Finding vulnerabilities by fuzzing, dynamic and static analysis

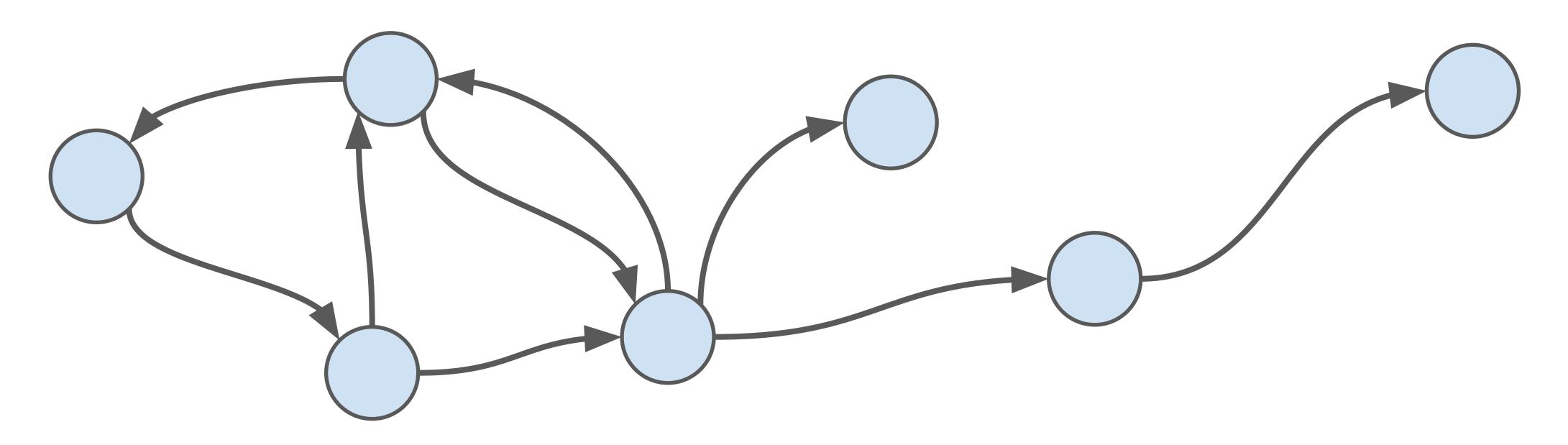
Brandon Azad Stanford CS155 guest lecture April 15, 2021

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh and Zakir Durumeric 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.

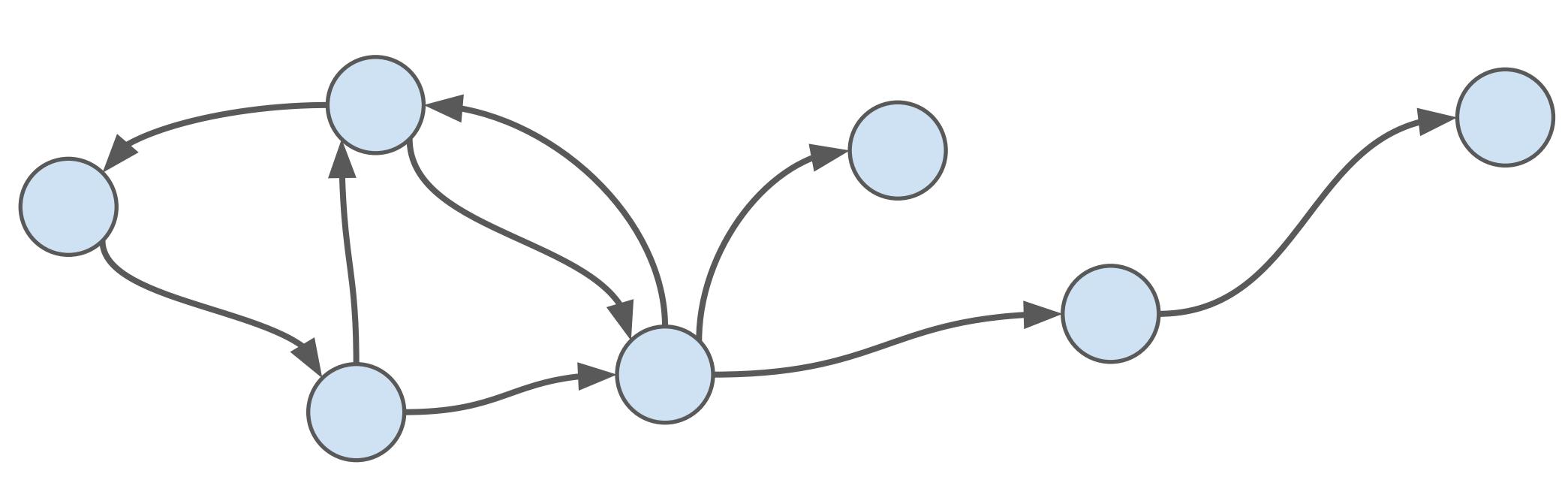


Conceptualizing vulnerabilities and exploits

Computer programs: finite state machines

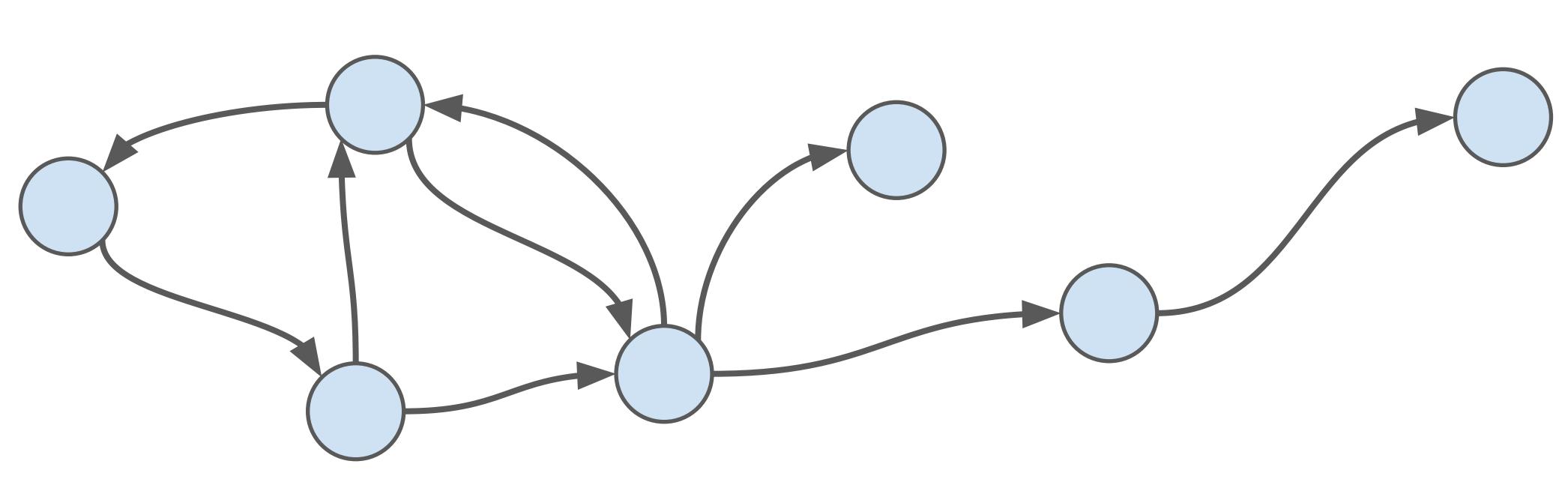


Computer programs: finite state machines



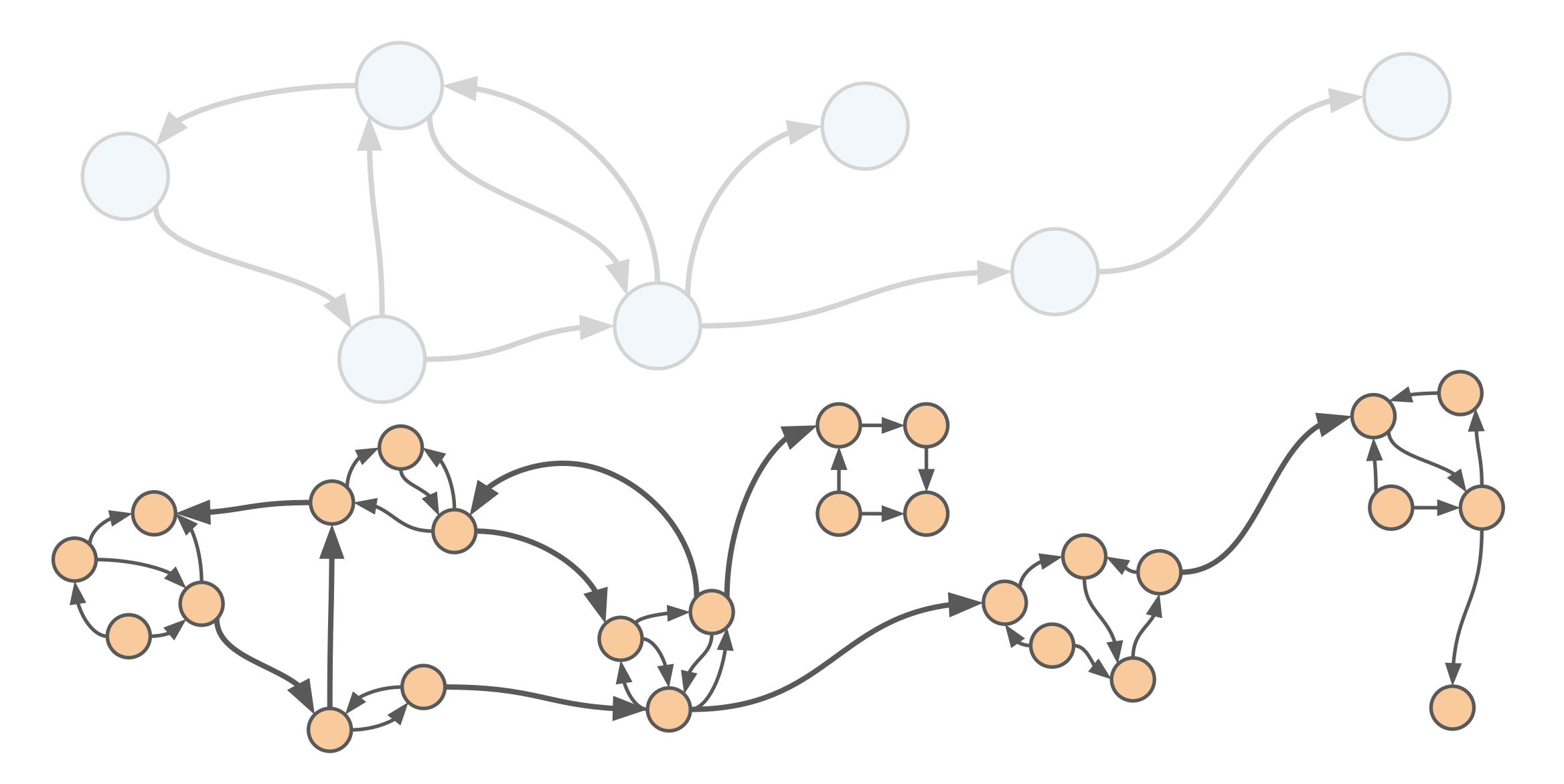
This is a *conceptual* state machine describing the *intended* operation of the program.

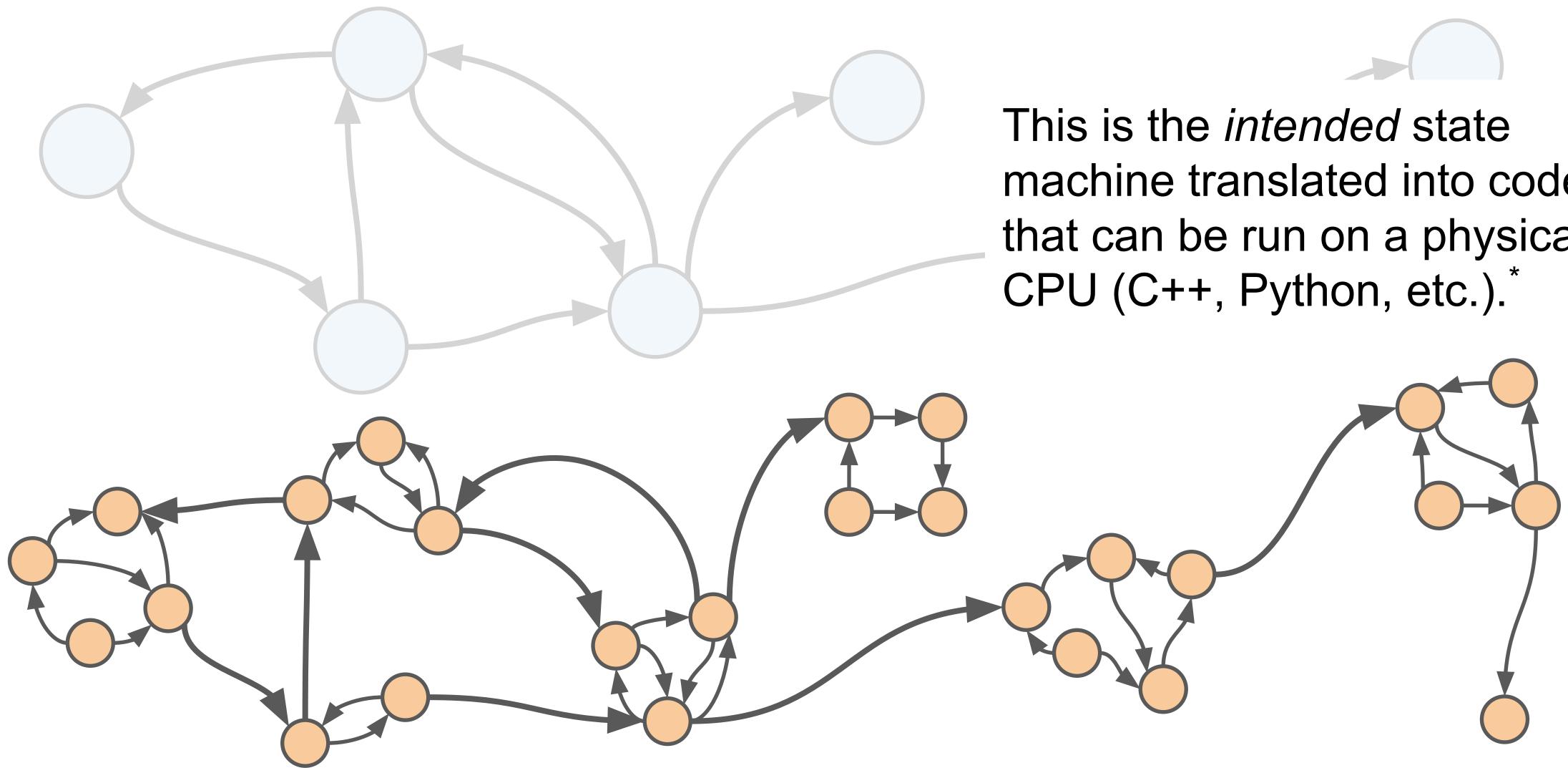
Computer programs: finite state machines



This is a *conceptual* state machine describing the *intended* operation of the program.

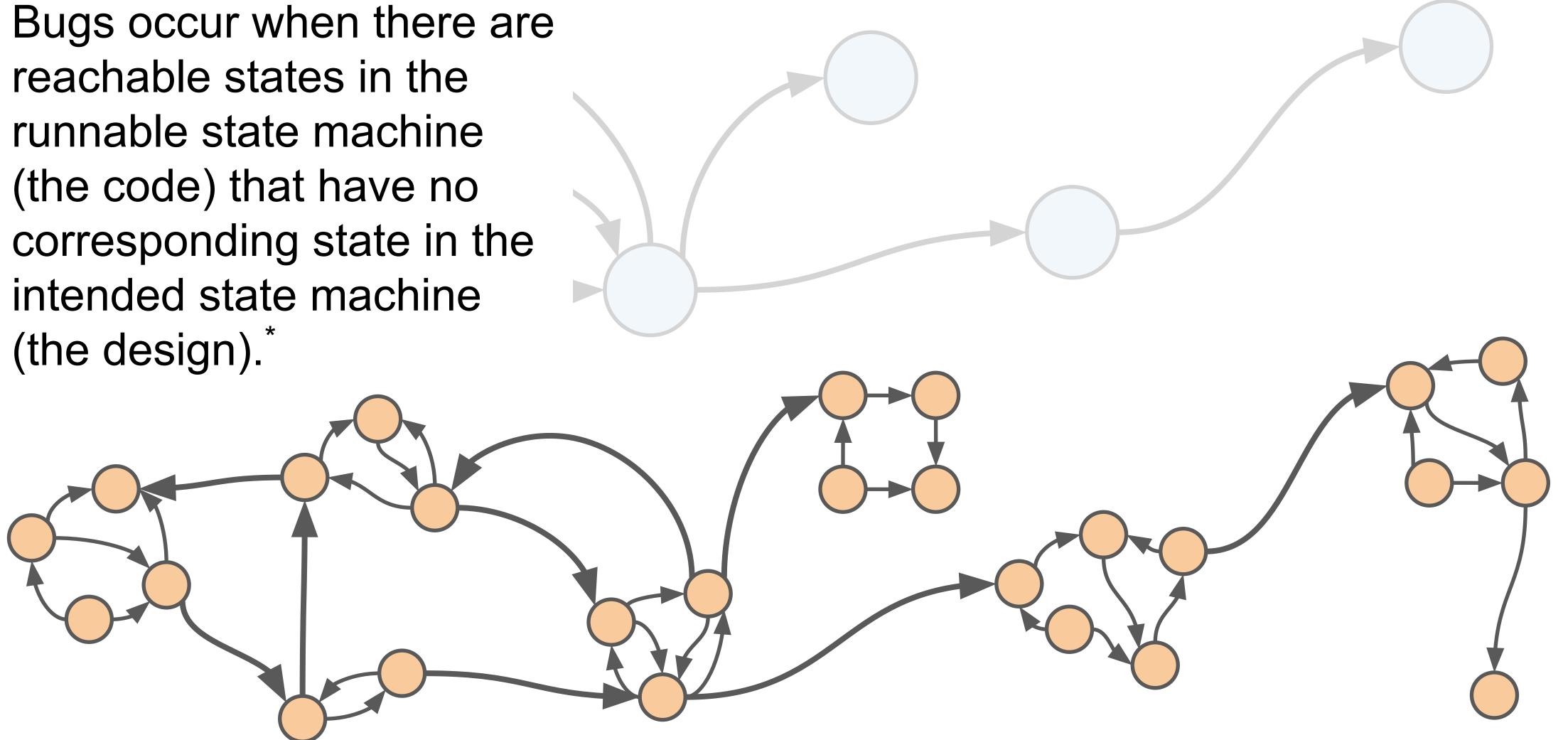
A physical CPU cannot directly execute this abstract state machine.





