Million Dollar Baby

An 'angr'y attempt at the Cyber Grand Challenge

Nick Stephens stephens@cs.ucsb.edu

Insomni'hack / March 17th, 2016

Shellphish

- Who are we?
  - A team of security enthusiasts
    - do research in System Security
    - play Capture the Flag competitions
    - released a couple of tools

Mini-primer: What's a CTF?

- Security competition:
  - exploit a vulnerable service / website / cryptosystem
  - reverse a binary
  - …
- Different formats
  - Jeopardy – Attack-Defense
  - Online – Live
  - …
- Basic idea: find the secret, submit to organizers, … *profit*

Shellphish

Roadmap

- **DARPA Cyber Grand Challenge (CGC)**
  The (almost-)Million Dollar Baby

- Our Cyber Reasoning System (CRS)
  Fancy term for auto-playing a CTF

- Automated Vulnerability Discovery
  Driller

- Automated Vulnerability Exploitation
  How it works

- Auto-exploitation demo using angr
  Open-source binary analysis framework

Cyber Grand Challenge (CGC)

- 2004: DARPA Grand Challenge
  - Autonomous vehicles

Cyber Grand Challenge (CGC)

- 20**14**: DARPA **Cyber** Grand Challenge
  - Autonomous **hacking!**

Cyber Grand Challenge (CGC)

- Qualification event: ONLINE
  - ~70 teams → 7 qualified teams
  - $750k per team

- Final event: DEFCON 24
  - Winning CRS will also play against humans!

cybergrandchallenge.com / cgc.darpa.mil


SHELLFISH

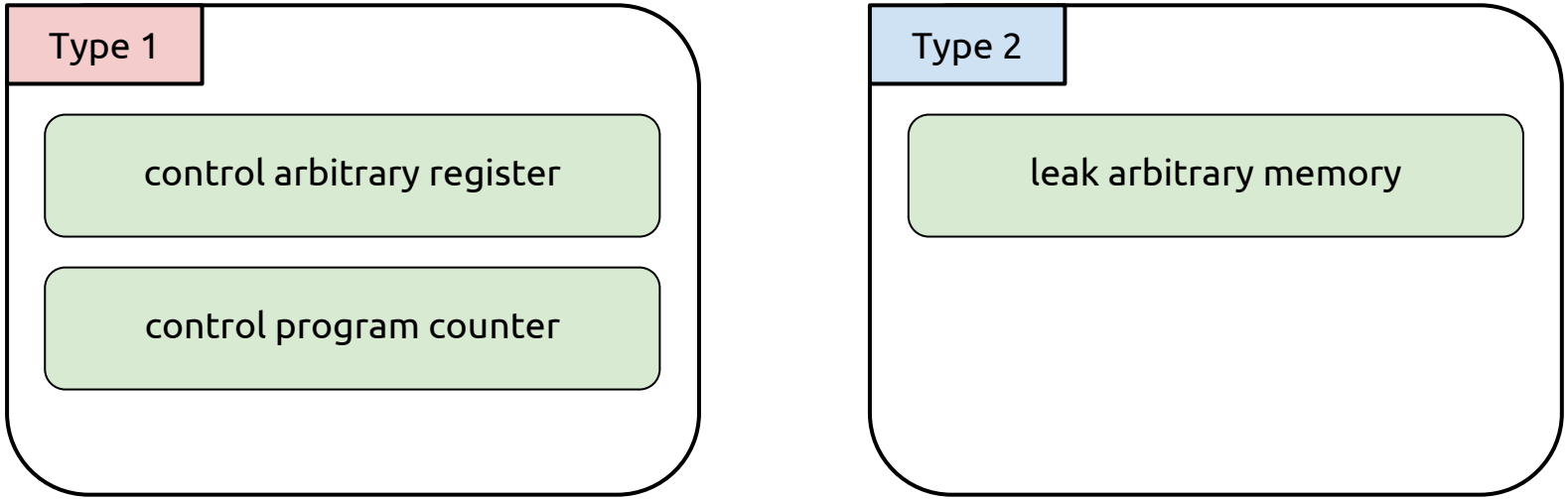
CGC Rules

- Attack-Defense CTF
- ~~Solving security challenges~~ → Developing a system that solves security challenges
- Develop a system that **automatically**
  - Exploits vulnerabilities
  - Removes vulnerabilities
- No human intervention

CGC – Rules

- Exploits
  - “getting a flag” (how? where?)
  - For the quals: exploit = **crash**
  - For the finals: exploit = **demonstrate PC control**
- Defend
  - int main() { return 0; }
    - Functionality checks
  - SIGSEGV => exit(0)
    - No easy “out-of-band” error handling
  - QEMU-style interpreter: interrupts => exit(0)
    - Performance cost (CPU, memory, file size)

