Automated Security Testing

CS155 Computer and Network Security

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh at Stanford University. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

Stanford University

Fuzzing

Fuzzing

Form of vulnerability analysis:

1. Feed large number of random anomalous test cases into program
2. Monitor for crashes or unexpected program behavior

    Some kinds of errors can be used to find an exploit

Commonly used to test file parsers (e.g., PDF readers) and network protocols

HTTP Fuzzing Example

Standard HTTP GET Request

```
GET /index.html HTTP/1.1
```

Anomalous Requests

```
GEEEE…EET /index.html HTTP/1.1
GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1
```

HTTP Fuzzing Example

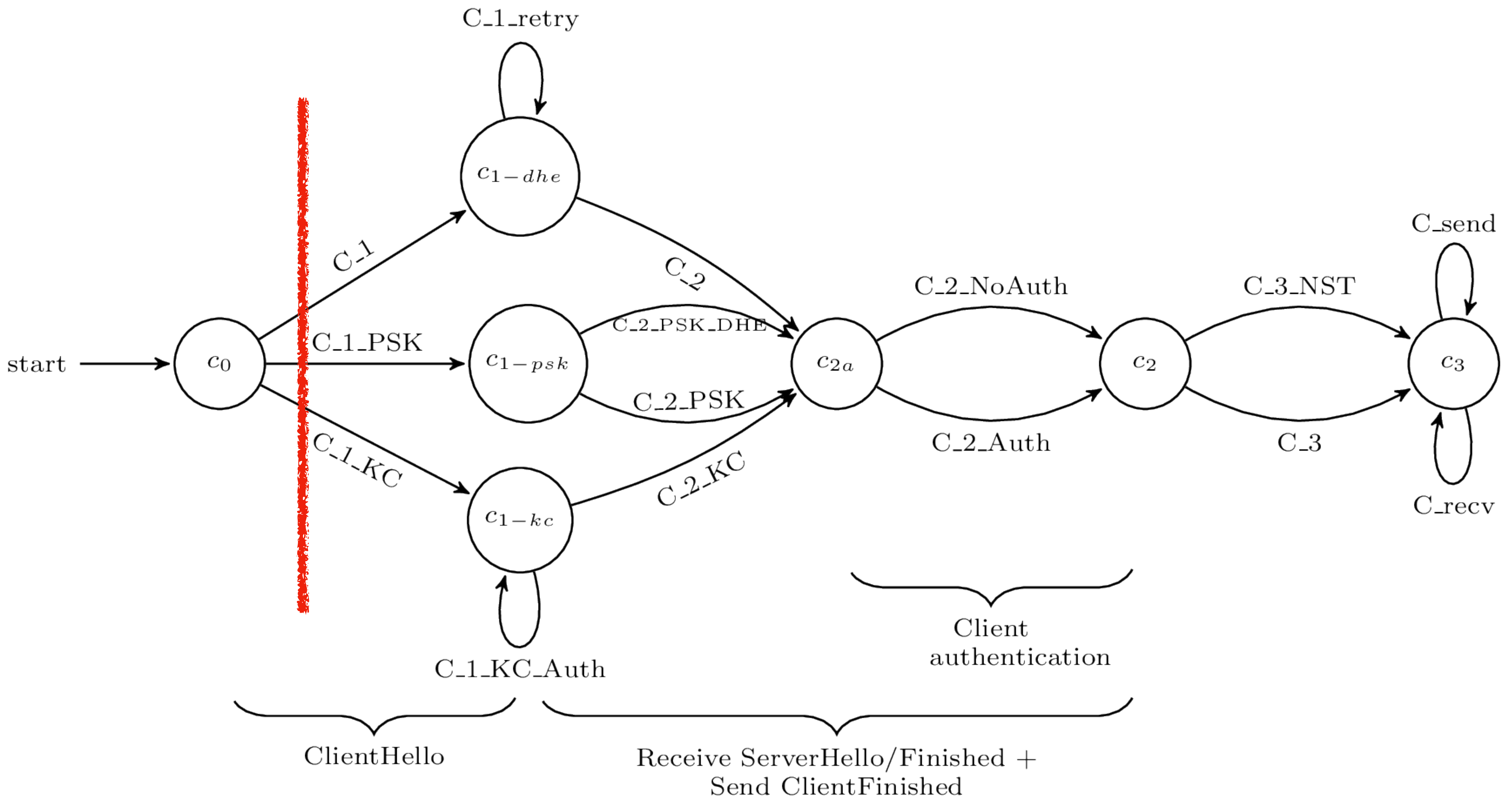
Standard HTTP GET Request

```
GET /index.html HTTP/1.1
```

Anomalous Requests

```
GEEEE…EET /index.html HTTP/1.1
GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAA.html HTTP/1.1
df%w3rasd8#r78jskdflasdjf
4isg8swksdfskdflsdgmsf$gkjs
```

Random Fuzzing



TLS 1.3 State Diagram

Types of Fuzzing

Mutation-based (Dumb) fuzzing

- Add anomalies to existing good inputs (e.g., test suite)

Generative (Smart) fuzzing

- Generate inputs from specification of format, protocol, etc

Evolutionary (Responsive) fuzzing

- Leverage program instrumentation, code analysis
- Use response of program to build input set

Mutation-Based Fuzzing

Basic Idea

- Take known good input and add anomalies
- Anomalies may be completely random or follow some heuristics
  - Large integers or strings
  - Randomly flip bits

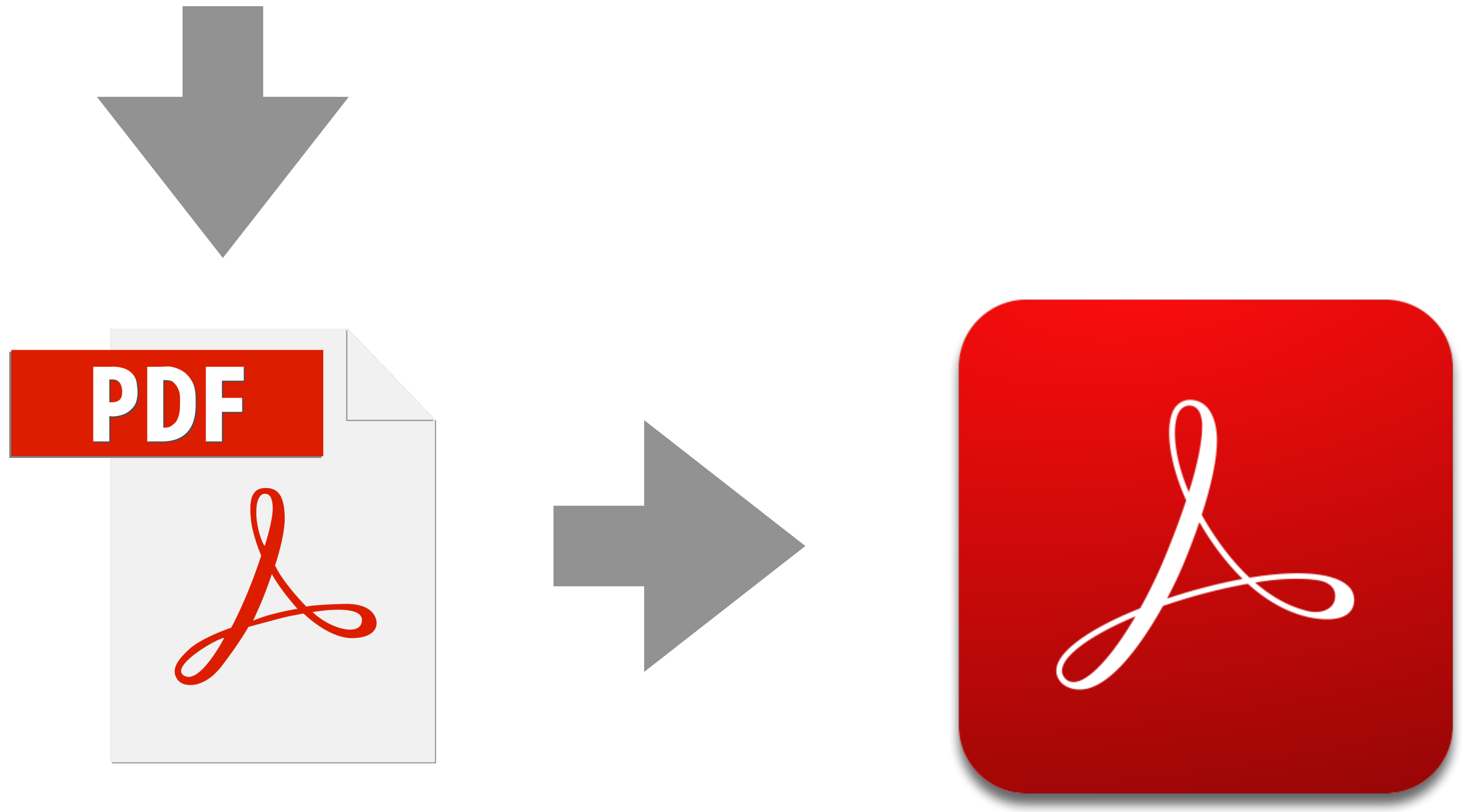
Fuzzing PDF Reader

Download 100s of random PDF files

Mutate content in the PDF file:

- flip bits
- increase size of integers or strings
- remove data

Limited by the functionality that the existing files happened to use — unlikely to hit less commonly tested code paths



Mutation-Based Fuzzing

Basic Idea

- Take known good input and add anomalies
- Anomalies may be completely random or follow some heuristics

Advantages

- Little or no knowledge of the structure of the inputs is assumed
- Requires little to no set up time

Disadvantages

- Dependent on the inputs being modified
- May fail for protocols with checksums, challenge-response, etc.

Generation Based Fuzzing

Basic Idea

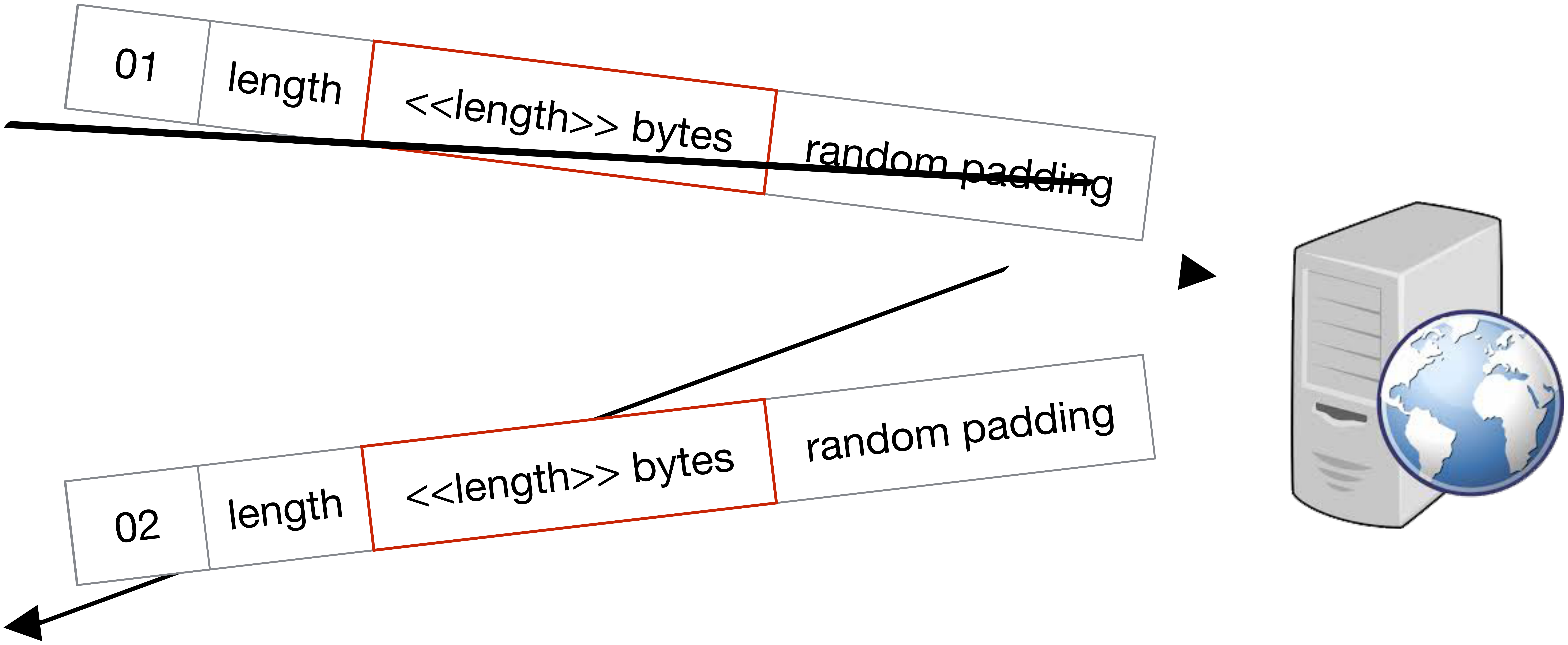
Test cases are generated from protocol description: RFC, spec, etc.