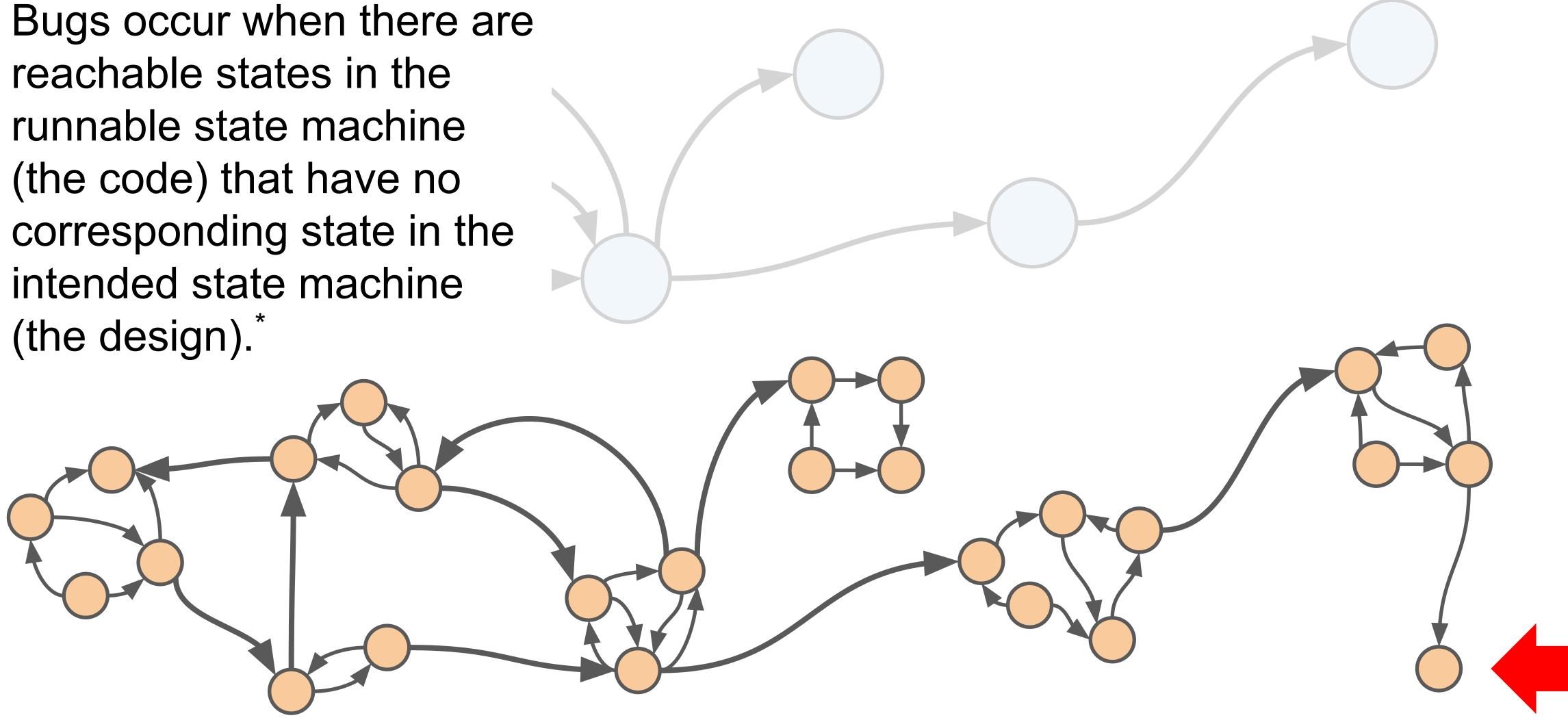
machine translated into code that can be run on a physical

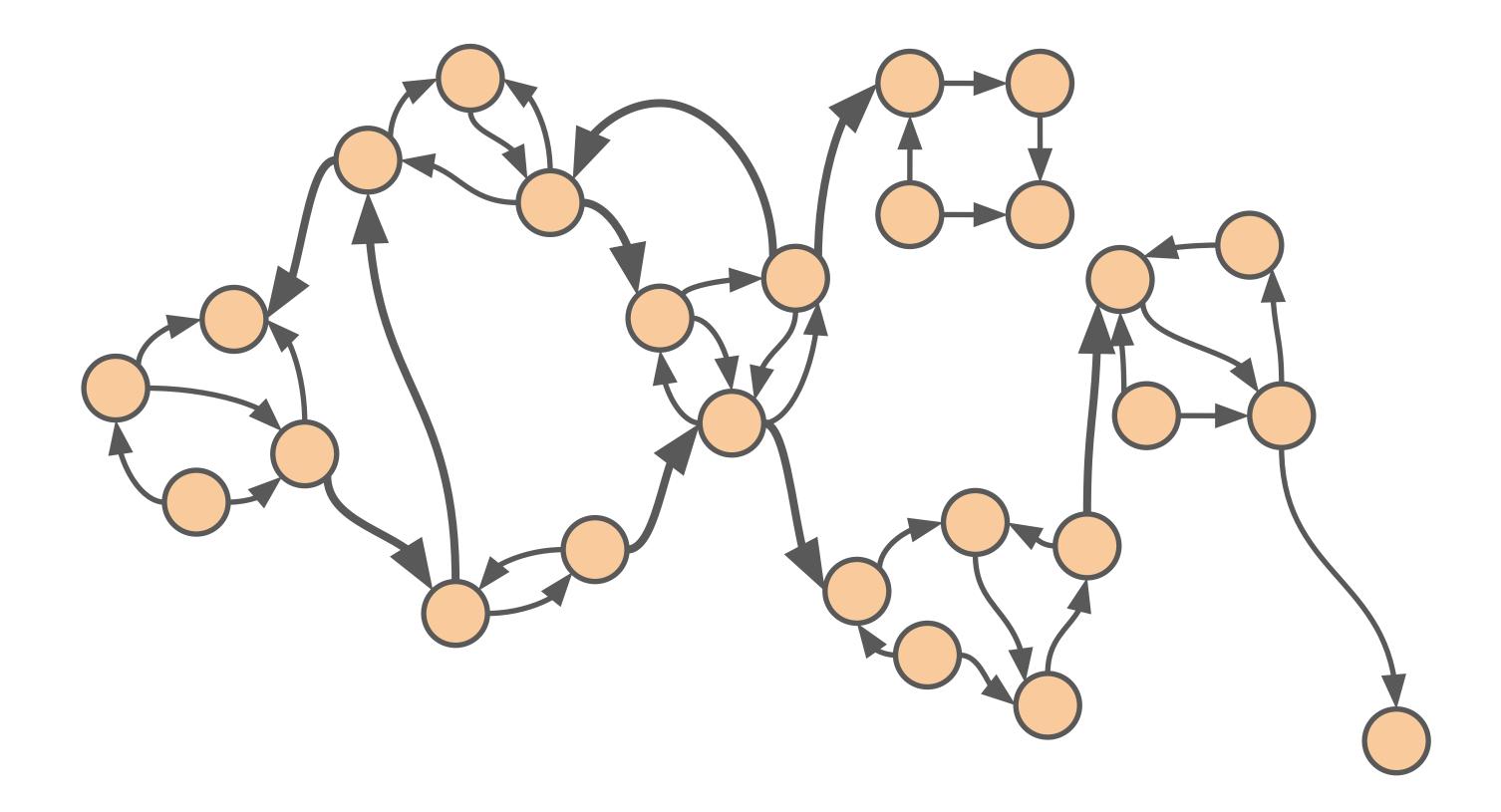
* Not quite true: that code still needs to be translated to machine code, which introduces another level of state machines emulating state machines.

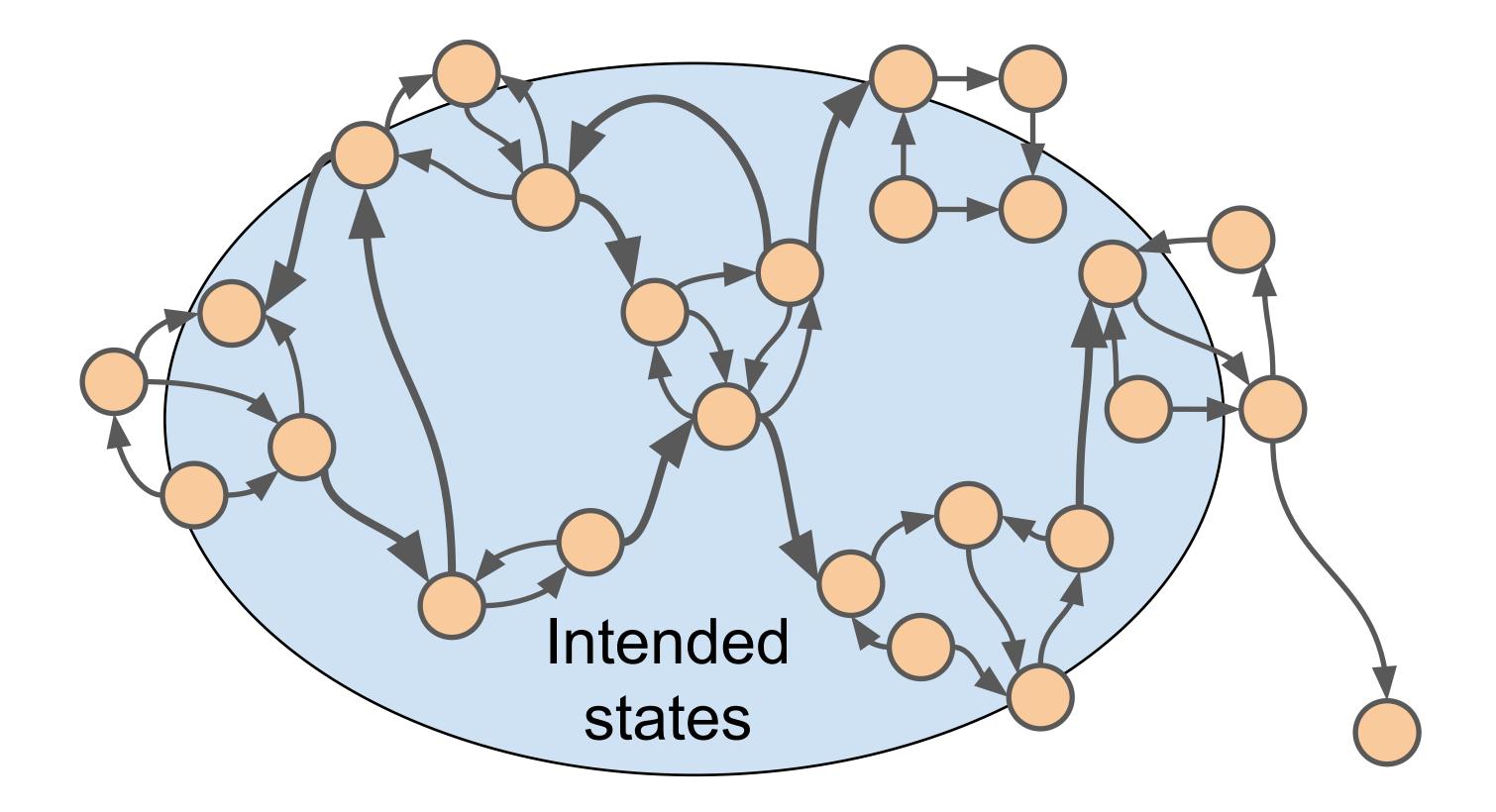


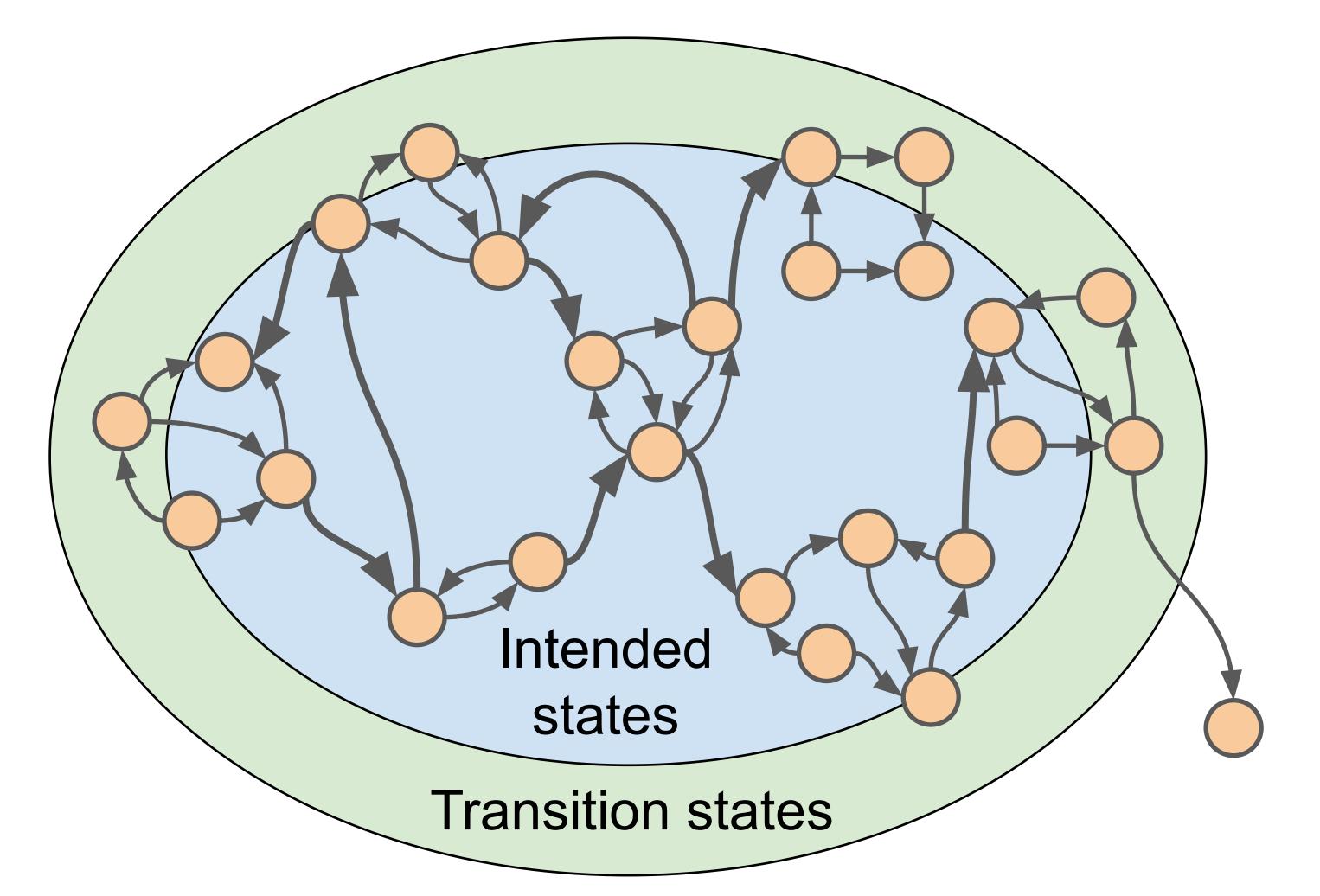
* Not the full picture: the initial design itself could have issues (design issues) which still count as software bugs.

