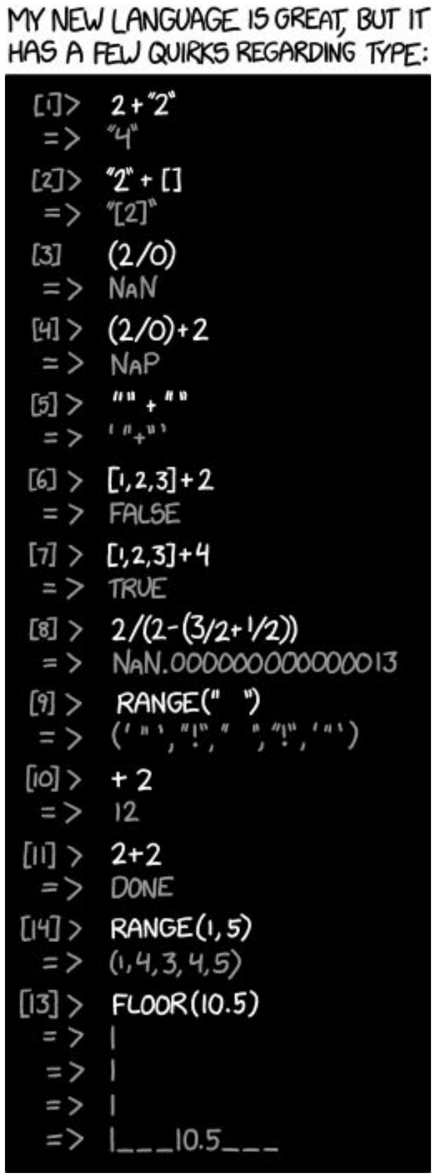
Διάλεξη #11 - Control Flow Integrity

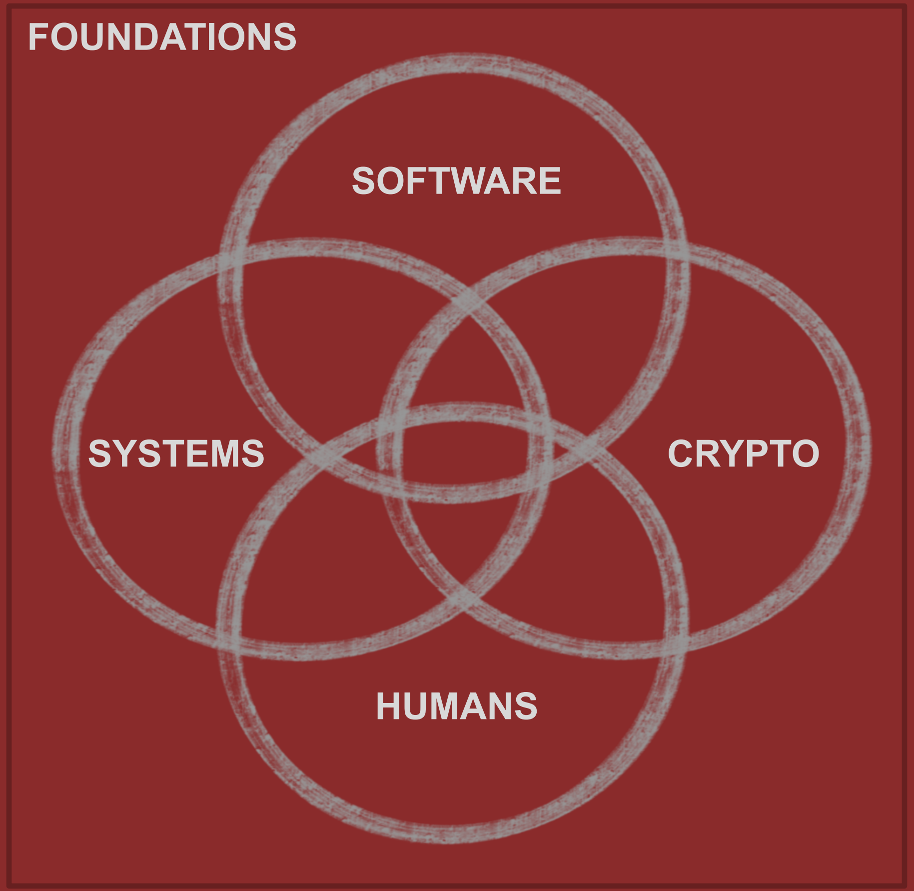
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Εισαγωγή στην Ασφάλεια

Θανάσης Αυγερινός



https://xkcd.com/1537/



Huge thank you to David Brumley from Carnegie Mellon University for the guidance and content input while developing this class (some slides from Dan Boneh @ Stanford!)

Ανακοινώσεις / Διευκρινίσεις

- Άρα γιατί δεν μπορώ να πάρω root access στον υπολογιστή μου τρέχοντας setresuid(0, 0, 0) ο ίδιος;
- Υπάρχουν άλλες μέθοδοι για access control enforcement;

Την Προηγούμενη Φορά

- Reference Monitors
- "Gold" (Au) Standard: Authentication + Authorization + Audit
- Authorization Mechanisms / Access Control
  - Access Control Lists (ACLs) and Capabilities (CAP)
  - Discretionary Access Control (DAC)
  - Role-Based Access Control (RBAC)

Σήμερα

- CFG and call graph definitions
- Insensitive and sensitive program analysis types
  - Soundness / Unsoundness
- Type safety

Our Security Journey So Far

Security Journey

Threat Modeling and Security Fundamentals

Get a handle on what you're trying to do.

Trusted Computing Base

Security needs to be rooted in something assumed. We try to limit that to a fixed, known set.

Program Exec

Let's understand software at a basic execution level.

Defenses

Compile-time and OS defenses.

Access Control

How to say who is who, who can do what, and check what they did.

Memory Safety Defenses

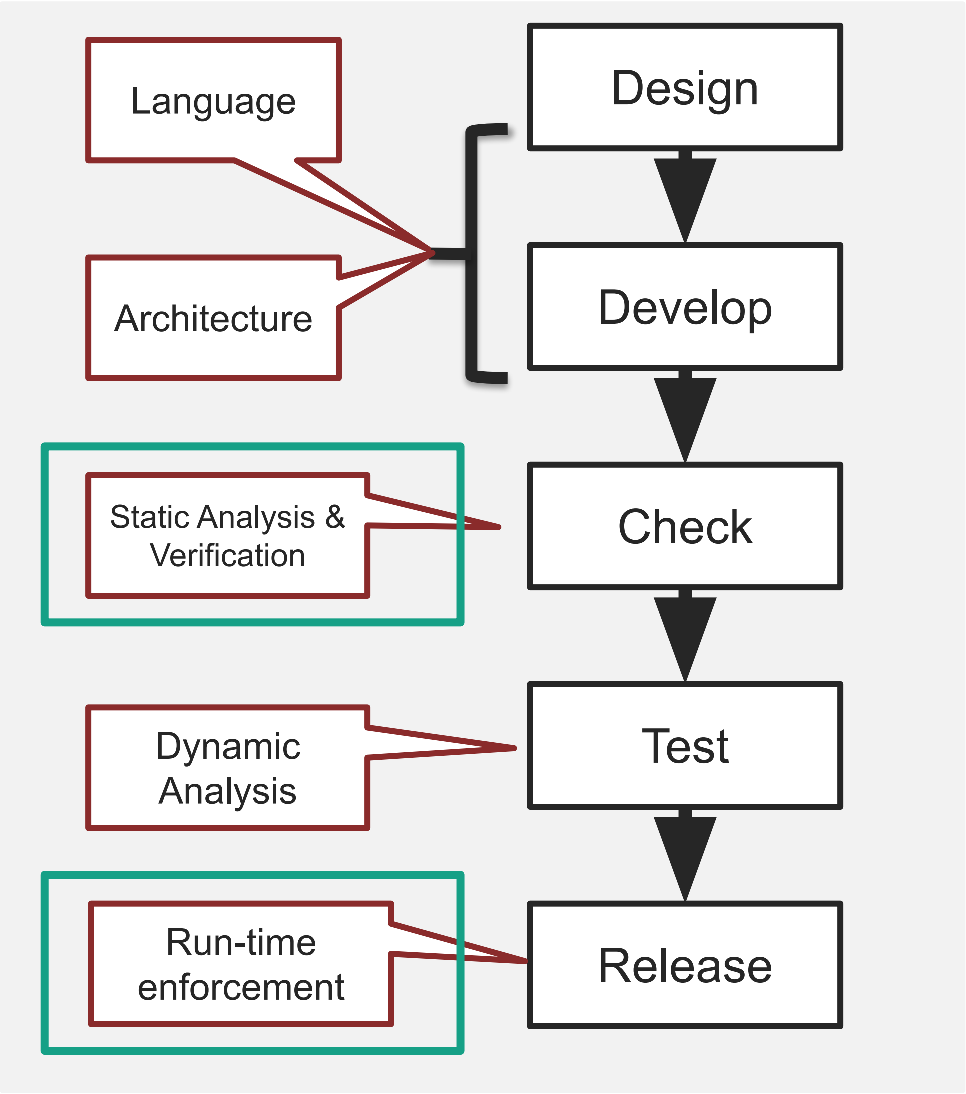
Control flow integrity, and by way of that, **introduction** to some program analysis terms.

Software Security Techniques

Different options at different stages of development lifecycle

Today: Types

Today: Control Flow Integrity



So far: Ad-hoc methods

All attacks rely on hijacking control!

http://propercourse.blogspot.com/2010/05/i-believe-in-duct-tape.html

Adversary Model Matters!

Cowan et al., USENIX Security 1998

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

*"Programs compiled with StackGuard are safe from **buffer overflow attack**, regardless of the software engineering quality of the program."*

Hmm. Live and learn.
What we missed was a formal model of security.

Motivating Example: Control Flow Integrity

- **protects against powerful adversary**
  - with full control over entire $\boxed{\text{data memory}}$
- **widely-applicable**
  - language-neutral; requires binary only
- **provably-correct & trustworthy**
  - formal semantics; small verifier
- **efficient**
  - hmm… 0–45% in experiments; average 16%

Intro to Program Analysis

Intelligence means the ability to question statements

Proof? That’s just an if-then statement.

- What is the “if” (assumptions).
- What is the implication of the “then”? Is it trivial or not?
- What are the requirements to make the proof? Is it rejecting good programs?

Let’s start by understanding program analysis terms

We'll divide this into three sections

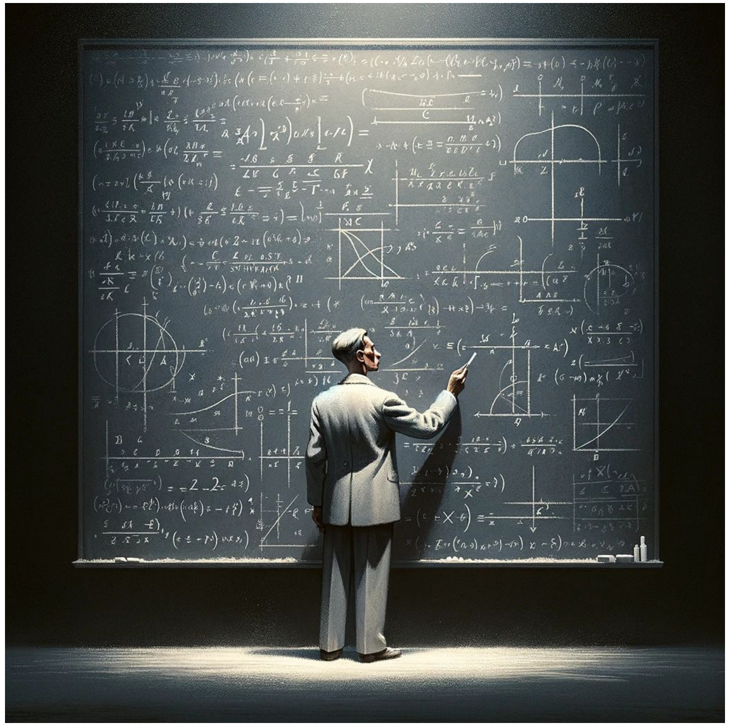
1. Soundness and completeness
2. Control flow reasoning
3. Data flow reasoning (types)

Two types of programs in this world

All programs

Ok Programs	Buggy programs

Defining buggy and ok



There is a catch-22. We need to define buggy and ok, but that requires a level of formalism we don't have.

We'll resolve this with examples and pseudo-logical statements

Examples

"If the program *p* is bad if it raises a unix signal"
$\forall p. \exists i. (run(p,i) = signal) \rightarrow bad(p)$

"Only in-bounds array accesses are allowed"
$\forall p. \forall i. isPointer(p) \wedge pointsTo(p, mem, l) \wedge , i \leq l \rightarrow ok(p[l]))$

"unprivileged users should not access privileged resources"
$\forall u. \forall r. unprivileged(u) \wedge access(r, privilged) \rightarrow deny(u,r)$

An *analysis* labels a program p as either ok or buggy.