Anomalies are added to each possible spot in the inputs

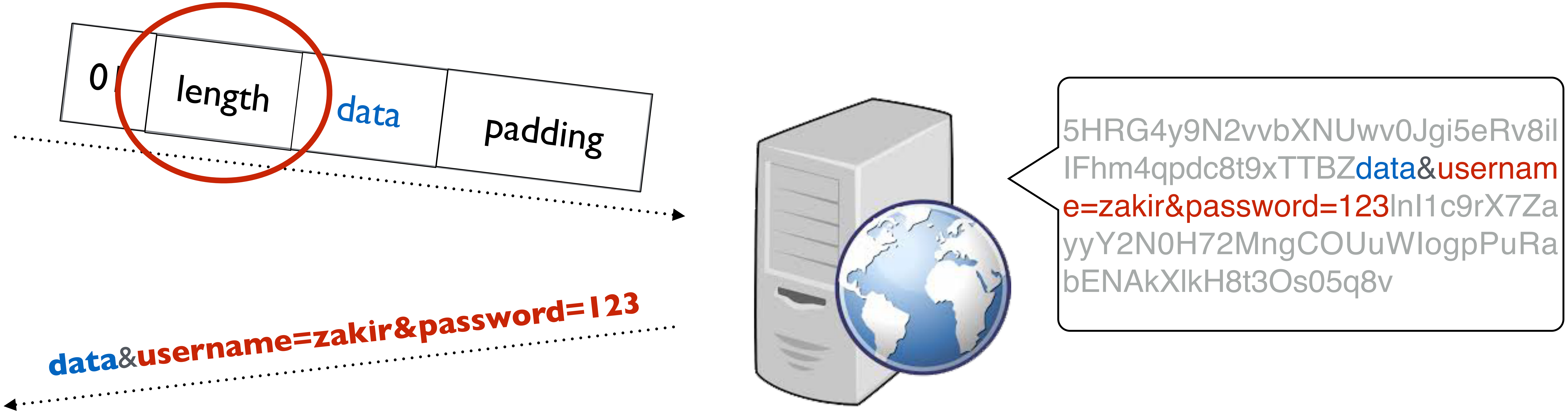
Generation Example

```
<!--  A.  Local file header -->
  <Block name="LocalFileHeader">
    <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
    <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
    ...
    [truncated for space]
    ...
    <Number name="lfh_CompSize" size="32" endian="little" signed="false">
      <Relation type="size" of="lfh_CompData"/>
    </Number>
    <Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
    <Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FileName"/>
    </Number>
    <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FldName"/>
    </Number>
    <String name="lfh_FileName"/>
    <String name="lfh_FldName"/>
    <!--  B.  File data -->
    <Blob name="lfh_CompData"/>
  </Block>
```

Generation Example: TLS Heartbeat



Generation Example: TLS Heartbeat



Heartbleed Vulnerability: server trusts user provided length field and echoes back memory contents following request data

Generation Based Fuzzing

Basic Idea

Test cases are generated from protocol description: RFC, spec, etc.

Anomalies are added to each possible spot in the inputs

Advantages

Knowledge of protocol may give better results than random fuzzing

Disadvantages

Can take *significant* time to set up. Requires understanding spec

Can you find anything with “dumb” fuzzing?

Charlie Miller’s 5 Lines

In 2010, Charlie Miller fuzzed Adobe Acrobat, Apple Preview, Powerpoint, and Open Office by downloading PDF and PPT files and five lines of simple fuzzing:

```
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
    rbyte = random.randrange(256)
    rn = random.randrange(len(buf))
    buf[rn] = "%c"%(rbyte)
```

Charlie Miller's 5 Lines

Collect a large number of pdf files

- Aim to exercise all features of pdf readers
- Found 80,000 PDFs on Internet

Reduce to smaller set with apparently equivalent code coverage

- Used Adobe Reader + Valgrind in Linux to measure code coverage
- Reduced to 1,515 files of 'equivalent' code coverage
- Same effect as fuzzing all 80k in 2% of the time

Charlie Miller’s 5 Lines

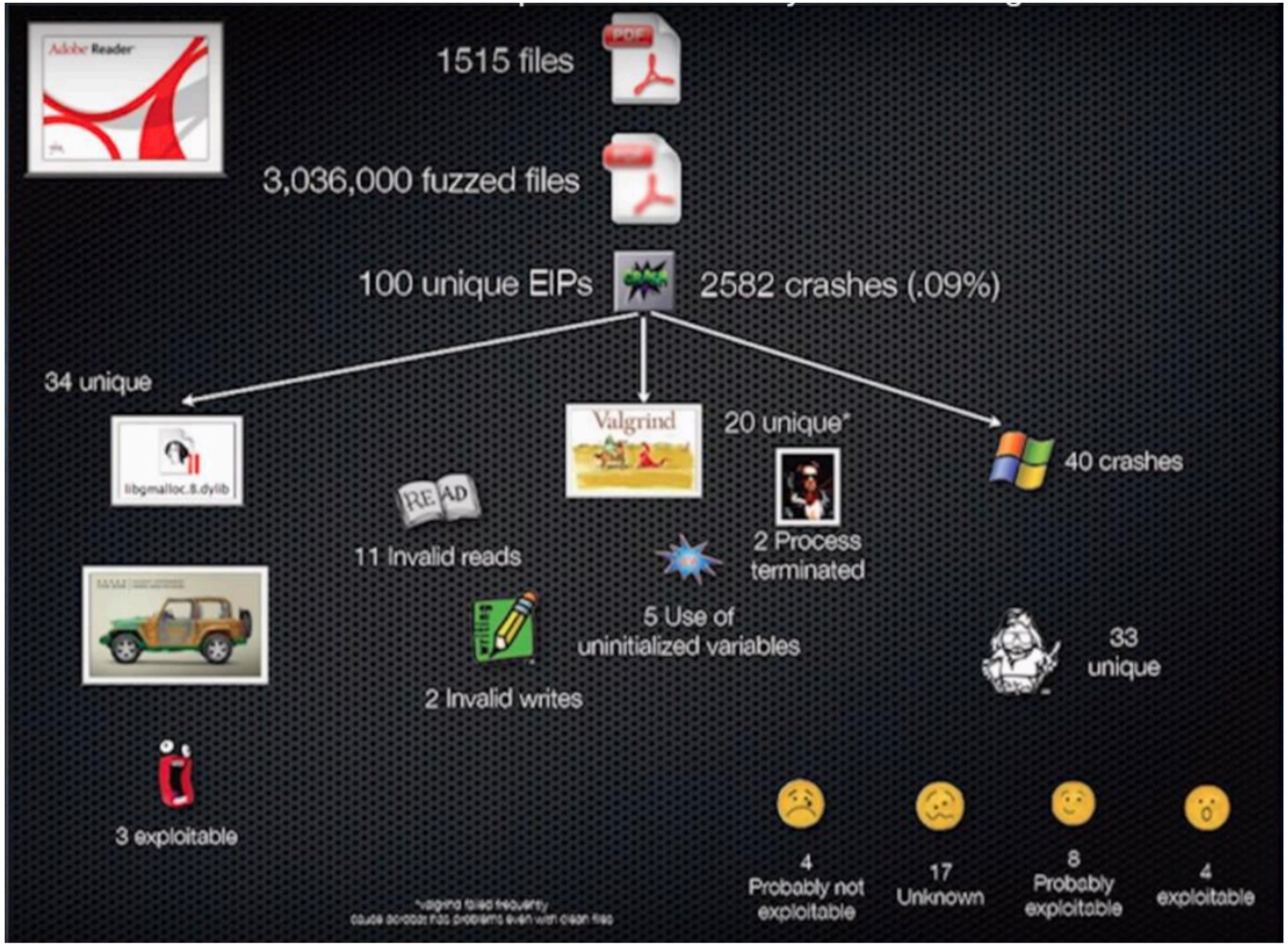
Randomly changed selected bytes to random values in files

Produce ~3 million test cases from 1,500 files

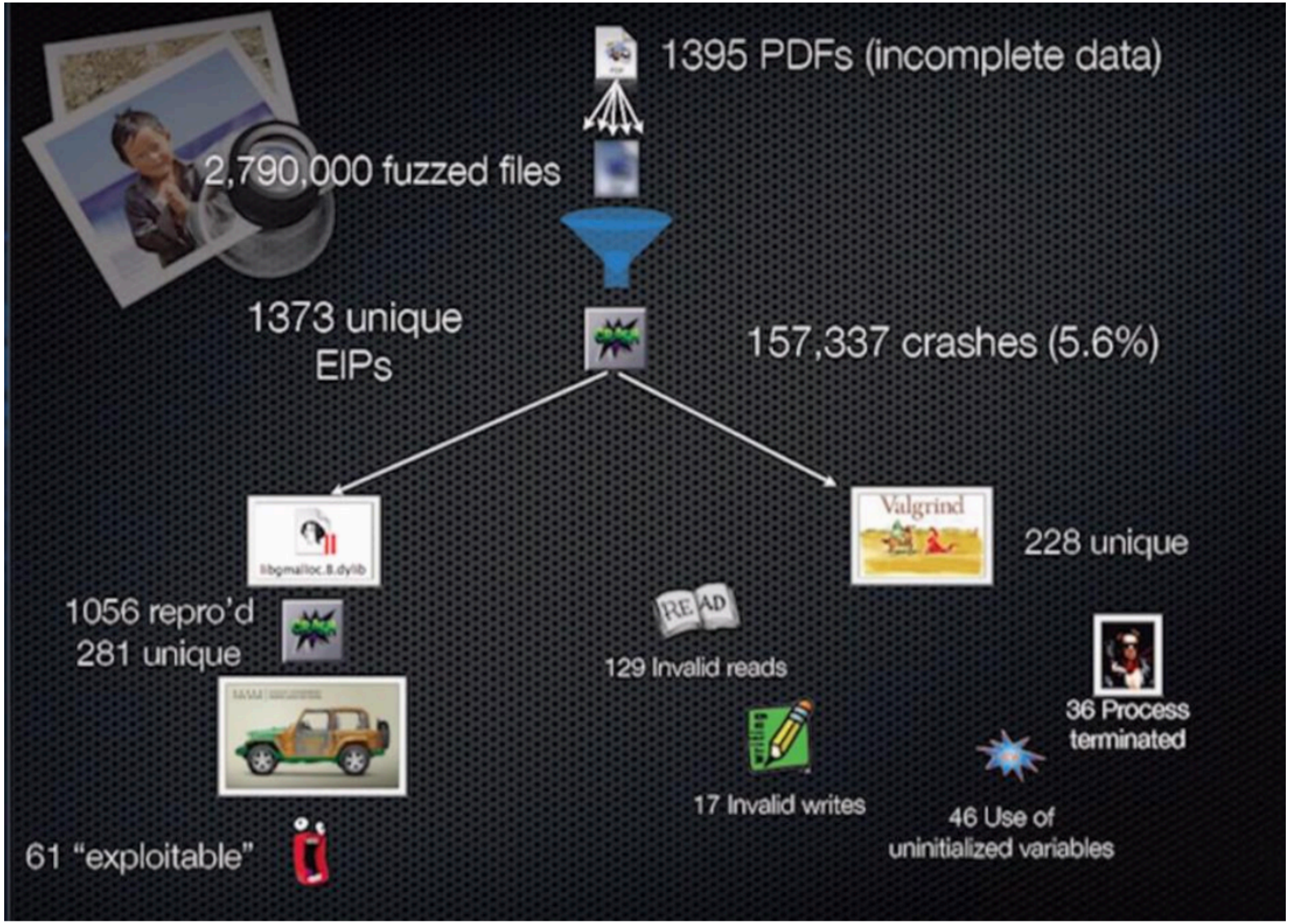
Use standard common tools to determine if crash represents a exploit

- Acrobat: 100 unique crashes, 4 actual exploits
- Preview: 250 unique crashes, 60 exploits (tools may over-estimate)

Adobe Acrobat



Apple Preview



Mutation vs Generation

	Ease of Use	Knowledge	Completeness	Complex Programs
Mutation	Easy to setup and automate	Little to no protocol knowledge required	Limited by initial corpus	May fail for protocols with checksums or other complexity
Generative	Writing generator is labor intensive	Requires having protocol specification	More complete than mutations	Handles arbitrarily complex protocols

Problems with Fuzzing

Mutation based fuzzers can generate an infinite number of test cases...
When has the fuzzer run long enough?