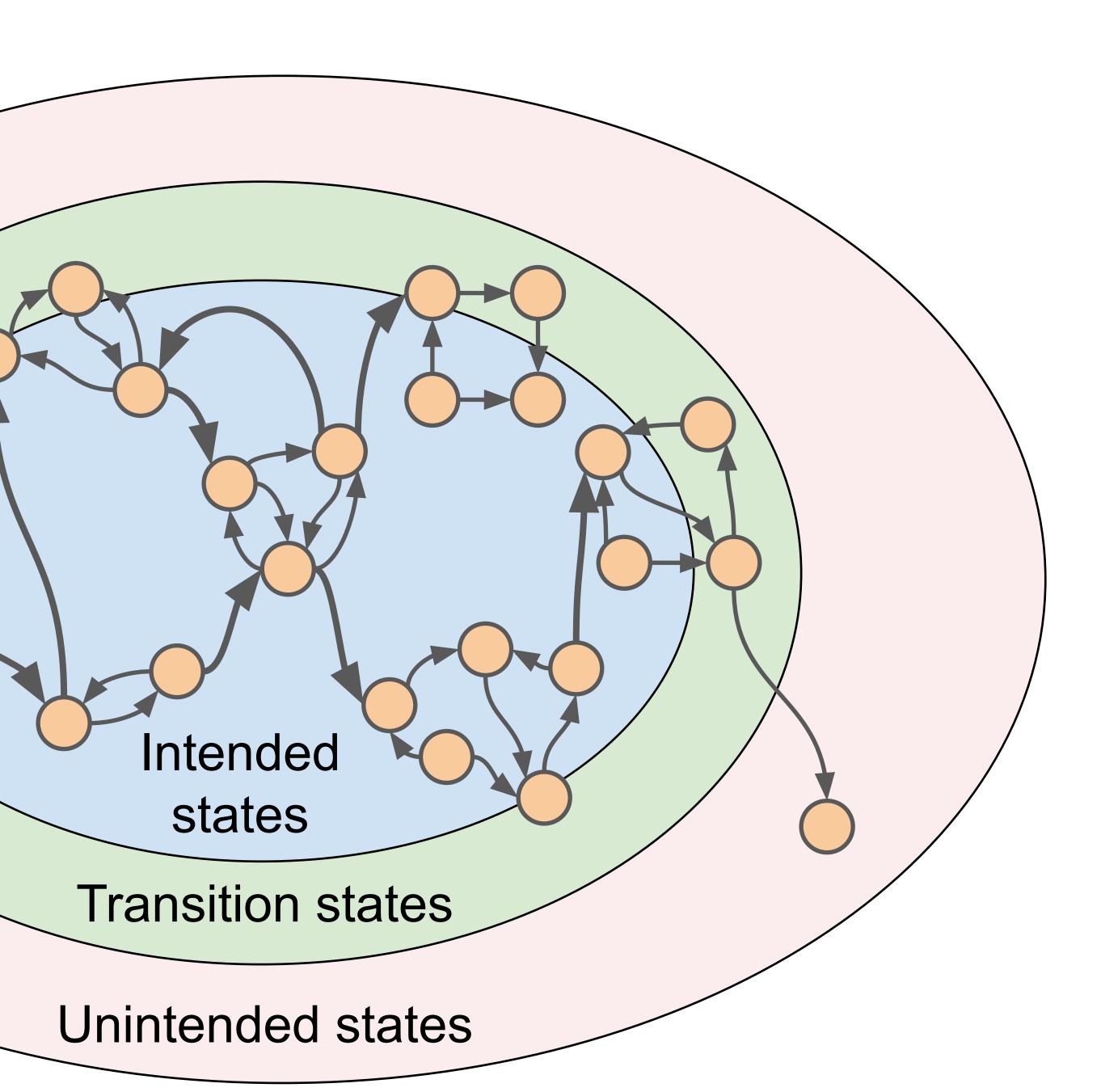




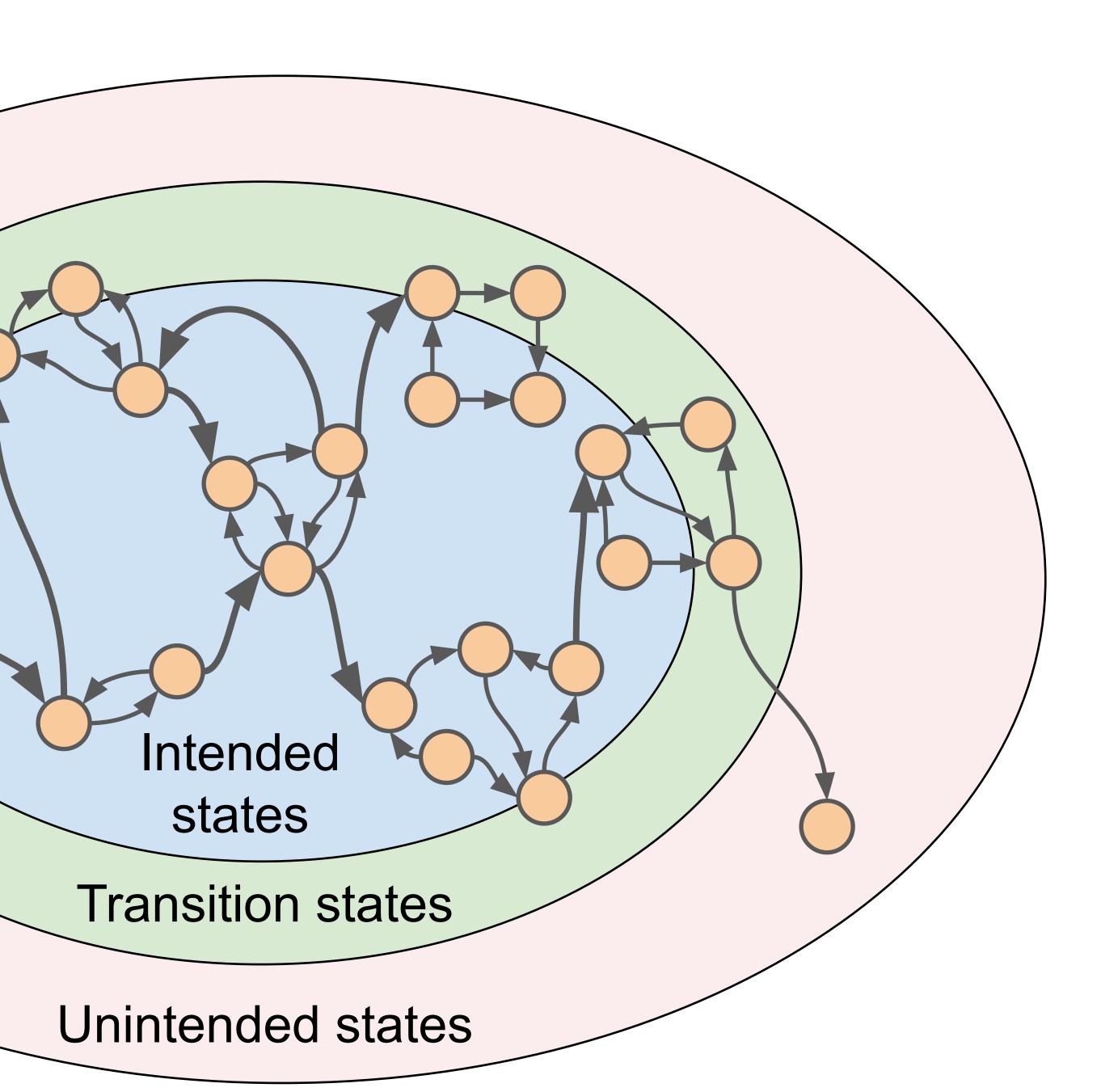


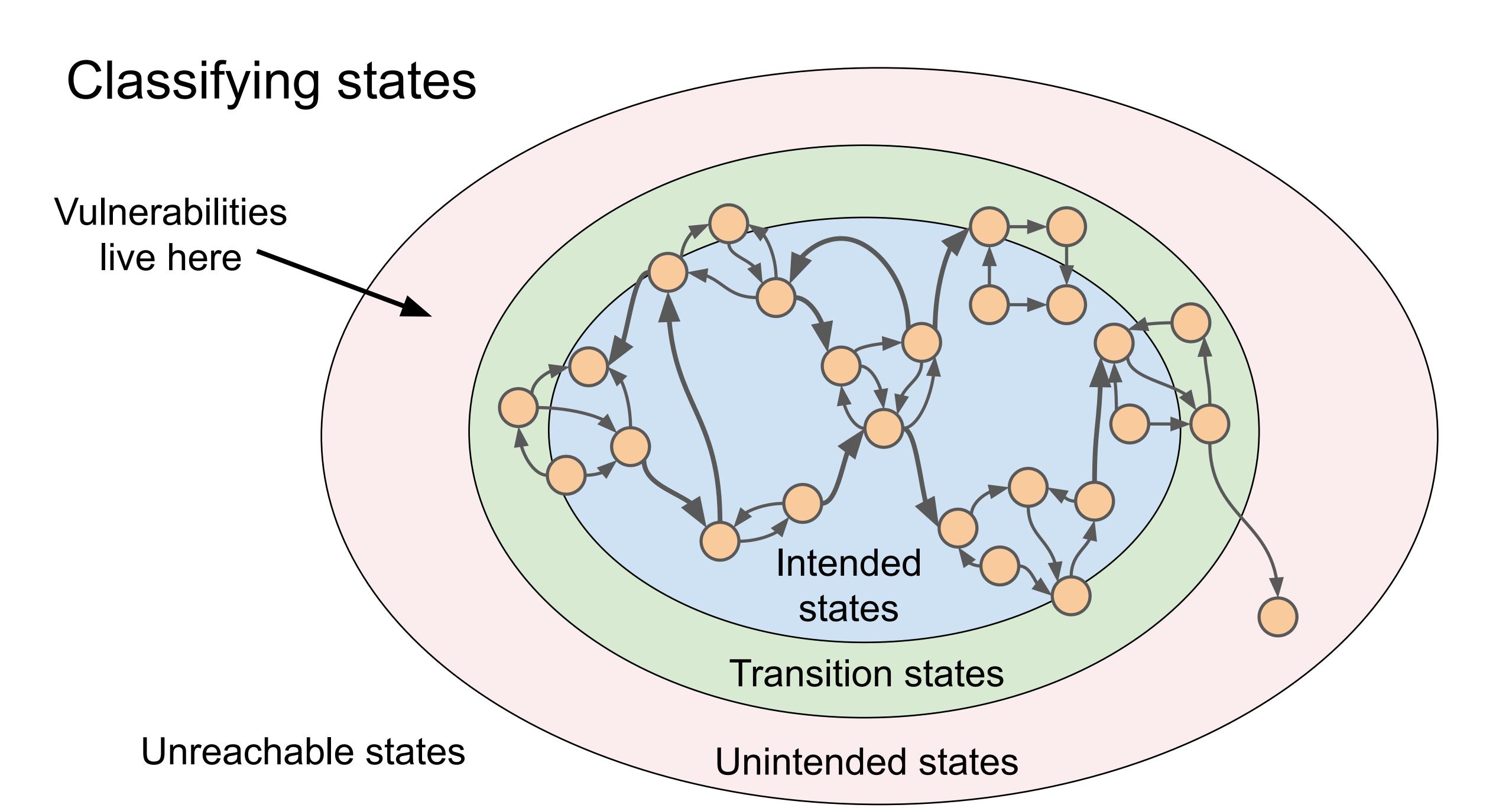






Unreachable states

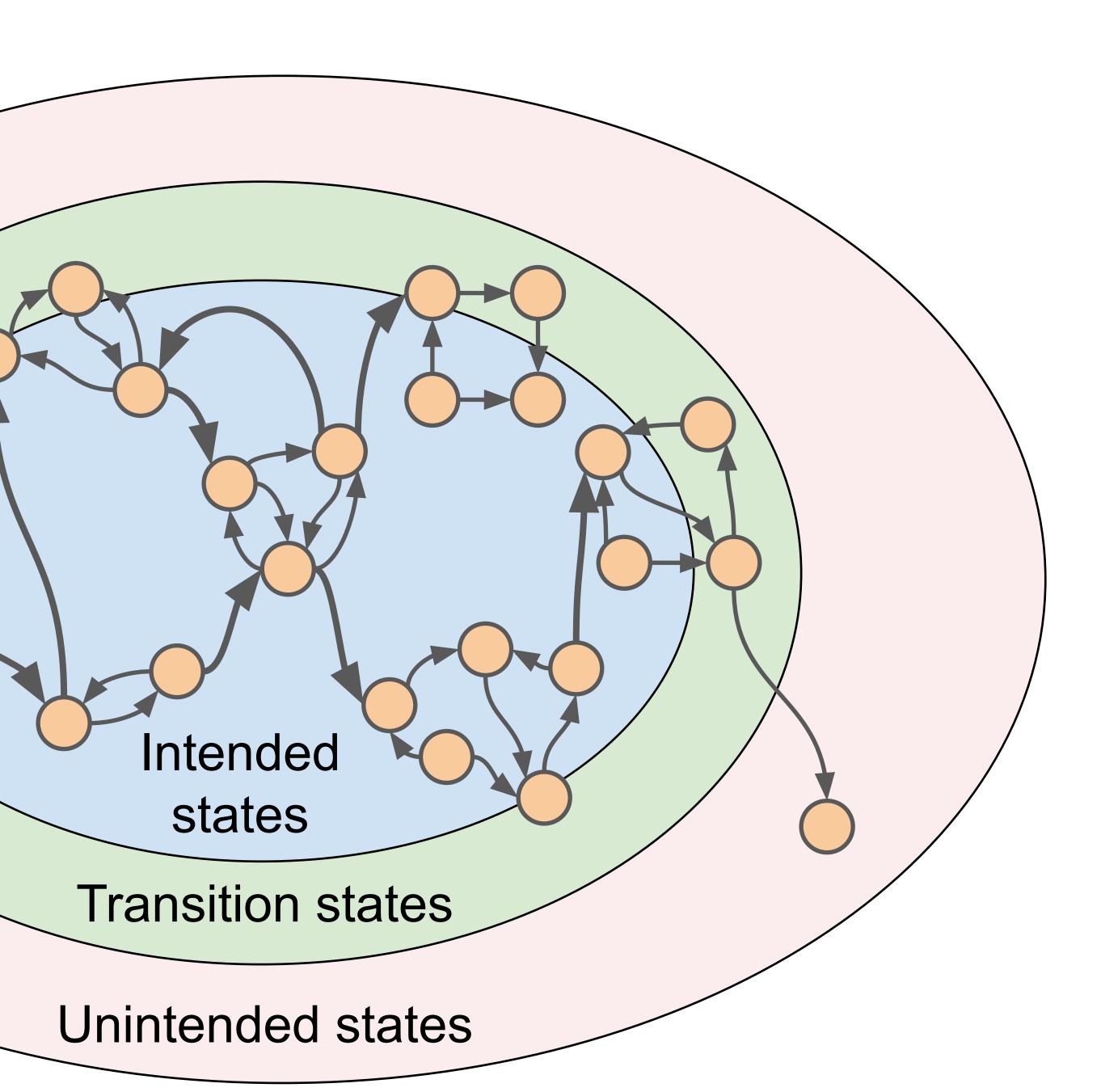




Vulnerabilities live here ~

Exploitation is making the program do "interesting" transitions in the unintended state space.

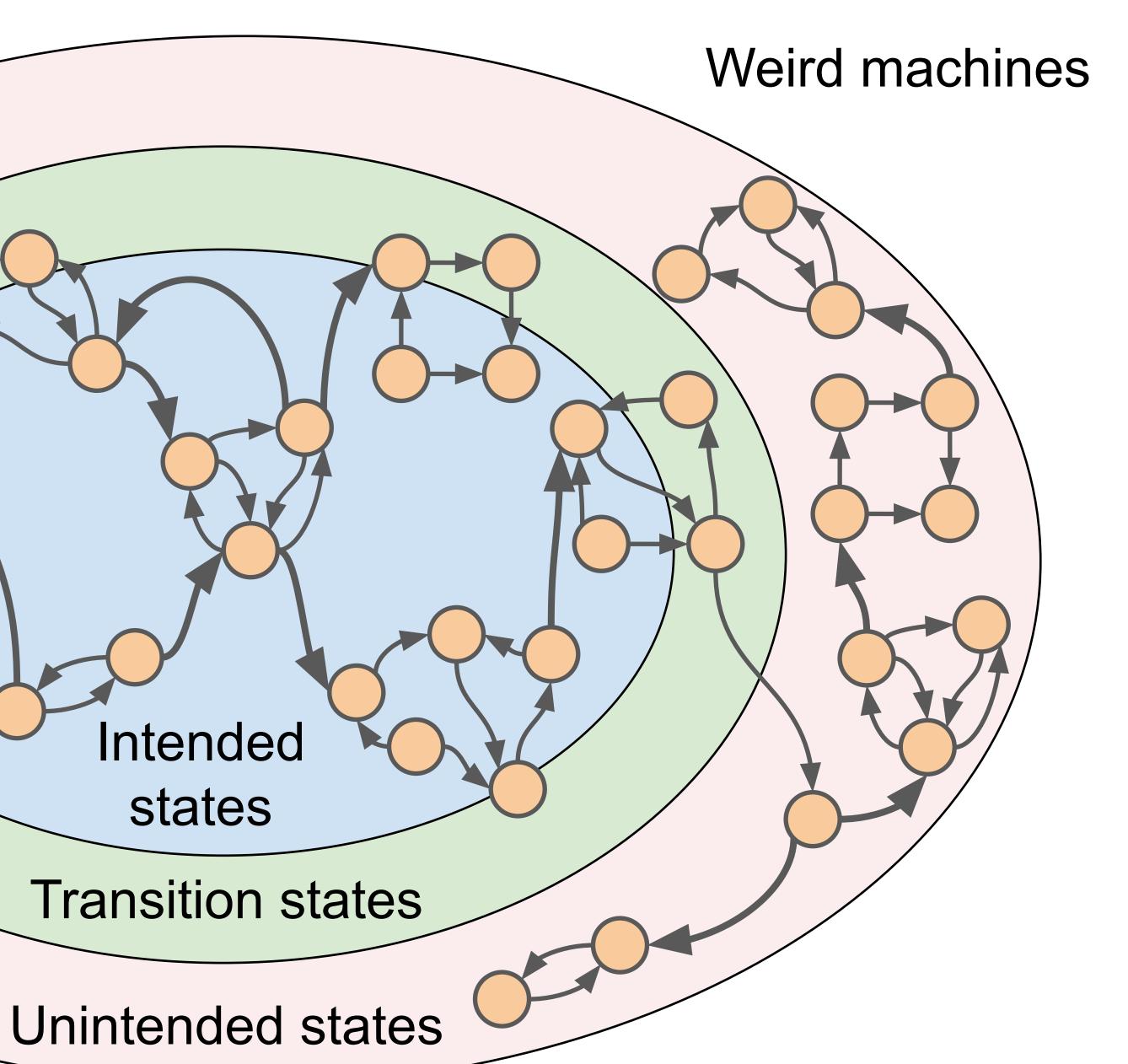
Unreachable states



Vulnerabilities live here ~

Exploitation is making the program do "interesting" transitions in the unintended state space.

Unreachable states



http://www.dullien.net/thomas/weird-machines-exploitability.pdf

Common categories of software bugs

Functionality bug: The code has bad transitions but only between validly represented states

Implementation bug: Code introduces new states not represented in the conceptual state machine

Lack of length checks introduces new "stack corruption" state

- **Design issue:** The conceptual state machine does not meet the intended goals
 - The firewall's remote interface is designed with a hardcoded admin password
 - The save button code is broken, no transition to "saving the file" state

Other ways to reach unintended states

Hardware fault: The hardware suffers a glitch that causes a transition to an unintended state even if the code is perfect

where one candidate has an impossible number of votes

Transmission error: The code is correct but is corrupted in-flight

- A cosmic ray causes a bit flip in a voting machine's memory, causing a state
- A program downloaded from the internet suffers packet corruption, so the program that is run has a different state machine from the one that was sent

This list is not intended to be exhaustive; merely to illustrate the myriad ways that unintended states may enter a system; deciding which ones to defend against is one step of proper threat modeling.



For any interesting program, it is essentially impossible to manually explore the full state space to find the unintended states

Fuzzing

Fuzzing

Find bugs in a program by feeding it random, corrupted, or unexpected data Idea: Random inputs will explore a large part of the state space Any crash is a bug, but only some bugs are exploitable Works best on programs that parse files or process complex input data

- Some unintended states are observable as crashes (SIGSEGV, abort())

Fuzzing example

Fuzzing can be as simple as:

How could we do better?

Randomly corrupt real JPEG files

Reference the JPEG spec so that we generate only "JPEG-looking" data

Measure the JPEG parser to see how deep we're getting in the code

cat /dev/random | head -c 512 > rand.jpeg; open rand.jpeg



Common fuzzing strategies

Mutation-based fuzzing

Randomly mutate test cases from some corpus of input files

Generation-based (smart) fuzzing

Generate test cases based on a specification for the input format

Coverage guided fuzzing

Measure code coverage of test cases to guide fuzzing towards new (unexplored) program states

This is not a rigid taxonomy: fuzzers often employ multiple strategies.



Mutation-based fuzzing

Randomly mutate test cases from some corpus of input files

- 1. Collect a corpus of inputs that explores as many states as possible
- 2. Perturb inputs randomly, possibly guided by heuristics

Modify: bit flips, integer increments

Substitute: small integers, large integers, negative integers

- 3. Run the program on the inputs and check for crashes
- 4. Go back to step 2

Can mutation-based "dumb" fuzzing be successful?

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)

Found 64 exploitable-looking crashes

Dumb fuzzing is often way more successful than it has any right to be

In 2010, Charlie Miller fuzzed PDF viewers using the following mutation program:

Mutation-based fuzzing

Advantages

Simple to set up and run

Can use off-the-shelf software (possibly with a harness) for many programs

Limitations

Results depend strongly on the quality of the initial corpus Coverage may be shallow for formats with checksums or validation

Generation-based (smart) fuzzing

Generate test cases based on a specification for the input format

- 1. Convert a specification of the input format (RFC, etc.) into a generative procedure
- 2. Generate test cases according to the procedure and introduce random perturbations
- 3. Run the program on the inputs and check for crashes
- 4. Go back to step 2

Syzkaller

A kernel system call fuzzer that uses test case generation and coverage

Test cases are sequences of syscalls generated from syscall descriptions

Runs the test case program in a VM

Kernel crashes in the VM indicate potential Local Privilege Escalation (LPE) vulnerabilities Raw Blame

304 lines (235 sloc) 15.7 KB

Syscall descriptions

syzkaller uses declarative description of syscall interfaces to manipulate programs (sequences of syscalls). Below you can see (hopefully self-explanatory) excerpt from the descriptions:

```
open(file filename, flags flags[open_flags], mode flags[open_mode]) fd
read(fd fd, buf buffer[out], count len[buf])
close(fd fd)
open_mode = S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_
```

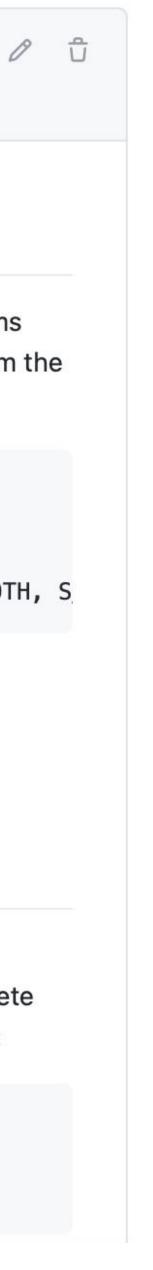
The descriptions are contained in sys/\$0S/*.txt files. For example see the sys/linux/dev_snd_midi.txt file for descriptions of the Linux MIDI interfaces.

A more formal description of the description syntax can be found here.

Programs

The translated descriptions are then used to generate, mutate, execute, minimize, serialize and deserialize programs. A program is a sequences of syscalls with concrete values for arguments. Here is an example (of a textual representation) of a program:

```
r0 = open(&(0x7f000000000)="./file0", 0x3, 0x9)
read(r0, &(0x7f000000000), 42)
close(r0)
```



Generation-based (smart) fuzzing

Advantages

Input format/protocol complexity is not a limit on coverage depth

Limitations

Requires a lot of effort to set up Successful fuzzers are often domain-specific

- Can get deeper coverage faster by leveraging knowledge of the input format

- Coverage limited by accuracy of the spec; implementation may diverge

Coverage guided fuzzing

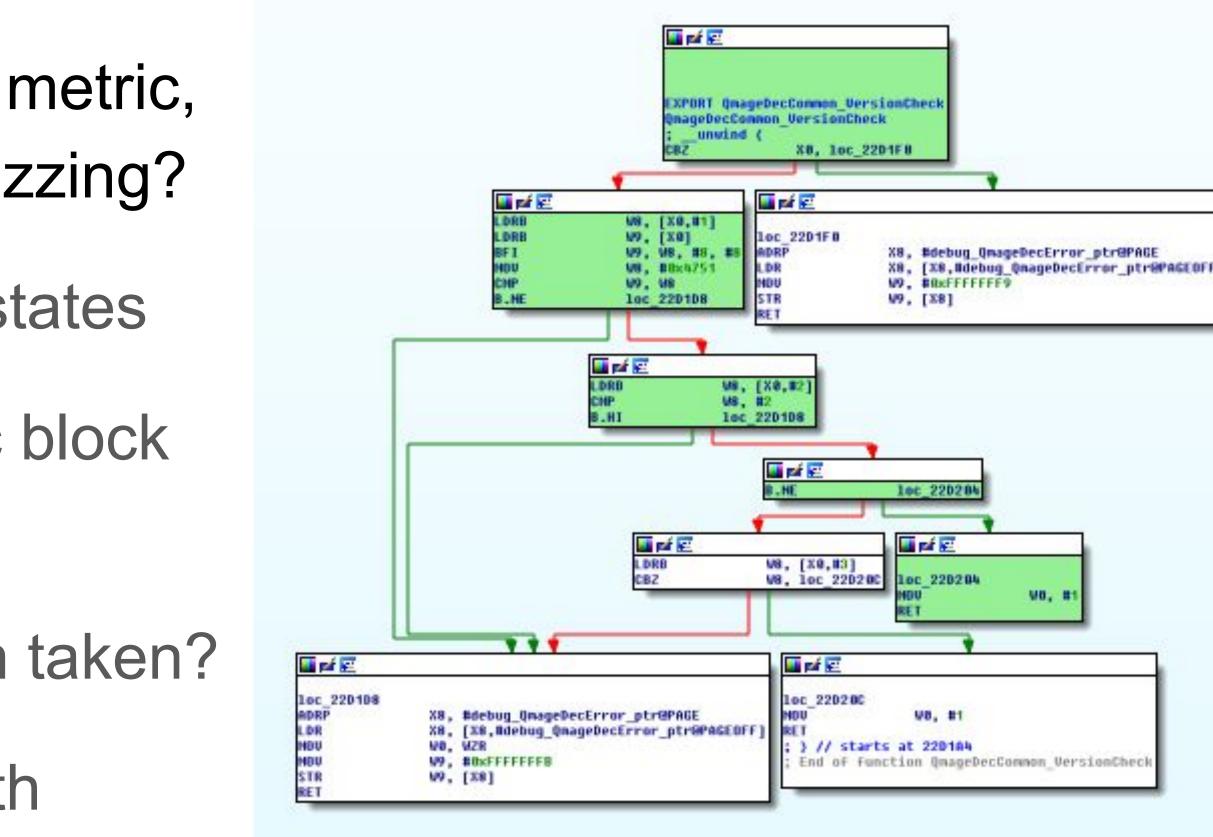
Key insight: code coverage is a useful metric, why not use it as **feedback** to guide fuzzing?

Prefer test cases that reach new states

Basic block coverage: Has this basic block in the CFG been run?

Edge coverage: Has this branch been taken?