Please differentiate between:

- What the program actually does for all inputs
- What the analysis says about the program

All program s

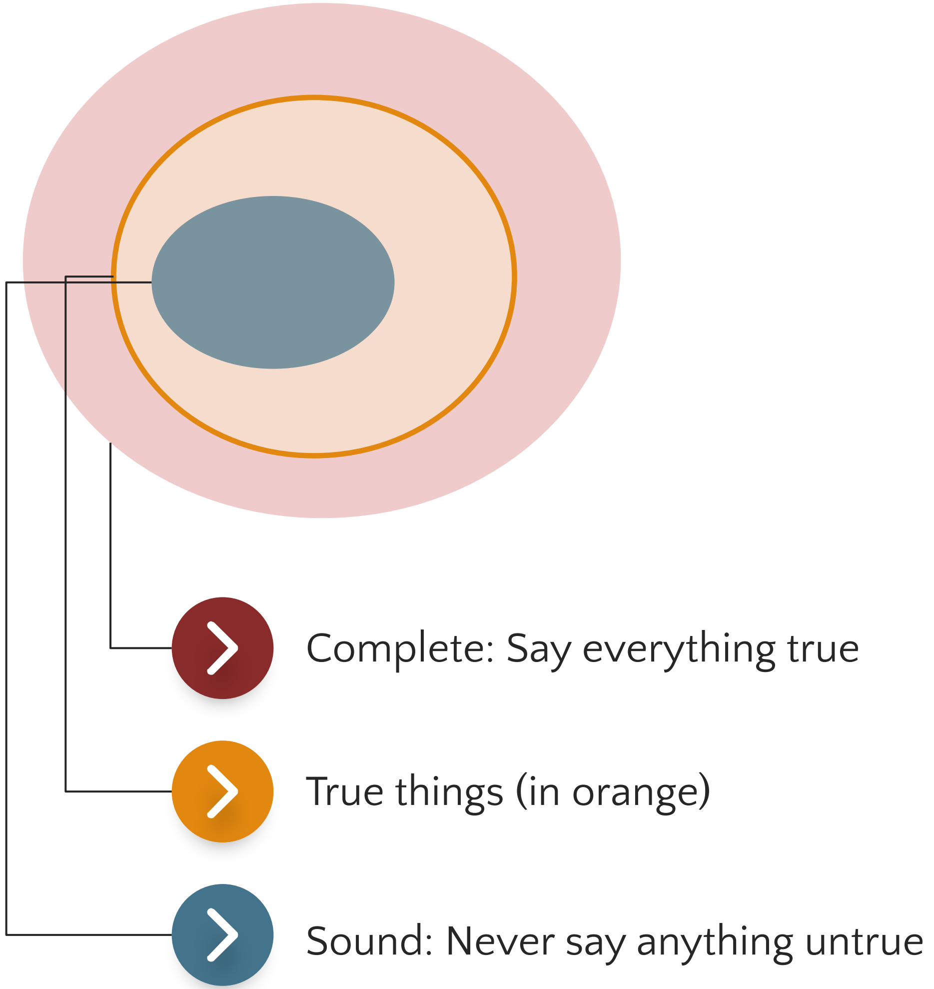
Analysis Sound/Complete Tradeoff

Sound: If the analysis says X is true, then X really is true.

- I.e., better to be quiet than tell a lie.
- Trivial example: never say anything is true.

Complete: If X is true, the analysis says X

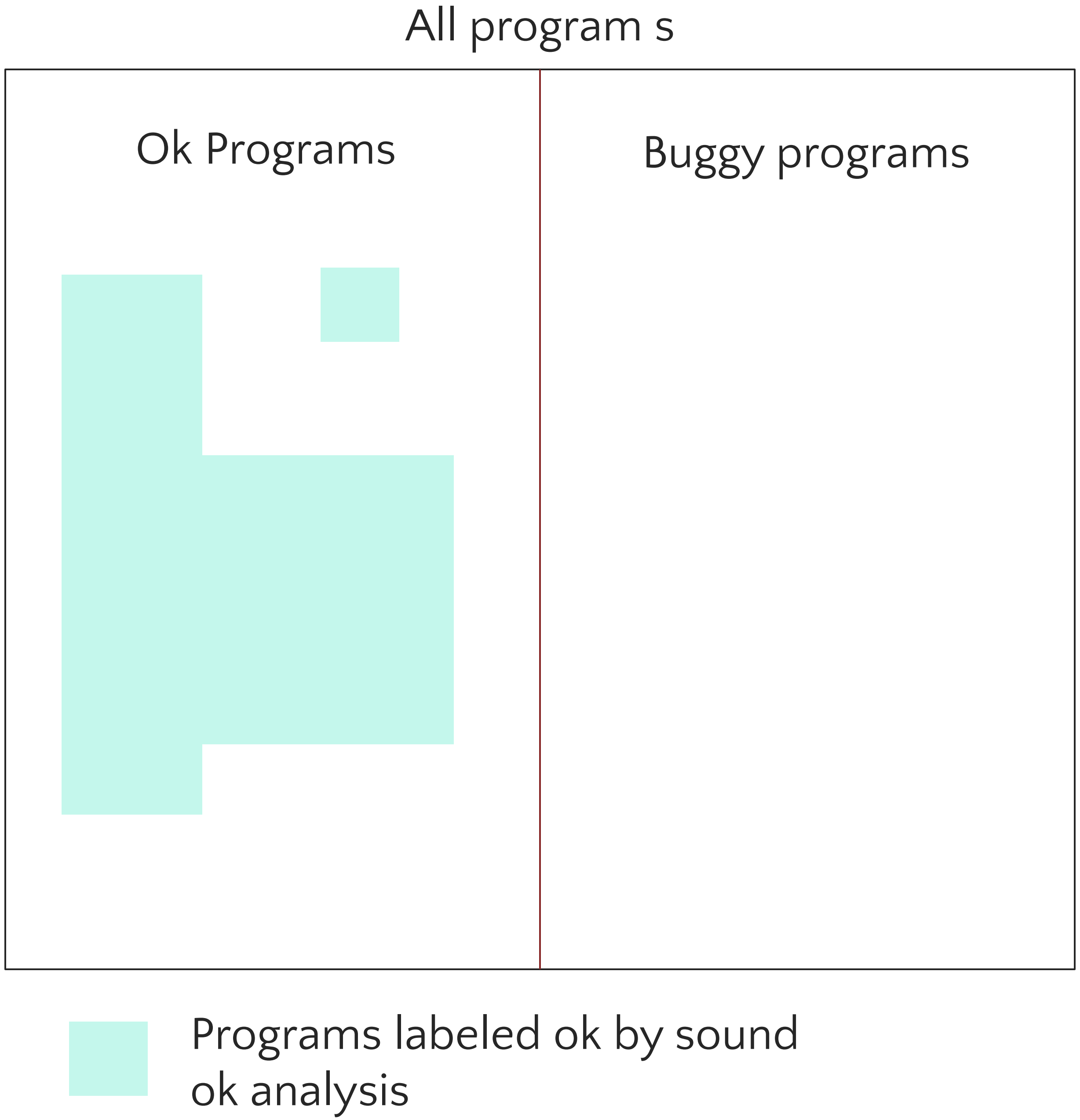
- I.e., better to say all things than miss a true fact
- Say everything



Extremes: can you provide examples of complete / sound analyses?

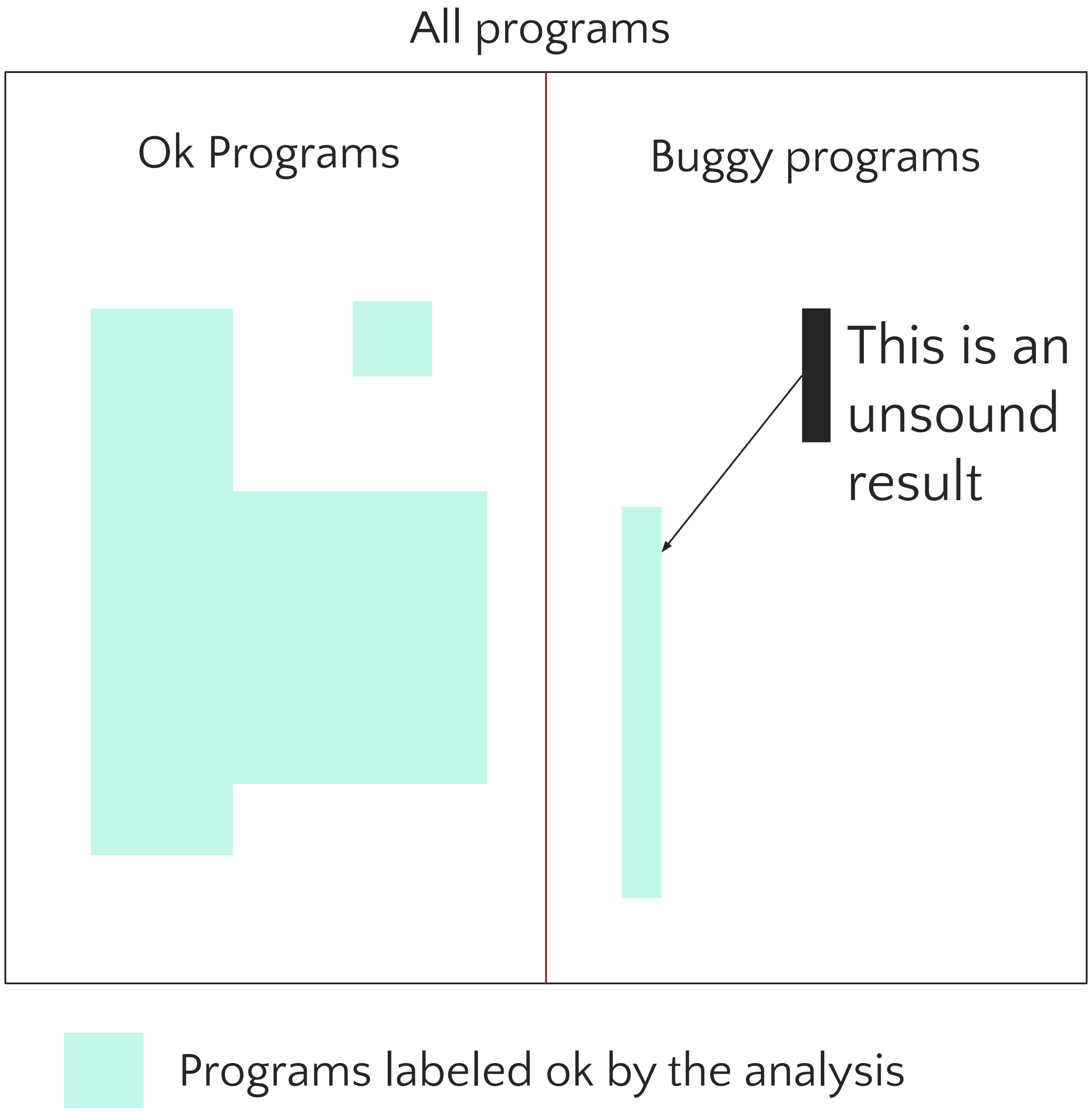
Sound ok analysis:
analysis(p) = ok,
then p is ok

(Notice the under-approximation)

