
```

```
B:
  nop IMM_1
  [...]
  if(**rsp != nop IMM_2) exit
  ret

C:
  nop IMM_1
  [...]
  if(**rsp != nop IMM_2) exit
  ret
```

Control Flow Integrity

- The same applies to the return (e.g., B returning to A or D)

```
A:
  [...]
  if(*func != nop IMM_1) exit
  call *func
  nop IMM_2
  [...]

D:
  [...]
  if(*func != nop IMM_1) exit
  call *func
  nop IMM_2
  [...]
```

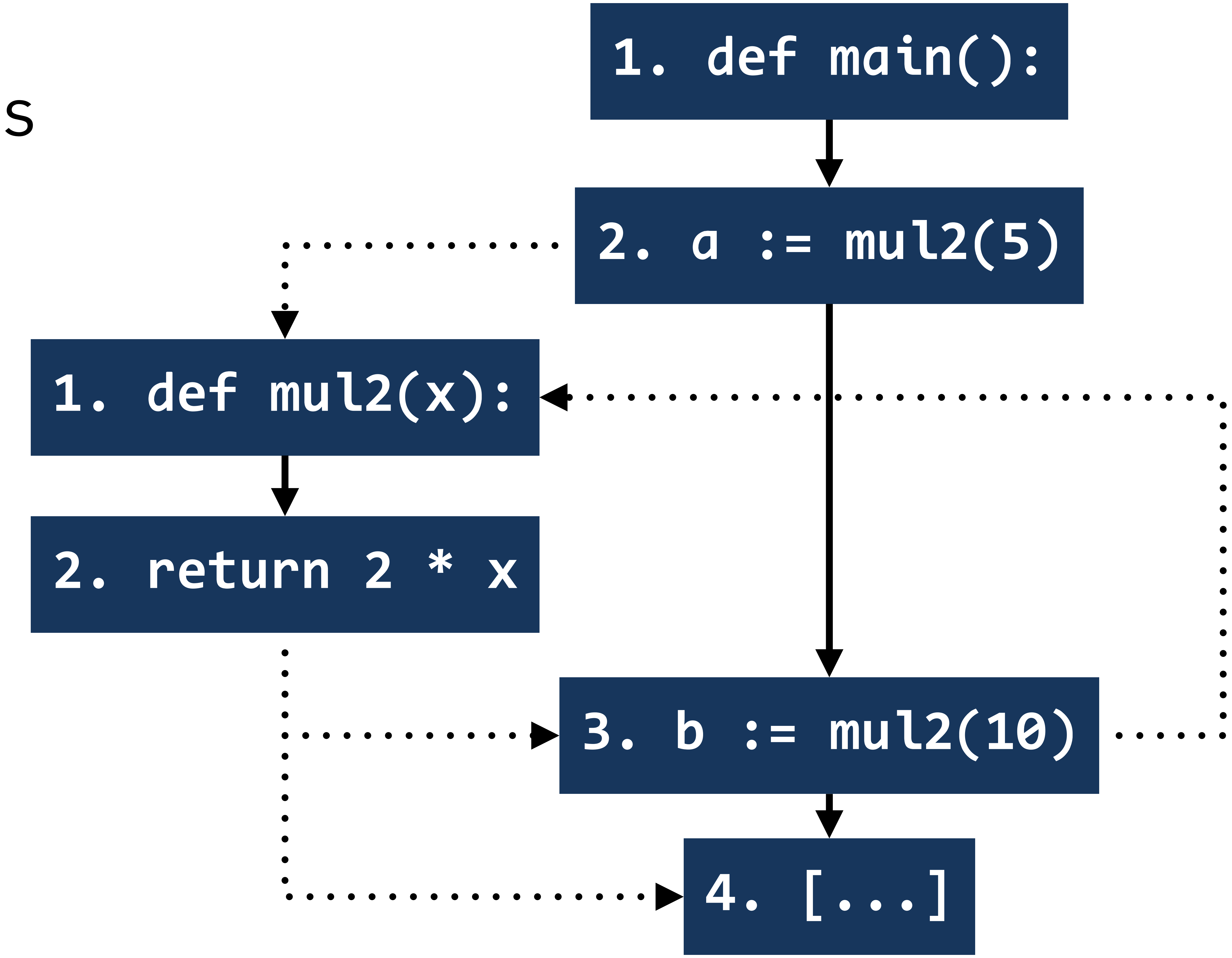
```
B:
  nop IMM_1
  [...]
  if(**rsp != nop IMM_2) exit
  ret
```

Backward Edge and Forward Edge

- We protect the <u>backward edge</u> if returns are CFG-enforced
  - This suffices to protect against basic ROP
- We protect the <u>forward edge</u> if calls/jumps are CFG-enforced
  - Needed to protect against other xOP variants (Jump/CallOP)
- For effective CFI, we need forward edge protection and backward edge protection

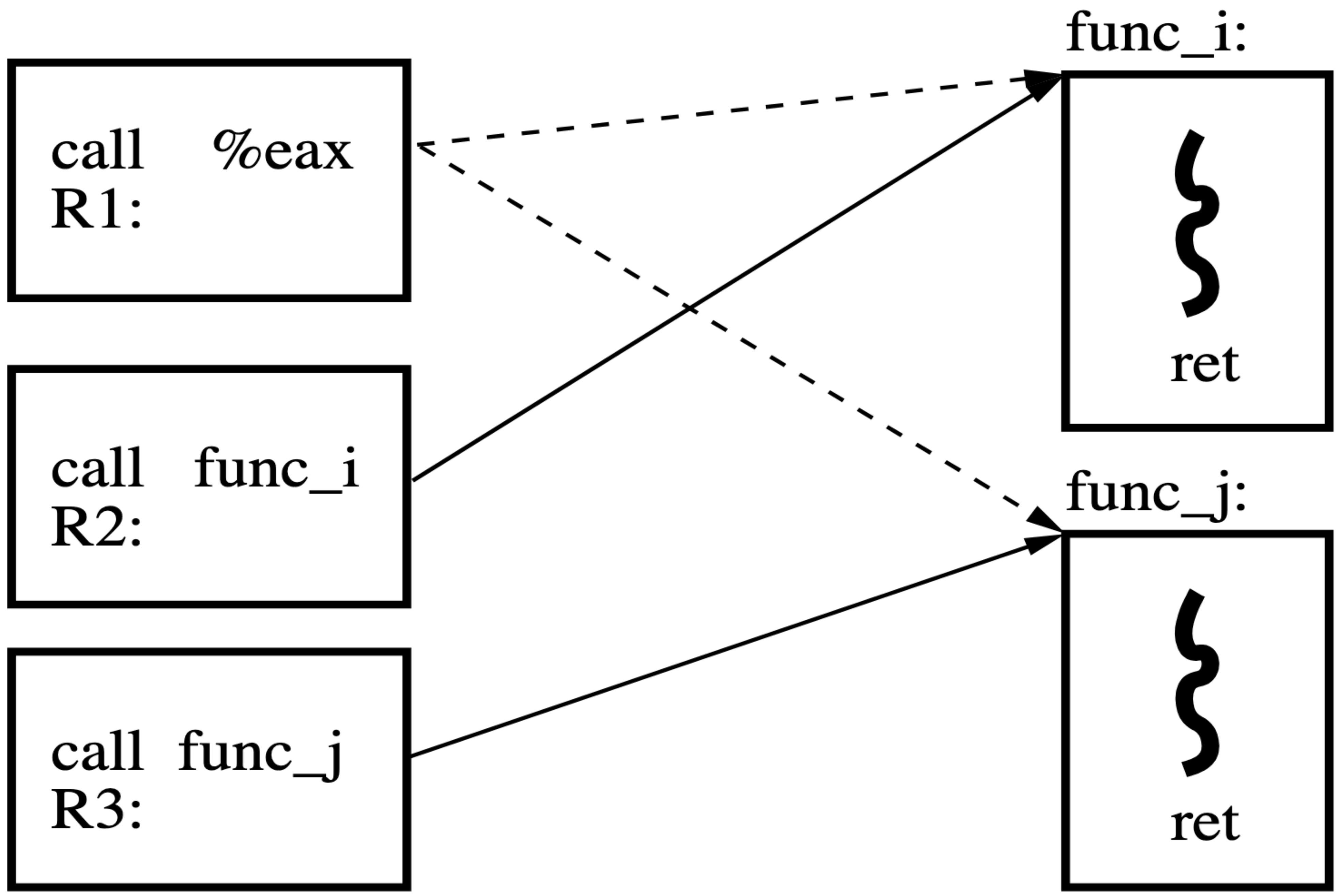
Context Sensitivity

- Slightly weaker because a CFG might have multiple destinations, but only one destination is the <u>supposed</u> destination.