Generation based fuzzers generate a finite number of test cases.
What happens when they're all run and no bugs are found?

How do you monitor the target application such that you know when something "bad" has happened?

Sometimes every anomalous test case triggers the same (boring) bug?

Code Coverage

What if we tried to build tests that try to reach code in the program?

Code coverage is a metric which can be used to determine how much code has been executed.

Function coverage: Has each function in the program been called?

Edge coverage: Has every edge in the Control flow graph been executed?

Branch coverage: Has each branch of each control structure been executed?

Predicate coverage: Has each boolean expression been evaluated to true and false?

Evolutionary Fuzzing

Basic Idea:
Generate inputs based on the structure and response of the program

Autodafe: Prioritizes based on inputs that reach dangerous API functions

EFS: Generates test cases based on code coverage metrics

Typically instrument program with additional instructions to track what code has been reached — or, if no source is available, track with Valgrind.

Tools

Two popular tools today are:

cross_fuzz — specifically targeted at browser and generating complex DOM sequences

American Fuzzy Lop (AFL) — most everything else

AFL Algorithm

1) Load user-supplied initial test cases into the queue,

2) Take next input file from the queue,

3) Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,

4) Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,

5) If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.

6) Go to 2.

Program Analysis

Program Analyzers

Program analysis — process of analyzing program behavior to determine correctness, robustness, safety and liveness

Static analysis

- Analyze source to find errors or check their absence
- Consider all possible inputs (in summary form)
- Can prove absence of bugs, in some cases

Dynamic analysis

- Run instrumented code to find problems
  - Need to choose sample test input
- Can find vulnerabilities but cannot prove their absence

Static Analysis

A static analysis tool ***S*** analyzes the source code of a program ***P*** to determine whether it satisfies a property ***ϕ***, such as:

- “***P*** never deferences a null pointer”
- “***P*** does not leak file handles”
- “No cast in ***P*** will lead to a ClassCastException”

Static Analysis

A sta
deter

- “
- “
- “

Unfortunately, it is impossible to write such a tool!

Rice's theorem states that all non-trivial, semantic properties of programs are undecidable

For any nontrivial property ϕ, there is no general automated method to determine whether P satisfies ϕ

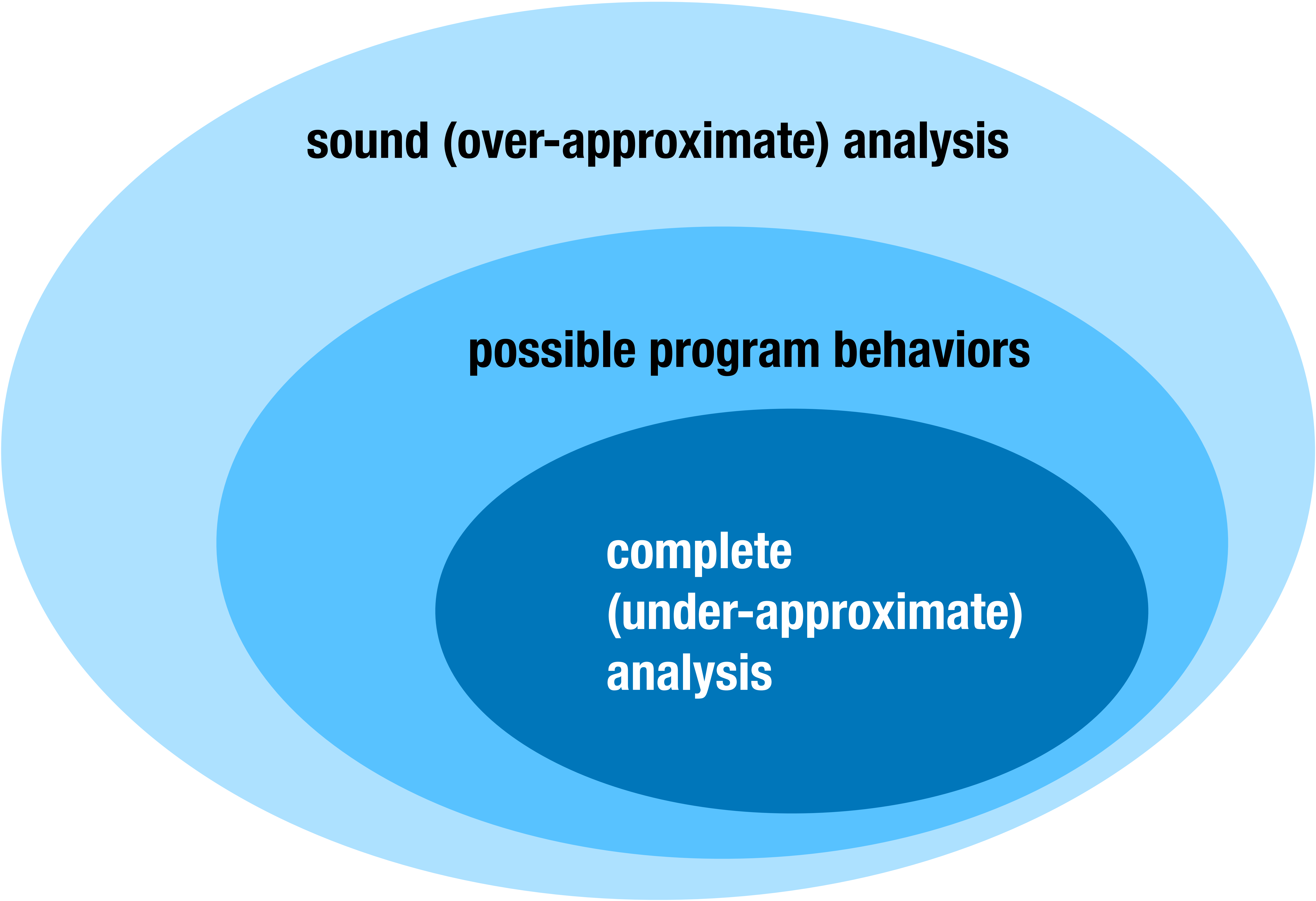
Two Imperfect Options

An analysis tool ***S*** analyzes the source code of a program ***P*** to determine whether it satisfies a property $\boldsymbol{\phi}$ can be wrong in one of two ways:

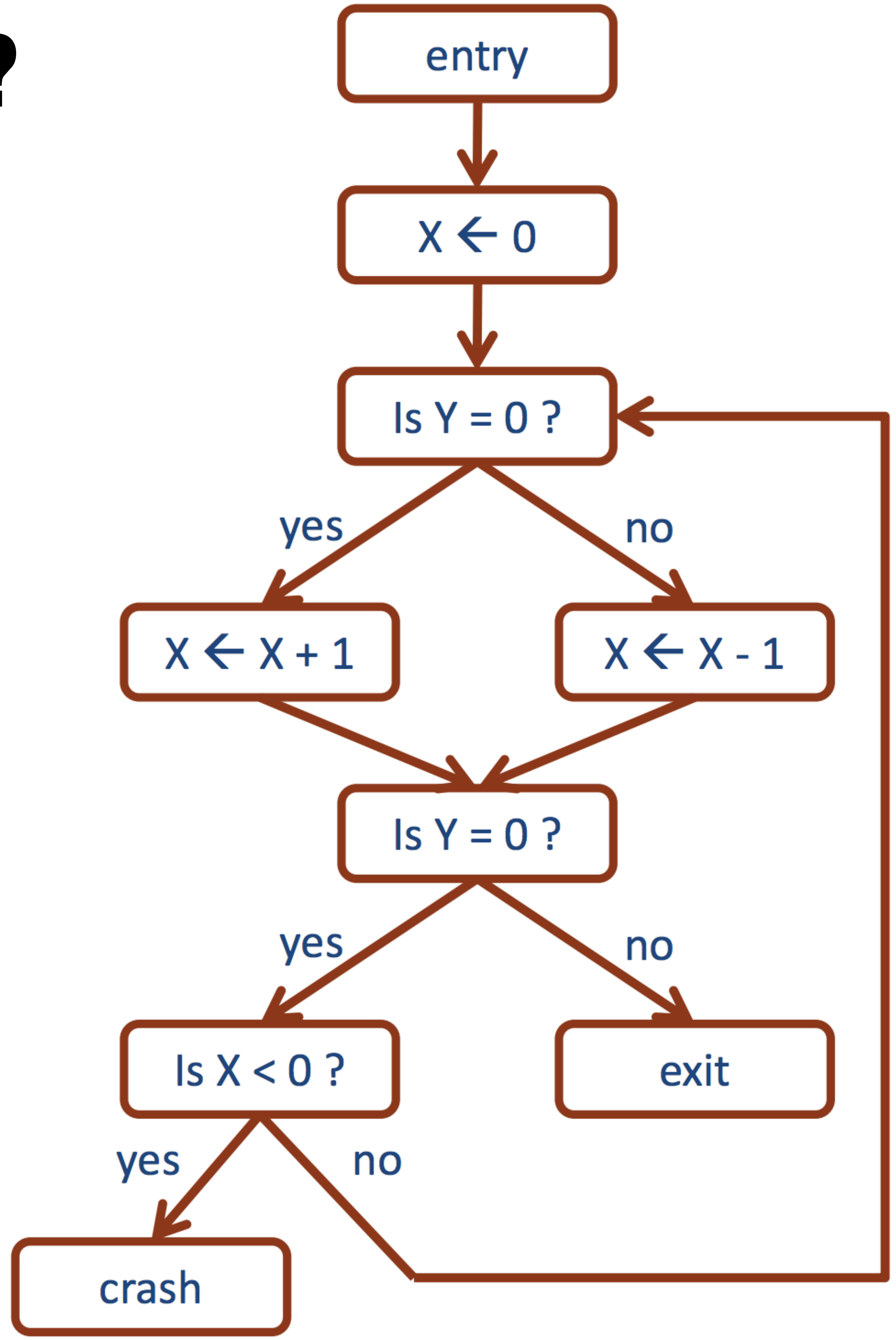
If ***S*** is **sound**, it will never miss violations, but it may say that **P** violates $\boldsymbol{\phi}$ even though it doesn't (resulting in false positives).

If **S** is **complete**, it will never report false positives, but it may miss real violations of $\boldsymbol{\phi}$ (resulting in false negatives).