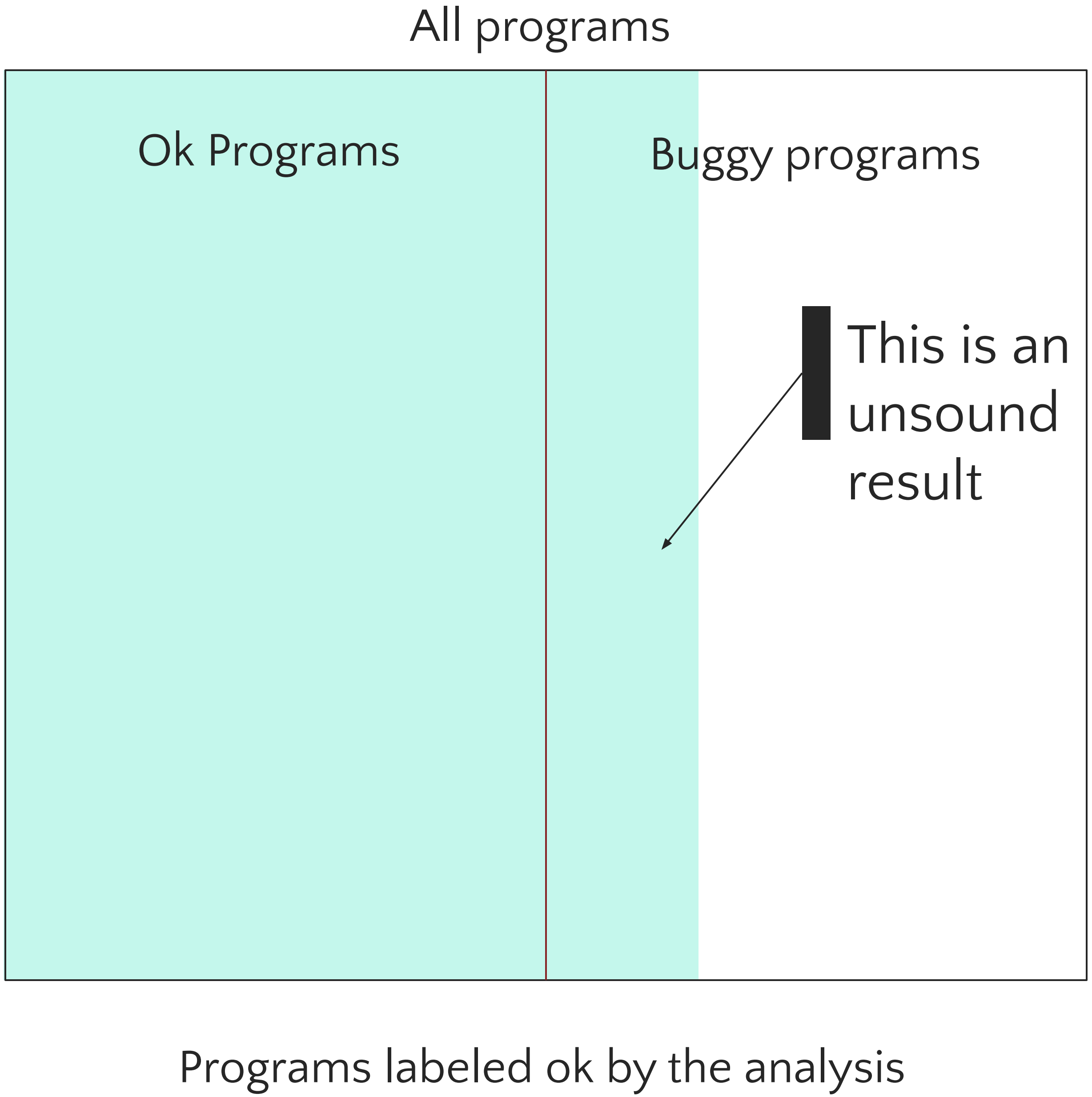


Analysis can also be *unsound*

Unsound ok analysis:
analysis(p) = ok, but
p is not ok

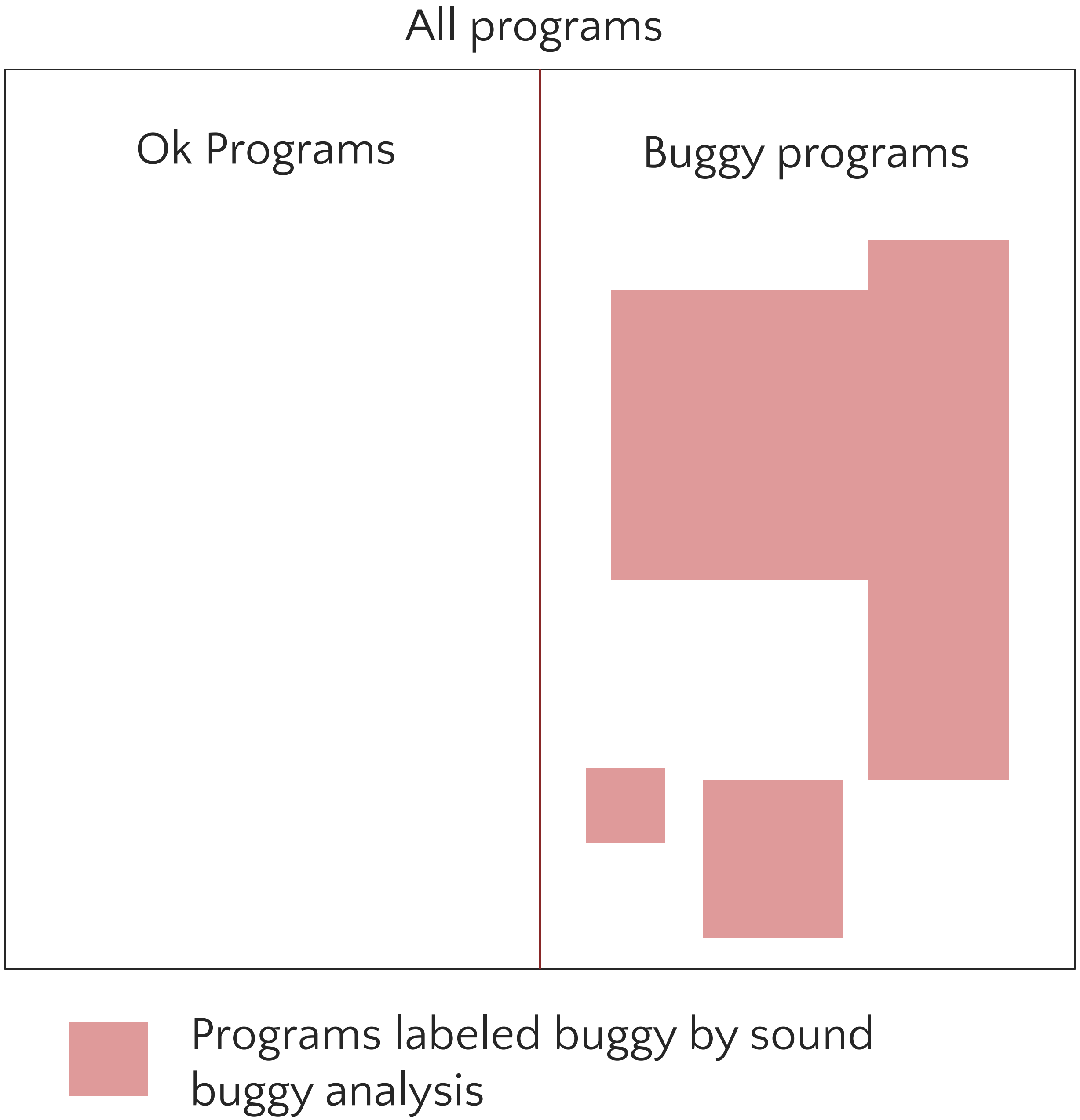


<u>Complete ok analysis:</u>
If p is ok, then analysis(p) = ok



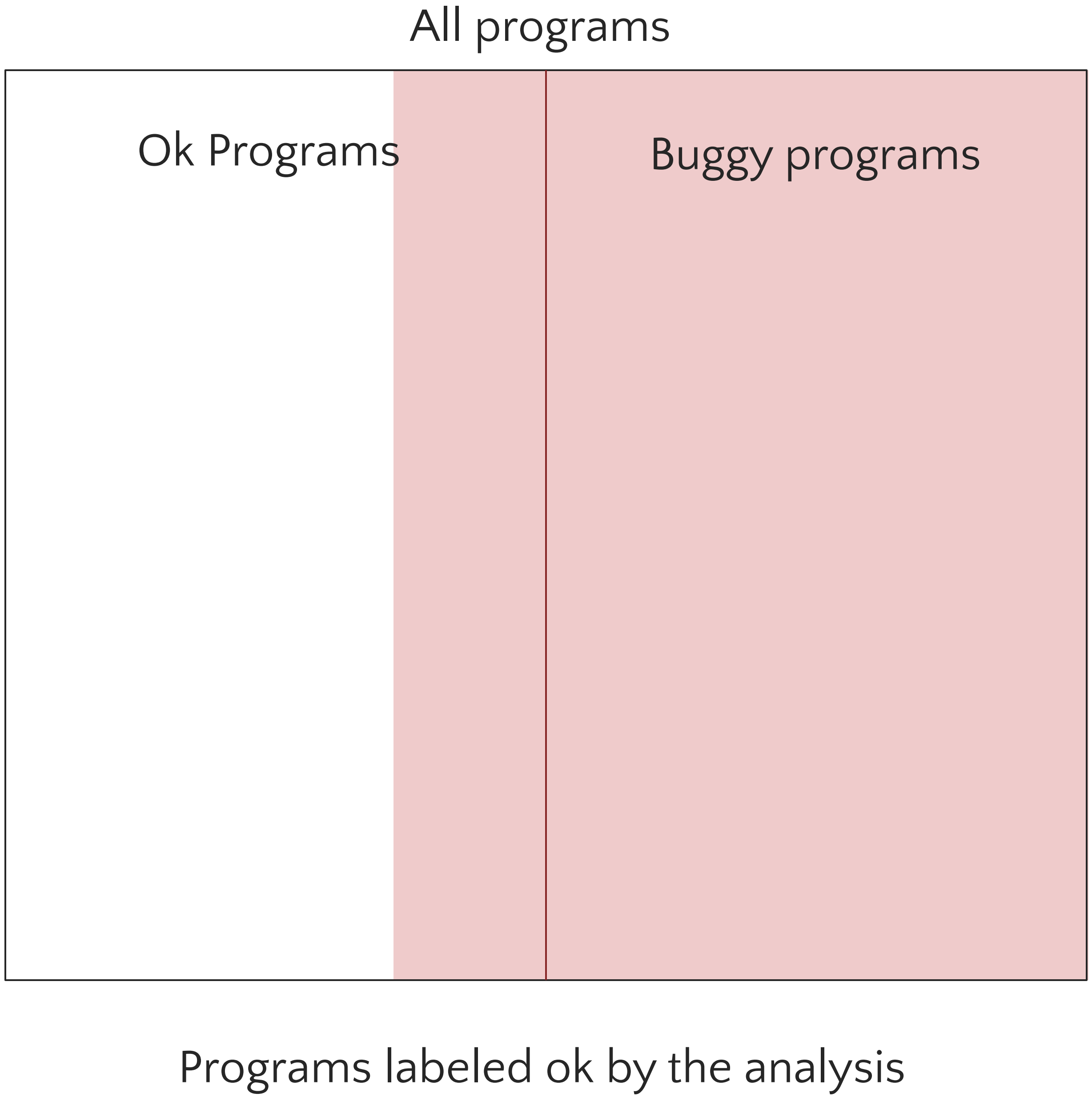
Soundness is an if-then statement, so we can also talk about sound buggy analysis.

Sound buggy analysis:
analysis(p) = buggy, then p is buggy

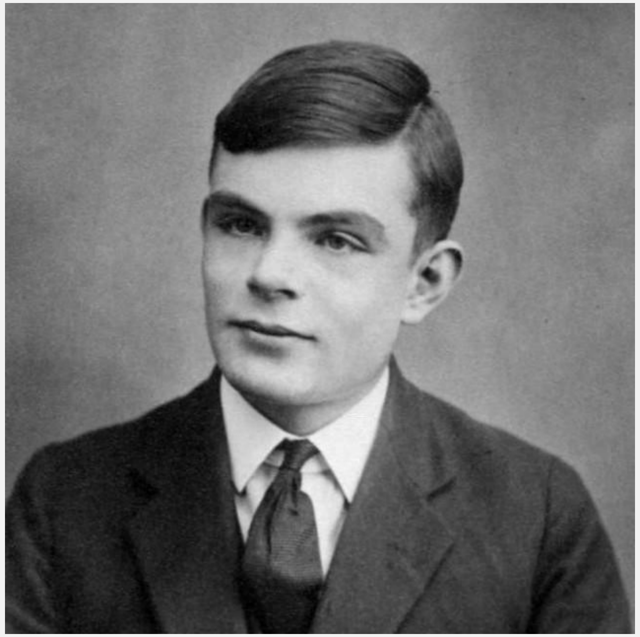


And complete buggy analysis

<u>Complete buggy analysis:</u>
If p is buggy, then analysis(p) = buggy



Halting Problem



Bad News?

Turing (1936): The **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. The halting problem is **undecidable**.

Rice Theorem

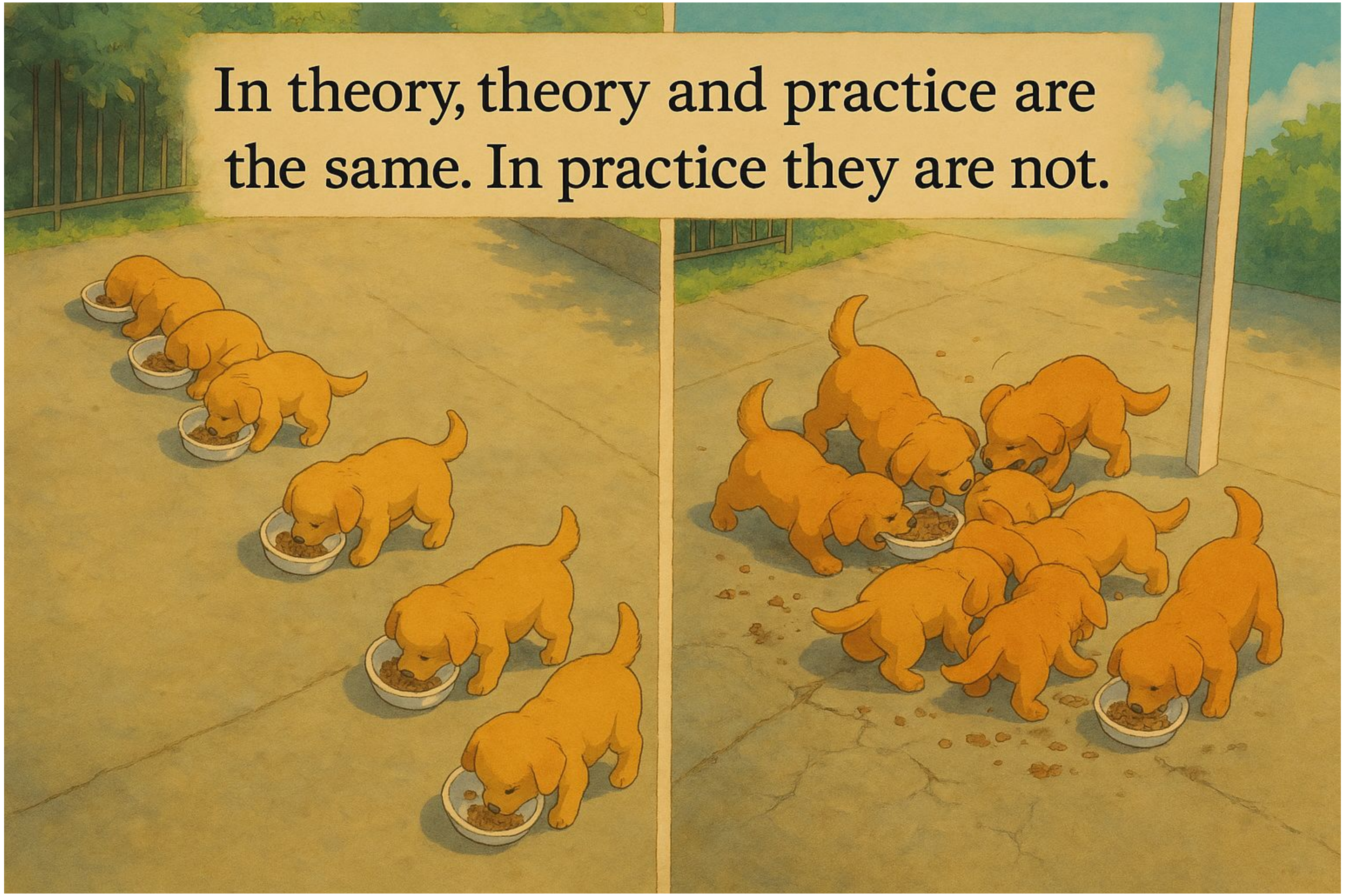
The Real Bad

News

Informal Statement of the Theory

There is no sound and complete analysis for any interesting (i.e., non-trivial) property for programs in Turing-complete languages.