  - A CFG is not context-sensitive, thus we cannot (necessarily) determine whether the return destination for double() should be
    - before: `3. b := mul2(10)`
    - or before: `4. [...]`



Destination Equivalence

- Two destinations are equivalent if they connect to a common source in the CFG



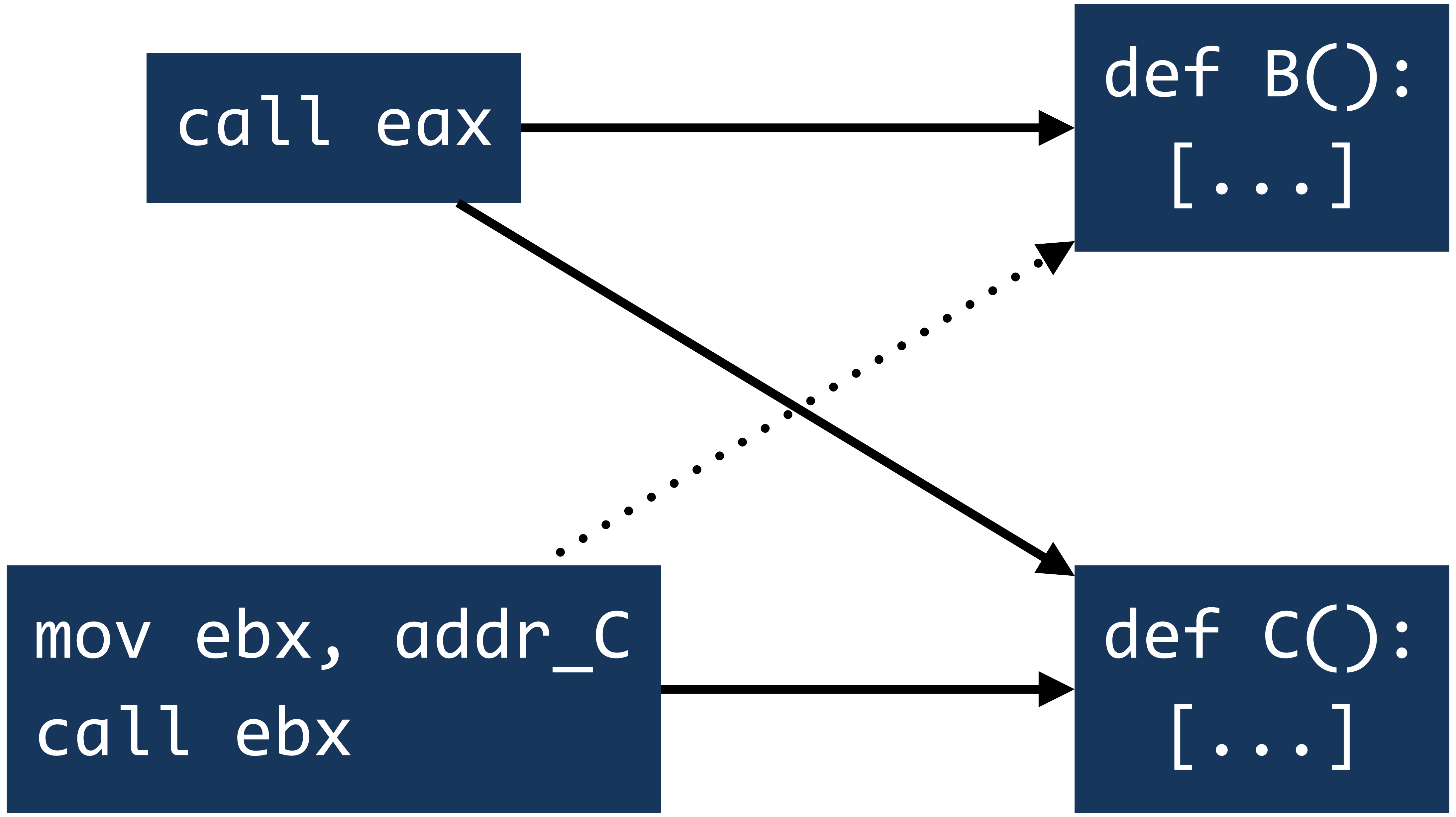
Can R2 be a return of target func_j?