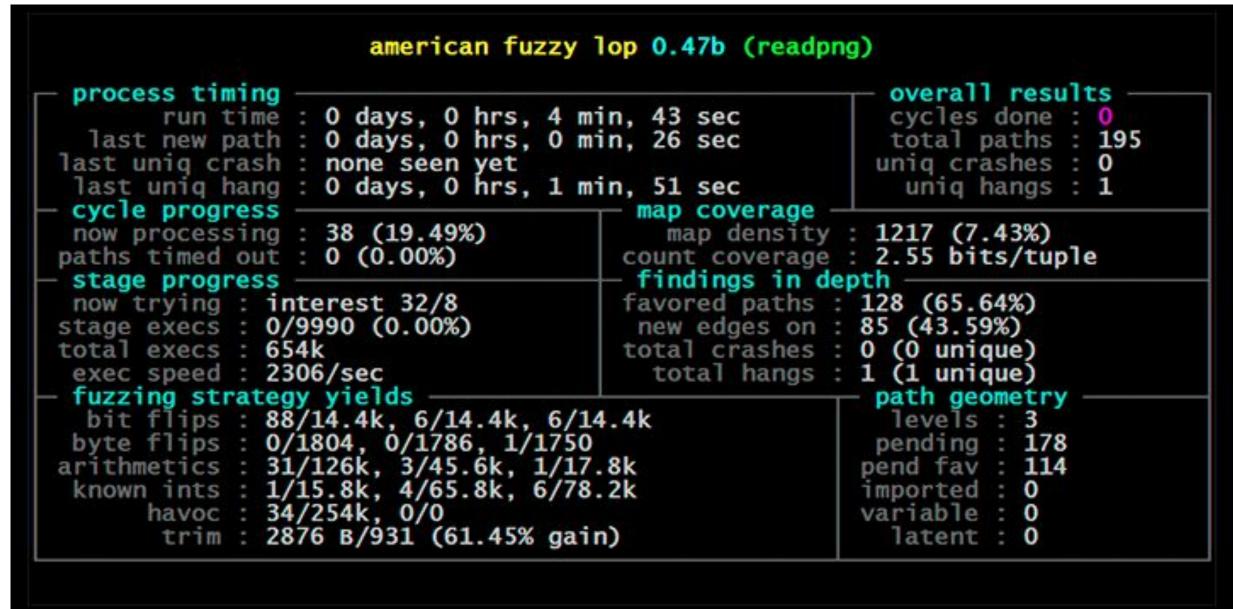
Path coverage: Has this particular path through the program been taken?





american fuzzy lop (AFL)

- Compile the program with instrumentation to measure coverage
- 2. Trim the test cases in the queue to the smallest size that doesn't change the program behavior
- 3. Create new test cases by mutating the files in the queue using traditional fuzzing strategies
- 4. If new coverage is found in a mutated file, add it into the queue
- 5. Go back to step 2



Coverage guided fuzzing

Advantages

Combines well with other fuzzing strategies Wildly successful track record Limitations

Not a panacea to bypass strong checksums or input validation Still doesn't find all types of bugs (e.g. race conditions)

Very good at finding new program states, even if the initial corpus is limited

Real world example: Fuzzing the Samsung Qmage codec

Thursday, July 23, 2020

MMS Exploit Part 2: Effective Fuzzing of the Qmage Codec

Posted by Mateusz Jurczyk, Project Zero

This post is the second of a multi-part series capturing my journey from discovering a vulnerable little-known Samsung image codec, to completing a remote zero-click MMS attack that worked on the latest Samsung flagship devices. New posts will be published as they are completed and will be linked here when complete.

- MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface
- [this post]
- MMS Exploit Part 3: Constructing the Memory Corruption Primitives
- MMS Exploit Part 4: MMS Primer, Completing the ASLR Oracle
- MMS Exploit Part 5: Defeating Android ASLR, Getting RCE

Introduction

In Part 1, I discussed how I discovered the "Qmage" image format natively supported on all modern Samsung phones, and how I traced its roots to Android boot animations and even some pre-Android phones. At this stage of the story, we also know that the codec seems very fragile and is likely affected by bugs, and that it constitutes a zeroclick remote attack surface via MMS and the default Samsung Messages app. I was at this point of the project in early December 2019. The next logical step was to thoroughly fuzz it – the code was definitely too extensive and complex to approach with a manual audit, especially without access to the original source or expertise of the inner workings of the format. As a big fan of fuzzing, I hoped to be able to run it in accordance with the current state of the art: efficiently (without unnecessary overhead), at scale, with code coverage information, reliable reproducibility and effective deduplication. But how to achieve all this with a codec that is part of Android, accessible only through Skia image API, and precompiled for the ARM/ARM64 architectures only? Read on to find out!

Writing the test harness

The fuzzing harness is usually one of the most critical pieces of a successful fuzzing session, and it was the first thing I started working on. I published the end result of my work as SkCodecFuzzer on GitHub, and it can be used as a reference while reading this post. My initial goal with the loader was to write a Linux command-line program that could run on physical Android devices, and use the Skia SkCodec interface to load and decode an input image file in exactly the same way (or at least as closely as

- In 2019, Mateusz Jurczyk discovered the Qmage image codec included on Samsung smartphones
 - Reachable via zero-click MMS
- The code looks fragile but the library is closed source
 - Few examples of Qmage files
- Mateusz developed a harness to enable large-scale coverage-guided fuzzing of the Qmage codec

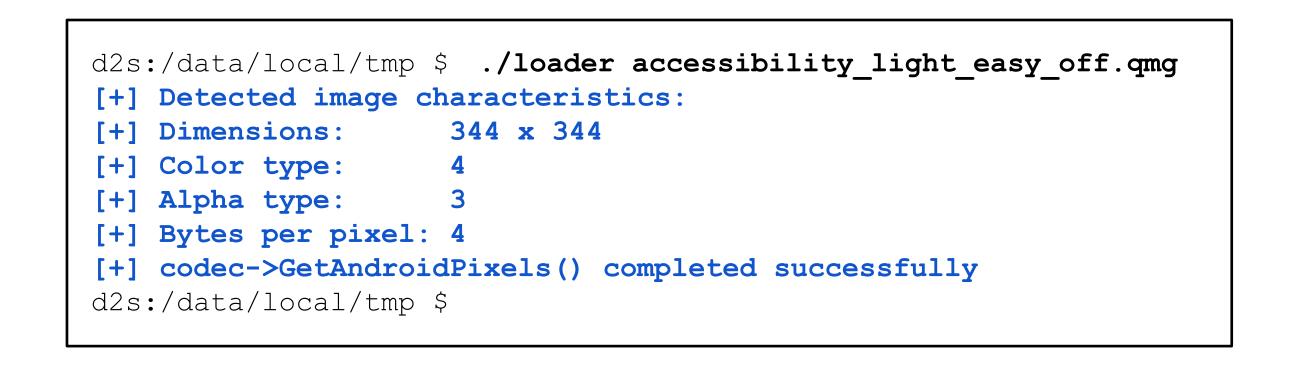


Fuzzing the Samsung Qmage image codec: harness

A fuzzing harness was written to call the interesting functions in the library and supply the test case input from the fuzzer

Linux machine

Easier to get 1000 Linux cores than 1000 Samsung Galaxy phones



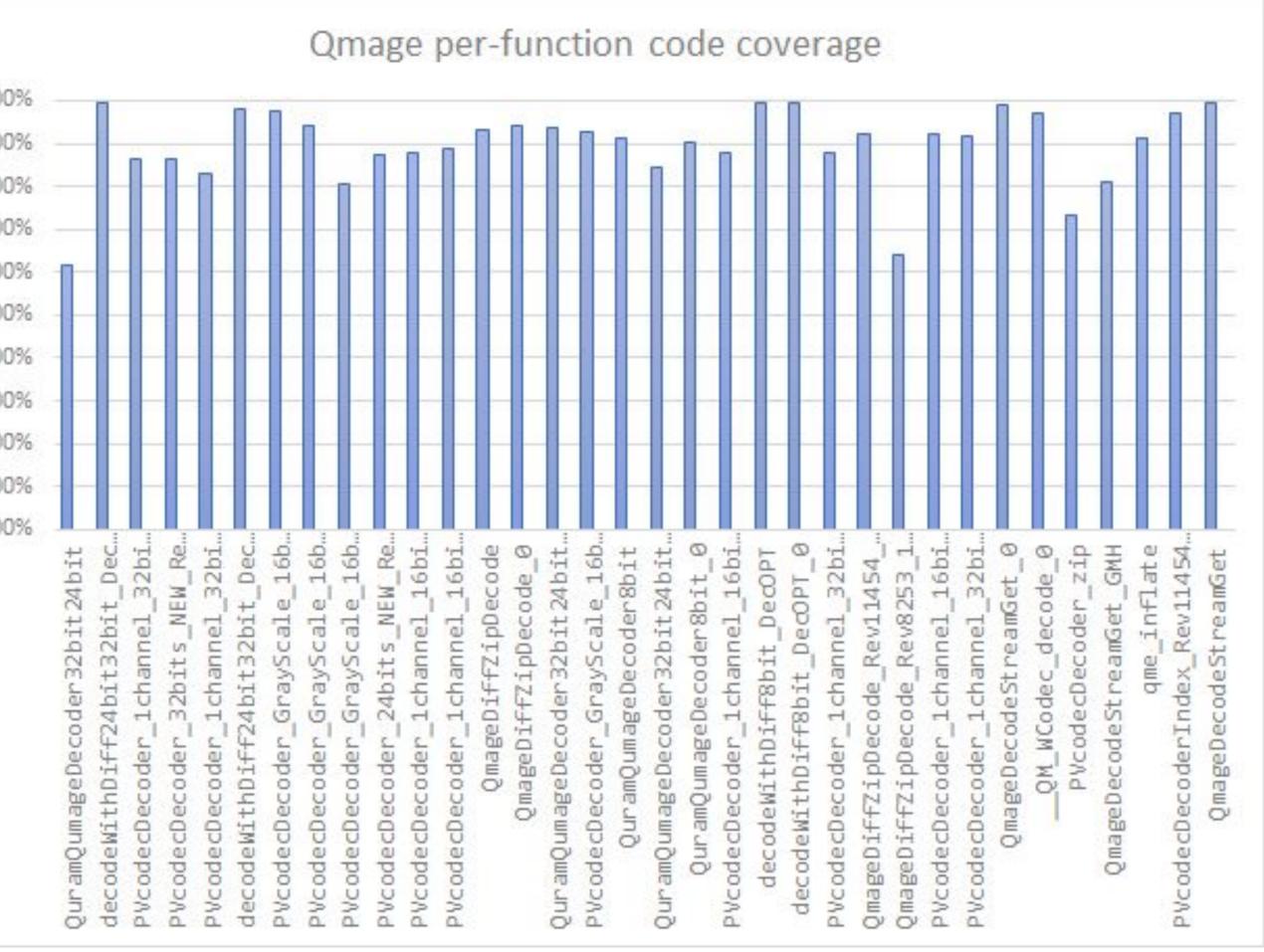
An emulator (gemu-aarch64) was used to run the harness and Qmage library on a



Fuzzing the Samsung Qmage image codec: coverage

Code coverage was collected by modifying qemu-aarch64 to trace executed PC addresses

Coverage feedback compensated for the small number of initial test cases 100.00% 90.00% 80.00% 70.00% 60.00% 50.00% 30.00% 20.00% 10.00%



Fuzzing the Samsung Qmage image codec: results

Category	Count	Pe
write	174	
read-memcpy	124	
read-vector	18	
read-32	3	
read-16	52	
read-8	34	
read-4	703	
read-2	393	
read-1	3322	(
sigabrt	3	
null-deref	392	

rcentage
3.33%
2.38%
0.34%
0.06%
1.00%
0.65%
3.47%
7.53%
63.66%
0.06%

7.51%

4 weeks of fuzzing

87.3% coverage of the Qmage codec

5218 unique crashes



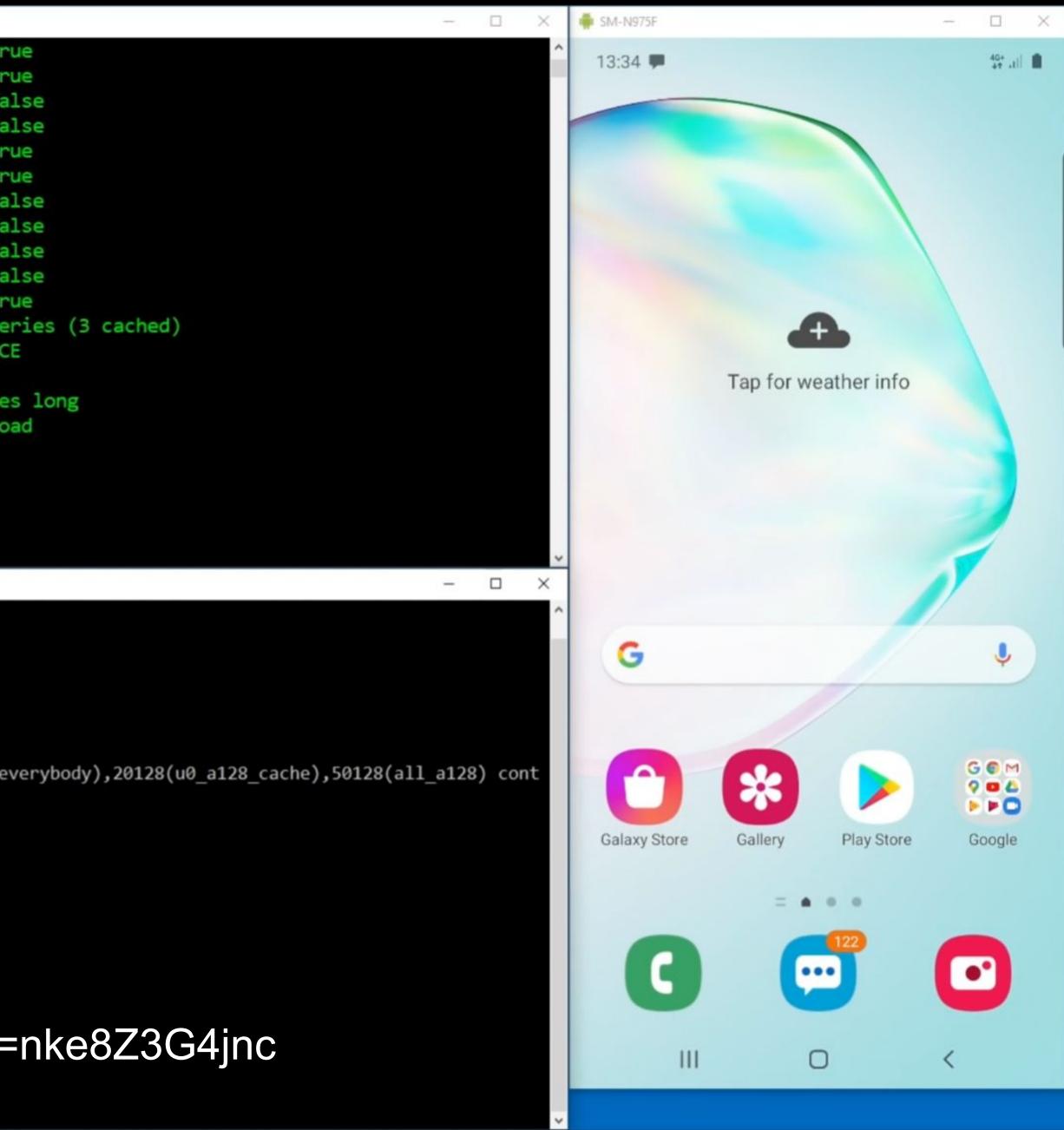
```
C:\Windows\System32\cmd.exe
```

2020-04-22	13:21:08,765	[INFO]	Range [765a760000 765a760fff] is readable: tr
2020-04-22	13:22:25,896	[INFO]	Range [765a760000 765a7e7fff] is readable: tr
2020-04-22	13:24:03,055	[INFO]	Range [765a760000 765a86ffff] is readable: fa
2020-04-22	13:25:10,218	[INFO]	Range [765a7e8000 765a82bfff] is readable: fa
2020-04-22	13:25:58,355	[INFO]	Range [765a7e8000 765a809fff] is readable: tr
2020-04-22	13:27:16,491	[INFO]	Range [765a80a000 765a81afff] is readable: tr
2020-04-22	13:28:53,653	[INFO]	Range [765a81b000 765a823fff] is readable: fa
2020-04-22	13:30:00,820	[INFO]	Range [765a81b000 765a81ffff] is readable: fa
2020-04-22	13:31:07,988	[INFO]	Range [765a81b000 765a81dfff] is readable: fa
2020-04-22	13:32:15,149	[INFO]	Range [765a81b000 765a81cfff] is readable: fa
2020-04-22	13:33:02,294	[INFO]	Range [765a81b000 765a81bfff] is readable: tr
2020-04-22	13:33:02,294	[INFO]	linker64 address 0x765a701000 found after 89 que
2020-04-22	13:33:02,295	[INFO]	ASLR defeated, crafting a corrupted image for RC
2020-04-22	13:33:02,341	[INFO]	Generator stdout: done!
2020-04-22	13:33:02,342	[INFO]	RCE exploit image successfully created, 533 byte
2020-04-22	13:33:02,342	[INFO]	Crashing Messages before sending the final paylo
2020-04-22	13:33:04,389	[INFO]	Cooldown, sleeping for 65 seconds
2020-04-22	13:34:09,390	[INFO]	Woke up, sending the exploit
2020-04-22	13:34:11,450	[INFO]	Exploit sent, enjoy your reverse shell!

13:34:11 Vexillium>

```
j00ru@vps12284:~$ # We will get the reverse shell here
j00ru@vps12284:~$ nc -l -p 1337 -v
Listening on [0.0.0.0] (family 0, port 1337)
                           54194 received!
Connection from
/bin/sh: can't find tty fd: No such device or address
/bin/sh: warning: won't have full job control
:/ $ id
uid=10128(u0_a128) gid=10128(u0_a128) groups=10128(u0_a128),3002(net_bt),3003(inet),9997(everybody),20128(u0_a128_cache),50128(all_a128) cont
ext=u:r:platform_app:s0:c512,c768
:/$
```

https://www.youtube.com/watch?v=nke8Z3G4jnc





Another cool fuzzer: Fuzzilli

Very successful JavaScript fuzzer

Principle: Translate JavaScript to a dense Intermediate Language (IL), and fuzz the IL

<> Code (!) Issue	s 17 11 Pull requests 5 🖓 Dis	scussions 🕑 Actions 🔟 Projects 🕕 Security 🗠 Ins	sights
	양 master → 양 1 branch ⊙ 2 tags	Go to fil	le
	wbowling Added entry for CVE-2019	-8844 to README.md (#190) 🗸 0a83e0c 17 days ago	1 216
	Cloud	Updated V8 and JSC targets	4 m
	Docs	Add documentation for type determination (#142)	7 m
	Sources	Reset any blocked signals after forking in libreprl	2
	Targets	Fixed QuickJS patch	2 m
	Tests	Implemented JavaScript Classes	5 m
	CONTRIBUTING.md	Fuzzilli is now open source!	2
	LICENSE	Fuzzilli is now open source!	2
	Package.swift	Added tiny benchmarking suite (#140)	7 m
	README.md	Added entry for CVE-2019-8844 to README.md (#190)	1
	i≣ README.md		

Googleprojectzero / fuzzilli

Fuzzilli

A (coverage-)guided fuzzer for dynamic language interpreters based on a custom intermediate language ("FuzzIL") which can be mutated and translated to JavaScript.

Written and maintained by Samuel Groß, saelo@google.com.

Usage

The basic steps to use this fuzzer are:

16 commits months ago months ago 25 days ago months ago 2 years ago 2 years ago months ago 17 days ago

业 Code -

Fuzzing summary

Off-the-shelf fuzzers are excellent at finding bugs

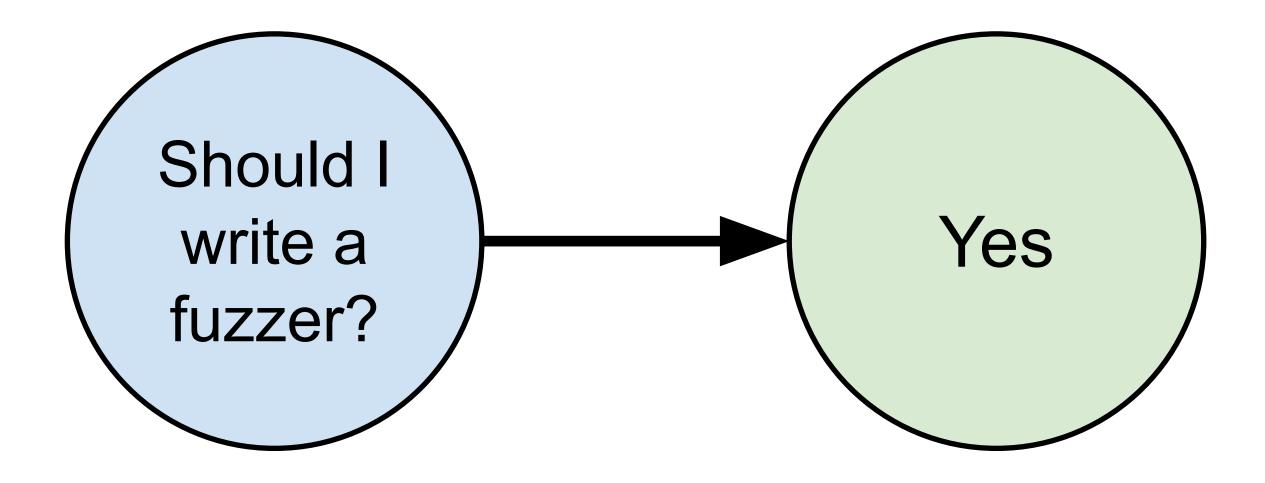
Custom fuzzers are also excellent at finding bugs

Different fuzzers often find different bugs

Relatively easy to get started

Fuzzing doesn't find all types of bugs

This code parses untrusted data





Dynamic analysis

Dynamic analysis

Analyze a program's behavior by actually running its code

May be combined with compile-time modifications like instrumentation

Can modify the program's behavior dynamically

Useful for rapid experimentation

Often complements fuzzing very well

Running A Program Under Valgrind

Like the debugger, Valgrind runs on your executable, so be sure you have compiled an up-to-date copy of your program. Run it like this, for example, if your program is named memoryLeak:

\$ valgrind ./memoryLeak

Valgrind will then start up and run the specified program inside of it to examine it. If you need to pass command-line arguments, you can do that as well:

\$ valgrind ./memoryLeak red blue

When it finishes, Valgrind will print a summary of its memory usage. If all goes well, it'll look something like this:

```
==4649== ERROR SUMMARY: 0 errors from 0 contexts
==4649== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4649== malloc/free: 10 allocs, 10 frees, 2640 bytes allocated.
==4649== For counts of detected errors, rerun with: -v
==4649== All heap blocks were freed -- no leaks are possible.
```

This is what you're shooting for: no errors and no leaks. Another useful metric is the number of allocations and total bytes allocated. If these numbers are the same ballpark as our sample (you can run solution under valgrind to get a baseline), you'll know that your memory efficiency is right on target.