Program Analysis: Making the Impossible, Possible



Control Flow Analysis

Basic Block

Defn Basic Block:

A consecutive sequence of instructions / code such that

- the instruction in each position always executes before (dominates) all those in later positions, and
- no outside instruction can execute between two instructions in the sequence

Note: dynamic analysis sometimes says "basic block" to mean no control flow change during an execution. Here we mean *statically*.

```
1. x = y + z
2. z = t + i
```

```
3. x = y + z
4. z = t + i
5. jmp 1
```

```
6. jmp 3
```

3 static basic blocks

execution is "straight"
(no jump targets except at the beginning,
no jumps except at the end)

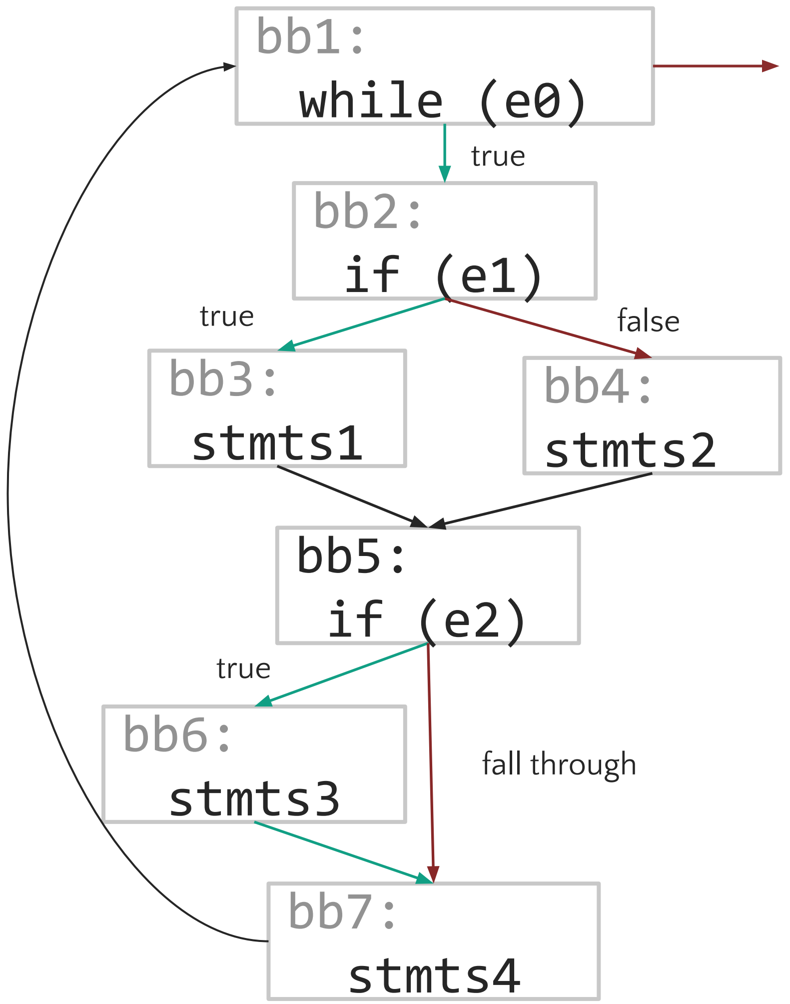
CFG Definition [Frances Allen - Turing Award 2006]

<u>Defn Control Flow Graph:</u>

A graph where

- each vertex bb_i is a (static) basic block, and
- there is an edge (bb_i, bb_j) if there ***may*** be a transfer of control from block bb_i to block bb_j.

Historically, the scope of a "CFG" is limited to a function or procedure, i.e., *<u>intra</u>*-procedural.



Call Graph

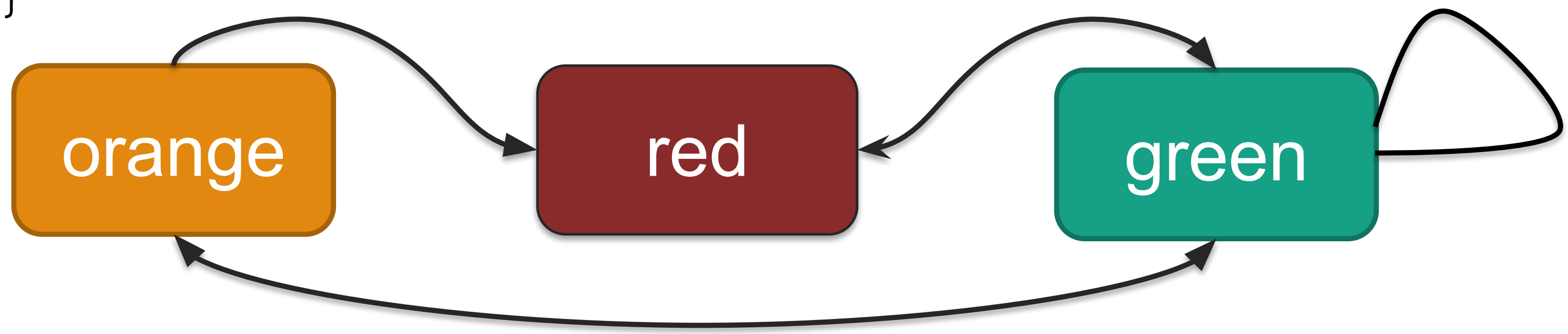
Defn Call Graph:

Nodes are functions. There is an edge (v_i, v_j) if function v_i calls function v_j

```
void orange()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
green();
...
}
```

```
void green()
{
  green();
  red();
}
```



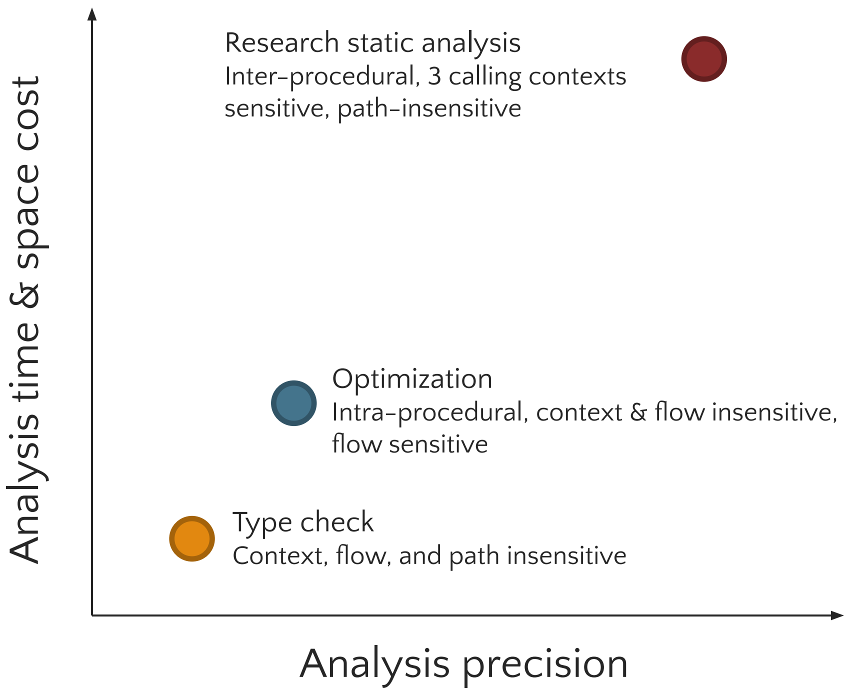
Note how different calls to red are not distinguished!
This is a source of imprecision. (More Later.)

Analysis-Precision Tradeoffs

Any static analysis chooses between:

- ***Intra vs Inter-procedural***
- ***Context*** (calling context) sensitive vs. insensitive
- ***Flow*** (CFG control flow) sensitive vs. insensitive
- ***Path*** (execution path flow) sensitive vs. insensitive

Trade off space examples



Context Sensitive Example

```
a = id(4);          void id(int z)
                    { return z; }
b = id(5);
```

Context-sensitive (color denotes matching call/ret)

Context sensitive can tell which call returns to which location
E.g., replace id(4) with value 4, id(5) with value 5.

Context Insensitive Example

```
a = id(4);          void id(int z)
                    { return z; }
b = id(5);
```

Context-insensitive (note merging)

Context insensitive will say both calls can return to both locations
E.g., id()□ {4,5}, so cannot safely optimize

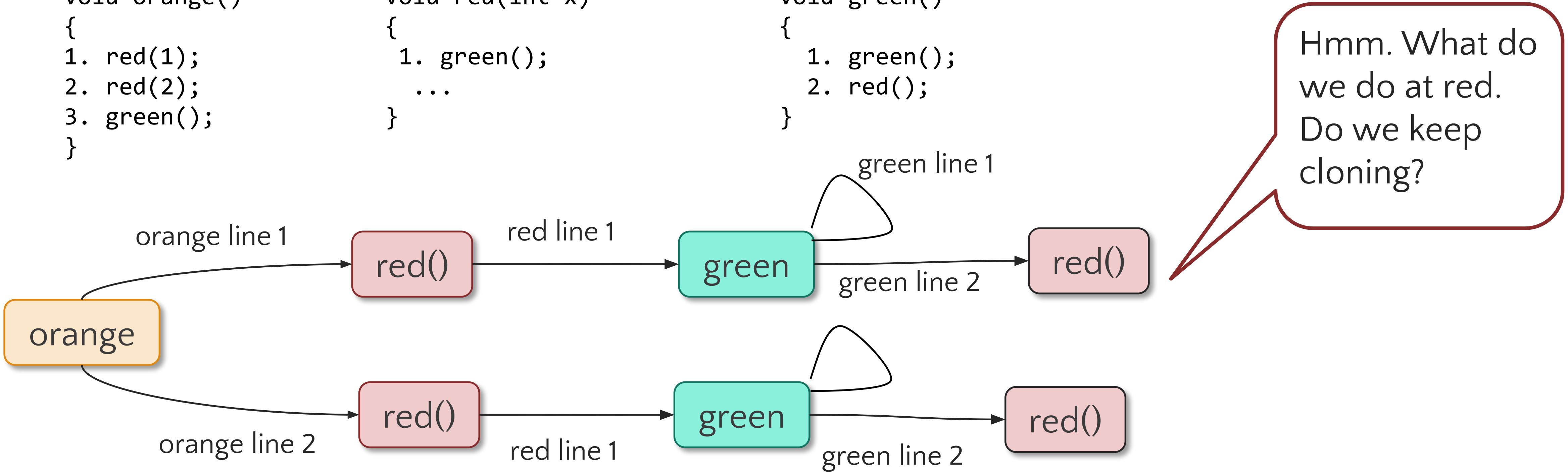
Intuition: Complete sensitivity is impossible

Analysis complexity comes from cloning results at each call.

```
void orange()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
 1. green();
  ...
}
```

```
void green()
{
  1. green();
  2. red();
}
```



Quiz Question

Consider the following definitions:
All possible statements: {a, b, c, d, e, f}
The true statements: {a, b, c}
Statements the analysis says are true: {a, b, c, d}

Which of the following is **TRUE**?

A. The analysis is sound and complete
B. The analysis is sound, but not complete
C. The analysis is complete, but not sound
D. The analysis is trivially complete

Type Safety Analysis

Types

- A *type* is a *specification* of data or code in a program
- Examples from C:
  - Basic types
    - `int, char, float, double, void`
    - `int x;` --- variable x will store an integer
  - Function types
    - `int -> int`
    - `int factorial(int);`
      factorial is a function that takes an integer as an argument and returns an integer:

Type safety is a little "ok" proof

- Type safety means the *running* program is guaranteed to manipulate values in a way that is compatible with its type
- Type safety checks are used to reject buggy programs:
  - Use strings as integers
  - Use integers as pointers
  - Cause null-pointer exceptions
  - Cause array overflows
  - Leak secret information
  - …

Type safety is an ok analysis

Type safety is proving just two theorems: progress and preservation

- Preservation: If the program is well-typed at step i, and then takes a step, it will be well-typed at step i+1
- Progress: If the program is well-typed at step i, it has either finished (safely) or it can take another step.

Dynamic Type Safety

Dynamic type safety: Preservation and progress are checked at runtime.

- Preservation: ensures that operations on values are type-safe as they occur.
- Progress: as long as the runtime type checks pass, the program can continue to execute and make progress.

Example: Python. Python checks that types don't change during execution, and will raise an exception if not. (Exceptions are well-typed.)

Static type safety

Static type safety: Preservation and progress are checked at compile time.

- Preservation: Guaranteed by the compiler, typically with a flow and context-insensitive analysis.
- Progress: as long as the compiler verifies the code, the program will execute to completion.