Figure via "Securing Virtualization: Techniques and Applications," by Zhi Wang

"Zig-Zag" Imprecision

Solution 1

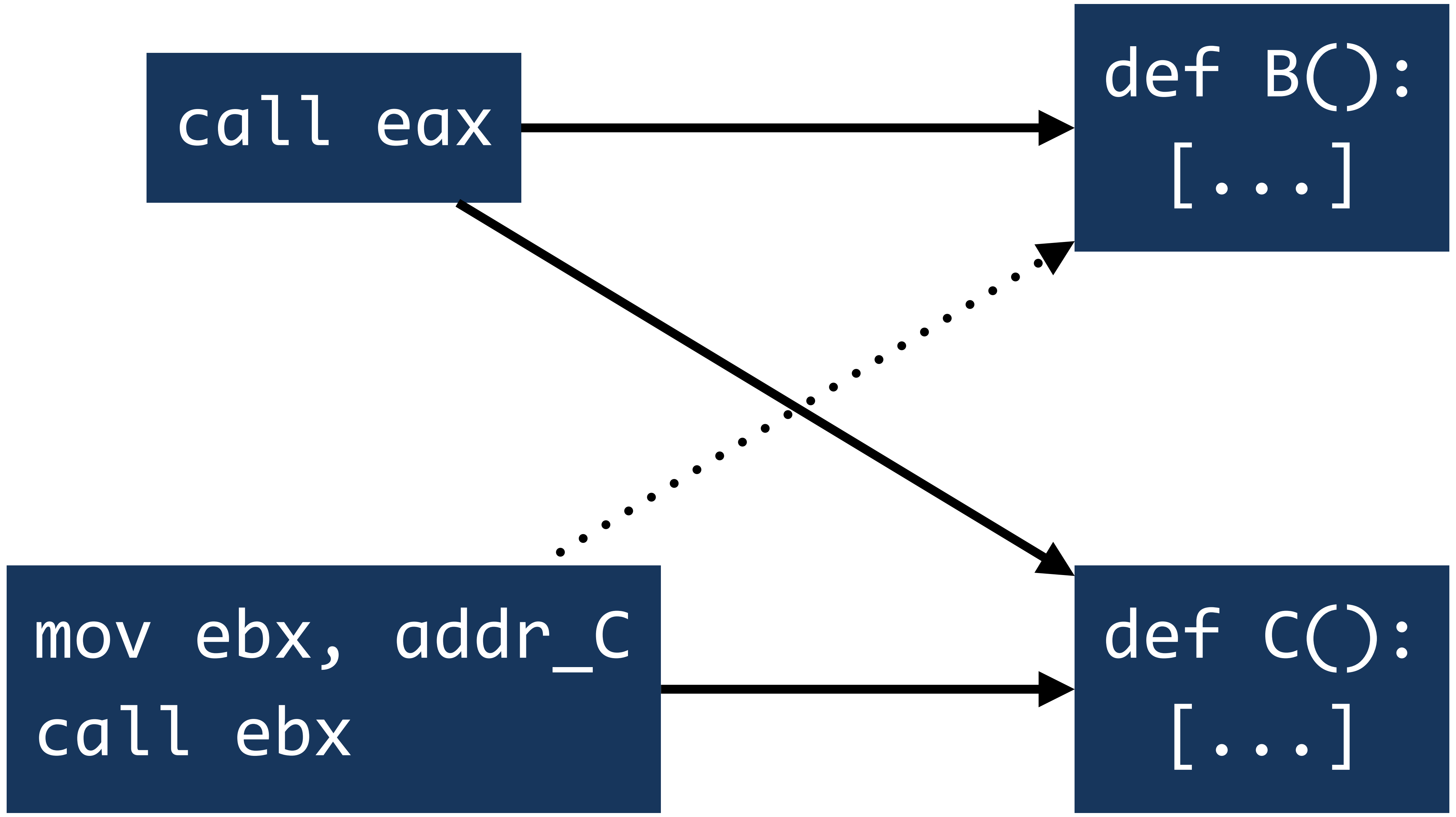
Allow the imprecision



dotted line is CFI-allowed

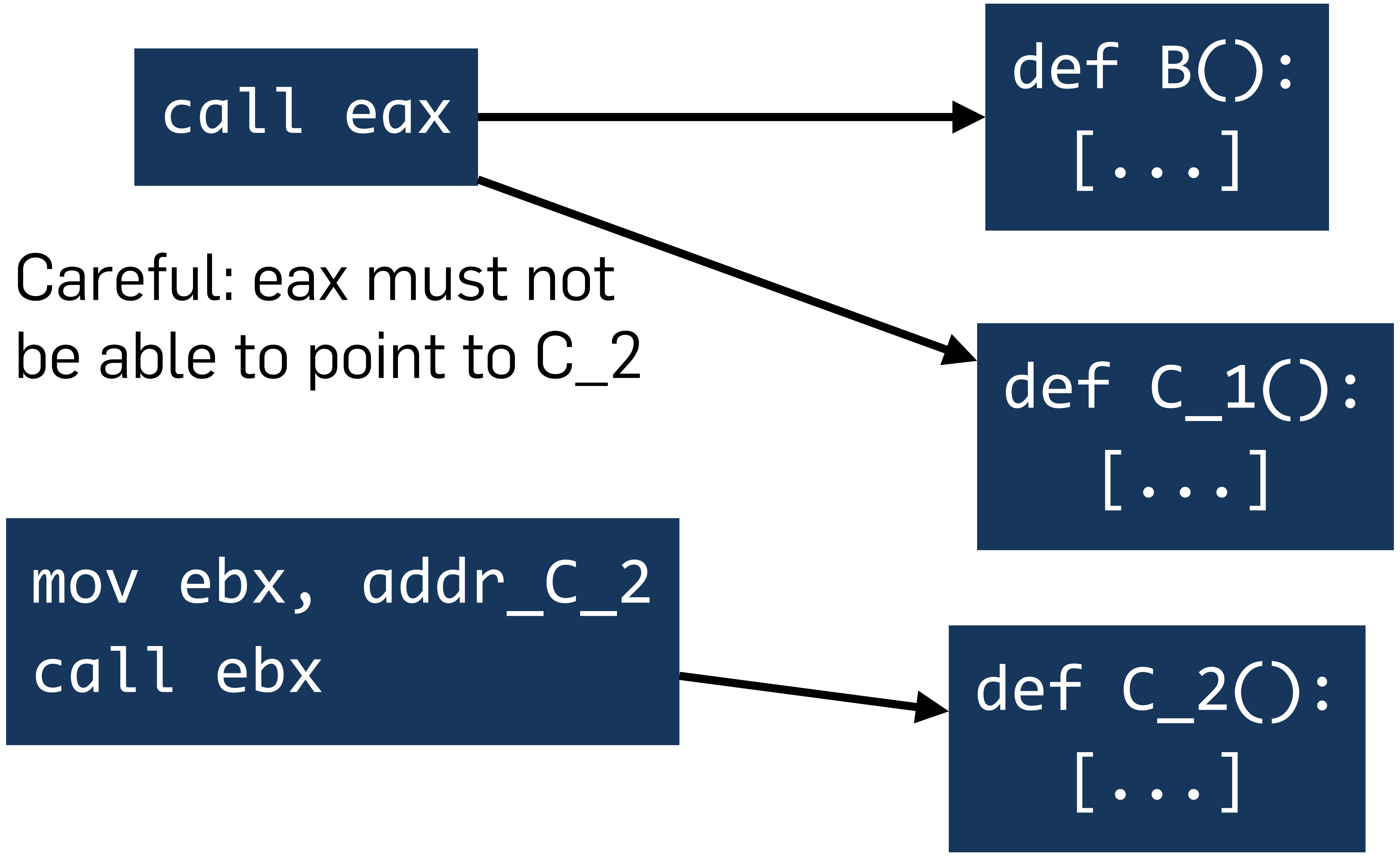
"Zig-Zag" Imprecision

Solution 1
Allow the imprecision



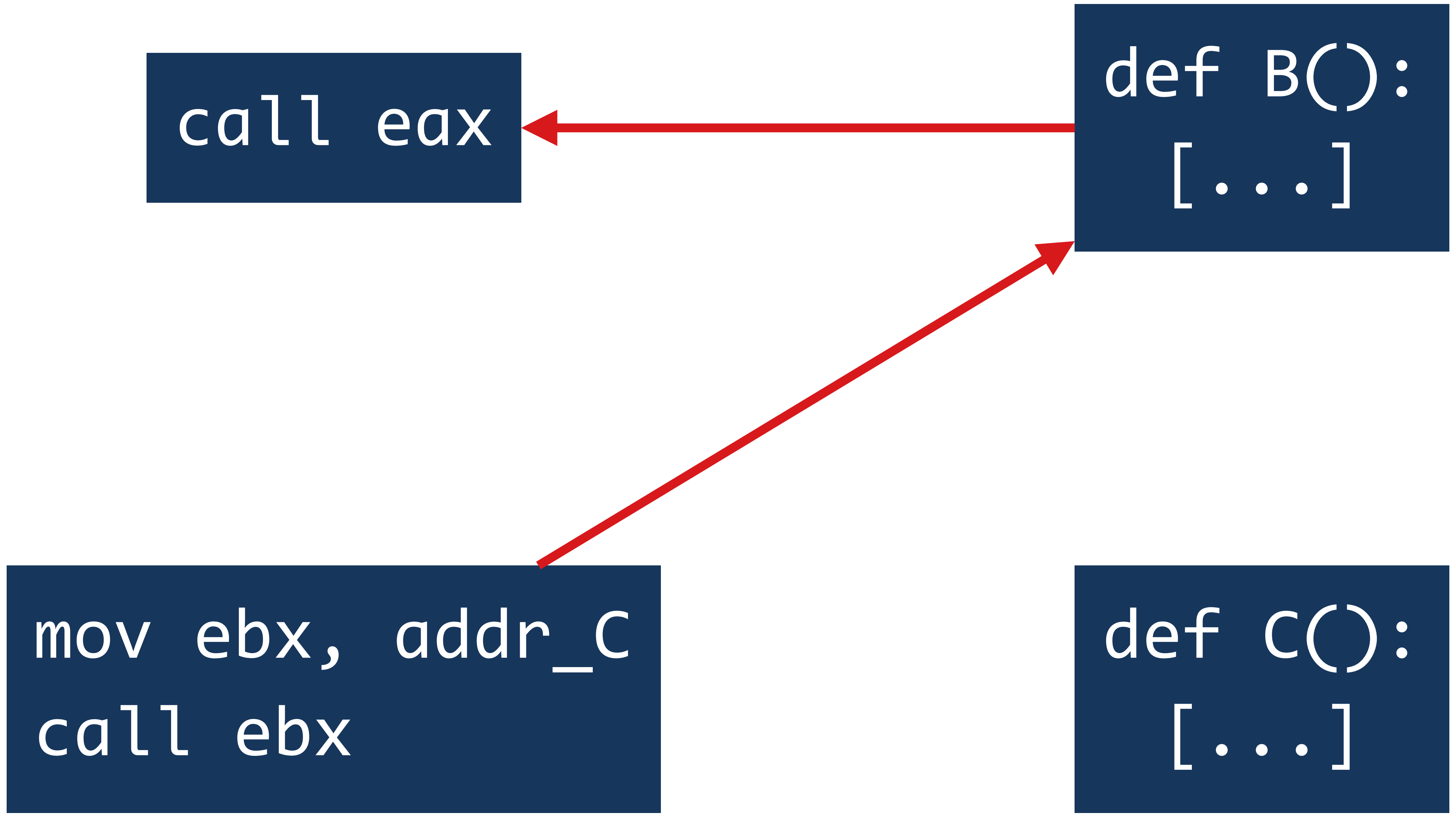
dotted line is CFI-allowed

Solution 2
Duplicate code to remove zig-zags



"Zig-Zag" Imprecision

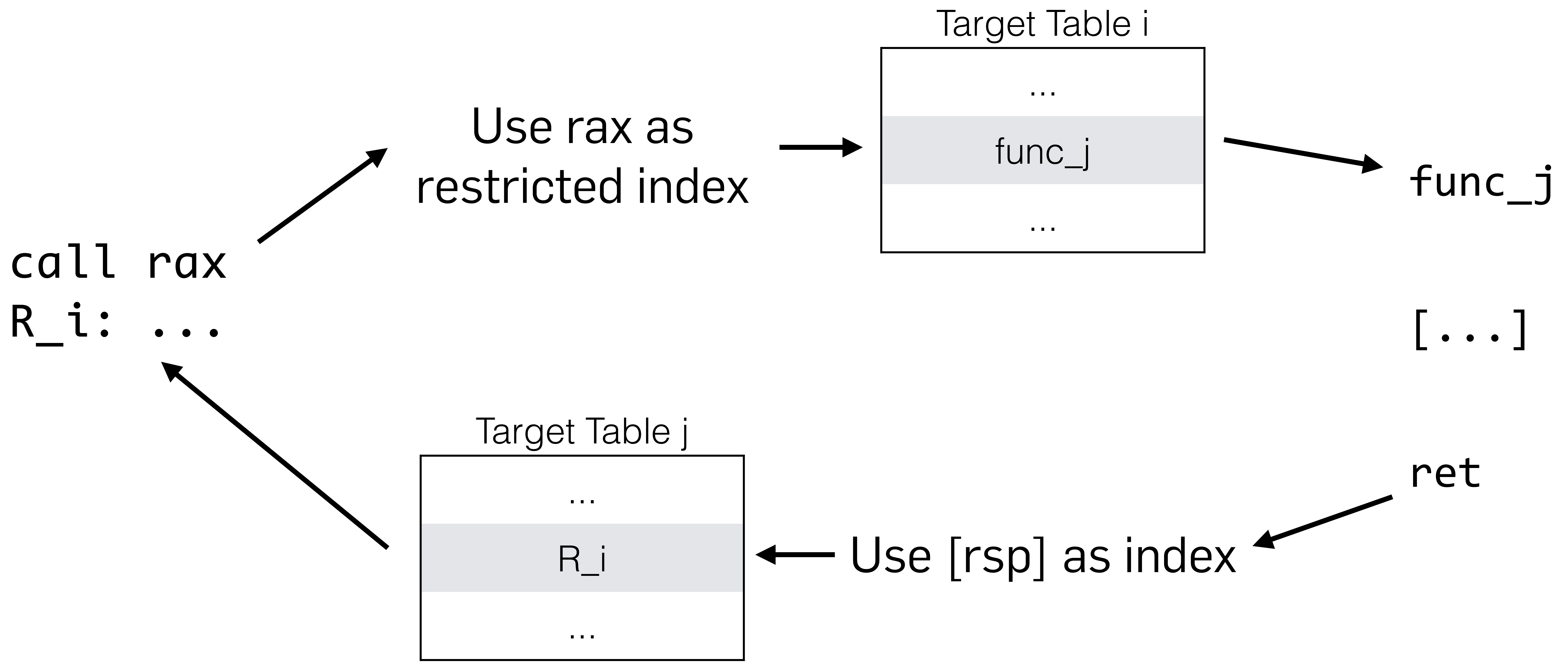
Solution 1
Allow the imprecision



Solution 2
Duplicate code to remove zig-zags

Restricted Pointer Indexing

- One table for call and one table for returns for each function



CFI and CFG

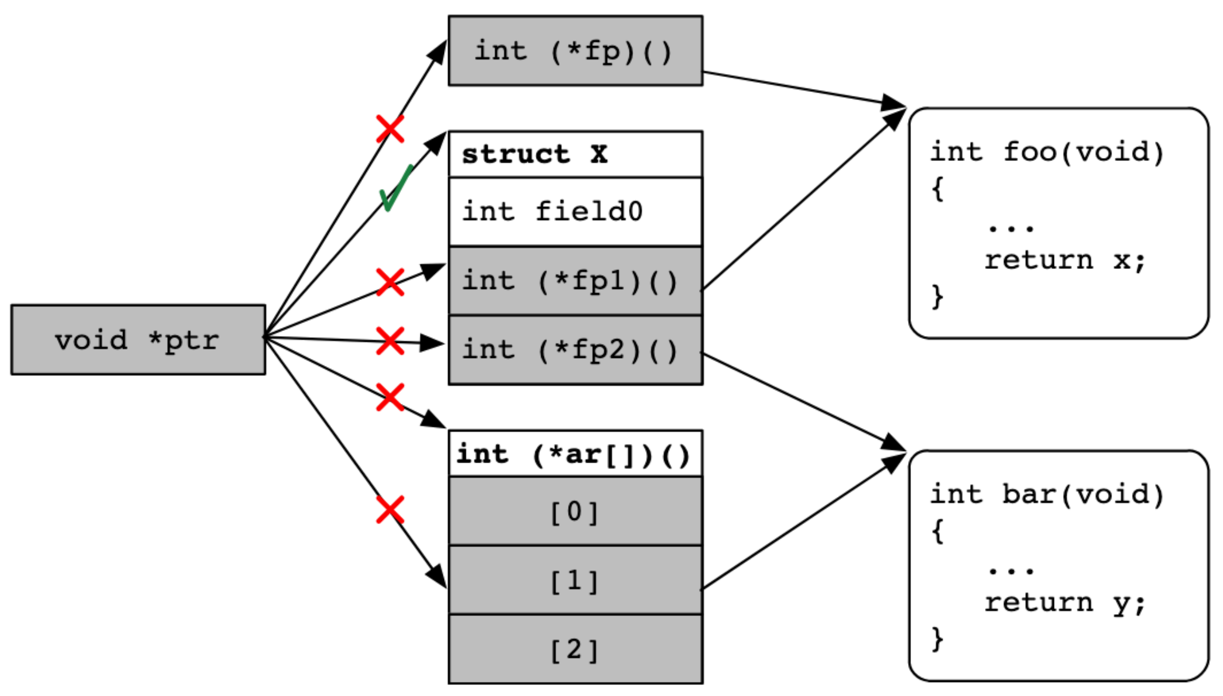
- CFI enforces an expected CFG
  - Each call at a call-site transfers to an expected instruction
  - Each return goes back to an expected call-site