Finding Memory Errors

Memory errors can be truly evil. The more overt ones cause spectacular crashes, but even then it can be hard to pinpoint how and why the crash came about. More insidiously, a program with a memory error can still seem to work correctly because you manage to get "lucky" much of the time. After several "successful" outcomes, you might wishfully write off what appears to be a spurious catastrophic outcome as a figment of your imagination, but depending on luck to get the right answer is not a good strategy. Running under valgrind can help you track down the cause of visible memory errors as well as find lurking errors you don't even yet know about.



AddressSanitizer (ASan)

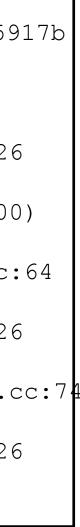
Fast memory error detector for C/C++ using compiler instrumentation and a

Out-of-bounds accesses Use-after-free Use-after-return Use-after-scope Double-free, invalid free Memory leaks

==9901==ERROR: AddressSanitizer:heap-use-after-free on address 0x60700000dfb5 at pc 0x45917b bp 0x7fff4490c700 sp 0x7fff4490c6f8 READ of size 1 at 0x6070000dfb5 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:7 #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

Typically 2x slowdown

runtime library that replaces malloc() to surround allocations with redzones



AddressSanitizer (ASan)

Fast memory error detector for C/C++ using compiler instrumentation and a

Out-of-bour

runtime library t

Use-after-fr

Use-after-re

Use-after-s

Double-free, invalid free Memory leaks

#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:7 #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

Typically 2x slowdown

Pro tip: Once coverage guided fuzzing plateaus, run the generated corpus under ASan to find bugs the fuzzer missed!

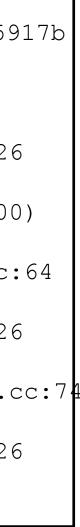
h redzones

s 0x60700000dfb5 at pc 0x45917b

bc-2.15/csu/libc-start.c:226

x6070000dfb0,0x6070000e000)

b/asan/asan_malloc_linux.cc:64



ThreadSanitizer (TSan)

Data race detector for C/C++

Similar in principle to AddressSanitizer but for race conditions

High overhead

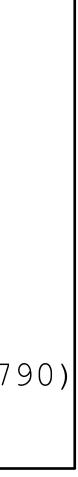
5-10x memory

5-15x slowdown

WARNING: ThreadSanitizer: data race (pid=19219) Write of size 4 at 0x7fcf47b21bc0 by thread T1: #0 Thread1 tiny race.c:4 (exe+0x0000000a360)

Previous write of size 4 at 0x7fcf47b21bc0 by main thread: #0 main tiny race.c:10 (exe+0x0000000a3b4)

Thread T1 (running) created at: #0 pthread_create tsan_interceptors.cc:705 (exe+0x0000000c790) #1 main tiny race.c:9 (exe+0x0000000a3a4)



Static analysis

Static analysis

Using a tool to analyze a program's behavior without actually running it

dynamic analysis can't

E.g., can prove that a program is free of NULL pointer dereferences

scope for improvements!

- Test whether a certain property holds or find places where it is violated
- Static analysis can prove some properties about the program that fuzzing and
- Despite lots of work in this area, there are countless interesting topics and huge

Undecidability of static analysis

Goal: Determine whether a given program satisfies a given property

This is theoretically undecidable: it reduces to the halting problem!

def solve halting problem(P, a): def new P(): **P(a)** bug() return static analyzer for bug(new P)

Soundness and completeness

- The best static analyzer can only satisfy one of the following:^{*}
 - **Soundness:** Everything that the static analyzer finds is a bug
 - But some bugs may be missed!
 - **Completeness**: The static analyzer finds every bug
 - But there may be false positives!
- Most static analyzers are neither sound nor complete



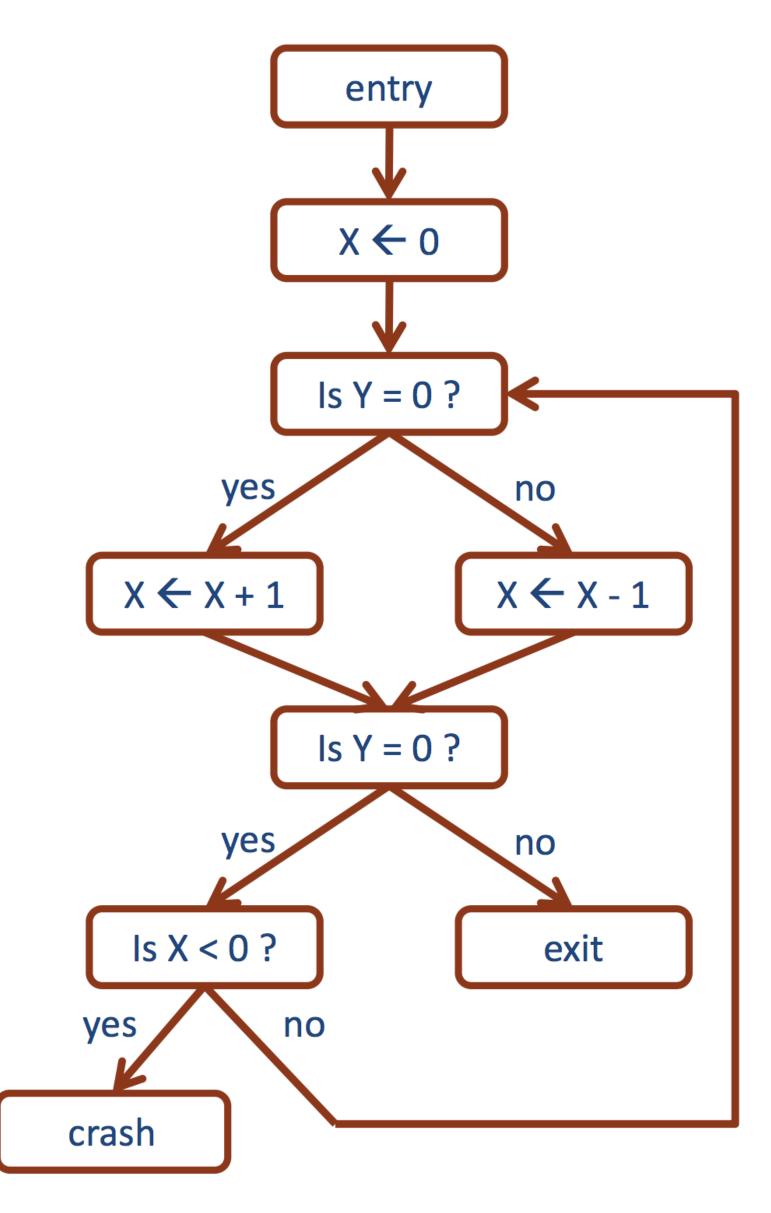
Soundness vs Completeness

sound (over-approximate) analysis

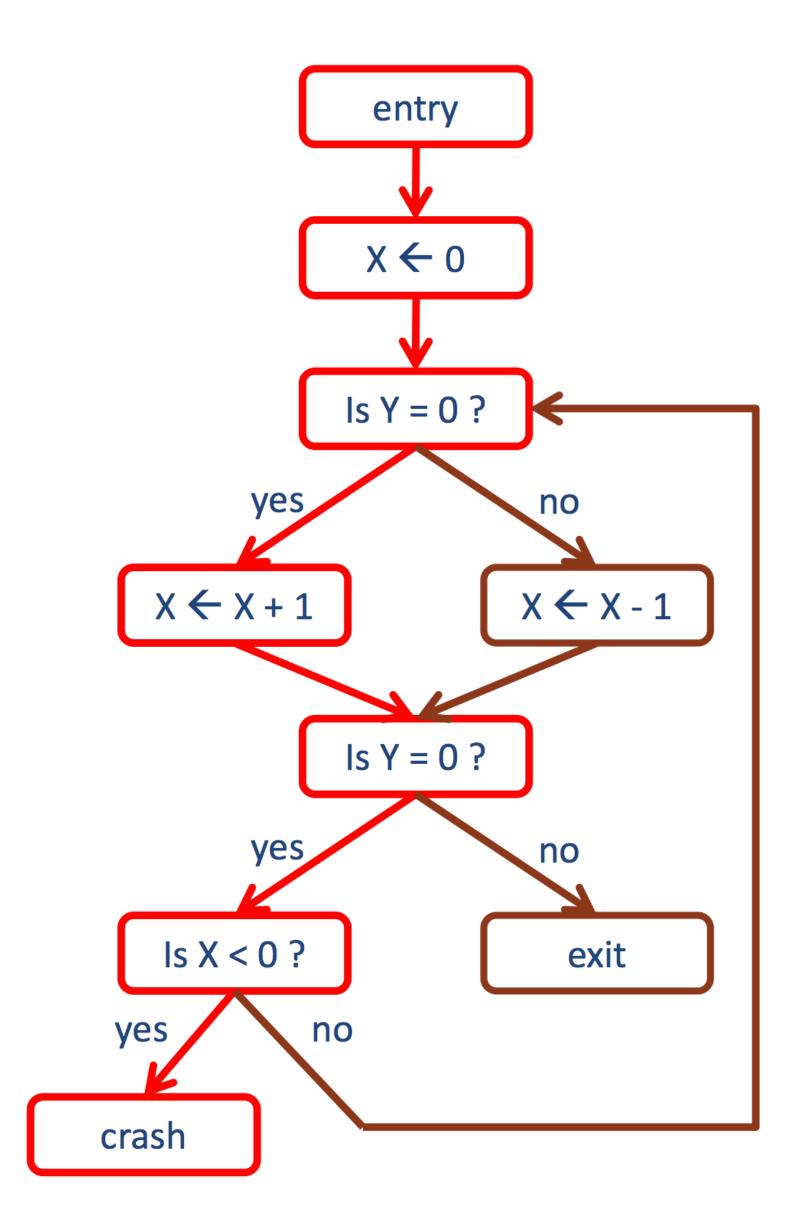
possible program behaviors

complete (under-approximate) analysis

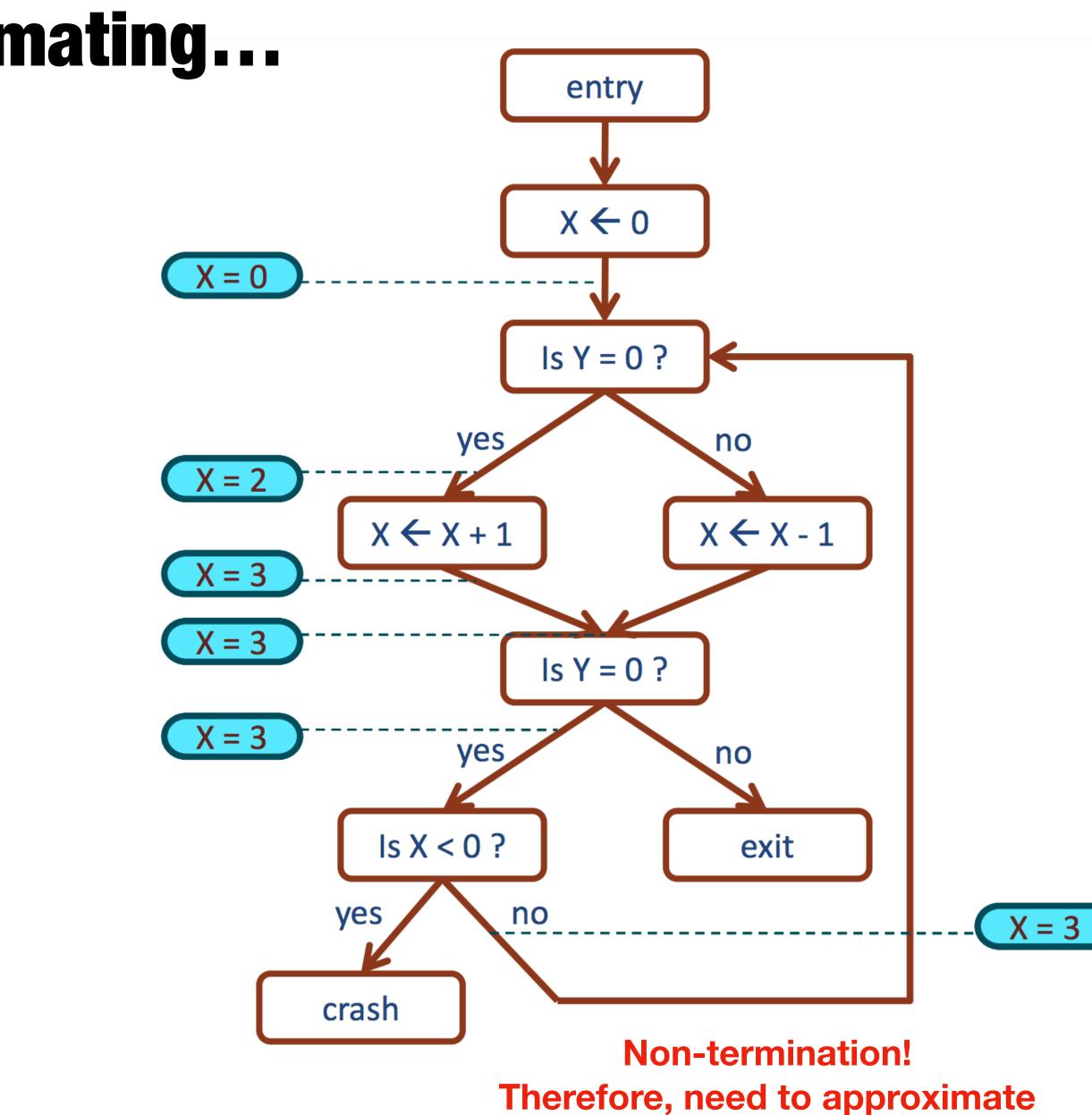
Is this program safe?



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

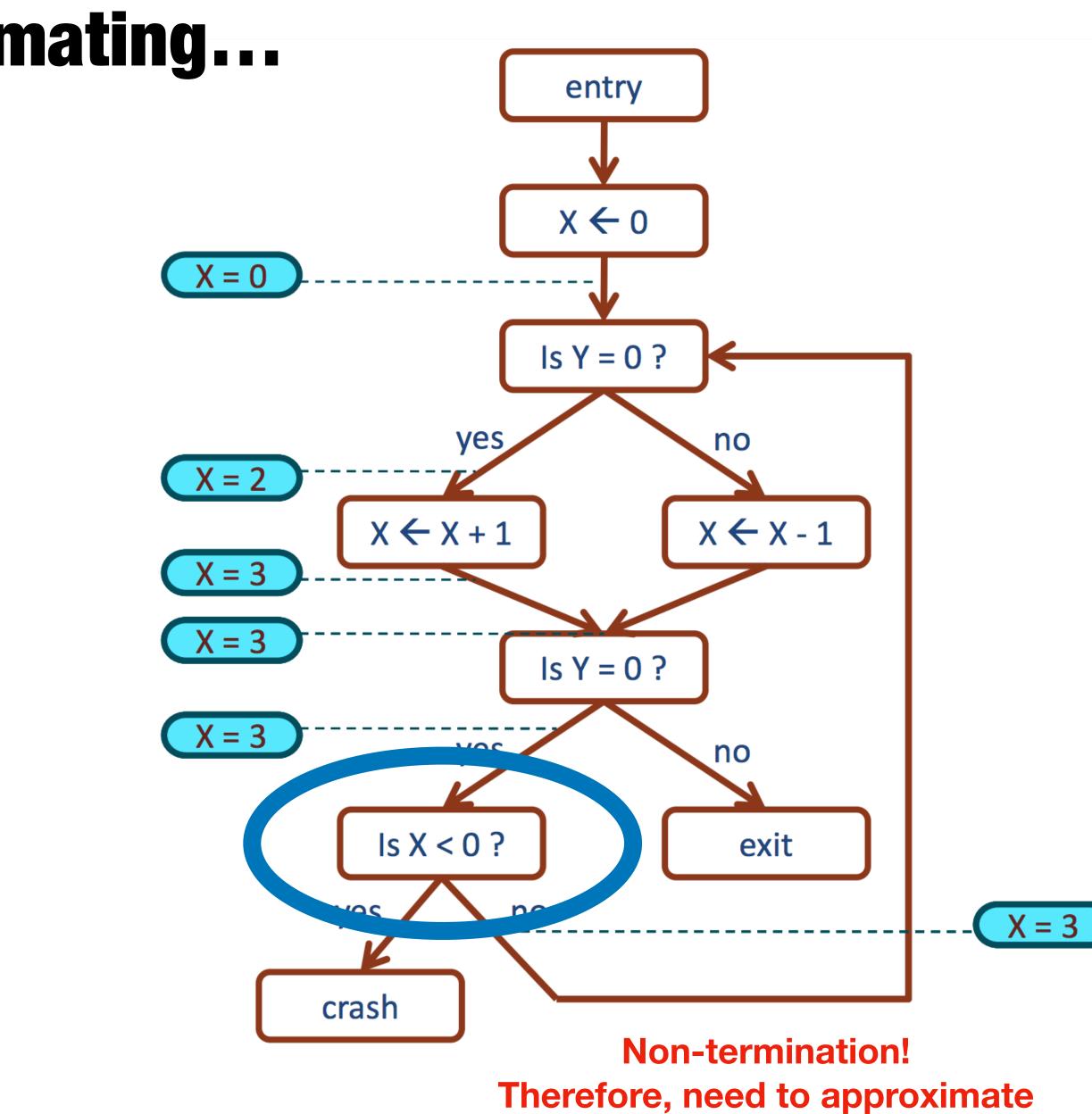


Try analyzing without approximating...



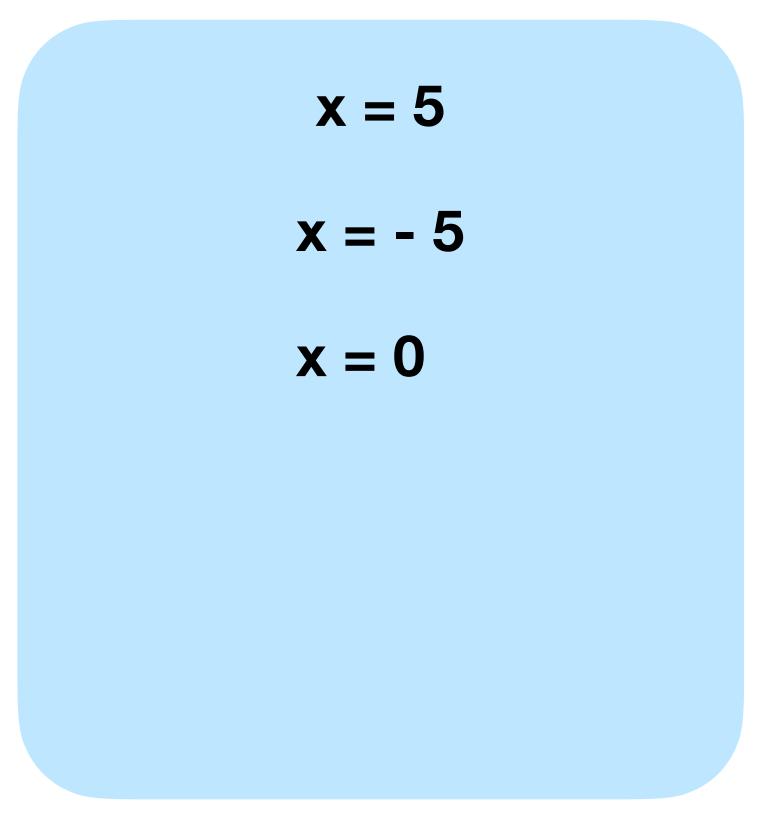


Try analyzing without approximating...