Example: Rust.

Java: Mix of both

Java will add dynamic bounds checks to arrays where it cannot prove them statically safe.

Absent, Weak, and Strong Static Typing

- Untyped languages don't have types (One type: Unityped). They are trivially type safe.
  - E.g., Bash, Perl, Ruby, …
  - Usually interpreted; difficult to compile without types
  - Program safety is programmer's responsibility: programs difficult to debug (really!)
- Weakly typed languages use types only for compilation; no type-safety
  - E.g., C, C++, etc.
  - Often allow unsafe casts, e.g., char[8] to char[] and int to char*
  - Program safety is programmer's responsibility: buffer overflows and segmentation faults are common in programs
- Strongly typed languages use types for compilation and guarantee type-safety
  - E.g., BASIC, Pascal, Cyclone, Haskell, SML, Java, etc.
  - No unsafe casts, e.g., an integer cannot be cast to a pointer, an array of length 8 is not an unbounded array, etc.
  - Safety is guaranteed but believed unsuitable for some low-level programs (debatable)

Buffer Overflow with scanf in C

This program is well-typed according to gcc, but can crash at runtime.

C is not type-safe

```
void readstring(char str[])
{
  char buf[8];
  scanf("%s",str);
}
```

User could provide input longer than 7 bytes, causing a buffer overflow
char [unlimited] != char[8]

Types simplify compilation

- Types are *necessary* to compile source code
  - What is the binary representation of variables?
    - `int x;` --- x is 4 bytes
    - `char x;` --- x is 1 byte
    - `int arr[8];` --- arr is 32 bytes
  - How to compute in assembly?
    - `int x; int y; x + y` □ add r1,r2
    - `float x; float y; x + y` □ fadd r1,r2
    - Difference is based on types

Types are used to compile code, but (usually) don't exist in the compiler's output

Type checking is a type of verification

no pun intended on the two uses of type

- Sufficiently rich type systems enable verification
  - Type-checking == Proof checking
- Example: Dependent types

```
x:int { x > 0 }
y:int { y % 2 == 0 }

method duplicate(input:array<int>{input!=null})
  returns (output:array<int>{array_equal(input,output)})
```

Most type systems are not this rich
□ Better automation

From Safety to Security

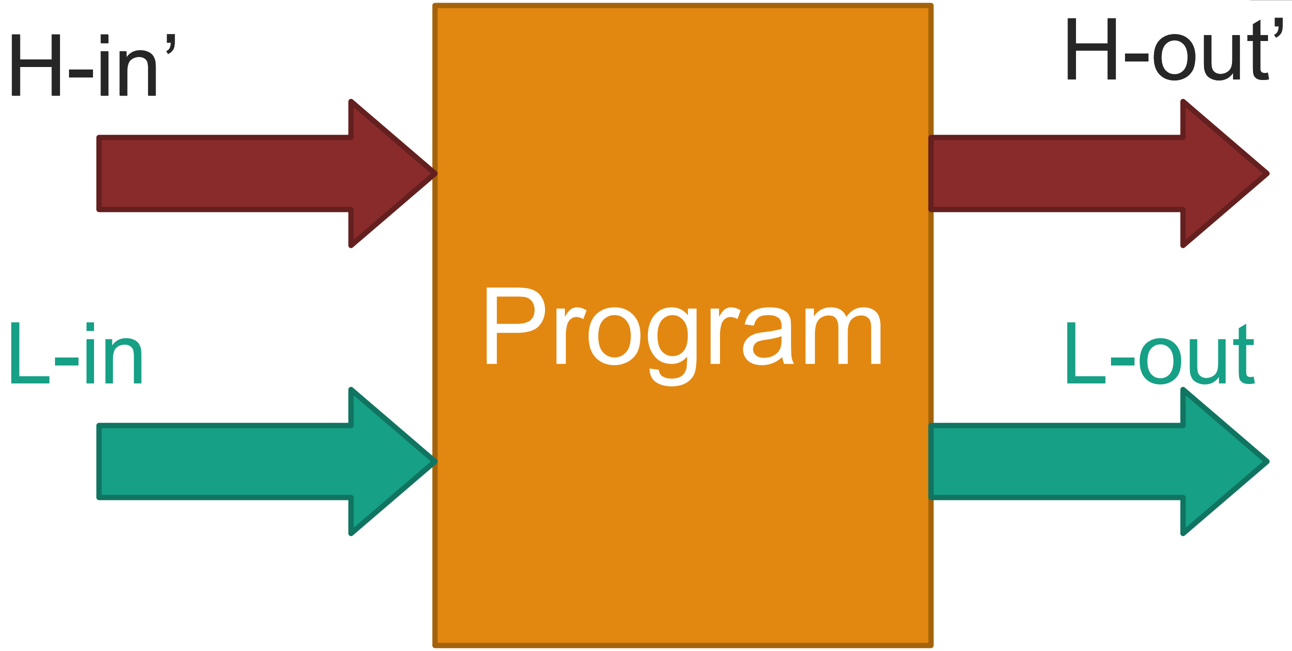
- Type safety in programming languages, despite the name, does not mean security safety.
- But types can be used to specify and enforce ***security*** properties
- Classic example: Information flow control
- Possible policy: Non-interference
  - Secret inputs cannot "interfere" with public outputs

Defining Security via Non-Interference

Security levels:

H: Classified

L: Unclassified



No information flows from high inputs to low outputs

Example

```
if x = 1 then
  y:=1
else
  y:=0
```

x	*y*	*NI*
H	H	Yes
L	L	Yes
H	L	No
L	H	Yes

Specification and Enforcement

- Approach
  - Use a typed programming language
  - Types represent *security levels*
    - H, L, …
  - Sub-typing captures partial order among security levels
    - $L \leq H$
  - Type system captures allowed information flows
  - Soundness theorem
    - Well-typed programs satisfy non-interference

Summary of Types

- Types are specifications of data and code
- Compiler may check well-typedness without executing the program
- Existence of type specifications may imply program safety (type-safety)
- Types can potentially specify deep program properties
- Not all languages with types are type-safe
  - E.g., C is not type-safe

Control Flow Integrity

Control-Flow Integrity: Principles, Implementation and Applications
by Abadi, Budiu, Erlingsson, and Ligatti

Control Flow Integrity

- **protects against powerful adversary**
  - with full control over entire $\boxed{\text{data memory}}$
- **widely-applicable**
  - language-neutral; support features; requires binary only
- **provably-correct & trustworthy**
  - formal semantics; small verifier
- **efficient**
  - hmm… 0–45% in experiments; average 16%

ADVERSARY CAN

Overwrite any data memory at any time, including stack, heap, data segments

ADVERSARY CANNOT

- Execute data (DEP enabled)
- Modify Code (.text RO)
- Write to %rip
- Overwrite registers in other contexts

ANALYSIS ASSUMES

- All code compiled w/ CFI
  (3rd party libraries problem)
- Compiler CFG is accurate
  (CFG both under and over approx. control flow)

CFI Overview

Invariant: Execution must follow a path in a $\underbrace{\text{control flow graph (CFG)}}_{\text{"static"}}$ created ahead of run time.

High-level method:

- build CFG statically, e.g., at compile time
- instrument (rewrite) binary, e.g., at install time
  - Add checks before each control transfer
- verify CFI instrumentation at load time
  - Make sure right checks are present and cannot be bypassed
- perform checks at run time
  - Crash/halt if checks are violated

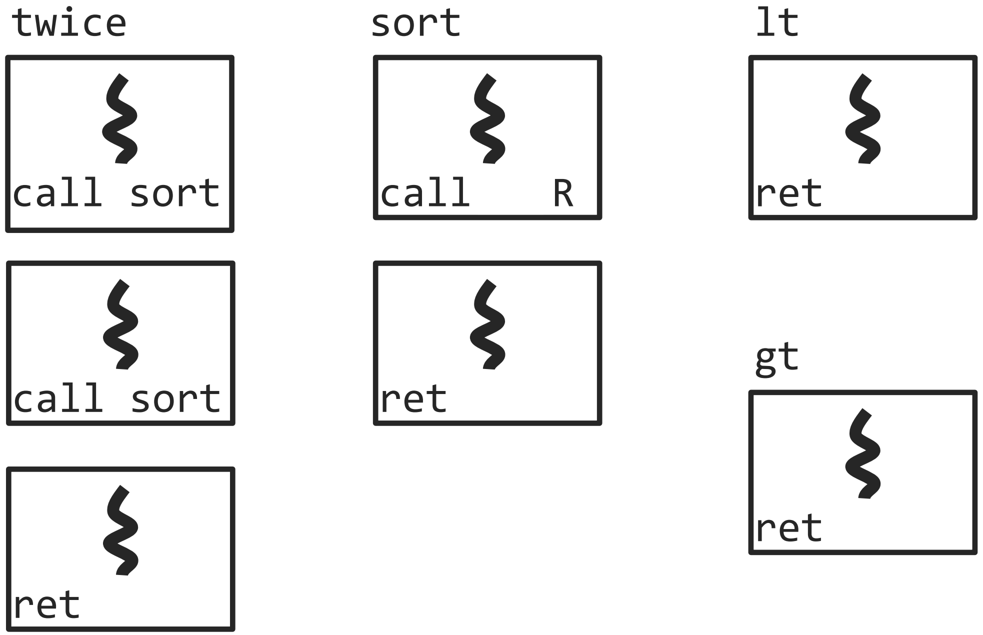
Ah, the Caveat

CFI accuracy is limited by how accurately we can statically understand runtime semantics. It will, after all, essentially be enforcing statically determined control flow.

It's important because the strong theoretical model means we "just" have to worry about implementation.