- Direct calls are no problem
- Indirect calls are problematic
  - All possible targets for function pointers need to be computed, might not be accurately possible
- Returns
  - Determined dynamically and stored somewhere, could be overwritten

Enforce CFG via CFI

- Computing an enforceable CFG is difficult
- Approaches
  - Coarse-grained CFG
    - Any function is a legal indirect call target
    - Any call-site is a legal return target
  - Signature-based
    - Functions with the same signature are valid as indirect call targets
      - Problematic: `system(const char *command)` vs `chdir(const char *path)`
  - Taint-based
    - Track function symbols/addresses that can reach an indirect call target

Taint-based CFG

- If function pointers are used in a restricted way, we can predict the indirect call targets using taint analysis
  - Assumption 1: The only allowed operations on a function pointer variable are assignments and dereferencing (for call)
  - Assumption 2: There is no data pointer to a function pointer



Shadow Stack

- Idea: When doing a call, store the return address in two separate places and compare them before returning
- Shadow stack is not in memory accessible by the program, but it needs to be separate
  - Set up by the OS/hypervisor
  - Protected via privilege levels
- Implies: Cannot be checked by program itself and requires OS support

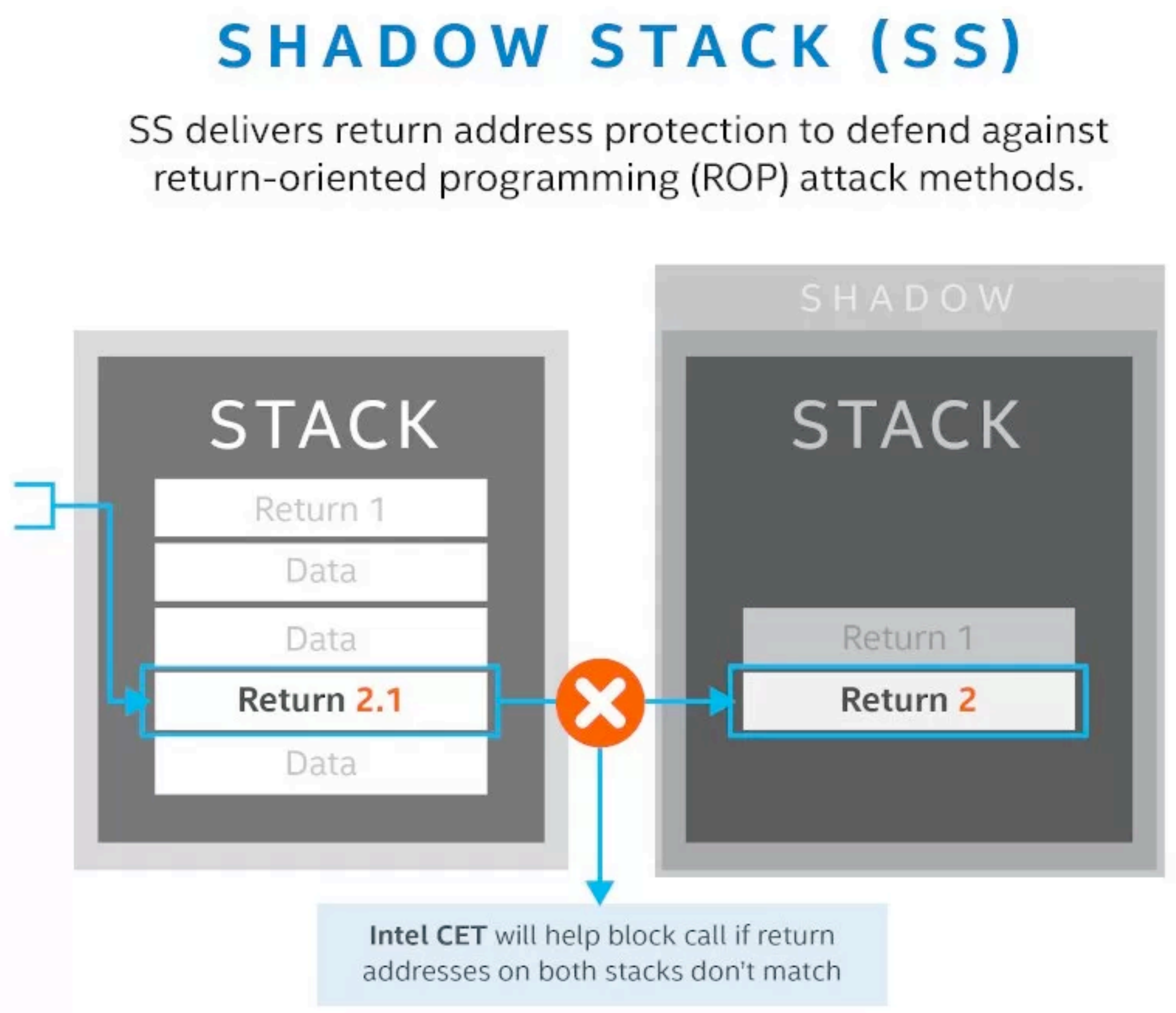


Figure via Intel

Defeating Control Flow Integrity

"Control-flow integrity principles, implementations, and applications", Abadi et al., https://dl.acm.org/doi/10.1145/1609956.1609960

Defeating Control Flow Integrity

- Not all locations of where you may return to may be known at compile time or initialization time, or they cannot be checked accurately.