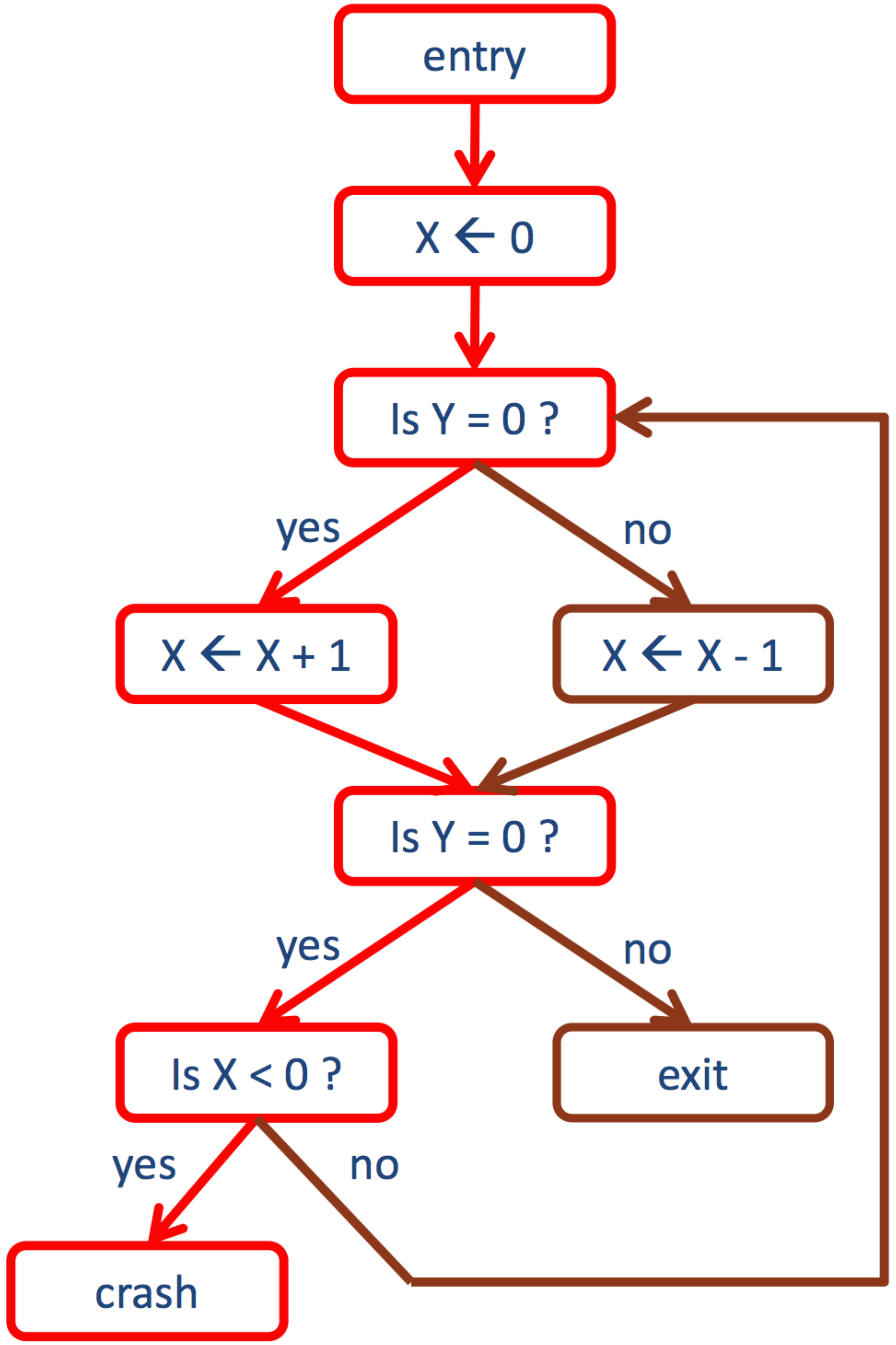
Soundness vs Completeness



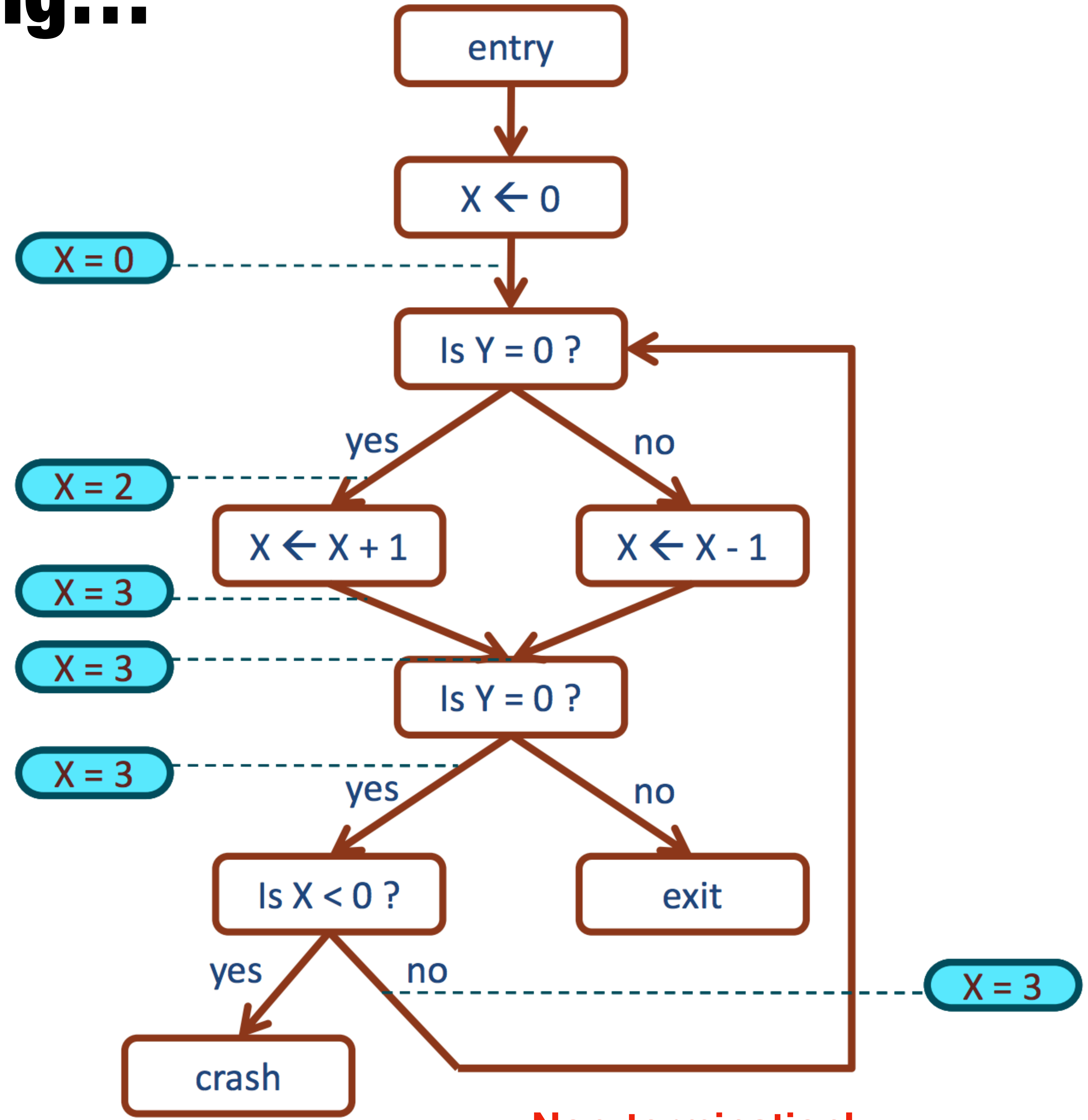
Is this program safe?



Yes, it is safe.
This program will not crash.