Build CFG: Basic Blocks

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

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

void sort(int w[], int len,
     bool (*sorted)(int, int))
{...}

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

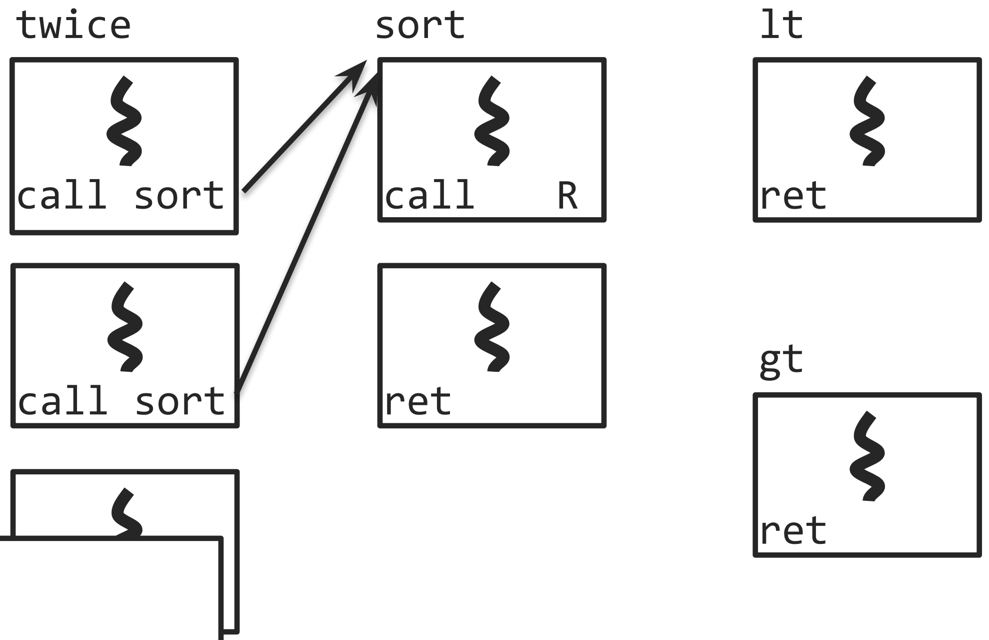


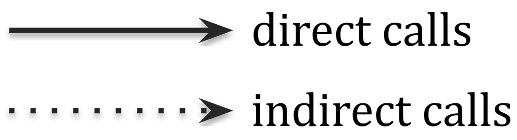
Build CFG: Forward Edges

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

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

void sort(int w[], int len,
          nt, int))
```

-230 in 2s complement

```
twice:
100000ed0: 55   push  %rbp
...
100000ef1: e8 1a ff ff ff     callq  -230 <sort>
...
100000f04: e8 07 ff ff ff     callq  -249 <sort>
```

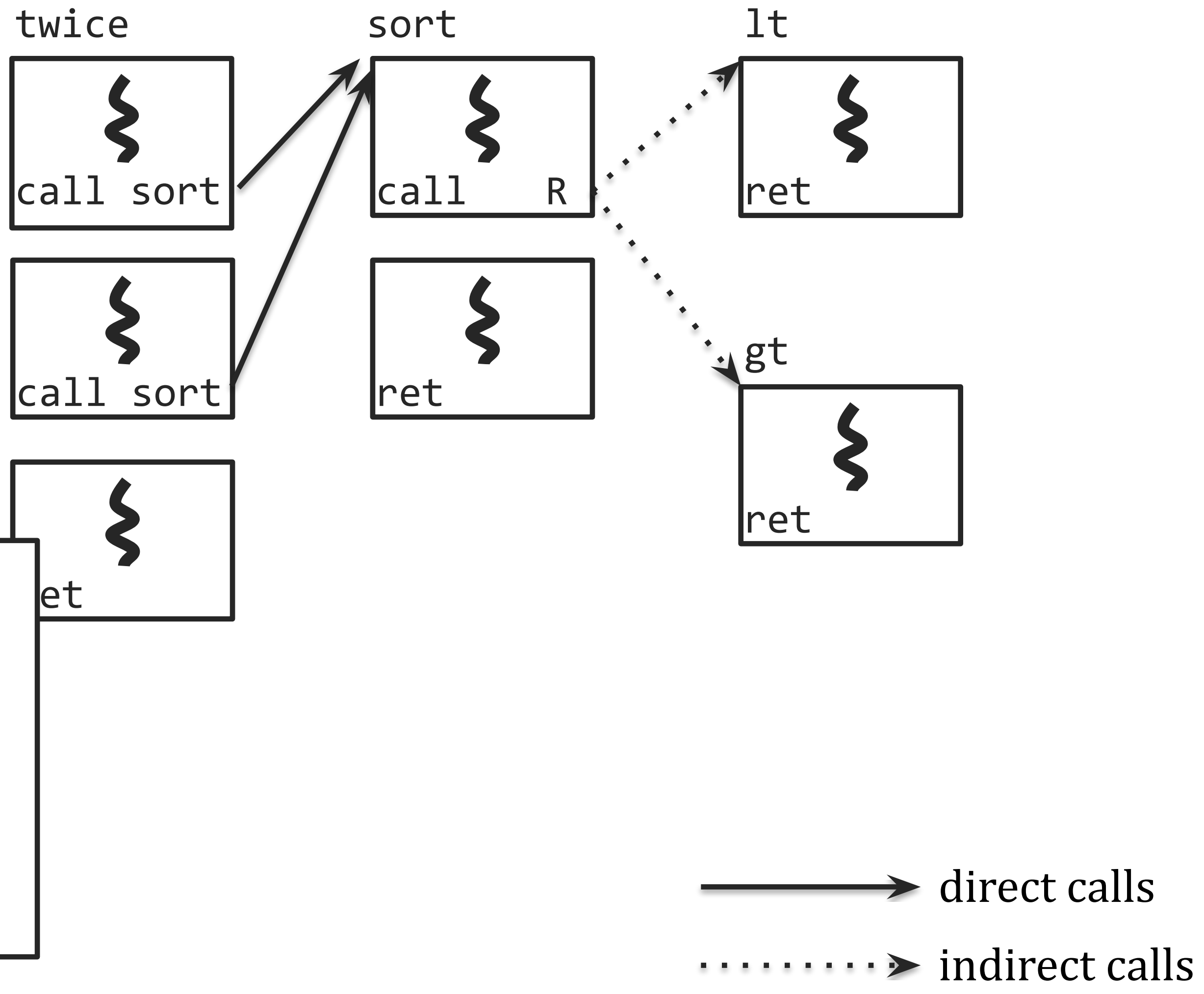




Build CFG: Forward Edges

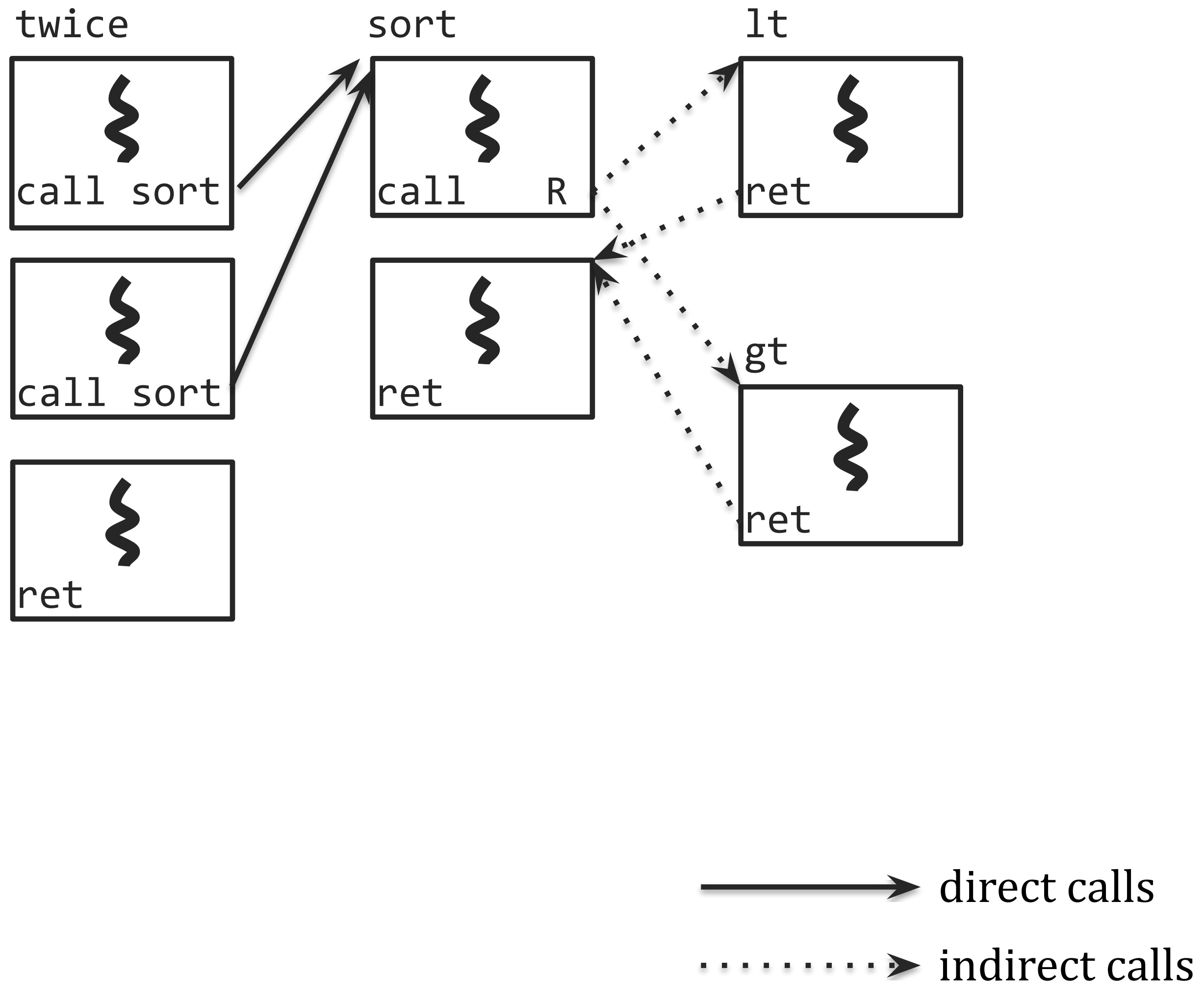
```
bool lt(int x, int y) {
  return x < y;
}

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

void sort(int w[], int len,
     bool (*sorted)(int, int))
{...}
```

```
sort:
100000e10: 55  pushq %rbp
...
100000e59: ff d0   callq *%rax
...
100000ec6: c3  ret
```



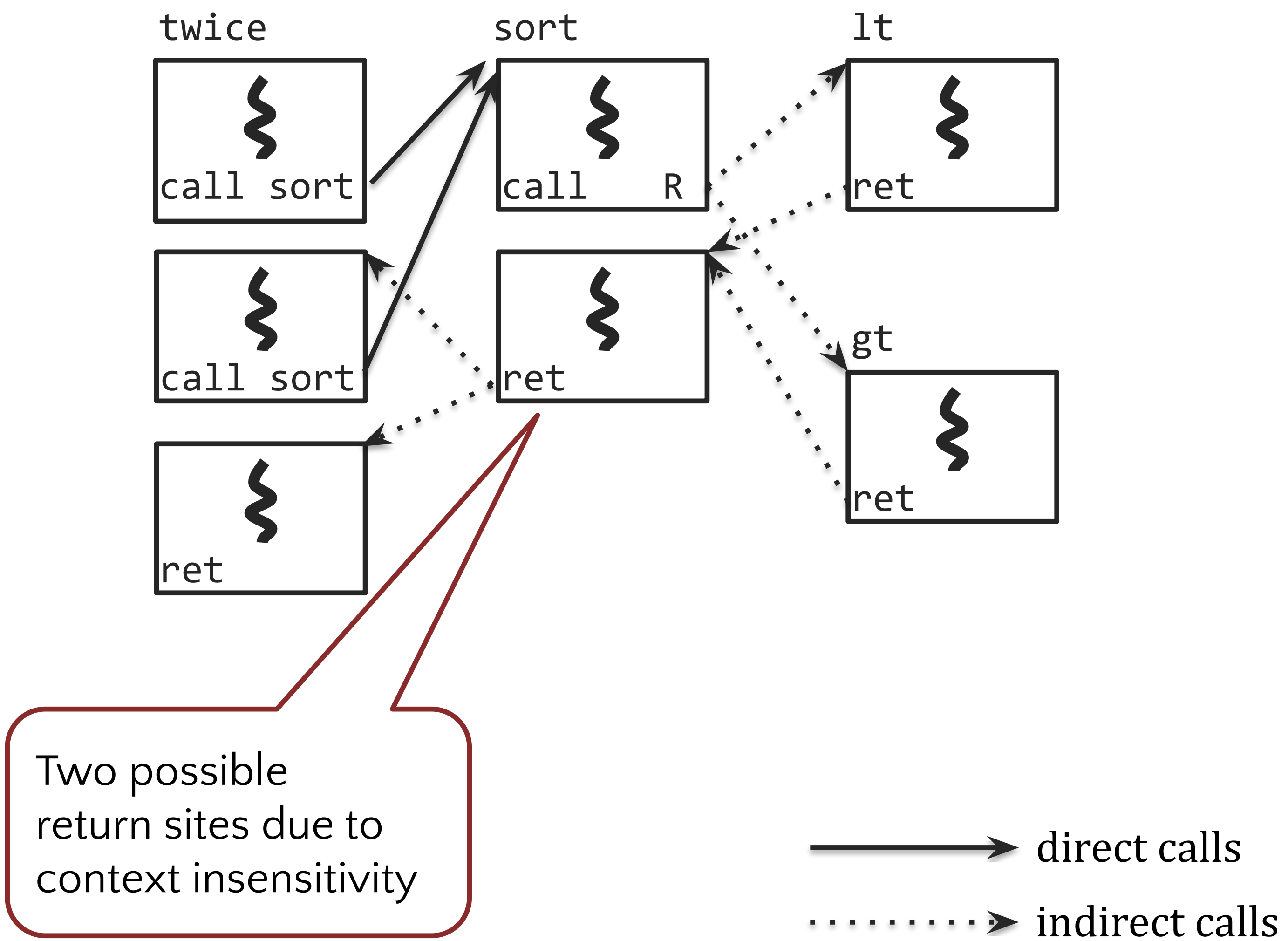
Build CFG: Backward Edges

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

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

void sort(int w[], int len,
     bool (*sorted)(int, int))
{...}

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



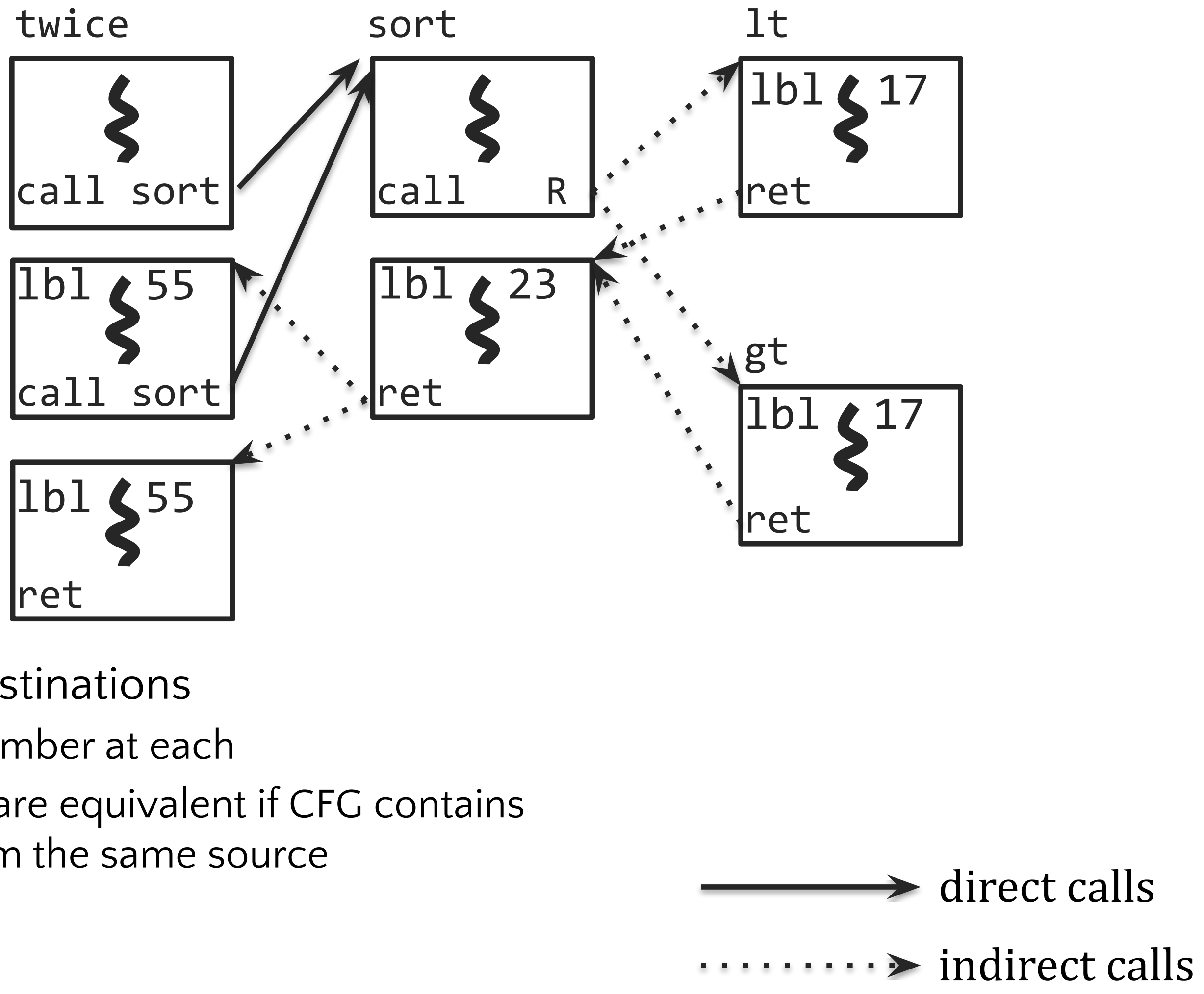
Build CFG: Backward Edges

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

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

void sort(int w[], int len,
     bool (*sorted)(int, int))
{...}

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



Instrument Binary: Labels

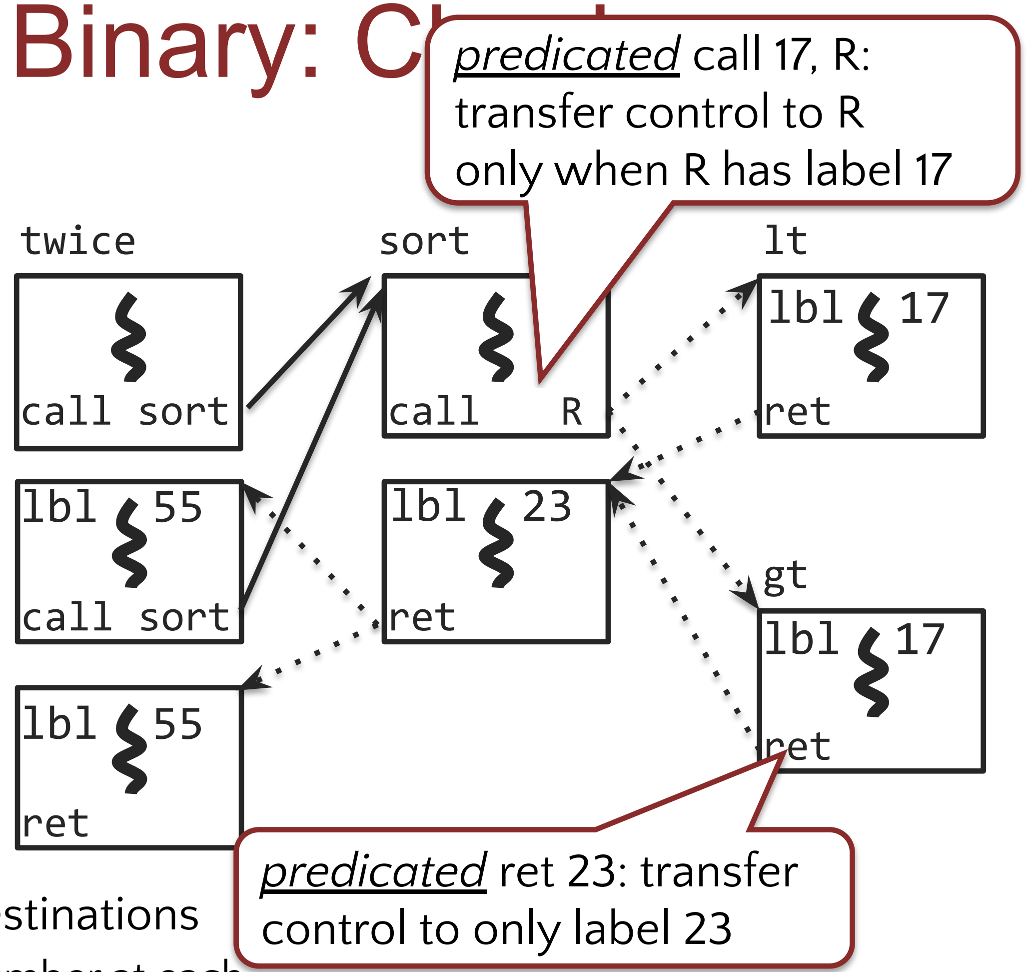
```
bool lt(int x, int y) {
  return x < y;
}