  - You need to overapproximate the set of locations. This may allow you to defeat it.

"Control-flow integrity principles, implementations, and applications", Abadi et al., https://dl.acm.org/doi/10.1145/1609956.1609960

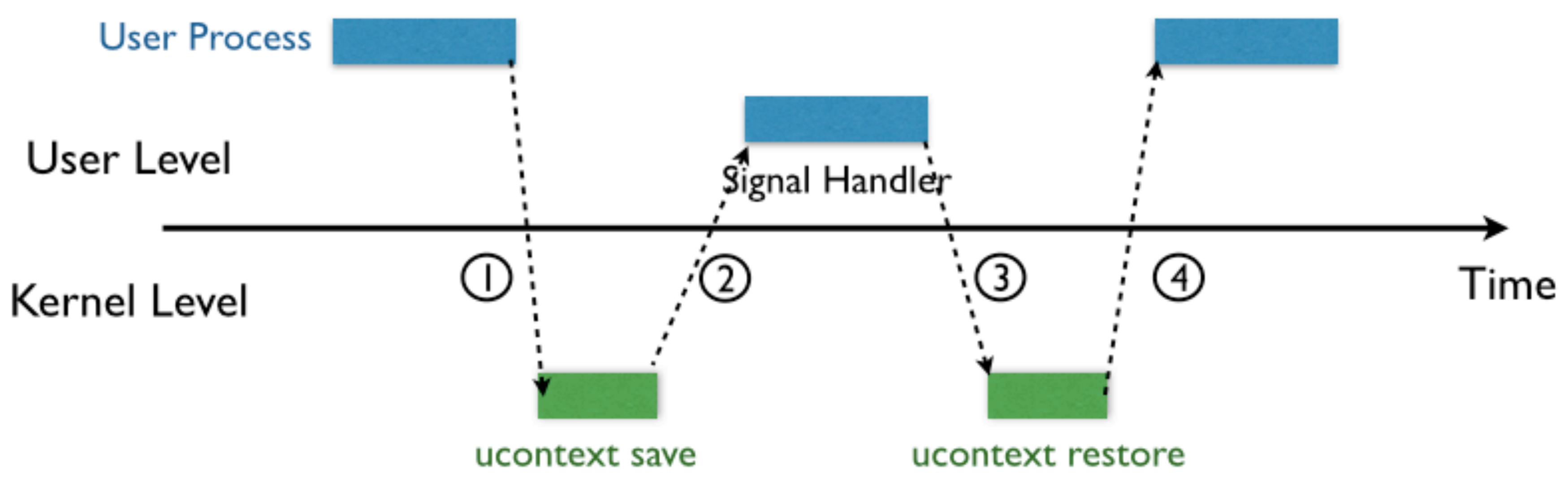
Defeating Control Flow Integrity

- Not all locations of where you may return to may be known at compile time or initialization time, or they cannot be checked accurately.
  - You need to overapproximate the set of locations. This may allow you to defeat it.
- Techniques
  - B(lock)OP: ROP on basic block (or multi-block) level
  - J(ump)OP: Use indirect jumps (with overly broad destinations)
  - C(all)OP: Use indirect calls (with overly broad destinations)
  - S(igreturn)OP: Use the sigreturn syscall and signals
  - D(ata)OP: Leave control flow intact, but change data values
    - It might call system() by itself, we can just change the argument

"Control-flow integrity principles, implementations, and applications", Abadi et al., https://dl.acm.org/doi/10.1145/1609956.1609960

Sigreturn-oriented Programming (SROP)

- During signal handling
    - The kernel saves the current context, calls the signal handler, restores the context

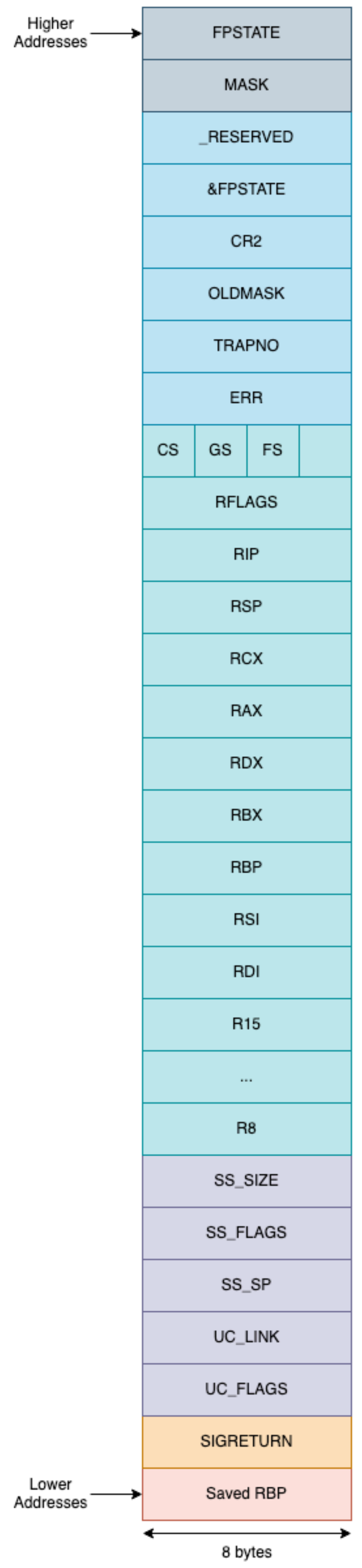


- The context is all registers + other state information of the process
- The context is pushed on the stack before the handler and restored after

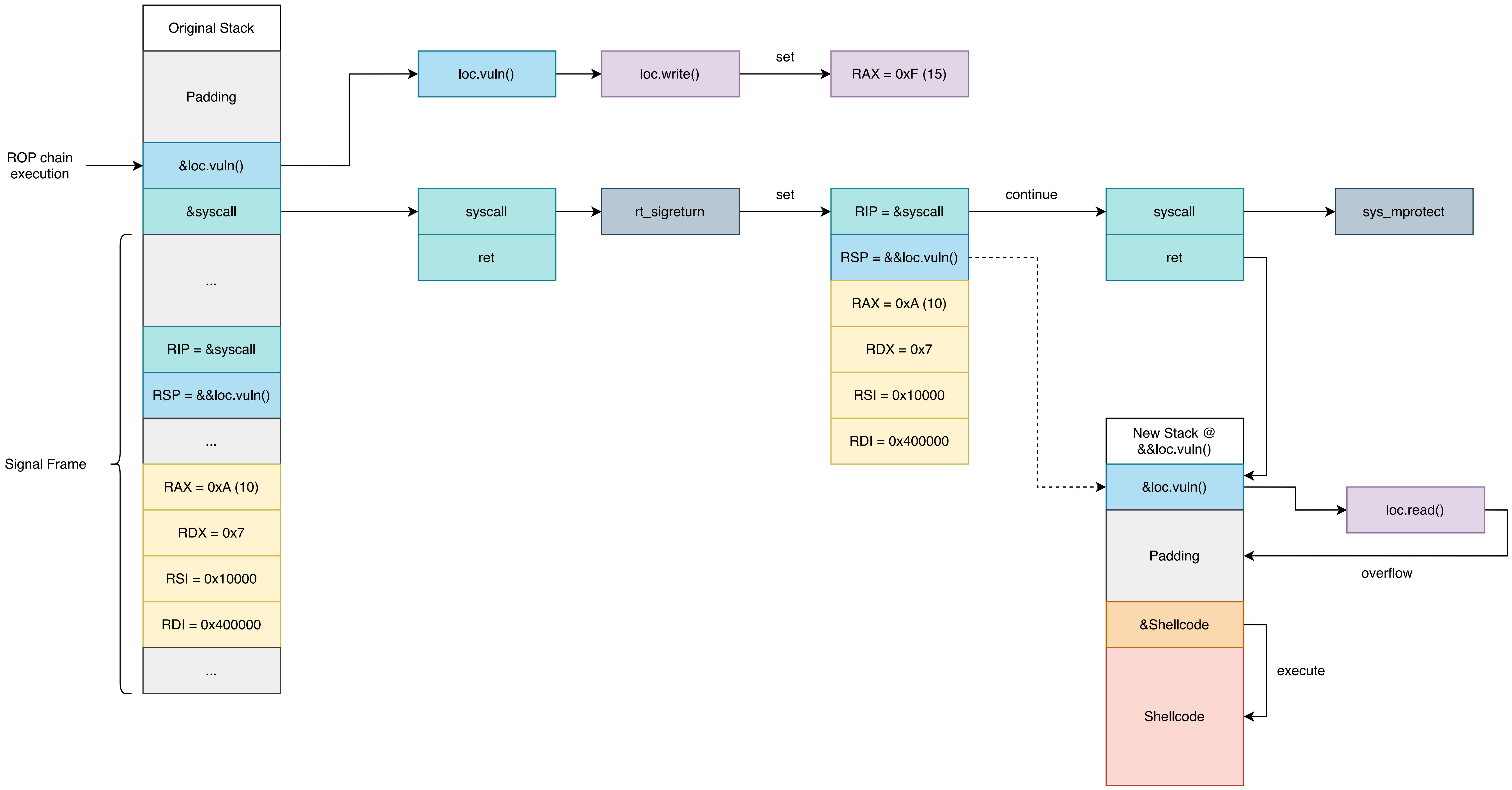
```
struct _fpstate
{
  /* FPU environment matching the 64-bit FXSAVE layout.  */
  __uint16_t        cwd;
  __uint16_t        swd;
  __uint16_t        ftw;
  __uint16_t        fop;
  __uint64_t        rip;
  __uint64_t        rdp;
  __uint32_t        mxcsr;
  __uint32_t        mxcr_mask;
  struct _fpxreg    _st[8];
  struct _xmmreg    _xmm[16];
  __uint32_t        padding[24];
};

struct sigcontext
{
  __uint64_t r8;
  __uint64_t r9;
  __uint64_t r10;
  __uint64_t r11;
  __uint64_t r12;
  __uint64_t r13;
  __uint64_t r14;
  __uint64_t r15;
  __uint64_t rdi;
  __uint64_t rsi;
  __uint64_t rbp;
  __uint64_t rbx;
  __uint64_t rdx;
  __uint64_t rax;
  __uint64_t rcx;
  __uint64_t rsp;
  __uint64_t rip;
  __uint64_t eflags;
  unsigned short cs;
  unsigned short gs;
  unsigned short fs;
  unsigned short __pad0;