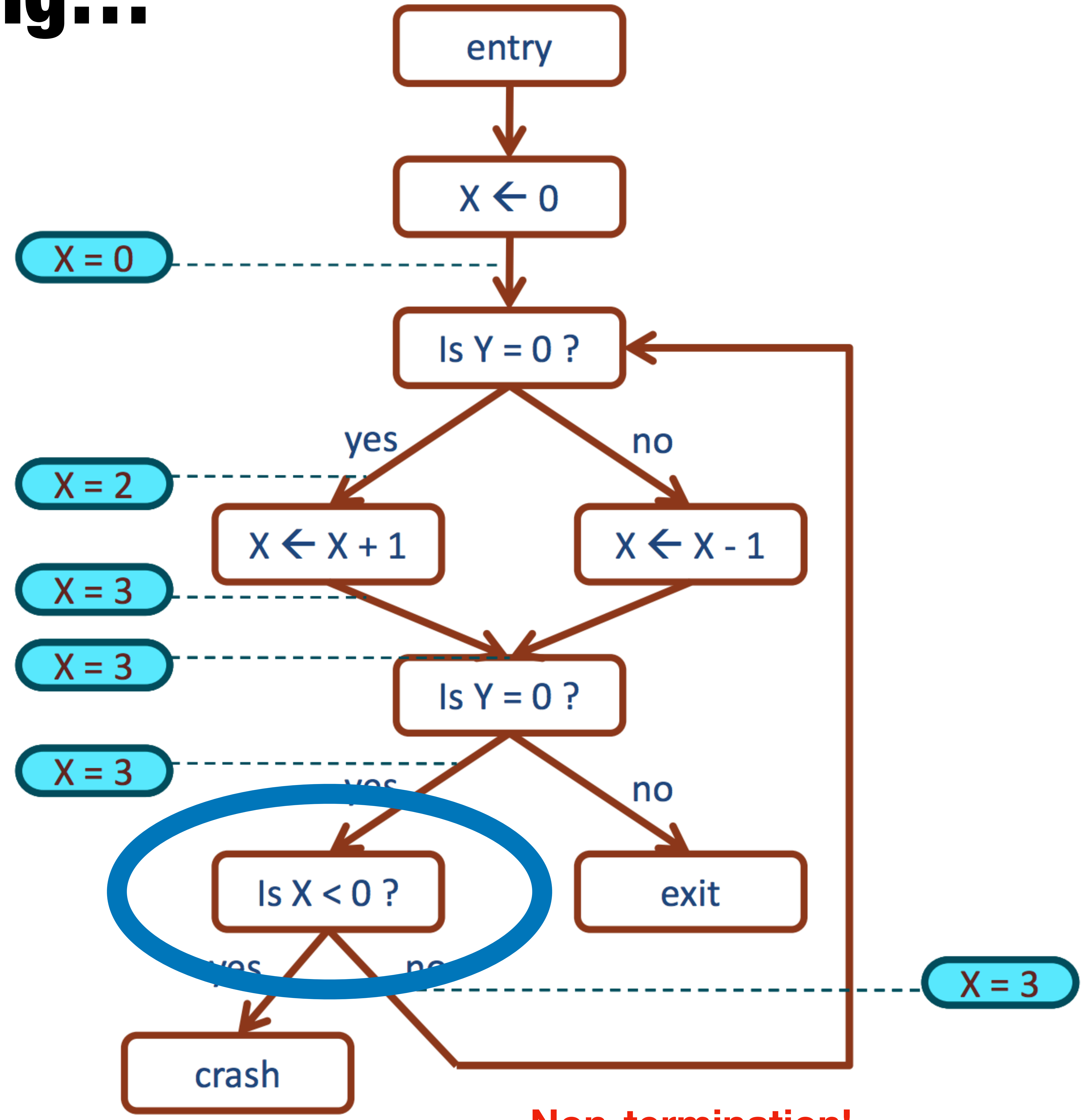


Try analyzing without approximating…



Non-termination!
Therefore, need to approximate

Try analyzing without approximating…



Non-termination!
Therefore, need to approximate

Abstraction

Concrete Domain of Integers

x = 5

x = - 5

x = 0

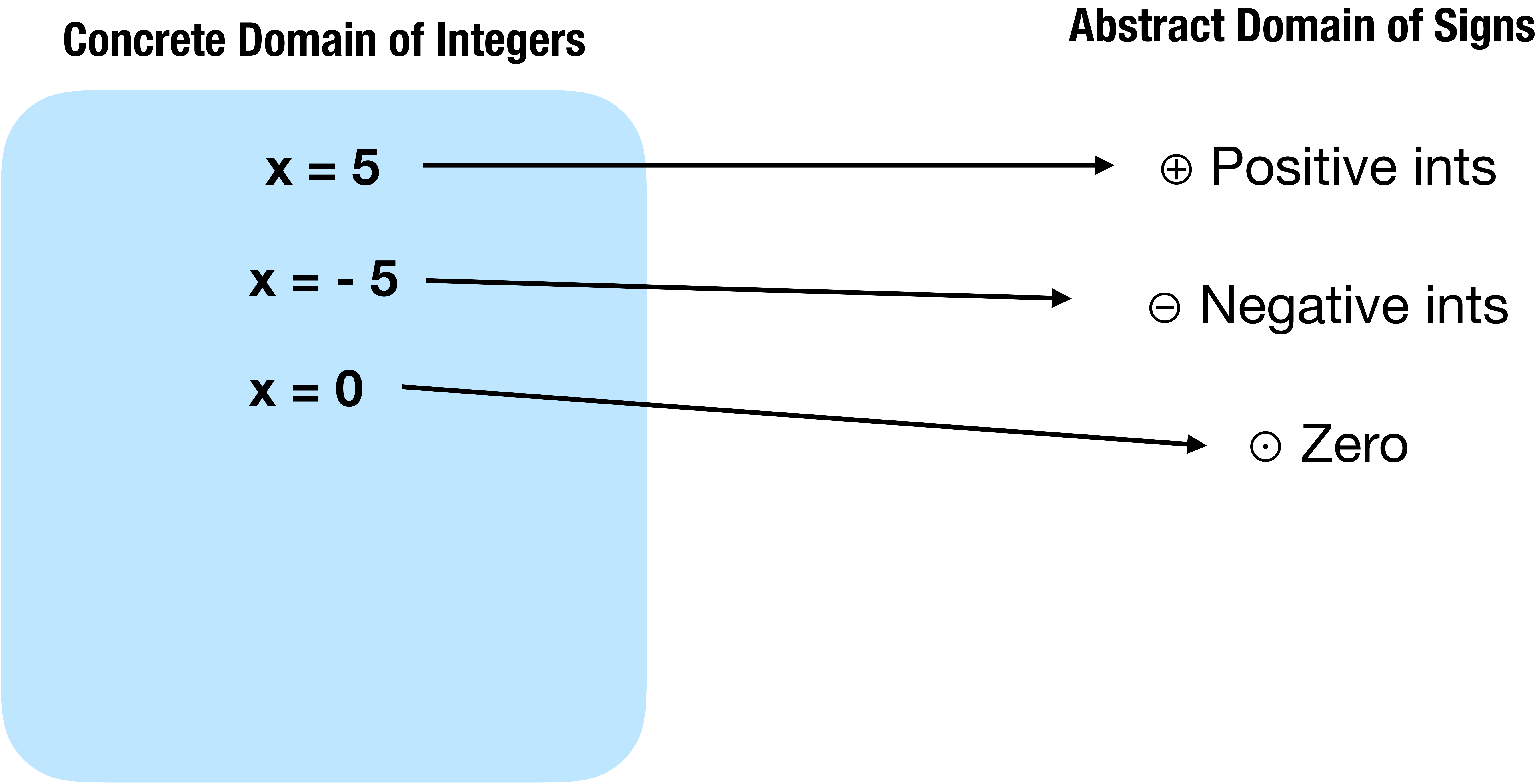
Abstract Domain of Signs

⊕ Positive ints

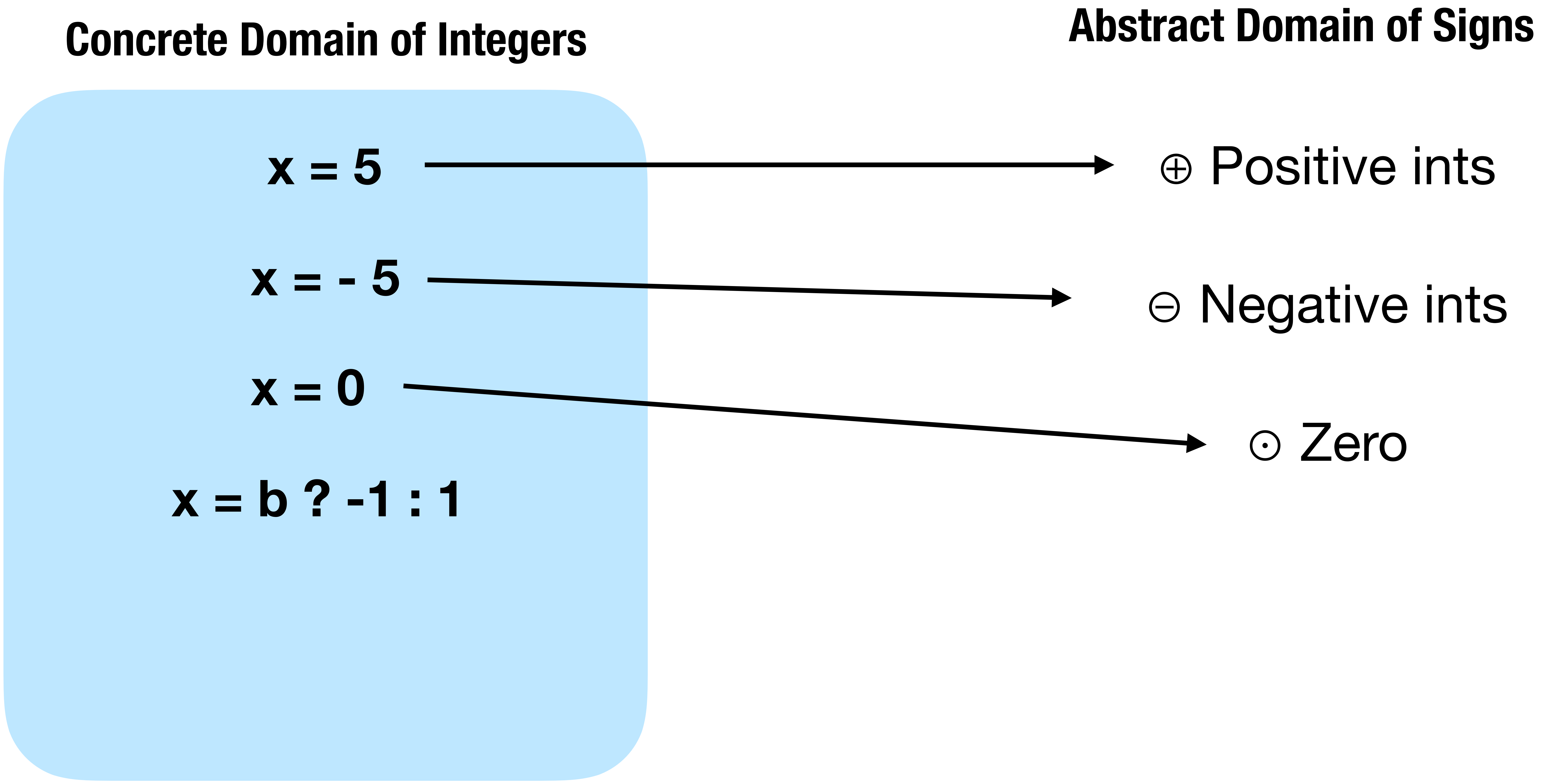