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

void sort(int w[], int len,
     bool (*sorted)(int, int))
{...}

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



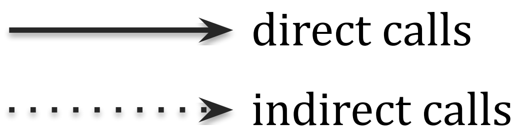
1. Label dynamic destinations
   – Insert a unique number at each
   – Two destinations are equivalent if CFG contains edges to each from the same source

Instrument Binary: Ch

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

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

void sort(int w[], int len,
     bool (*sorted)(int, int))
{...}

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



1. Label dynamic destinations
   – Insert a unique number at each
   – Two destinations are equivalent if CFG contains edges to each from the same source
2. Add checks at each control transfer



Example of Instrumentation

Original code

Source

```
Opcode bytes          Instructions
FF E1                 jmp   ecx                  ; computed jump
```

Destination

```
Opcode bytes          Instructions
8B 44 24 04           mov   eax, [esp+4]       ; dst
```

Instrumented code

```
B8 77 56 34 12      mov   eax, 12345677h       ; load ID-1
40                  inc   eax                  ; add 1 for ID
39 41 04            cmp   [ecx+4], eax         ; compare w/dst
75 13               jne   error_label          ; if != fail
FF E1               jmp   ecx                  ; jump to label
```

```
3E 0F 18 05         prefetchnta                ; label
78 56 34 12             [12345678h]            ;    ID
8B 44 24 04         mov   eax, [esp+4]         ; dst
...
```

Jump to the destination only if the tag is equal to “12345678”

Abuse an x86 assembly instruction to insert “12345678” tag into the binary

Verify CFI Instrumentation

- **Jump targets** (e.g. `call 0x12345678`)
  - are all targets valid according to CFG?
- **IDs**
  - is there an ID right after every entry point?
  - does any ID appear in the binary by accident?
- **ID Checks**
  - is there a check before every control transfer?
  - does each check respect the CFG?

Performance in 2005

Size: increase 8% avg

Time: increase 0–45%; 16% avg

– I/O latency helps hide overhead

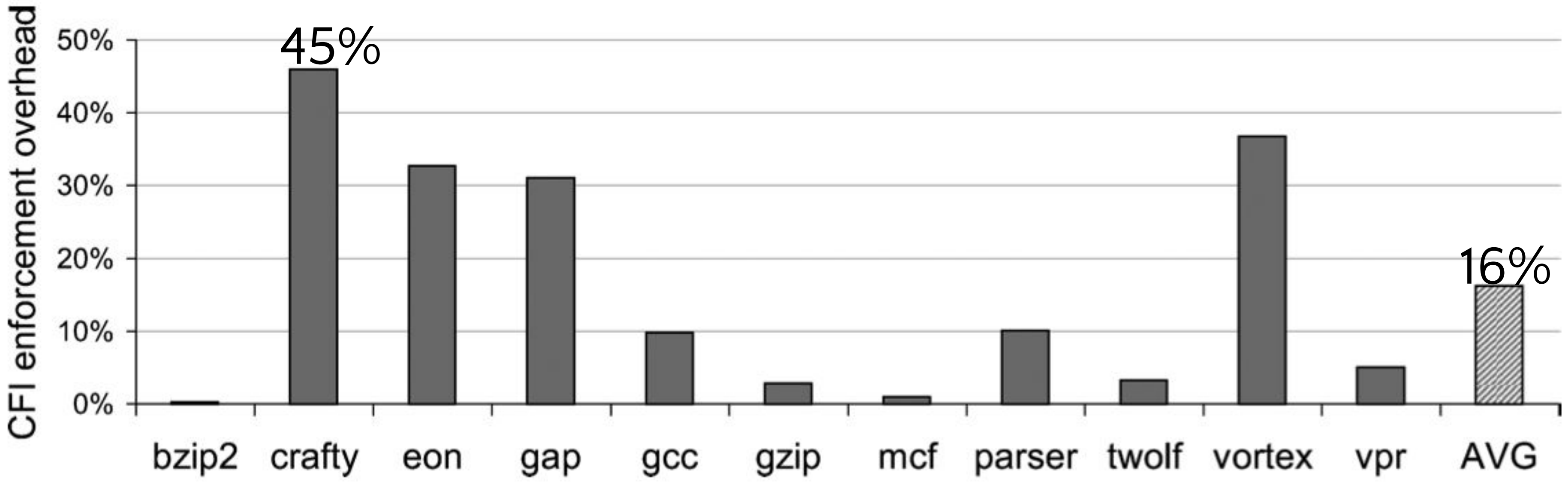
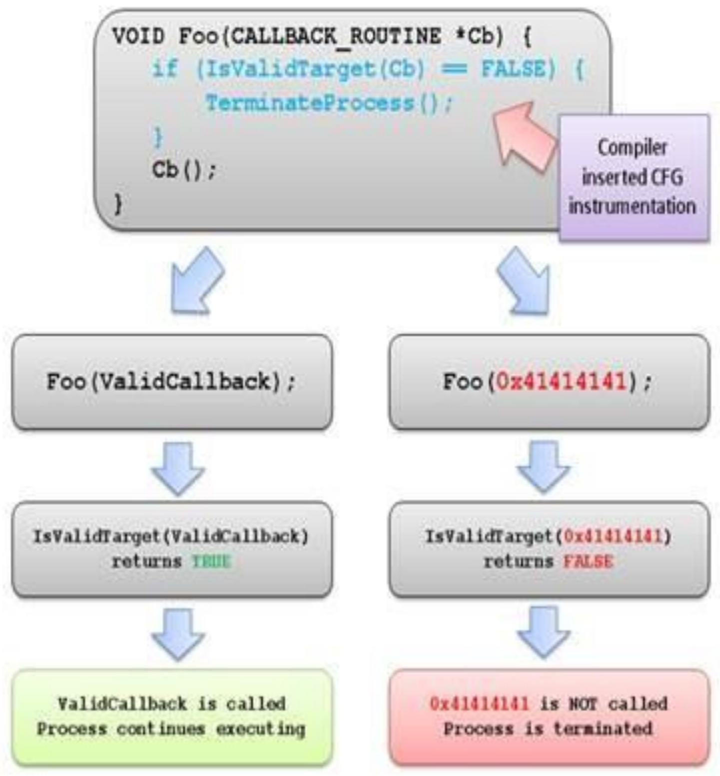


Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

Fast Forward to 2020

- CFI introduced in CCS 2005
- Many new developments since then
  - E.g., “Coarse-grained” CFI
- Deployments:
  - Forward edge
    - gcc >= 4.9, llvm >=3.7, msvc >= 2015 & win >= 8.1
  - Backward edge
    - Software implementations of shadow stack
    - Intel Control-flow Enforcement Technology (CET)
    - MS Control Flow Guard