  __uint64_t err;
  __uint64_t trapno;
  __uint64_t oldmask;
  __uint64_t cr2;
  __extension__ union
    {
      struct _fpstate * fpstate;
      __uint64_t __fpstate_word;
    };
  __uint64_t __reserved1 [8];
};
```

Sigreturn-oriented Programming (SROP)

- Interesting for us is the sigreturn syscall
  - Reads off the stack and restores context
  - Does not validate the integrity of the context
- Effectively, we spoof that a signal was handled
- Idea: Use a syscall to set up a syscall
  - Write signal frame into the stack
  - Return to sigreturn to set registers etc.
  - Return to syscall



Sigreturn-oriented Programming (SROP) Example



Control Flow Integrity: CPU Support

- Intel introduced some CPU support for control-flow integrity in June 2019, "Control-flow Enforcement Technology" (CET)
- Shadow stack protects backward edge
- Endbranch protects forward edge
  - Jumps must land on an endbreak instruction (endbr64) (= one destination ID)
  - Compiler adds the endbreak instruction where needed
  - If the target of an indirect jump is not an endbreak instruction, then the program will terminate
- Makes some xOP attacks more difficult, but also by-passable:
  - SROP: Typically a valid target (proper POSIX control flow transfer)
  - Data-only attacks
  - Simply put: It makes programming your weird machine weirder, and restrict the instructions you can use (or in Lego terms: you cannot use blue Lego pieces anymore)

"Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity", Shanbhogue et al., https://dl.acm.org/doi/pdf/10.1145/3337167.3337175

Software Security 1

Fundamentals of Data-only Attacks

Kevin Borgolte kevin.borgolte@rub.de

Data-only Attacks

- Slight misnomer: Basically all attacks are data-only attacks, since control data is also data
  - Previously also called non-control-data attacks
- Difference to control flow hijacking:
  - We do not overwrite control flow data in any way or form
  - Execution remains CFG-enforceable (CFI does not help)
- Instead, we overwrite some data that is used normally and then continue with <u>normal</u> execution

Data-only Attacks

Vulnerable code on the server

```
char some_buffer [256];
char cgi_bin_path [MAX_PATH_LEN];

void handle_request(...) {
    ...
    // Read up to 512B into a 256B buffer
    read (fd, some_buffer, 512);  // Overflow!
    ...