⊖ Negative ints

⊕ Zero

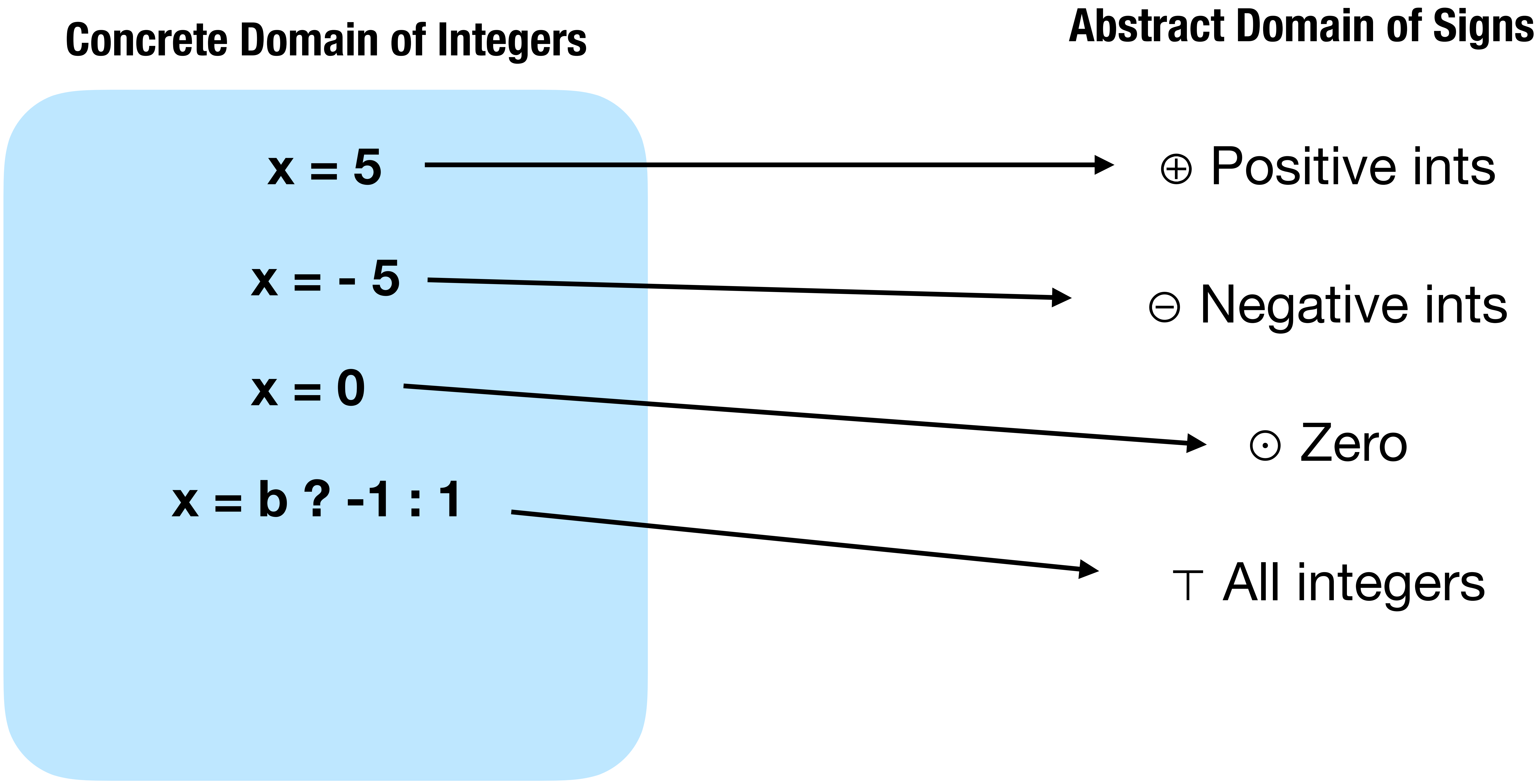
Abstraction



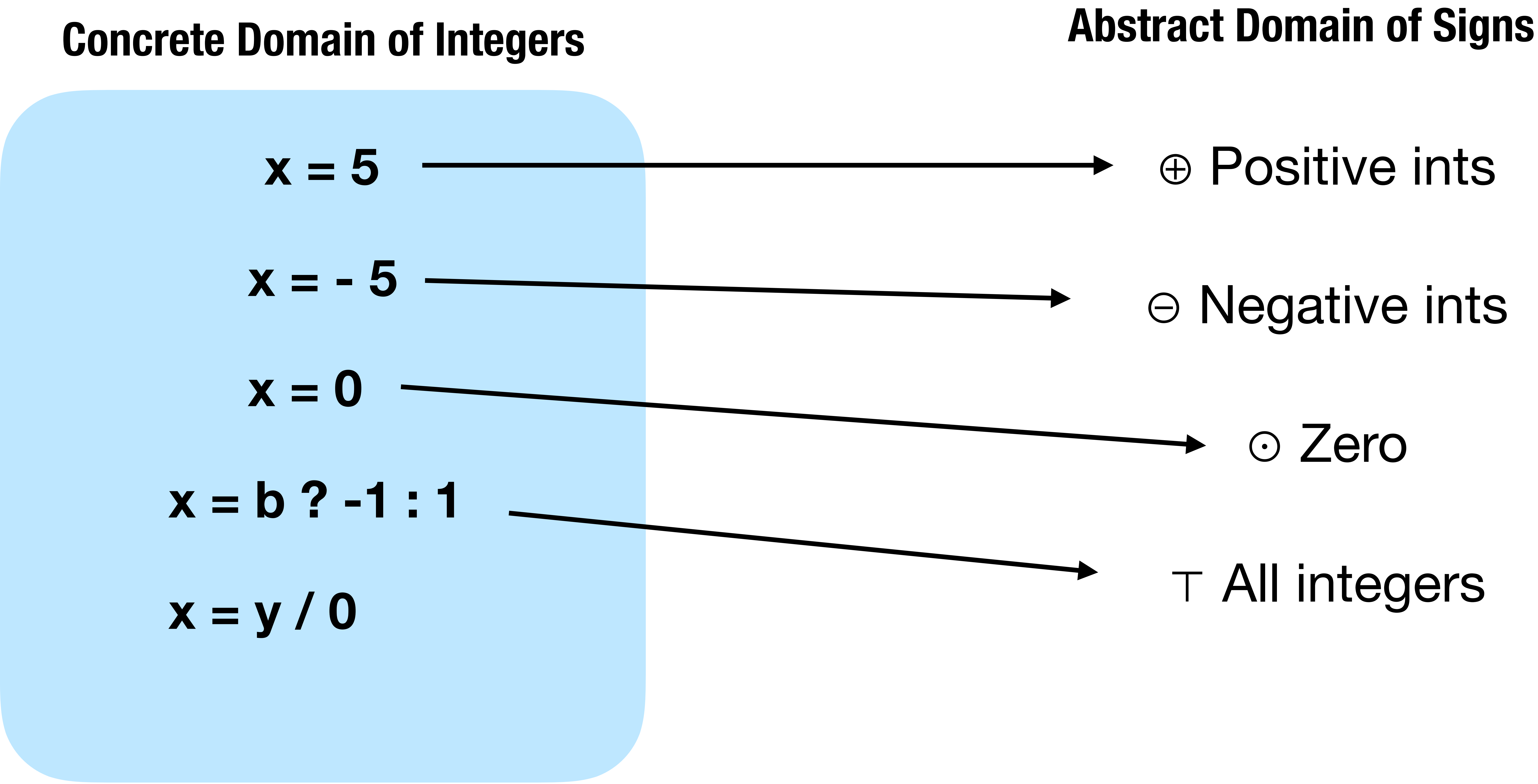
Abstraction



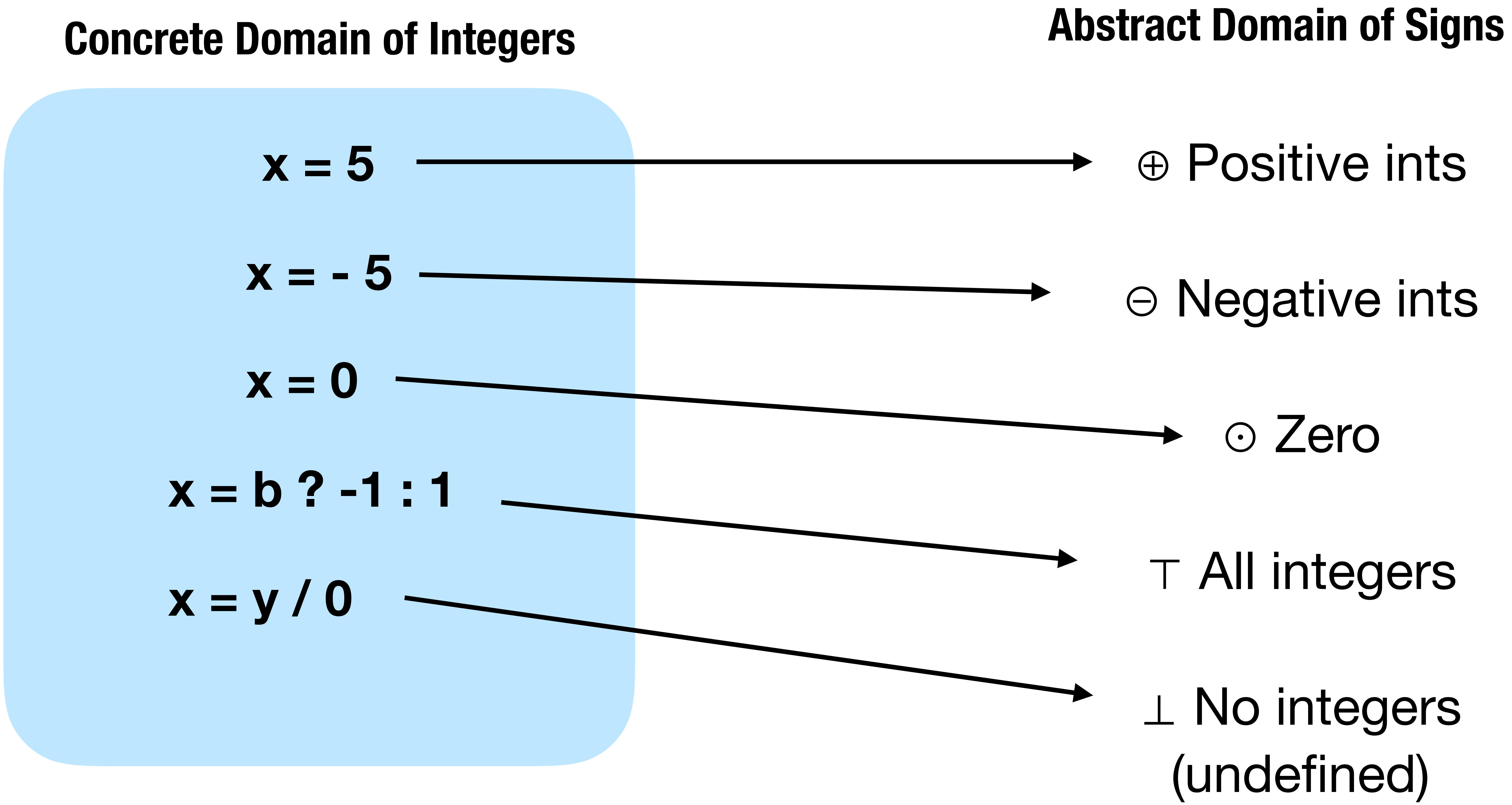
Abstraction



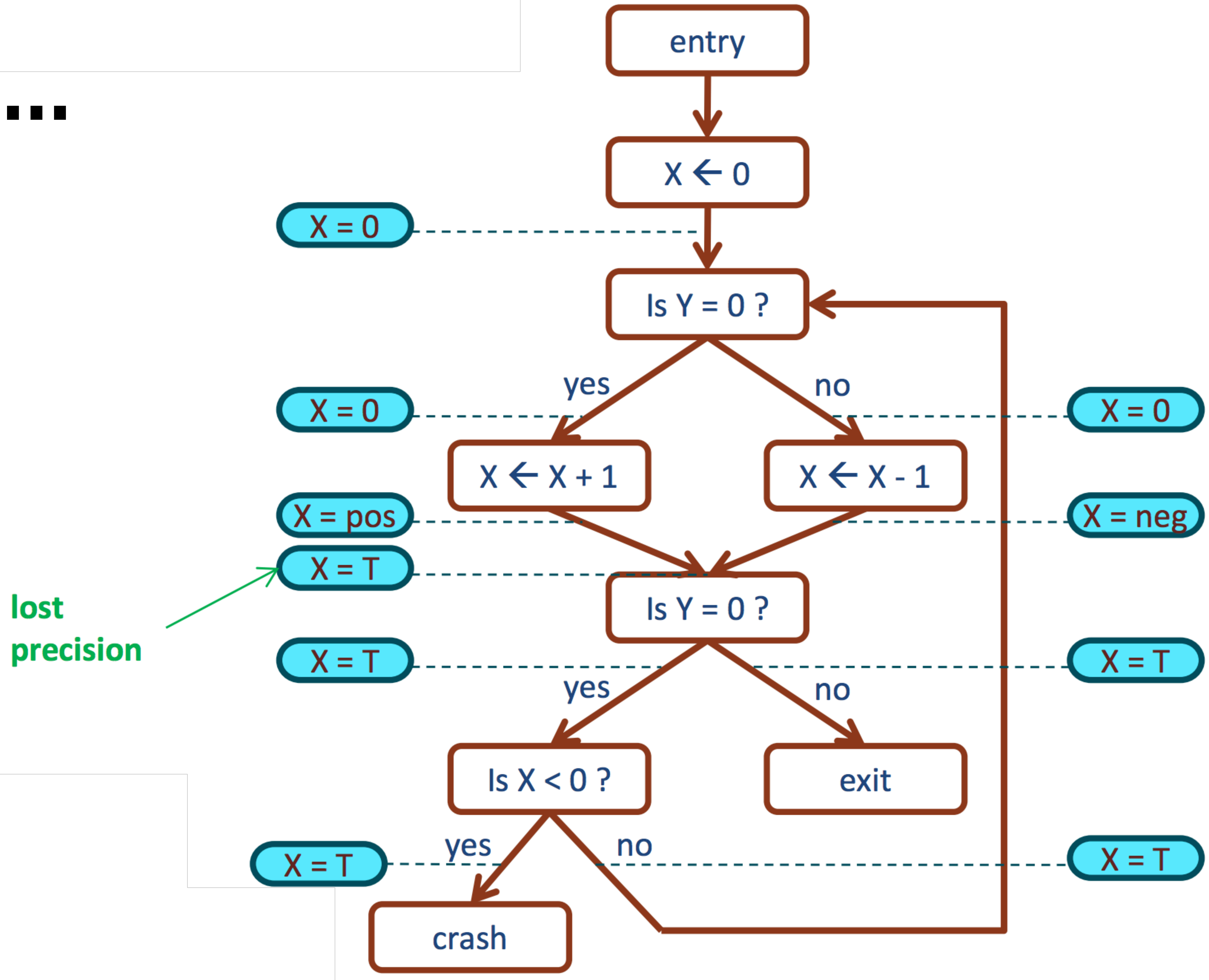
Abstraction



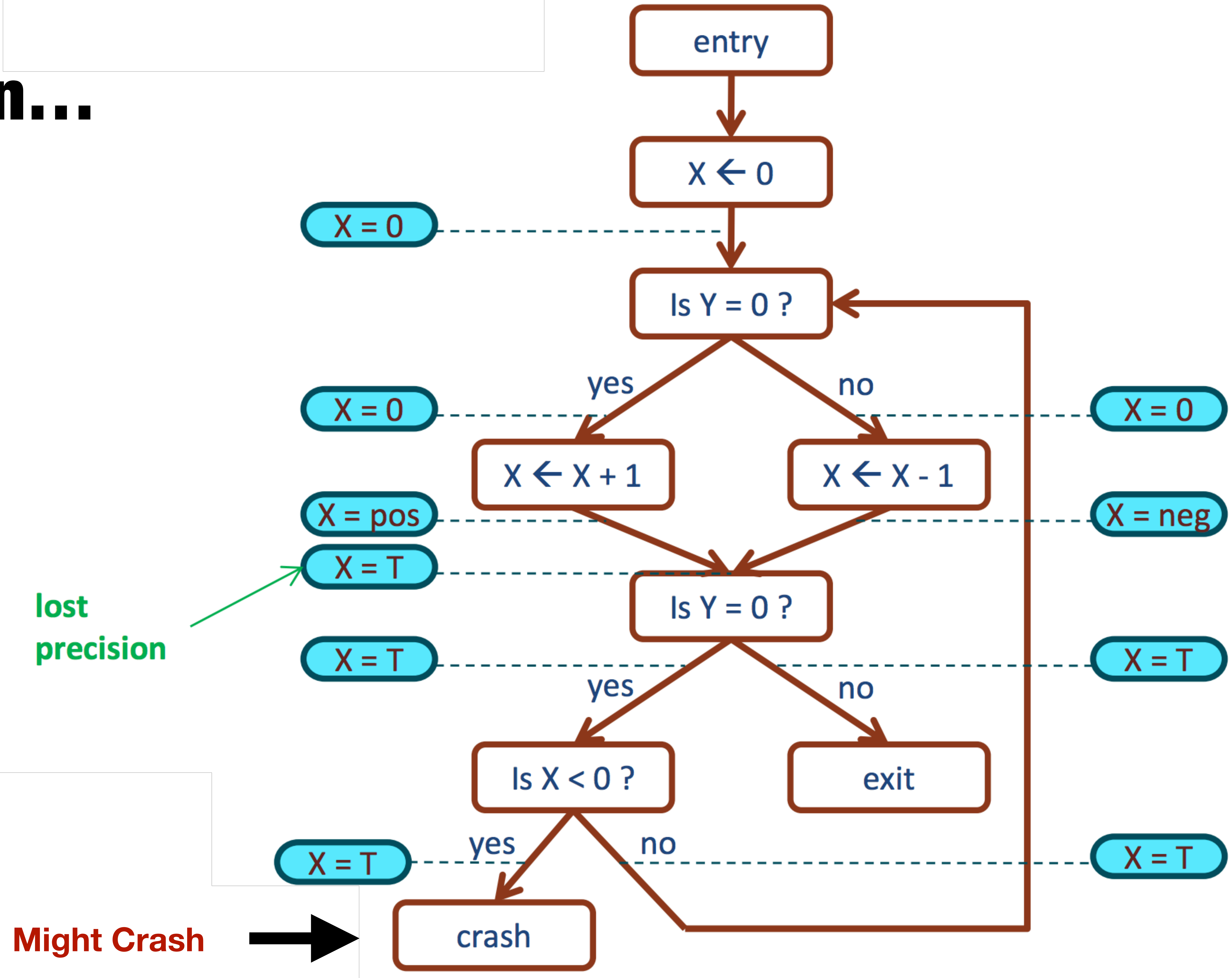
Abstraction



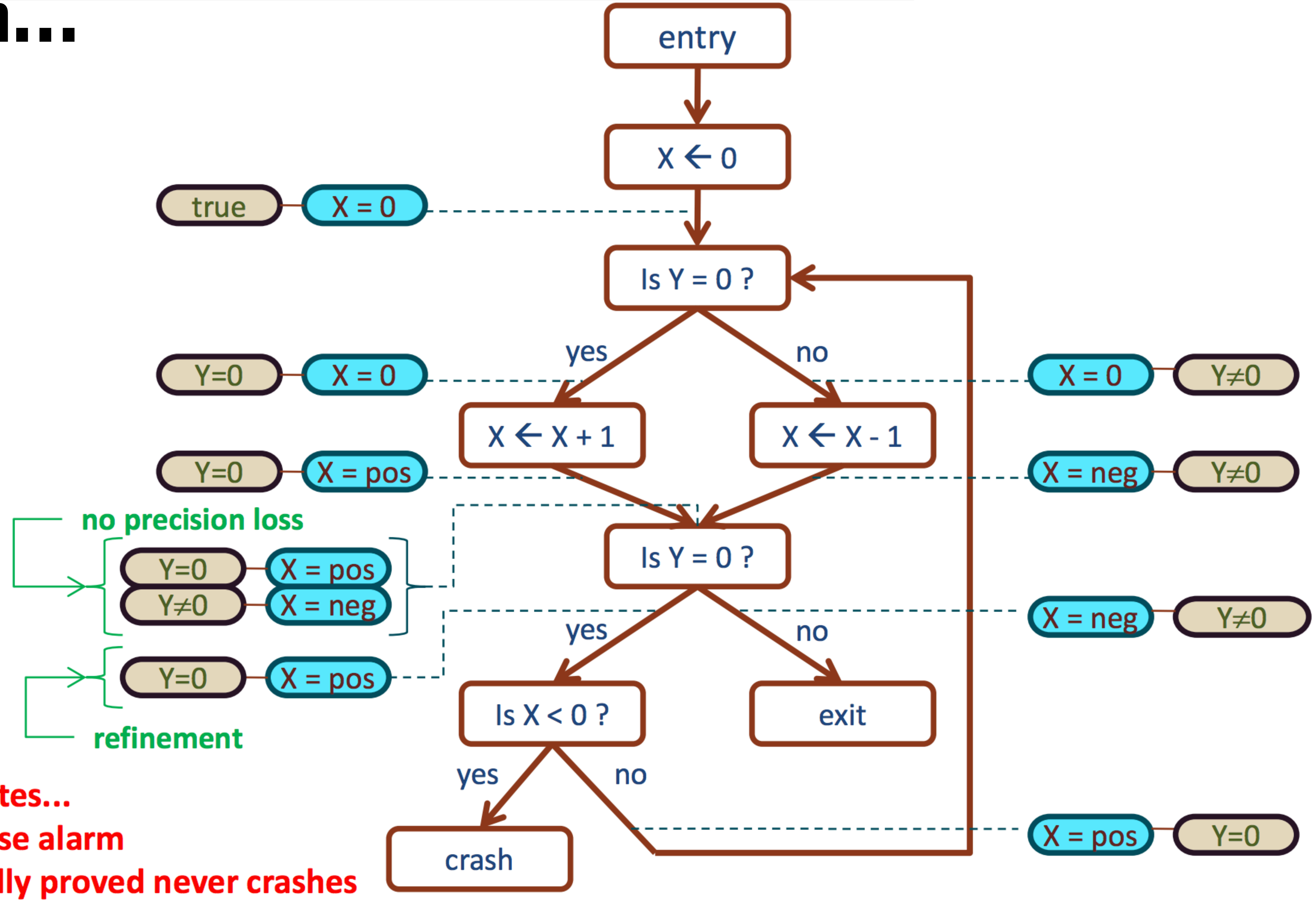
Try analyzing with "signs" approximation…