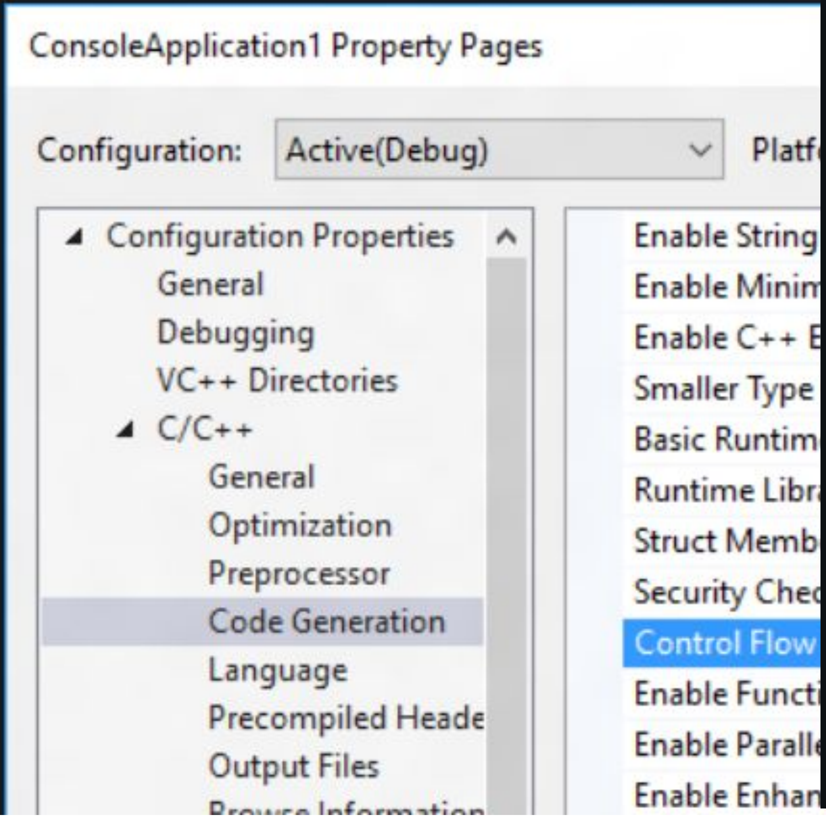


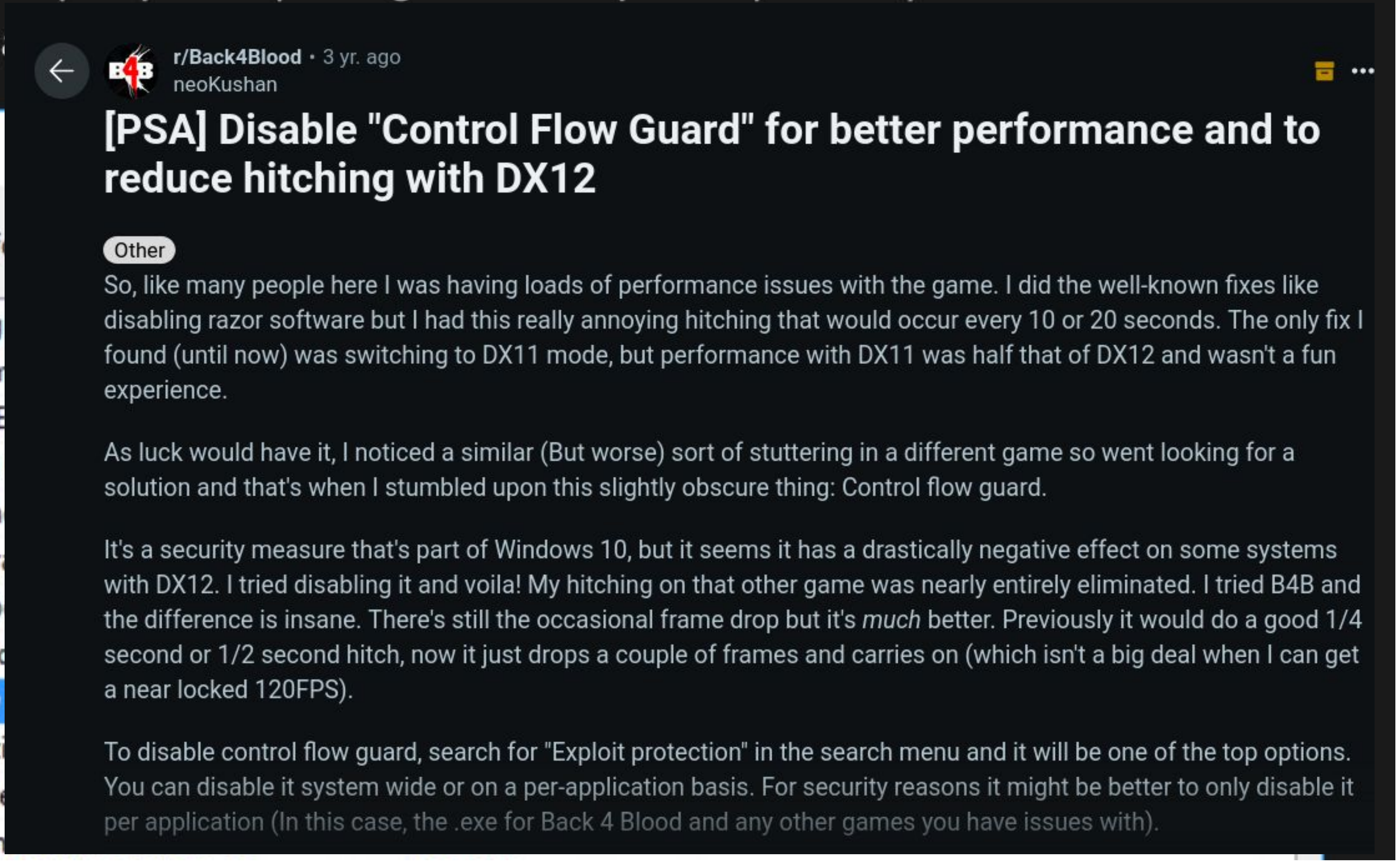
MS Control Flow Guard

How Can I Enable CFG?

In most cases, there's no need to change source code. All you have to do is add an option to your Visual Studio project, and the compiler and linker will enable CFG.

The simplest method is to navigate to **Project | Properties | Configuration Properties | C/C++ | Code Generation** and choose **Yes (/guard:cf)** for Control Flow Gu





r/Back4Blood • 3 yr. ago
neoKushan

[PSA] Disable "Control Flow Guard" for better performance and to reduce hitching with DX12

Other

So, like many people here I was having loads of performance issues with the game. I did the well-known fixes like disabling razor software but I had this really annoying hitching that would occur every 10 or 20 seconds. The only fix I found (until now) was switching to DX11 mode, but performance with DX11 was half that of DX12 and wasn't a fun experience.

As luck would have it, I noticed a similar (But worse) sort of stuttering in a different game so went looking for a solution and that's when I stumbled upon this slightly obscure thing: Control flow guard.

It's a security measure that's part of Windows 10, but it seems it has a drastically negative effect on some systems with DX12. I tried disabling it and voila! My hitching on that other game was nearly entirely eliminated. I tried B4B and the difference is insane. There's still the occasional frame drop but it's *much* better. Previously it would do a good 1/4 second or 1/2 second hitch, now it just drops a couple of frames and carries on (which isn't a big deal when I can get a near locked 120FPS).

To disable control flow guard, search for "Exploit protection" in the search menu and it will be one of the top options. You can disable it system wide or on a per-application basis. For security reasons it might be better to only disable it per application (In this case, the .exe for Back 4 Blood and any other games you have issues with).

ADVERSARY CAN

Overwrite any data memory at any time, including stack, heap, data segments

ADVERSARY CANNOT

- Execute data (DEP enabled)
- Modify Code (.text RO)
- Write to %rip
- Overwrite registers in other contexts

ANALYSIS ASSUMES

- All code compiled w/ CFI
  (3rd party libraries problem)
- Compiler CFG is accurate
  (CFG both under and over approx. control flow)

Assumptions are often vulnerabilities or limitations!

CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software

Xiaoyang Xu, *University of Texas at Dallas*
Masoud Ghaffarinia*, *University of Texas at Dallas*
Wenhao Wang*, *University of Texas at Dallas*
Kevin W. Hamlen, *University of Texas at Dallas*
Zhiqiang Lin, *Ohio State University*

Abstract

CONFIRM (CONtrol-Flow Integrity Relevance Metrics) is a new evaluation methodology and microbenchmarking suite for assessing compatibility, applicability, and relevance of *control-flow integrity* (CFI) protections for preserving the intended semantics of software while protecting it from abuse. Although CFI has become a mainstay of protecting certain classes of software from code-reuse attacks, and continues to be improved by ongoing research, its ability to preserve intended program functionalities (*semantic transparency*) of diverse, mainstream software products has been under-studied in the literature. This is in part because although CFI solutions are evaluated in terms of performance and security, there remains no standard regimen for assessing compatibility. Researchers must often therefore resort to anecdotal assessments, consisting of tests on homogeneous software collections with limited variety (e.g., GNU Coreutils), or on CPU benchmarks (e.g., SPEC) whose limited code features are not representative of large, mainstream software products.

Reevaluation of CFI solutions using CONFIRM reveals that there remain significant unsolved challenges in securing many large classes of software products with CFI, including software for market-dominant OSes (e.g., Windows) and code employing certain ubiquitous coding idioms (e.g., event-driven callbacks and exceptions). An estimated 47% of CFI-relevant code features with high compatibility impact remain incompletely supported by existing CFI algorithms, or receive weakened controls that leave prevalent threats unaddressed (e.g., return-oriented programming attacks). Discussion of these open problems highlights issues that future research must address to bridge these important gaps between CFI theory and practice.

1 Introduction

Control-flow integrity (CFI) [1] (supported by vtable protection [29] and/or software fault isolation [73]), has emerged as one of the strongest known defenses against modern control-flow hijacking attacks, including return-oriented programming (ROP) [60] and other code-reuse attacks. These attacks trigger dataflow vulnerabilities (e.g., buffer overflows) to manipulate control data (e.g., return addresses) to hijack victim software. By restricting program execution to a set of legitimate control-flow targets at runtime, CFI can mitigate many of these threats.

Inspired by the initial CFI work [1], there has been prolific new research on CFI in recent years, mainly aimed at improving performance, enforcing richer policies, obtaining higher assurance of policy-compliance, and protecting against more subtle and sophisticated attacks. For example, between 2015–2018 over 25 new CFI algorithms appeared in the top four applied security conferences alone. These new frameworks are generally evaluated and compared in terms of performance and security. Performance overhead is commonly evaluated in terms of the CPU benchmark suites (e.g., SPEC), and security is often assessed using the RIPE test suite [80] or with manually crafted proof-of-concept attacks (e.g., COOP [62]). For example, a recent survey systematically compared various CFI mechanisms against these metrics for precision, security, and performance [13].

While this attention to performance and security has stimulated rapid gains in the ability of CFI solutions to efficiently enforce powerful, precise security policies, less attention has been devoted to systematically examining which general classes of software can receive CFI protection without suffering compatibility problems. Historically, CFI research has struggled to bridge the gap between theory and practice (cf., [84]) because code hardening transformations inevitably run at least some risk of corrupting desired, policy-permitted program functionalities. For example, introspective programs that read their own code bytes at runtime (e.g., many VMs, JIT compilers, hot-patchers, and dynamic linkers) can break after their code bytes have been modified or relocated by CFI.

Compatibility issues of this sort have dangerous security ramifications if they prevent protection of software needed in mission-critical contexts, or if the protections must be weak-

*These authors contributed equally to this work.

ConFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software . USENIX 2019

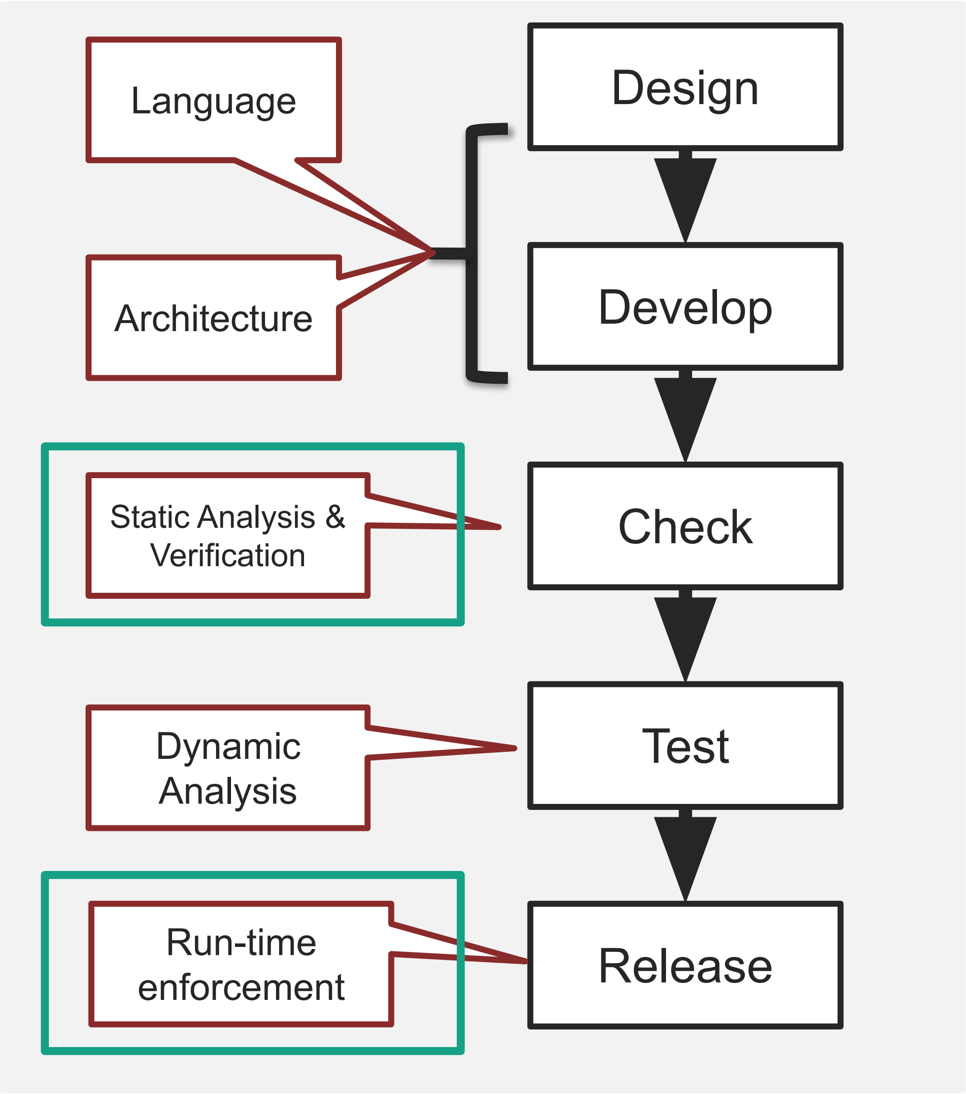
Reevaluation of CFI solutions using CONFIRM reveals that there remain significant unsolved challenges in securing many large classes of software products with CFI, including software for market-dominant OSes (e.g., Windows) and code employing certain ubiquitous coding idioms (e.g., event-driven callbacks and exceptions). An estimated 47% of CFI-relevant code features with high compatibility impact remain incompletely supported by existing CFI algorithms, or receive weakened controls that leave prevalent threats unaddressed (e.g., return-oriented programming attacks). Discussion of these open problems highlights issues that future research must address to bridge these important gaps between CFI theory and practice.

Summary: CFI

- Provides a strong theoretic guarantee against a strong attacker model
- Relies on the precision of static analysis to insert runtime checks
- Theoretic guarantees are assumptions, and not easy to satisfy with fully featured real systems.

Key Takeaways!

- Many techniques exist to create more secure software
- Understand the tradeoffs



Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!