}
```

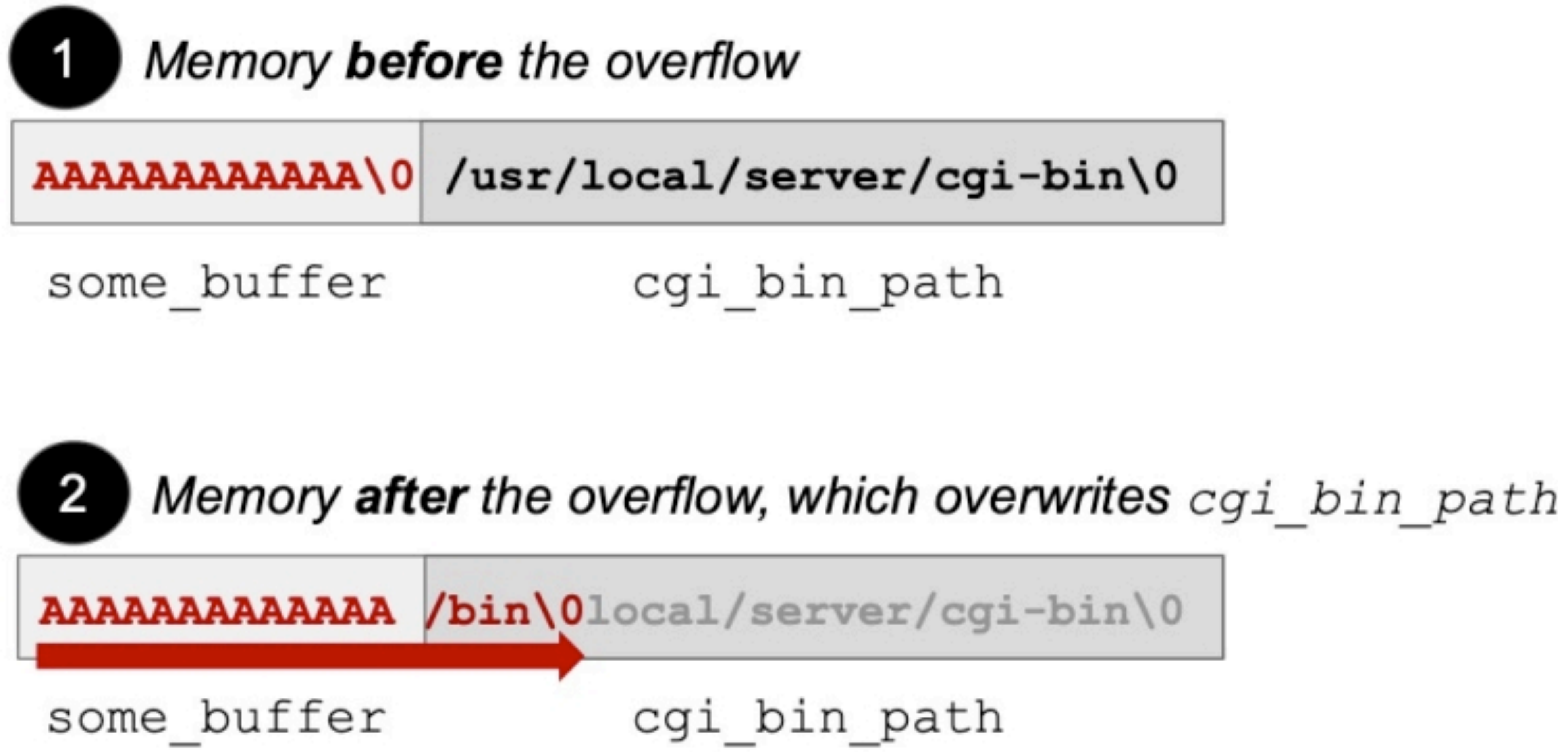


Figure via "Data-Only Attacks Are Easier than You Think," Johannesmeyer et al.

Data-only Attacks

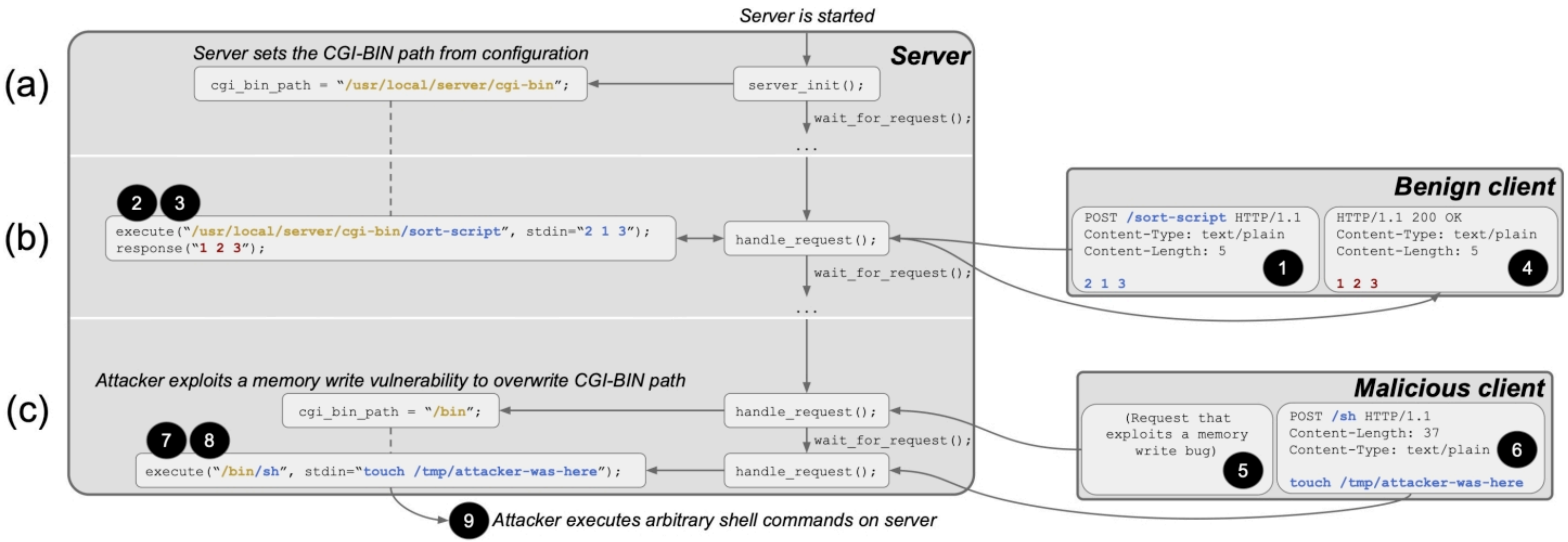


Figure via "Data-Only Attacks Are Easier than You Think," Johannesmeyer et al.

Data-only Attacks

- Data-only attacks are no "exploit silver bullet"
  - Targets need to include dangerous function calls/syscalls that are normally called (in a safe way)
    - However, programs often use many of them
    - Libraries in your address space further increase attack surface
- Typically non-control data is <u>less expressive</u>
  - Real expressive power is (often) Turing-complete though
- See also
  - "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks" by Hu et al. https://doi.org/10.1109/SP.2016.62
  - "Block Oriented Programming: Automating Data-Only Attacks" by Ispoglou et al. https://doi.org/10.1145/3243734.3243739