Concrete Domain of Integers



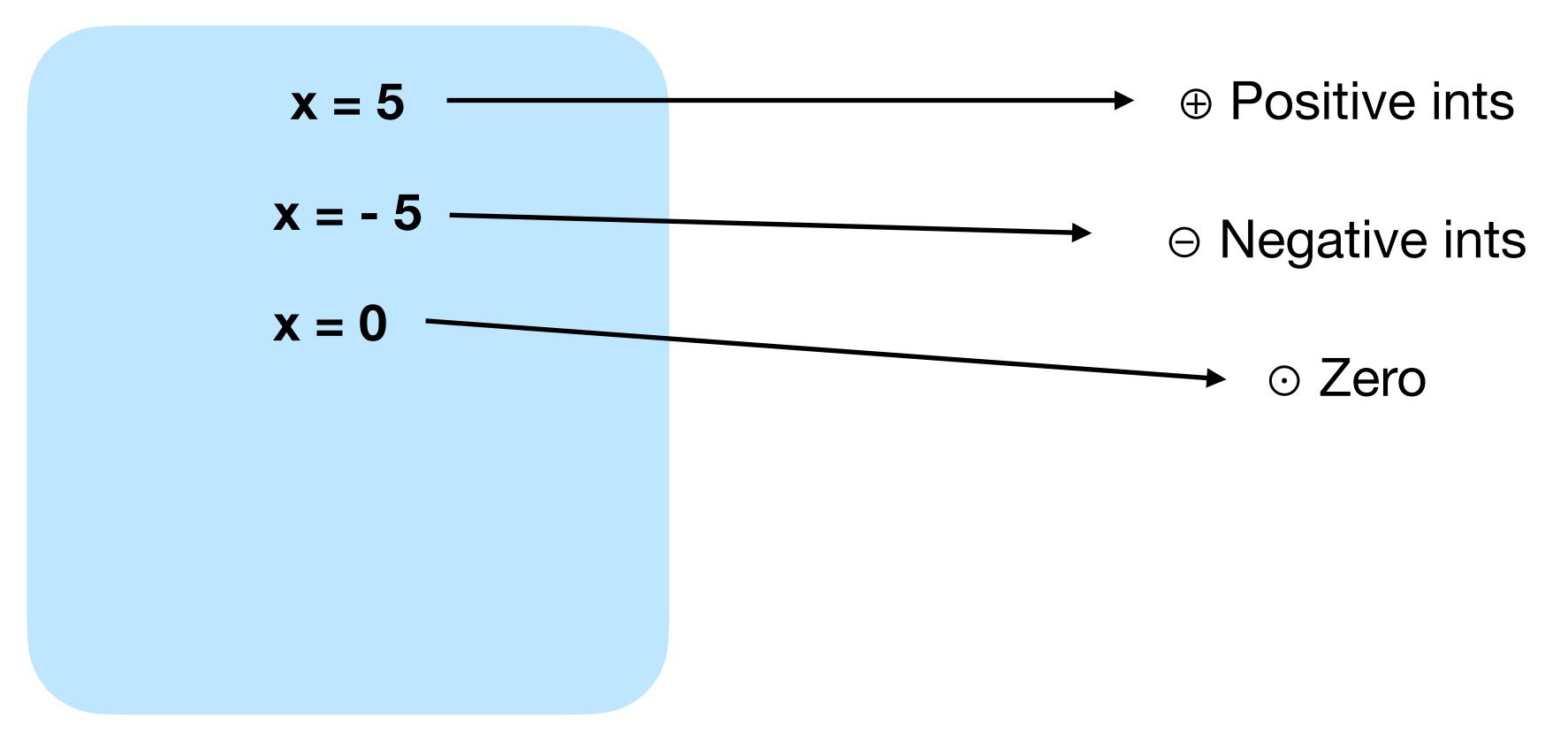
Abstract Domain of Signs

⊕ Positive ints

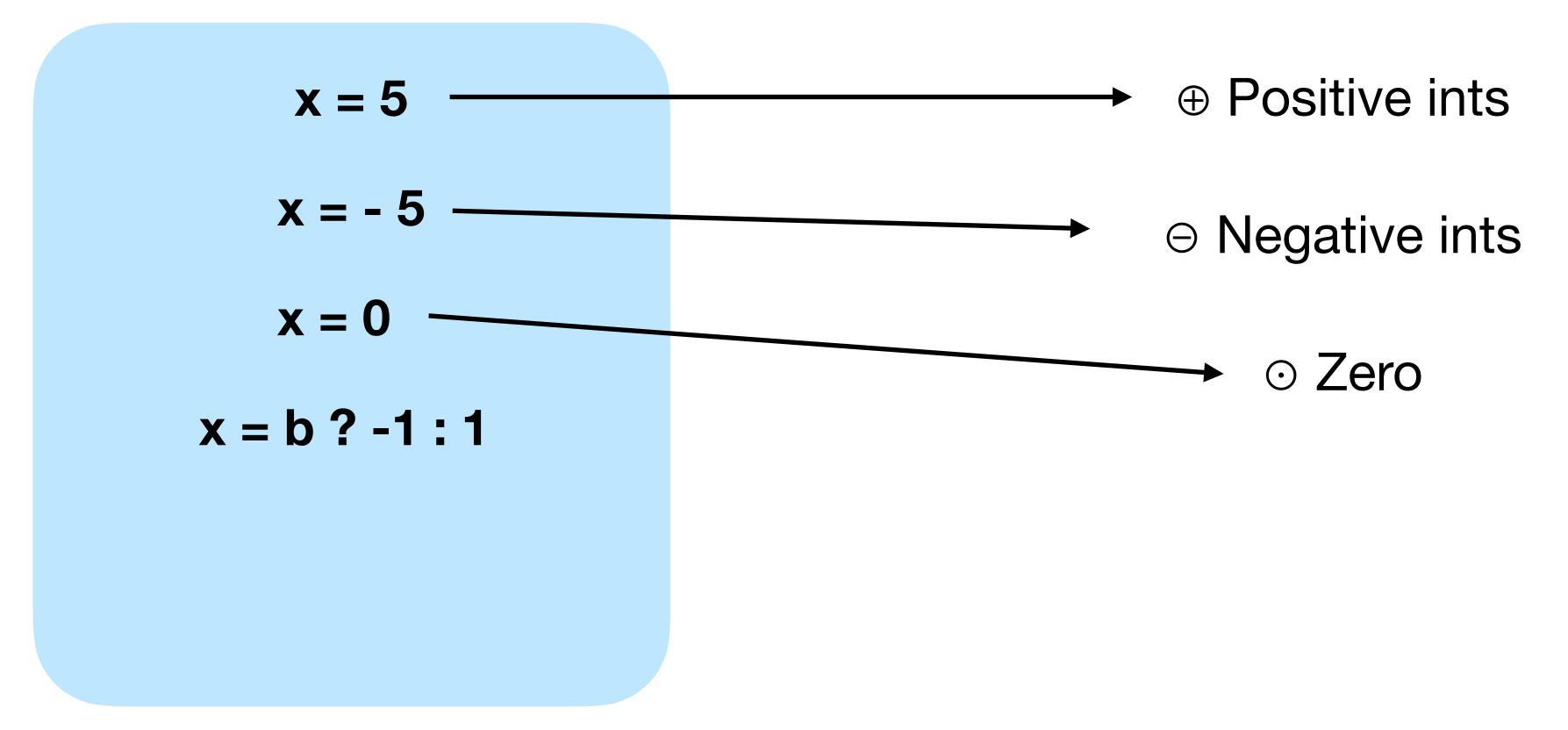
⊖ Negative ints

 \oplus Zero

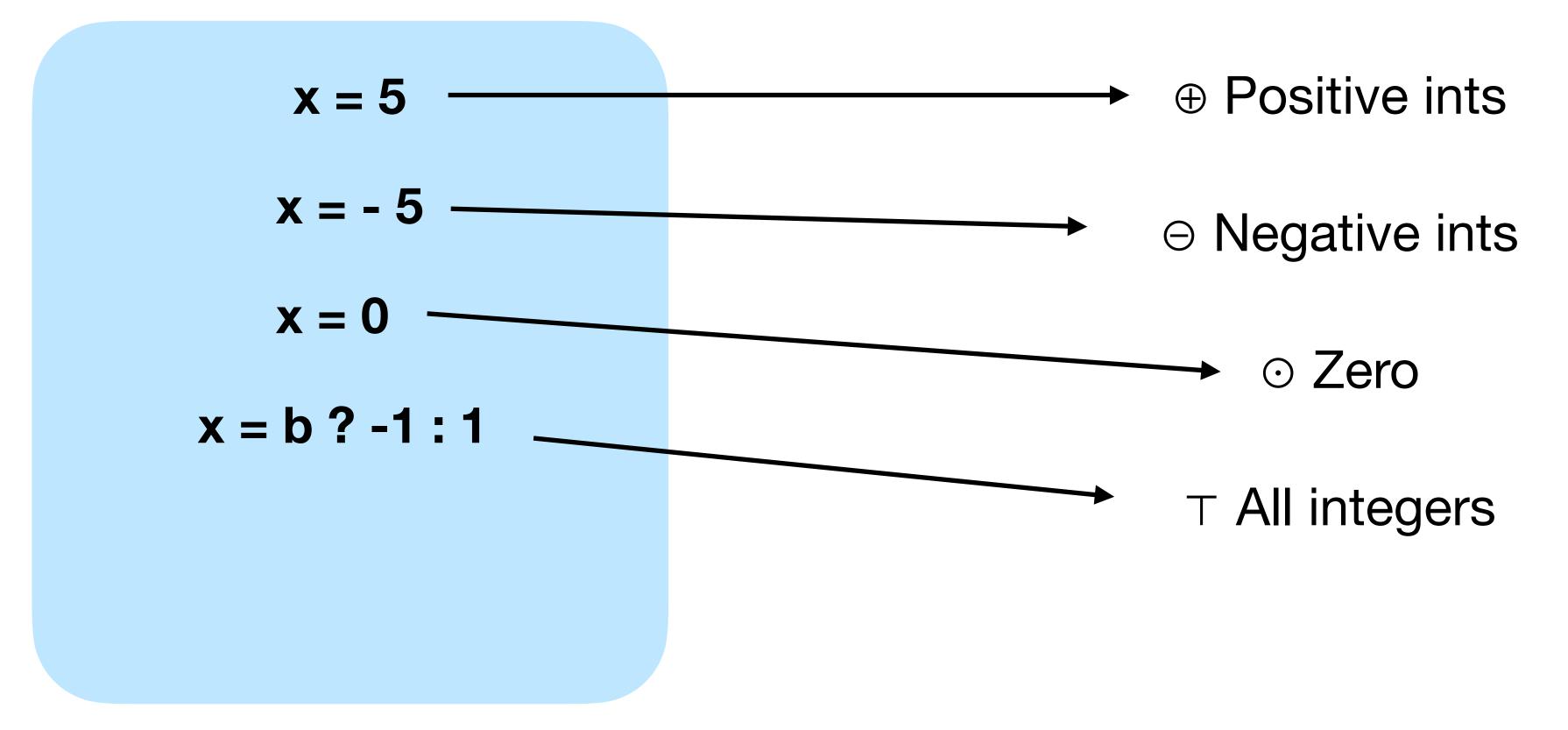
Concrete Domain of Integers



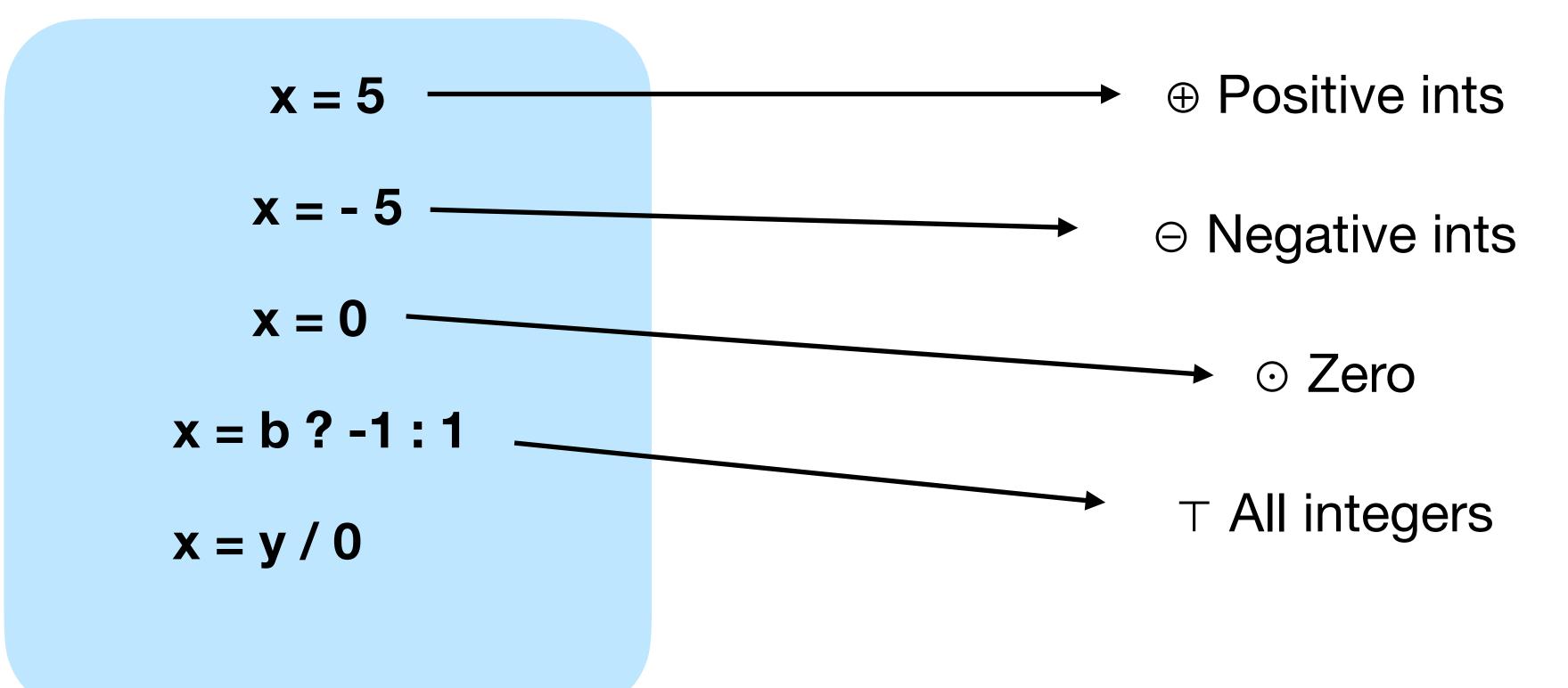
Concrete Domain of Integers



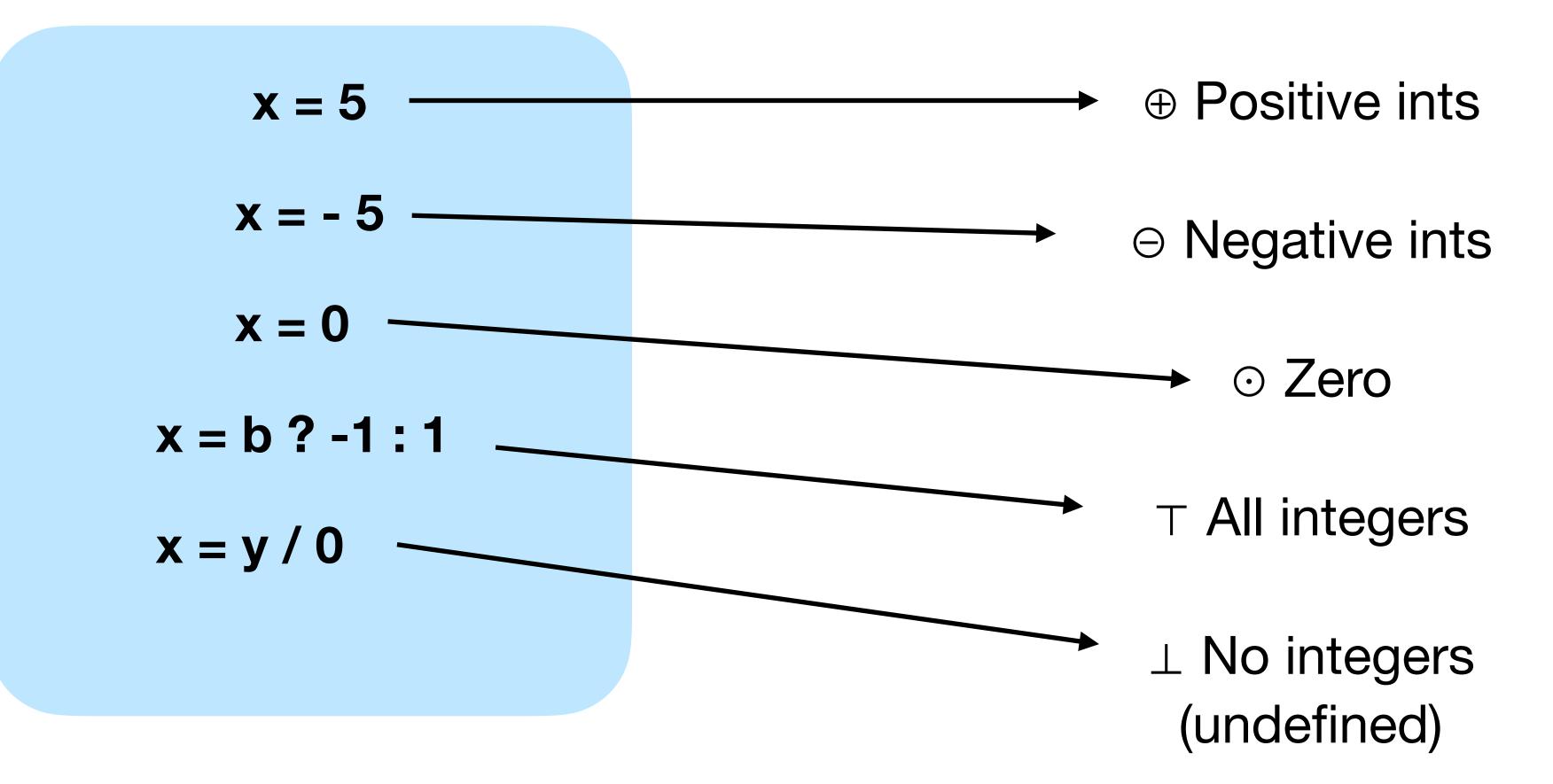
Concrete Domain of Integers



Concrete Domain of Integers

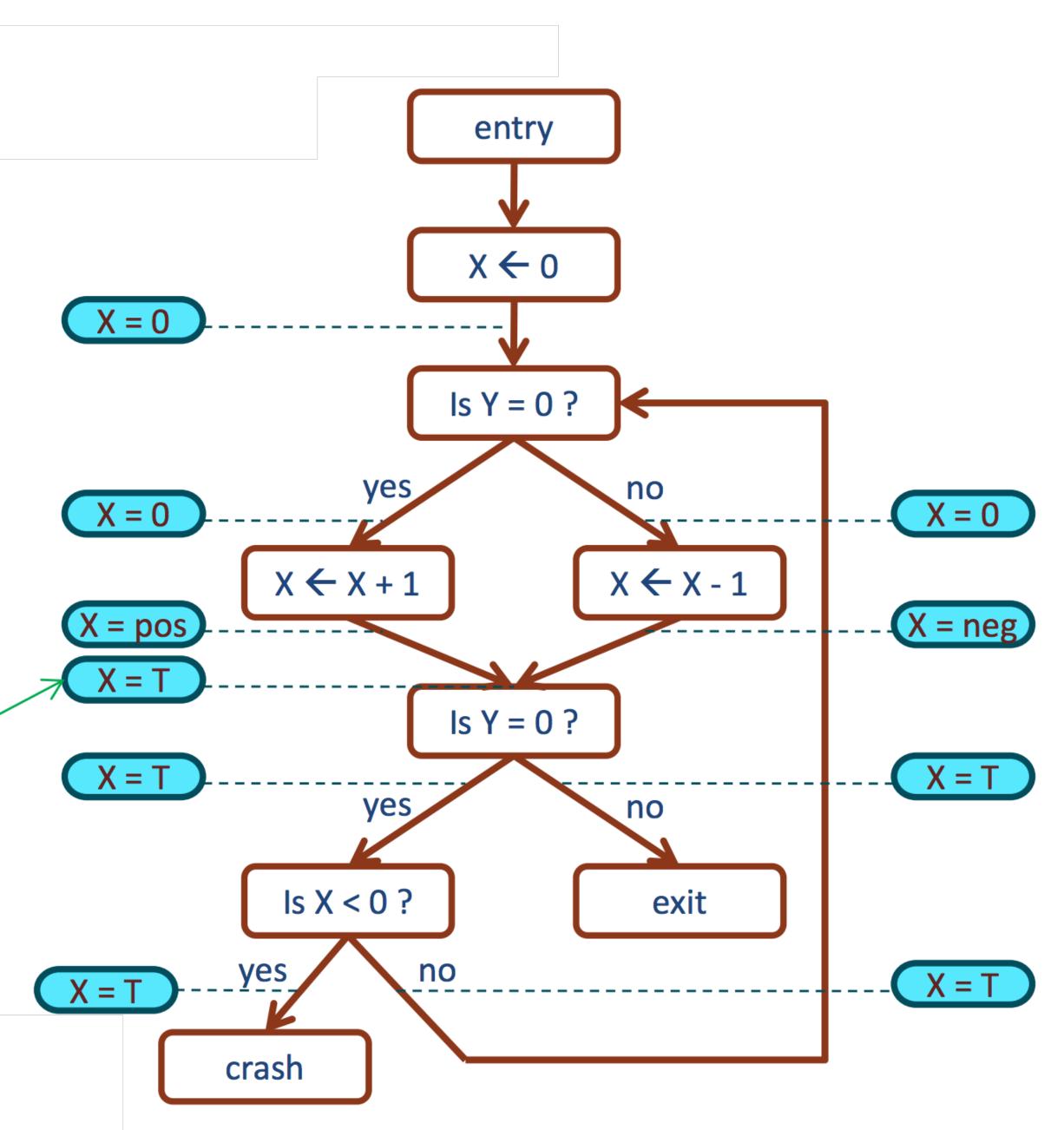


Concrete Domain of Integers



Try analyzing with "signs" approximation...