Try analyzing with “signs” approximation…



Try analyzing with "path-sensitive signs" approximation…



terminates…

… no false alarm

… soundly proved never crashes

Bugs to Detect

Uninitialized variables
Null pointer dereference
Use after free
Double free
Array indexing errors
Mismatched array new/delete
Potential stack overrun
Potential heap overrun
Return pointers to local variables
Logically inconsistent code

Invalid use of negative values
Passing large parameters by value
Underallocations of dynamic data
Memory leaks
File handle leaks
Network resource leaks
Unused values
Unhandled return codes
Use of invalid iterators

Example: Check for missing optional args

Prototype for open() syscall:

```
int open(const char *path, int oflag, /* mode_t mode */...);
```

Typical mistake:

```
fd = open("file", O_CREAT);
```

Result: file has random permissions

Check: Look for oflags == O_CREAT without mode argument

Example: Chroot protocol checker

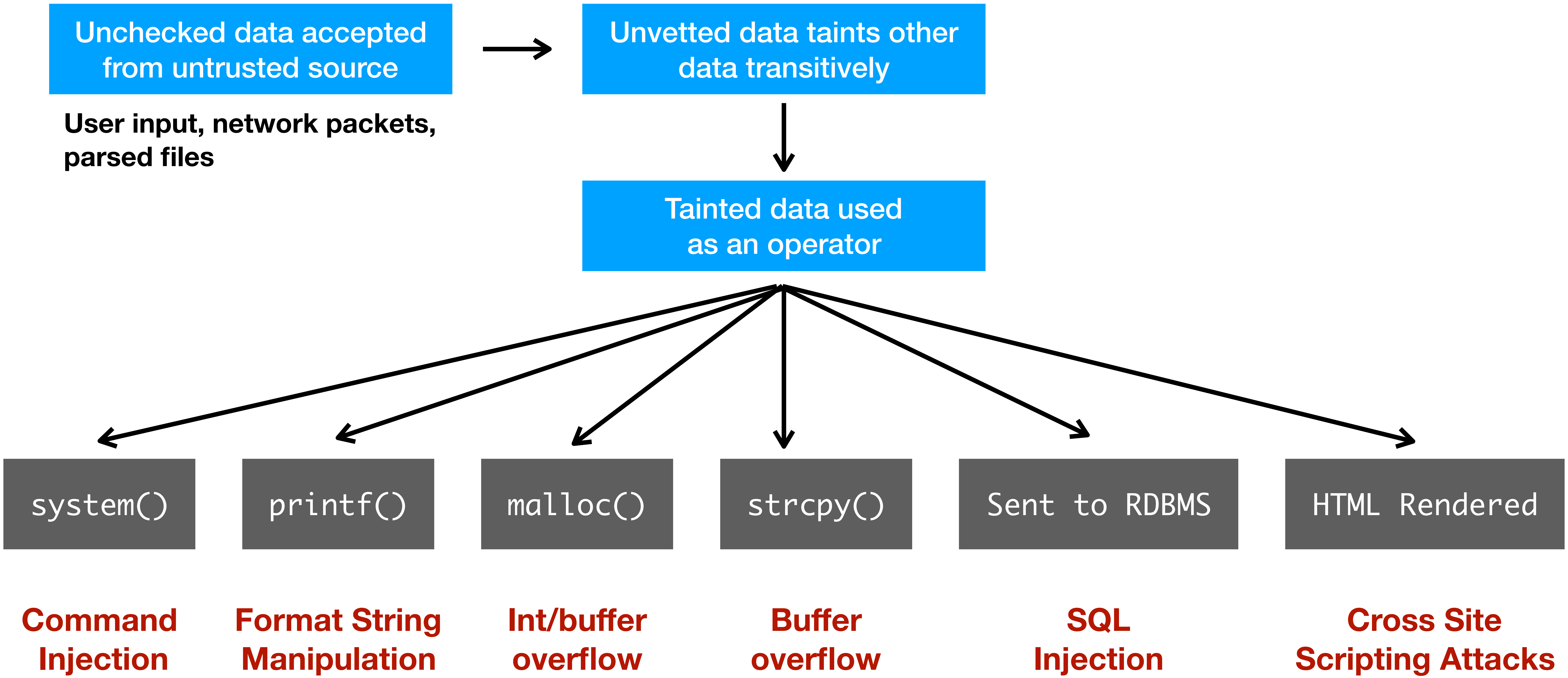
Goal: confine process to a “jail” on the filesystem

chroot() changes filesystem root for a process

Problem: chroot() itself does not change current working directory

Check: check if any sys calls (e.g., open) are called before chdir is called

Tainting Checkers



Finding Vulnerabilities

Stanford Research

Using Programmer-Written Compiler Extensions to Catch Security Holes

Ken Ashcraft and Dawson Engler

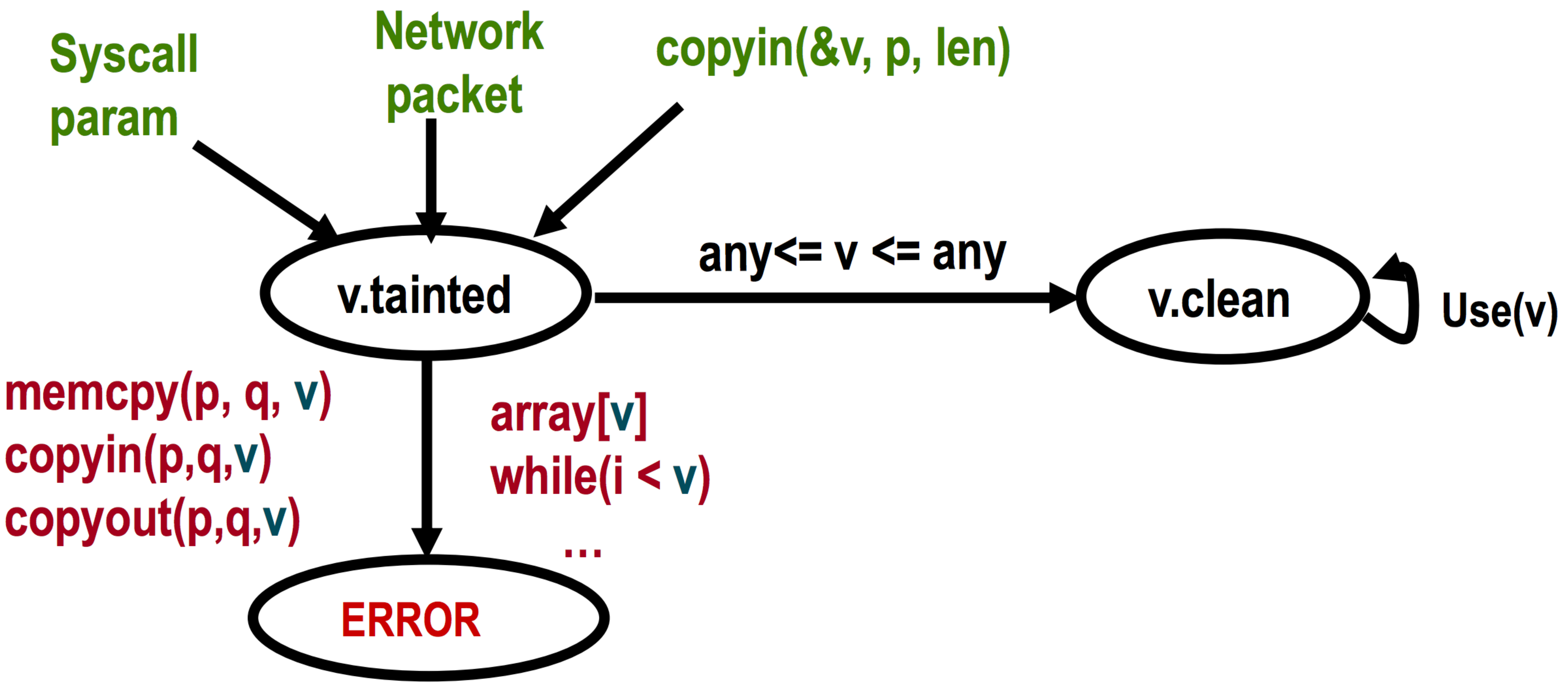
IEEE Security and Privacy ("Oakland") 2002

Used modified compiler to find over 100 security holes in Linux and BSD

Longterm, commercialized and extended tools

Checking for Unsanitized Integers

Warn when unchecked integers from untrusted sources reach trusting sinks



Linux: 125 errors, 24 false; BSD: 12 errors, 4 false

Example Untrusted Integer

Remote exploit, no length checks

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
msg = skb->data;
...
memcpy(cmd.parm.setup.phone,
msg->msg.connect_ind.addr.num,
msg->msg.connect_ind.addr.len - 1);
```

Overview of Static Analysis

Automated method to find errors or check their absence

Consider all possible inputs (in summary form)

Can prove absence of bugs, in some cases

Very well-studied part of computer science, but tools will inherently always over- or under-report problems

Dynamic Analysis

Dynamic (Program) Analysis analyzes computer software while it is operating (in contrast to static which looks only at code)

Unit tests, integration tests, system tests and acceptance tests are all a form of dynamic testing.

However, typically like to instrument code to understand where the problem occurred

Valgrind

```
==25832== Invalid read of size 4
==25832== at 0x8048724: BandMatrix::ReSize(int, int, int) (bogon.cpp:45)
==25832== by 0x80487AF: main (bogon.cpp:66)
==25832== Address 0xBFFFF74C is not stack’d, malloc’d or free’d
```

Isn’t that a Debugger?

Traditional debuggers typically focus on allow programmers to find the source of fatal errors (e.g., NULL pointer deref)

Not all bugs lead to crashes — especially for inputs that typically don’t crash.

In contrast, security tools attempt to uncover non-fatal problems — potential race conditions or overflows

Google AddressSanitizer (ASan)

AddressSanitizer is a memory error detector for C/C++ that finds:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Google AddressSanitizer (ASan)

LLVM Pass
Modifies the code to check the shadow state for each memory access and creates poisoned redzones around stack and global objects to detect overflows and underflows

A run-time library that replaces the malloc function
The run-time library replaces malloc, free and related functions, creates poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting.

Google AddressSanitizer (ASan)

```
==9901==ERROR: AddressSanitizer: heap-use-after-free on address 0x60700000dfb5 at pc 0x45917b
bp 0x7fff4490c700 sp 0x7fff4490c6f8
READ of size 1 at 0x60700000dfb5 thread T0
    #0 0x45917a in main use-after-free.c:5
    #1 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
    #2 0x459074 in _start (a.out+0x459074)
0x60700000dfb5 is located 5 bytes inside of 80-byte region [0x60700000dfb0,0x60700000e000)
freed by thread T0 here:
    #0 0x4441ee in __interceptor_free projects/compiler-rt/lib/asan/asan_malloc_linux.cc:64
    #1 0x45914a in main use-after-free.c:4
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
previously allocated by thread T0 here:
    #0 0x44436e in __interceptor_malloc projects/compiler-rt/lib/asan/asan_malloc_linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