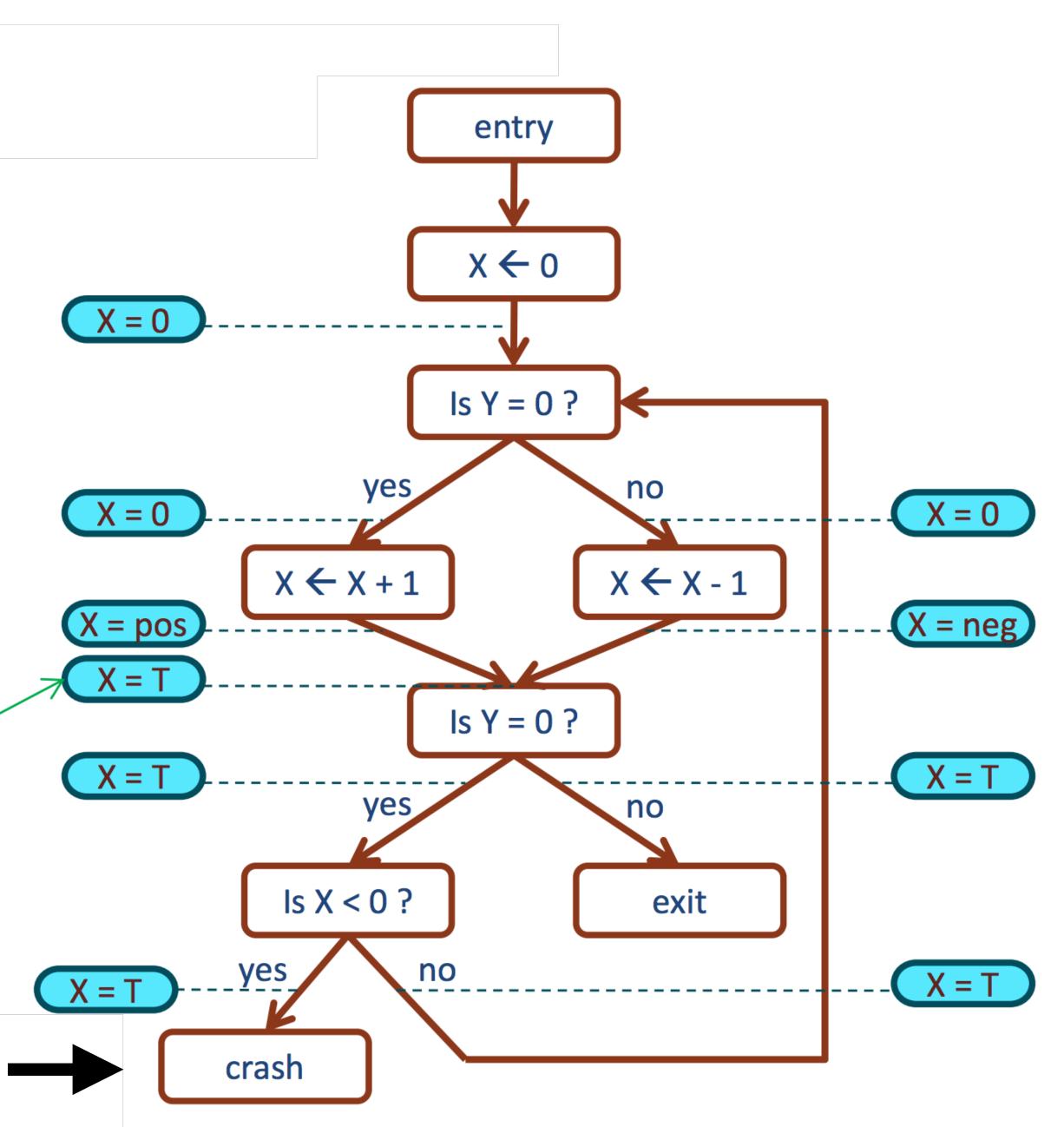
lost precision



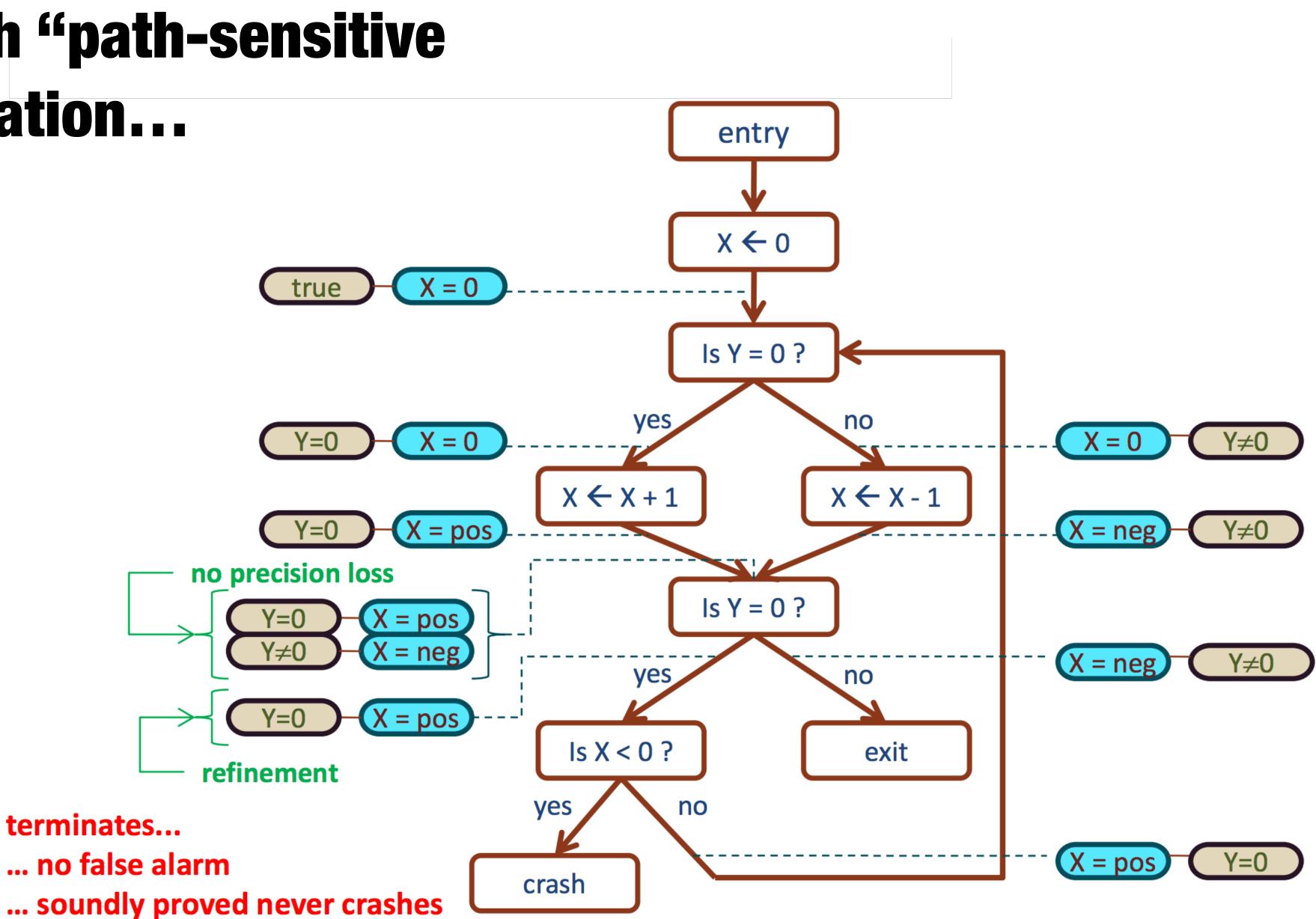
Try analyzing with "signs" approximation...

lost precision

Might Crash

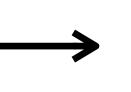


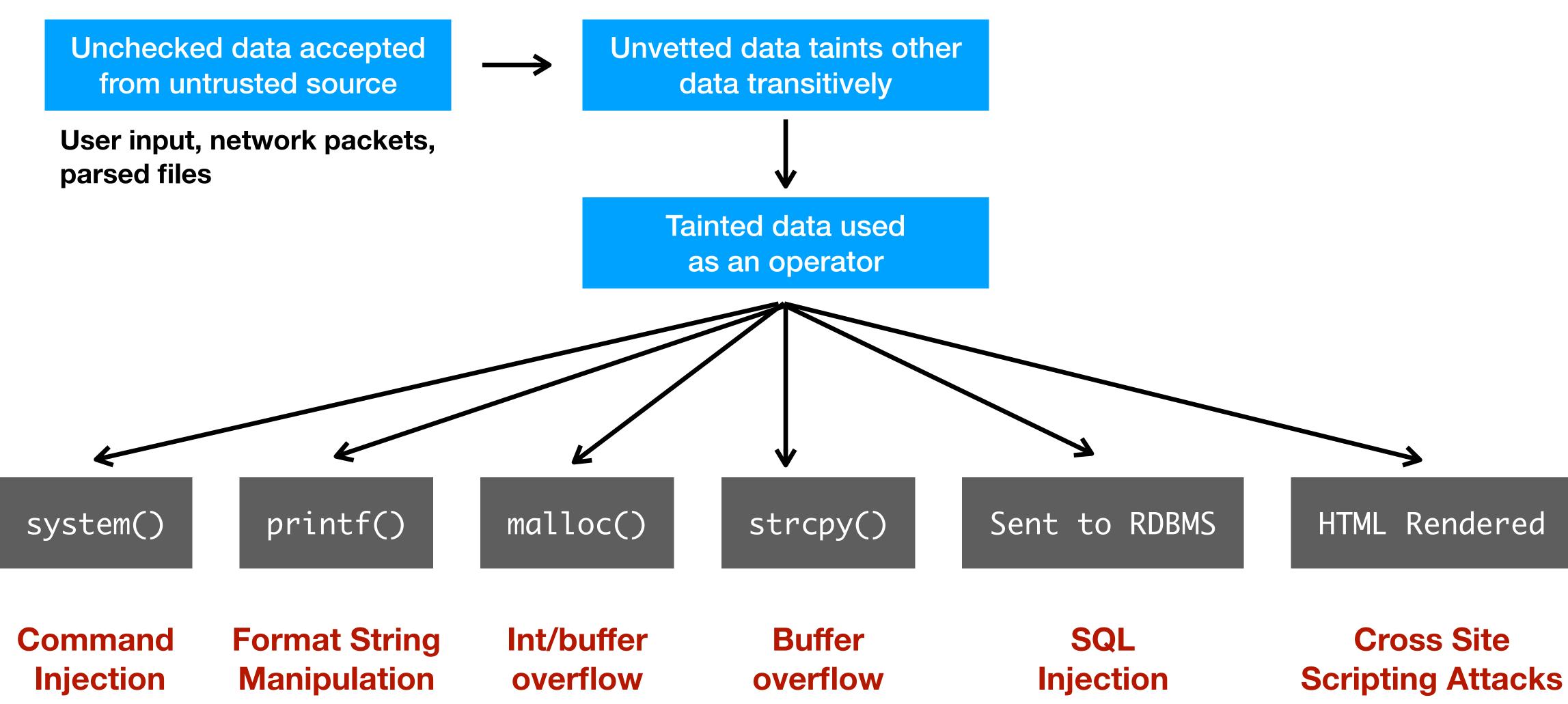
Try analyzing with "path-sensitive signs" approximation...



- ... no false alarm

from untrusted source





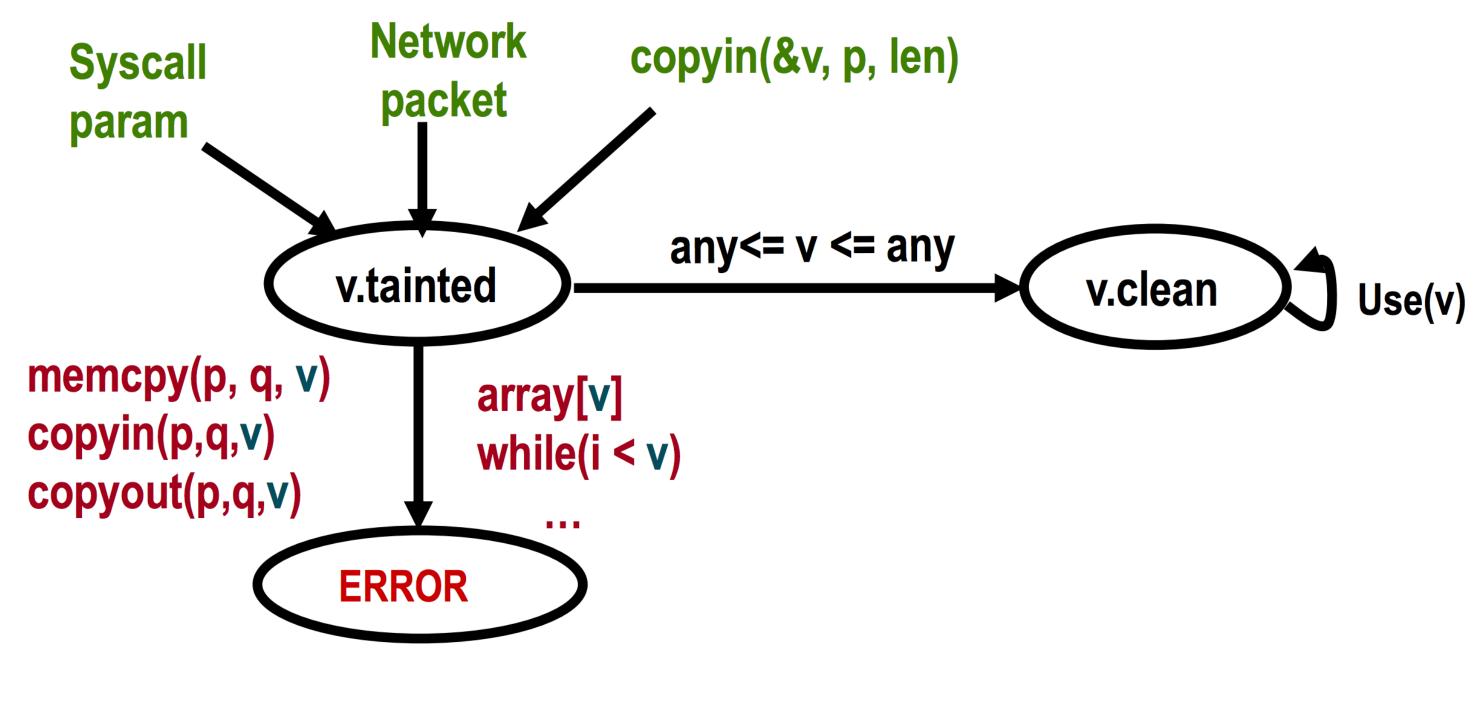
Tainting Checkers





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

• • •

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);

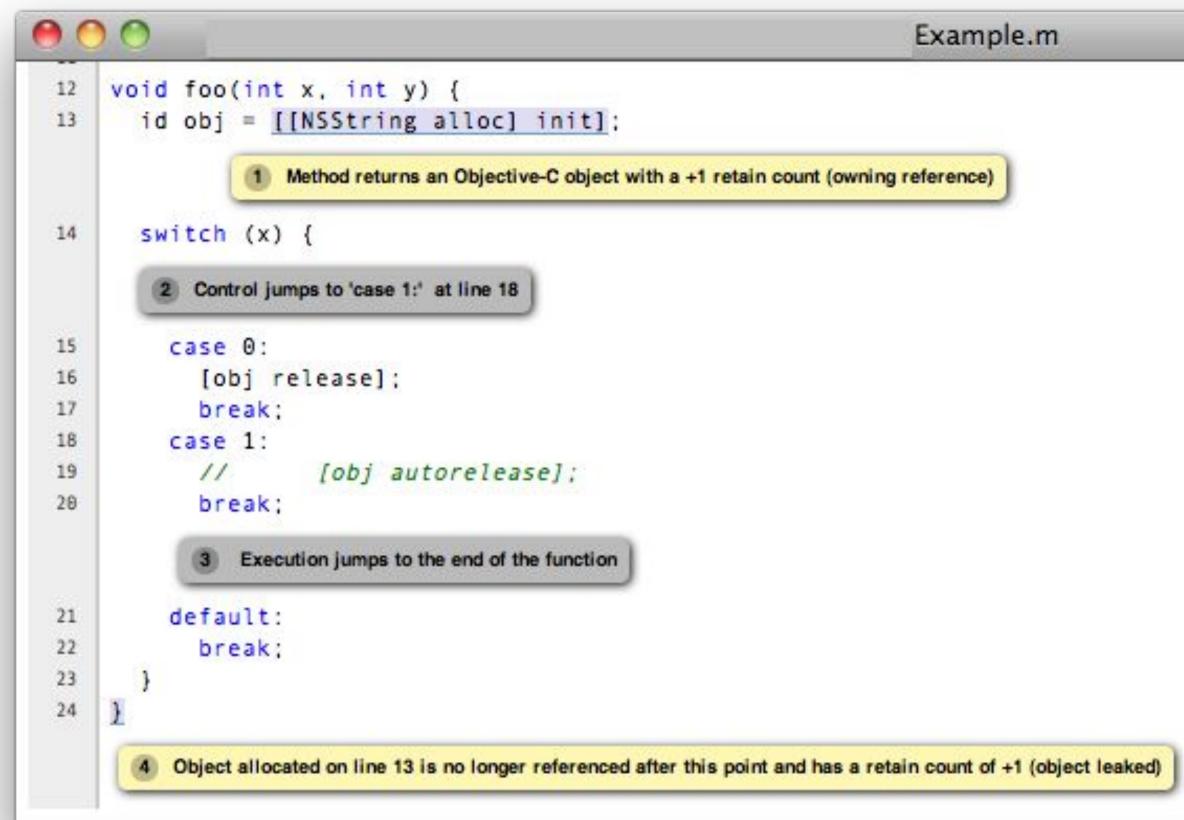
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */

Clang static analyzer

Check for common security issues with a static analysis framework in the compiler

Built in checkers:

Buffer overflows (with taint) Refcount errors malloc() integer overflows Insecure API use Uninitialized value use





CodeQL (Semmle)

```
class PotentialOverflow extends Expr {
 PotentialOverflow() {
   (this instanceof BinaryArithmeticOperation
                                                // match x+y x-y x*y
      and not this instanceof DivExpr
                                                // but not x/y
      and not this instanceof RemExpr)
                                                        or x%y
                                                11
   or (this instanceof UnaryArithmeticOperation // match x++ x-- ++x --x -x
         and not this instanceof UnaryPlusExpr) // but not +x
   // recursive definitions to capture potential overflow in
   // operands of the operations excluded above
   or this.(BinaryArithmeticOperation).getAnOperand() instanceof PotentialOverflow
   or this.(UnaryPlusExpr).getOperand() instanceof PotentialOverflow
from PotentialOverflow po, SafeInt si
where po.getParent().(Call).getTarget().(Constructor).getDeclaringType() = si
select
   po,
   po + " may overflow before being converted to " + si
```

Query language for finding patterns in large codebases

"SQL for searching code" Works best when you have a

specific bad code pattern in mind



Manual analysis

project-zero	project-zero
--------------	--------------

New issue

Open issues

 \mathbf{v}

· · · · · · · · · · · · · · · · · · ·		
☆ s	tarred by 4 users	Issue 2085: God
Owner:	natashenka@google.com	callee to leak vi
CC:	proje@google.com	Reported by natashenka@ Project Member
Status:	Fixed (Closed)	When Duo accepts an inco based on the remote offer,
Components:		in an executor from onSet
Modified:	Dec 2, 2020	is disabled on another, so sending traffic.
Finder-natashenka Deadline-90 Vendor-Google CCProjectZeroMem Severity-High Methodology-CodeF Product-Duo Reported-2020-Sep Fixed-2020-Oct-26	Review	Usually setting up the combecause it is run on the sattraffic occurs at the same to until the incoming data is provide the callee user. This could surroundings. To reproduce this issue: 1) run track.py on the attacking python3 track.py "Attacking 2) run exploit_sender.py or
		python3 exploit_sender.py
https://buga.abra	mium ora/n/project zero/iccues/detail2id=2085	

ogle Duo: Race condition can cause

ideo packets from unanswered call

G⊃ Code

1 of 9 Back to list

 $\mathbf{\nabla}$

google.com on Wed, Sep 2, 2020, 5:02 PM PDT

coming call, it starts the WebRTC connection by calling setLocalDescription on the answer it generates r, and then disables outgoing video traffic by disabling all encoders by calling RtpSender.setParameters etSuccess. This creates a race condition, as the connection gets set up by one thread, but outgoing traffic there is no guarantee that outgoing traffic will be disabled before the connection is set up and starts

nnection takes a long time, and disabling traffic is very fast, but it is possible to slow down disabling traffic, same thread queue that processes incoming messages from data channels, so if a lot of data channel time a new SDP offer is received, the method to disable video transmission needs to wait in the queue processed.

is a caller on Duo to receive a small amount of video from the callee even if the call is not answered by d allow an attacker to enable the camera on a remote user's device and take pictures of their

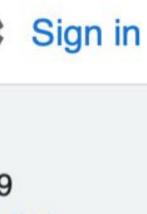
acker device

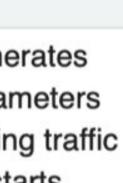
ng Pixel"

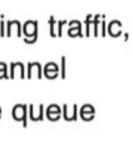
on the same attacker device in another window, with exploit_sender.js in the same directory

y "Attacking Pixel"

3) make a video call to the target device and hang up after one second (this populates some difficult-to-generate memory in the











Reverse engineering

Usually assisted by tools Disassembler Decompiler Strings Often aided by dynamic analysis Tracing

Looking at a compiled program in order to figure out what it does and how it works



IDA Pro

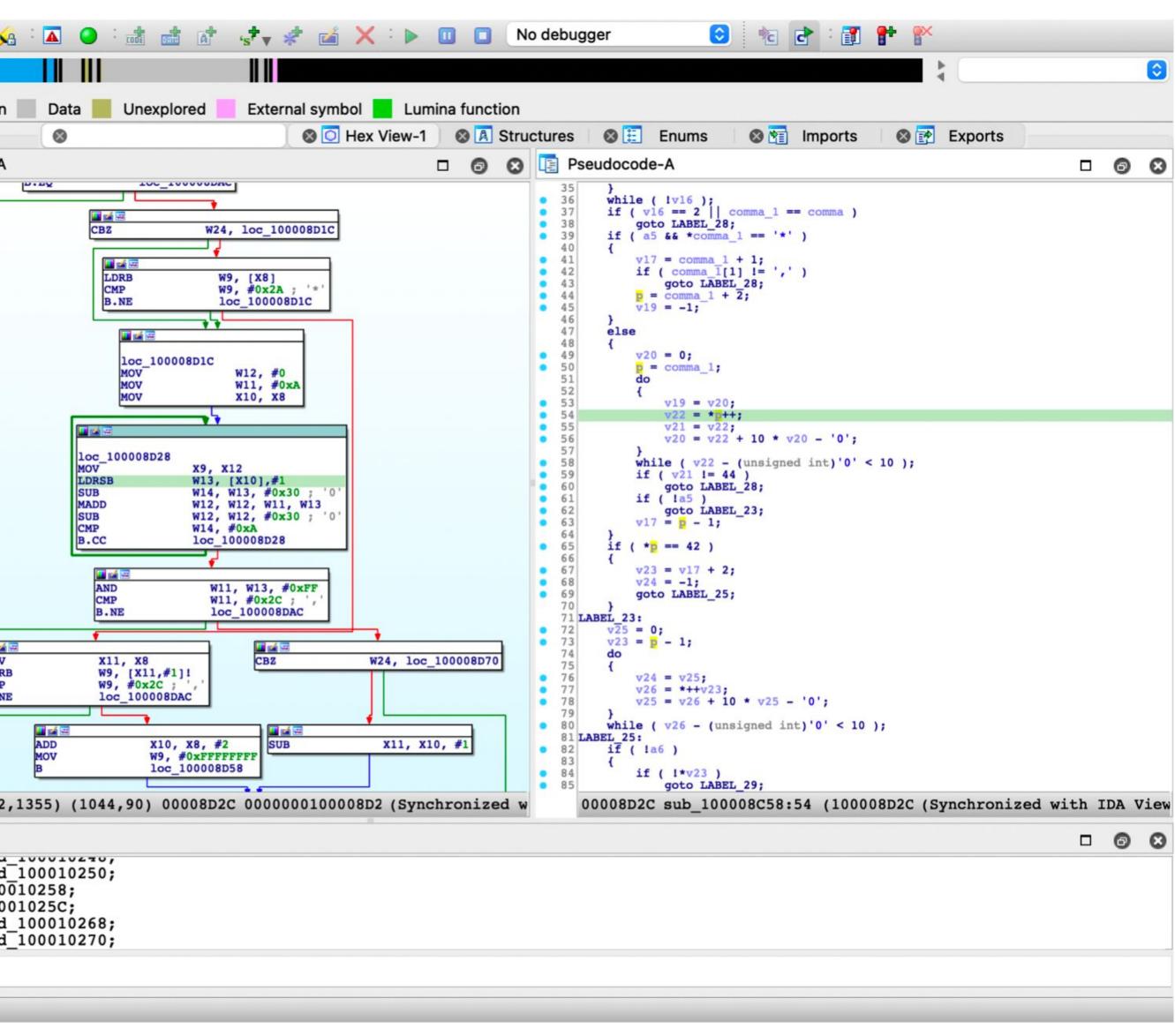
Disassembly

Decompilation

Binary analysis

Scripting

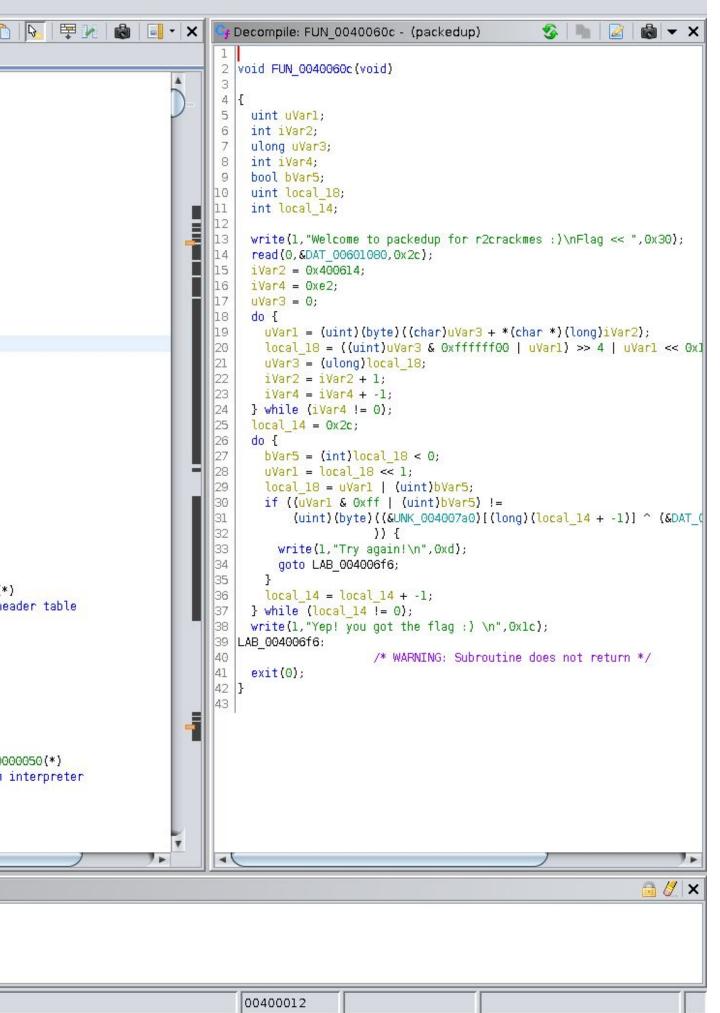
: 🚰 🔚 : 🗢 🗸 🔿 🗸	, : 0 <u>n</u> 0		6	4	} 5
1				1	Ţ.
Library function	Regular	r funct	ion	Ins	truction
f Functions window		0	8		
Function name				IDA	View-A
f sub_100007E7C					
f sub_100007F3C					
f sub_100007FA4					
f sub_1000081DC					
f sub_100008414					
f_ sub_100008468					
f sub_1000085B8					
f sub_100008884					
f sub_100008C24					
f sub_100008C58					
f sub_100008E00					
f sub_100009CD4					
f sub_100009E0C					
f sub_100009E64					
f sub_100009E80					
f sub_100009FBC					
f sub_100009FE4					
f sub_10000A05C					
f sub_10000A06C					
Line 50 of 194					
🖪 Graph overview		0	0		LDR CMP
				111-	B.N
E E			1	00.00)% (322
Output window					
100010240. using y		CYPE			4 WOLG
100010250: using g 100010258: using g	uessed	type	int	dwo	qword rd 100
10001025C: using g	uessed	type	int	dwo	rd_100
100010268: using g 100010270: using g					qword qword
Python					
AU: idle Down [Disk: 339	GB			



0010258; 001025C; d_100010268; d 100010270;

Ghidra

• preskedup // segment_2.1 // Loadable seguent [0x400000 - 0x4007/b] (disabled execute // rais // segment_2.1 // Loadable seguent [0x400000 - 0x4007/b] (disabled execute // rais // segment_2.1 // Loadable seguent [0x400000 - 0x4007/b] (disabled execute // rais // segment_2.1 // Loadable seguent [0x400000 - 0x4007/b] (disabled execute // rais 0.000000 rd rd rd s de 0.00000 rd rd rd s de 0.00000 rd rd rd s de 0.00000 rd rd s de de 0.0000 rd rd s de de <th></th> <th></th> <th>and the second second</th> <th></th> <th></th> <th></th> <th></th> <th></th>			and the second					
Busic // segment [2.1.// Loadbable segment [0.400000 - 0x400070] (disabled execute // iman: 0.4000700 - 0x40070) (disabled execute // iman: 0.4000700 - 0x40070) (disabled execute // iman: 0.400070 - 0x40070) (disabled execute // iman: 0.400070) (disabled execute // iman: 0.4000700 (disabled execute // iman: 0.4000700 (disabled execute								D 🗋
<pre> ddsa jdt f jdt i j</pre>		*pack	kedup 🗙					
<pre>pdt.pdt dpt.p</pre>		\rightarrow			11			
<pre> fni text text</pre>					// segment	_2.1		
<pre> fni text text</pre>	🔄 .got.plt				// Loadabl	e segment [0x400000	- 0x40097b] (disabled execu-	te
<pre> fni text text</pre>	🖹 .got							
<pre> fni text text</pre>	🔄 .dynamic							
<pre> fni text text</pre>	🗟 .icr				0 (Default)			
<pre> fmi text text</pre>	🔄 .fini arrav				Elf64	Ehdr		
<pre> fmi text text</pre>	init array							
<pre> fmi text text</pre>	ab frome							
<pre> fni text text</pre>								
<pre> fni text text</pre>	en_trame_nor			00400001 <mark>45 4</mark> c 4				
Dit 0040000700 00 00 00 00 00 00 00 00 00 00 00 00 00	🔄 .rodata			00400004 02	db	2h	e_ident_class	
Dit CO400007 00 00 00 00 00 00 00 00 00 00 00 0	🗟 .fini			00400005 01	db	lh	e_ident_data	
Dit CO400007 00 00 00 00 00 00 00 00 00 00 00 0	🗟 .text			00400006 01	db	lh	e_ident_vers	
Program Tree 00 00 00 00 00 00 00 00 00 00 00 00 00	🗟 .plt	T I	_					
Program Tree × 00400010 02 00 00 dv 2h e_type Symphol Tree 00400010 02 00 00 00 dv 2h e_machine 00400010 02 00 00 00 00 dd 1h e_wersion 00400010 02 00 00 00 dd e_ft/r e_machine 00400010 02 00 00 00 00 dd e_ft/r e_machine 00400010 02 00 00 00 dd e_ft/r e_ft/r 00400020 00 00 00 00 dd e_ft/r e_ft/r 00400020 00 00 00 dd ddw dh e_ft/r 00400020 00 00 00 dd dw dw dh e_ft/r 00400020 00 00 00 dd dw dw dh e_ft/r 00400020 00 00 dd dw dw dw e_ft/r e_ft/r 004000020 10 00 dd dw dw						S. 1		
Dota Type Manager X Image: Second condition of the second cond condition of the second condition of the second conditi	Program Tree ×					2h	e tvpe	
Sympositive 00400014 to 00 00 00 dd 00 00 00 dd entry e_entry • f extry • 60400012 d0 00 00 00 dd entry e_entry e_entry • f extry • 60400012 d0 00 00 00 dd entry e_entry e_entry • f extry • 60400012 d0 00 00 00 dd ettra_start e_entry e_entry • f extry • 60400012 d0 00 00 00 dd e_entry e_entry e_entry • f extry • 60400000000000 e_ghord e_ftags e_entry • f extry • 604000000000 00 00 00 dd e_ftags e_entry • f extry • 60400000000 00 00 00 dd e_ftags e_entry • f extry • 60400000000 00 00 00 dd 00 00 00 dd 00 00 00 dd • f extry • 604000000000 00 00 00 dd 00 00 00 dd 00 00 00 dd • f extry • 604000000000000000000000 00 00 00 dd 00 00 00 dd 00 00 00 dd • f extra • 60400000000000000000000000000000000000		→						
if entry e_entry e_entry if entry e_entry e_entry if exit 00400028 00 00 00 00 00 00 00 00 00 00 00 00 00	Symbol Tree	X						
<pre>0 00 00 00 0 00 00 00 00 00 00 00 00 00 00 00 00 0</pre>								
<pre></pre>		A					,	
<pre></pre>			_			Elf64 Phdr ABE	RAY 00400e nhoff =	
<pre>> FUN_00400770 > FUN_00400500 > FUN_00400500 > FUN_00400500 > FUN_00400700 > FUN_004000700 > FUN_00400700 > FUN_0040070 > FUN_0040070 > FUN_0040070 > FUN_0040070 > FU</pre>						Ector_indi_Art	un_uunun o_phorn	
<pre>></pre>						Elf64 Shdr &RE	RAY elfs e shoff	
<pre> Full 00400030 00 00 00 00 00 dw 0h e_flags full 0040030 00 00 00 00 dw 40h e_htsize full 0040033 80 00 dw 38h e_phentsize full 0040033 40 00 dw 40h e_htsize full 0040033 40 00 dw 40h e_htsize full 0040033 10 00 dw 10h e_htsize full 00400033 10 00 dw 10h e_htsize full 0040033 10 00 dw 10h e_h</pre>						Etto4_Shar_An		
<pre> FUN_00400700 F FUN_0040070 F FUN_00400770 F read Guddoosa 40 600 dv</pre>			0			Ob	e flans	
<pre> FON_00400700 FON_00400051 FON_00400700 FON_00400700 FON_00400040 FON_00400700 FON_00400700 FON_00400700 FON_00400700 FON_00400700 FON_00400040 FON_0040040 FON_004004 FON_0040040 FON_0040040 FON_0040 FON_004004 FON_004</pre>								
<pre>PL0_00400700 + Console - Scripting PL0_00400700 + Console - Scripting PL0_00400700 + Console - Scripting PL0_00400036 40 00 dv 9h = phnum plnum p</pre>	FUN_00400700							
<pre> read</pre>	FUN 00400770							
<pre>> * * * * * * * * * * * * * * * * * *</pre>								
<pre> write</pre>								
* * * write * * * * * * * * * * * * * * * * * * *	* write							
Labels Elf64_Phdr_ARRAY_00400040 XREF[2]: 00400020(*), 00400050(*) Classes 004000400 66 00 00 Elf64_Ph PT_PHOR - Program head Image:	A write			0040003e 16 00	aw	IBN	e_snstrnax	
Classes PT_PHDR - Program head () Namespaces 00 000 40 06 00 00 Elife4_Ph 00 00 00 40 () Namespaces () Namespaces () Data Type Manager () Namespaces () Star Type Manager () Namespaces () Star Type S () Namespaces () BuiltinTypes () Namespaces () BuiltinTypes () Namespace () Console - Scripting () Namespace () Console - Scripting () Namespace <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td>were fel</td> <td></td>							were fel	
Console - Scripting							XREF[2];	
⁽¹⁾ (1) ⁽¹⁾			(±		ELT64_	Ph		PI_PHDR - Program head
Filter:	[] Namespaces	V						
<pre>// .interp // .interp // SH_PROGBITS [0x400238 - 0x400253] // ram: 00400238-00400238 // ram: 00400238 - 0x400253 // ram: 00400238 - 0x400253 // s_/lib64/ld-linux-x86-64.so.2_00400238 XREF[2]: 00400088(*), elfSectionHeaders::00000 00400238 2f 6c 69 ds "/lib64/ld-linux-x86-64.so.2" Initial Elf program in 62 36 34 2f 6c 64 // // .note.ABI-tag // .note.ABI-tag</pre>				00 00 40 .				
Data Type Manager <td>Filter:</td> <td>2</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>	Filter:	2						
<pre>// ram: 00400238 00400253 // Data Types BuiltinTypes BuiltinTypes</pre>								
<pre>// ram: 00400238-00400253 // * Data Types * BuiltinTypes * BuiltinTypes * Generic_clib_64 * Windows_vs12_32 // Console - Scripting // ram: 00400238 2f 6c 69 ds */lib64/ld-linux-x86-64.so.2*</pre>	Data Type Manager	• X					(400253]	
<pre>\$ Data Types BuiltInTypes BuiltInTypes</pre>					// ram: 00	400238-00400253		
<pre>s_/fib64/fib-fifthdx-k86-64.so.2"</pre>	(= - ⇒ - 🔽 - 📉 🞯 🖃				11			
<pre>s_/fib64/fib-fifthdx-k86-64.so.2"</pre>	T B Data Tara							
<pre>BuiltInTypes BuiltInTypes Generic_clib_64 BuiltInTypes BuiltInTypes Generic_clib_64 BuiltInTypes Generic_clib_64 BuiltInTypes BuiltInTypes Generic_clib_64 BuiltInTypes Generic_clib_64 BuiltInTypes BuiltInTypes Generic_clib_64 BuiltInTypes BuiltInTypesBuil</pre>					s /lib64/l	d-linux-x86-64.so.2 0	0400238 XREF[2]:	00400088(*),
<pre>> @ opackedup > @ generic_clib_64 > @ windows_vs12_32</pre> Outdool238 2f 6c 69 ds "/lib64/ld-linux-x86-64.so.2" Initial Elf program in 62 36 34 2f 6c 64 // // .note.ABI-tag // Console - Scripting	🕨 🥩 BuiltInTypes				.78			
<pre> generic_clib_64 windows_vs12_32</pre>				00400238 2f 6c 69	ds	"/lib64/ld-linu	x-x86-64.so.2"	
<pre>> generic_clib_s4 > i windows_vs12_32</pre>						,,		The set brogram to
<pre>windows_vs12_32 // // .note.ABI-tag // // clrr_NOTE{Outdoo272} </pre>	generic_clib_64							
// .note.ABI-tag	🕨 🧊 windows vs12 32			21 00 04 .	11			
Console - Scripting					// note A	BT-tan		
Console - Scripting							1-1	
Filter:		Con:	sole - Scripting	3				
Filter:								
Filter:								
Filter:								
Filter:								
		THE R A						
	Filter:							



Similar to IDA Open source Written by the NSA (no, really)



Tips for writing (more) secure software

Software tests

- One of the most effective ways to reduce bugs
- Unit tests: Check that each piece of code behaves as expected in isolation
 - Goal: Unit tests should cover all code, including error handling
 - So many exploitable bugs would be eliminated with basic unit tests
- Regression tests: Check that old bugs haven't been reintroduced
 - If you don't run regression tests, attackers will run them for you!
- Integration tests: Check that modules work together as expected

General tips

- Use a modern, memory safe language where possible: Go, Rust, etc.
- Understand and document your threat model early in the design process
- Treat all input from outside your process adversarially, even if you trust the sender
- Use a clean, consistent style throughout the codebase

bazad@cs.stanford.edu

Thank you!