Summary of Program Analysis

	Pros	Cons
Static	Enables quickly finding bugs at development time Can detect some problems that dynamic misses	Either over or under reports. Misses complex bugs. Generally requires code.
Dynamic	May uncover complex behavior missed by static. Can run on blackbox.	Depends on user input—only checks executed code

Reverse Engineering

Reverse Engineering

reverse engineering: process of discovering the technological principles of a [insert noun] through analysis of its structure, function, and operation

In security, this is typically uncovering the human readable code for a binary:

- Vulnerability or exploit research
- Malware analysis
- Check for copyright/patent violations
- Interoperability (e.g. understanding a file/protocol format)
- Copy protection (e.g., DRM or software licensing) removal

Techniques

Static Code Analysis (structure)

* Disassemblers

Dynamic Code Analysis (operation)

* Tracing / Hooking

* Debuggers

Disassembly

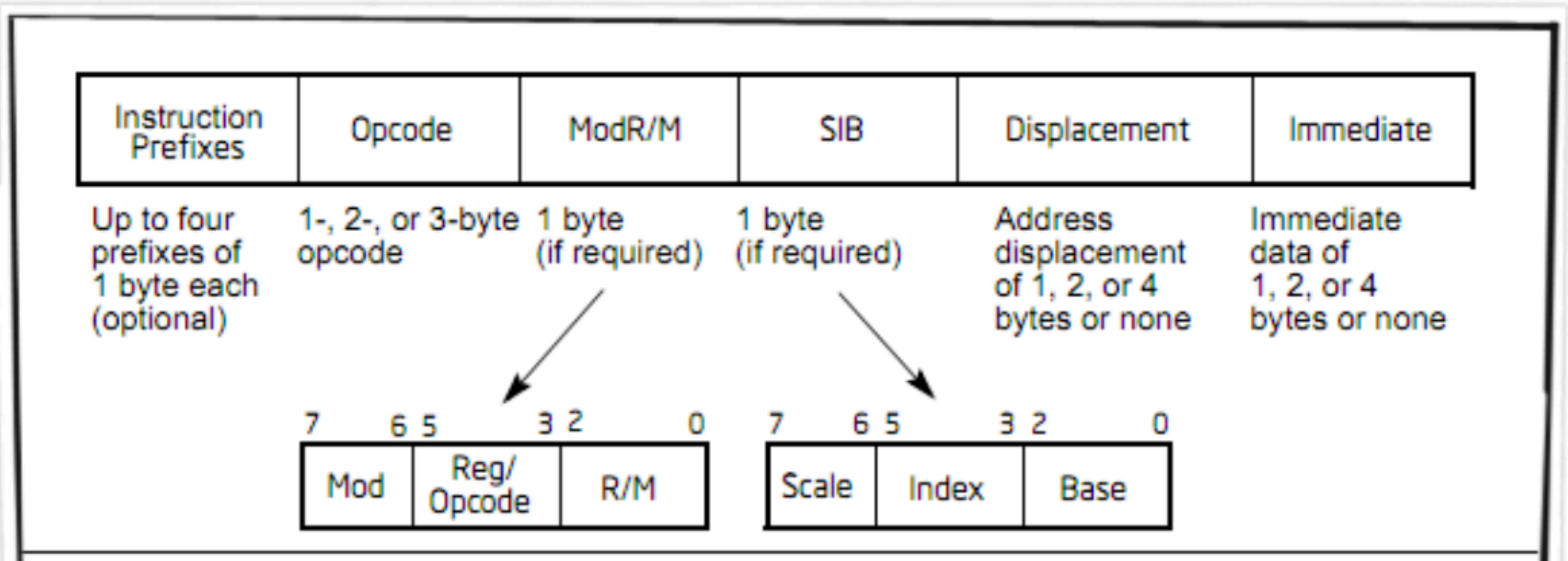
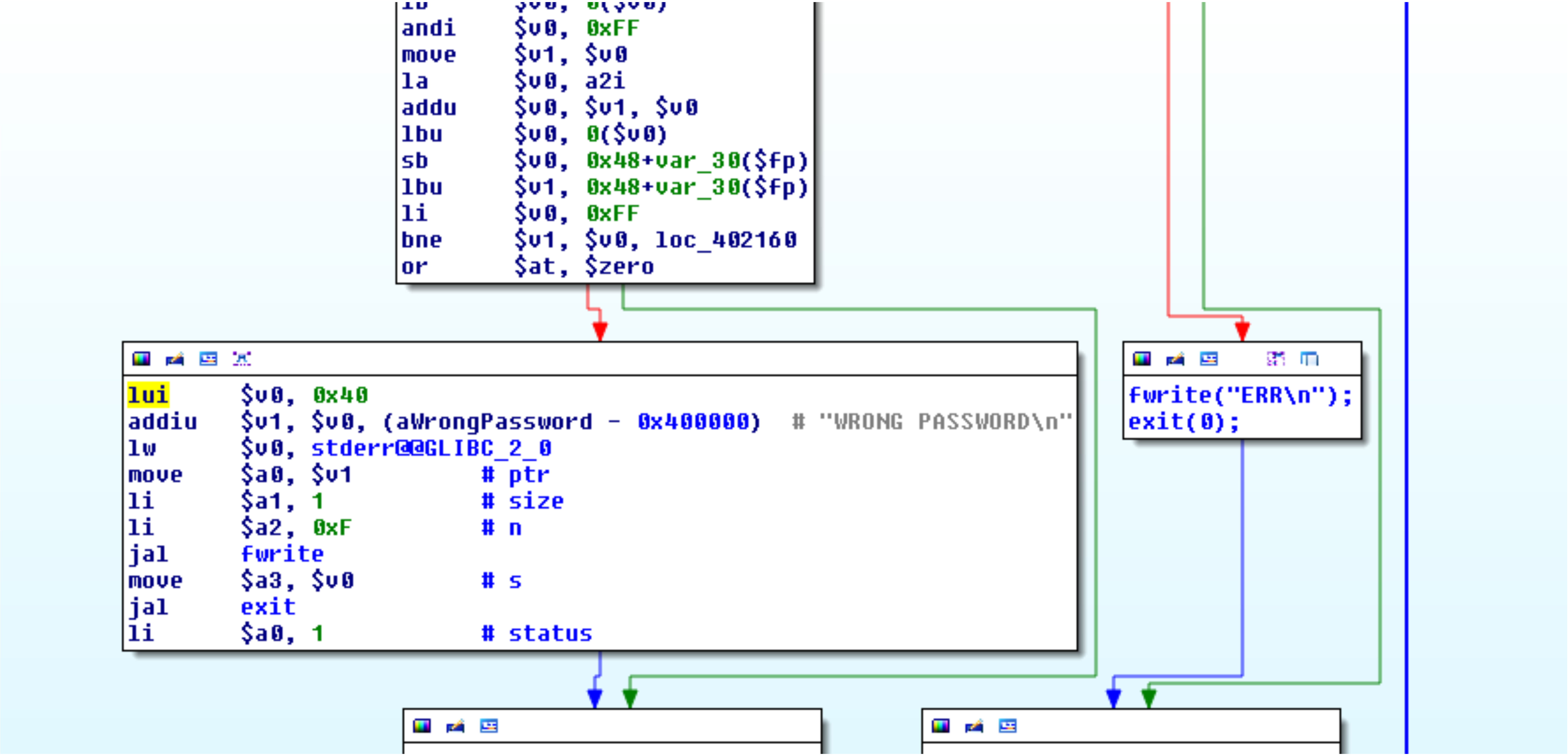


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Bits	Hex Bytes	Instructions (human-readable)
11101011 00000110	0xEB 0x06	JMP +6
01010000	0x50	PUSH EAX

Decompilation



Decompilation

IDA View-A | Pseudocode-A

```
{
  unsigned int v2; // [sp+20h] [bp-28h]@1
  __int64 *v3; // [sp+28h] [bp-20h]@1
  signed int i; // [sp+30h] [bp-18h]@1
  va_list va; // [sp+58h] [bp+10h]@1

  va_start(va, a1);
  v3 = (__int64 *)va;
  v2 = 0;
  for ( i = 0; i < (signed int)a1; ++i )
  {
    ++v3;
    v2 += *((_DWORD *)v3 - 2);
  }
  printf("va_ri/count = %d\n", a1);
  printf("va_ri/res   = %d\n", v2);
  return v2;
}
```

00003615 va_ri:2

Difficulties

Disassembly is imperfect

Benign Optimizations

- Constant folding
- Dead code elimination
- Inline expansion
- etc...

Intentional Obfuscation

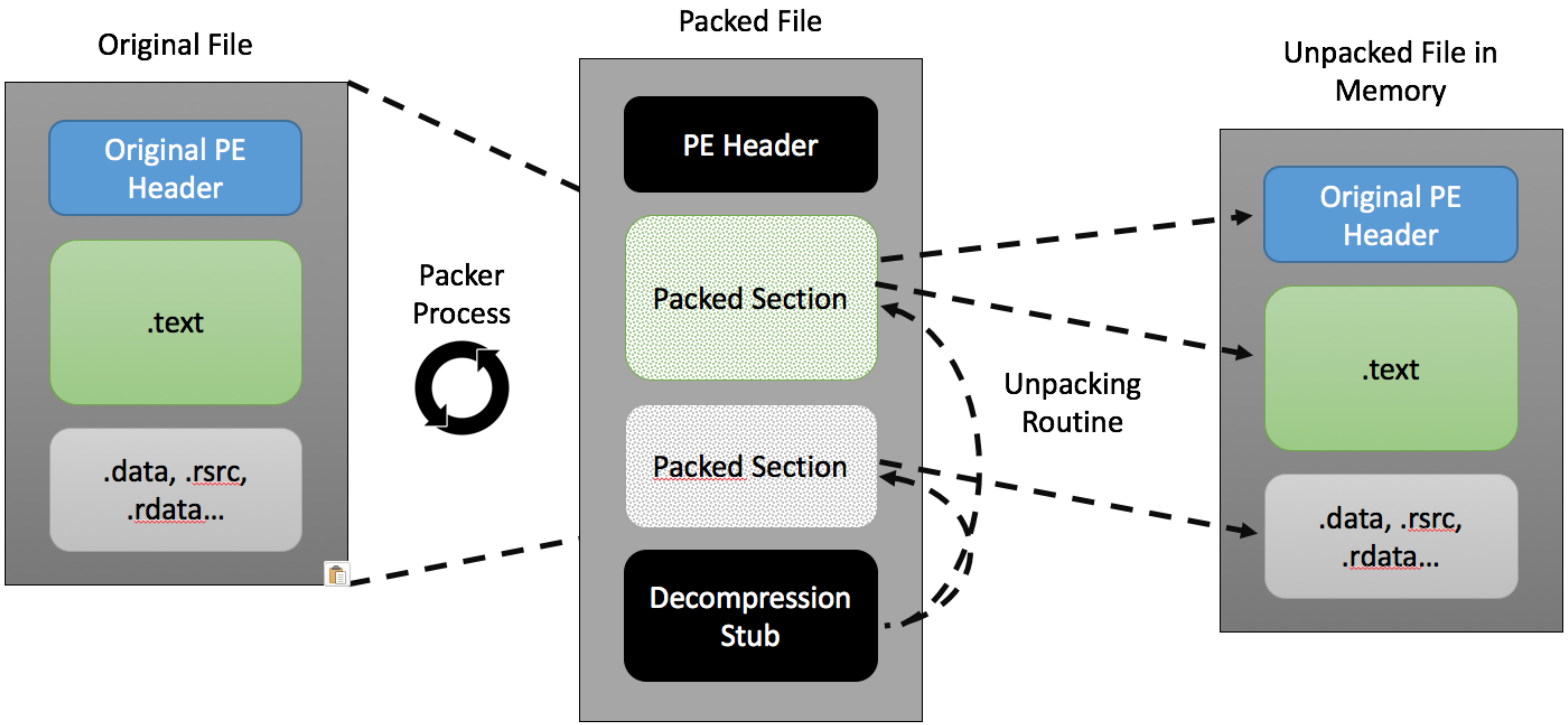
- Packing

Packing

Packing: technique to hide the real code of a program through one or more layers of compression/encryption

At run-time the unpacking routine restores the original code in memory and then executes it

Packing


Original File
Original PE Header
.text
.data, .rsrc, .rdata...
Packer Process
Packed File
PE Header
Packed Section
Packed Section
Decompression Stub
Unpacking Routine
Unpacked File in Memory
Original PE Header
.text
.data, .rsrc, .rdata...

IDA
Pro

GHIDRA

A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission

Automated Security Testing

CS155 Computer and Network Security

Stanford University