CGC Final Round Exploits



CGC Qualification Event – Rules

- Basic idea:
  - Real(istic) programs
  - No “extra” complications

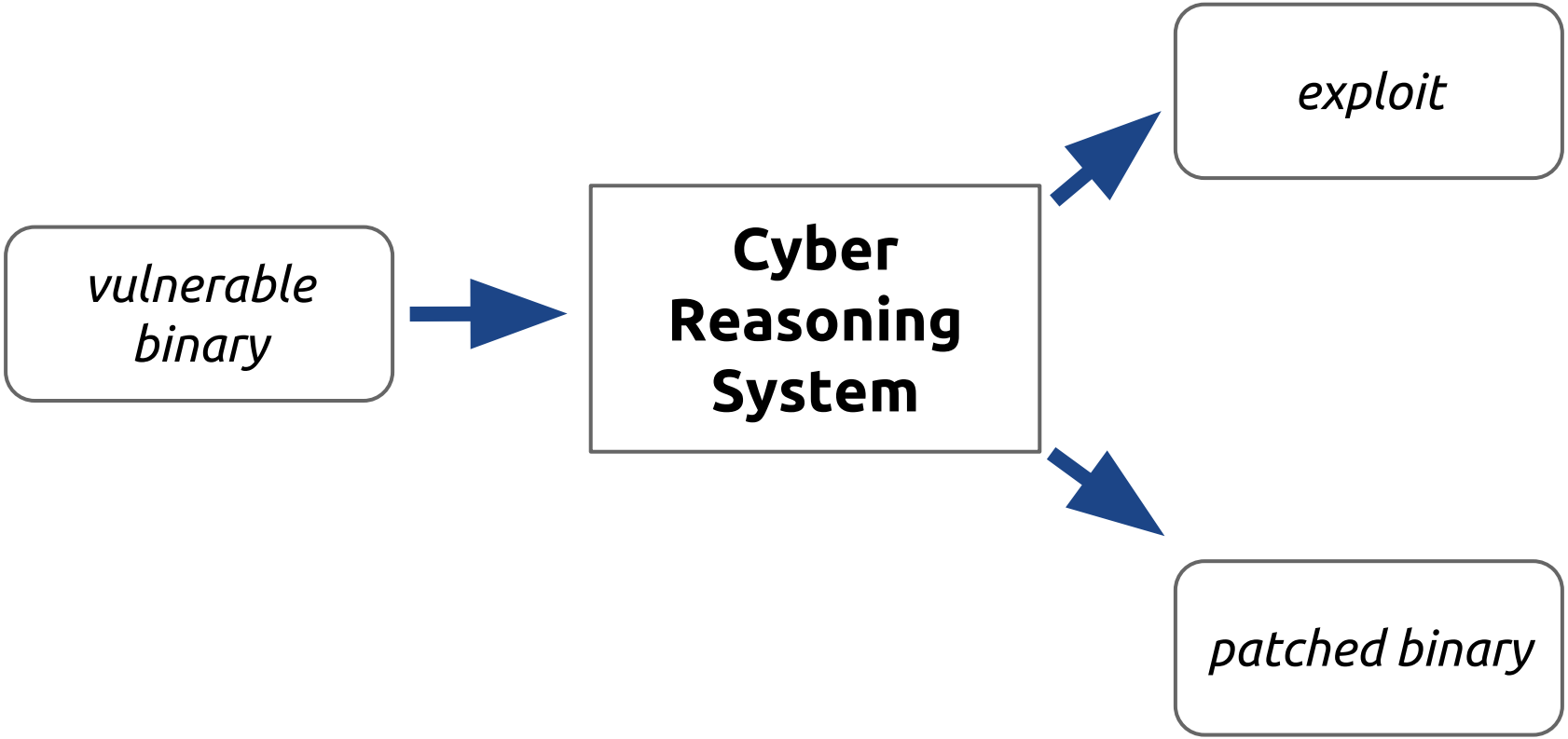
CGC Qualification Event – Rules

- Architecture: Intel x86, 32-bit
- OS: DECREE
  - Linux-like, but with 7 syscalls only
    - `transmit / receive / fdwait (≈select)`
    - `allocate / deallocate` (even executable!)
    - `random`
    - `_terminate`
- no signals, threads, shared memory
- “Bring Your Own Defense” approach (and pay for it)
  - Not even “the usual”: stack is executable, no ASLR, …

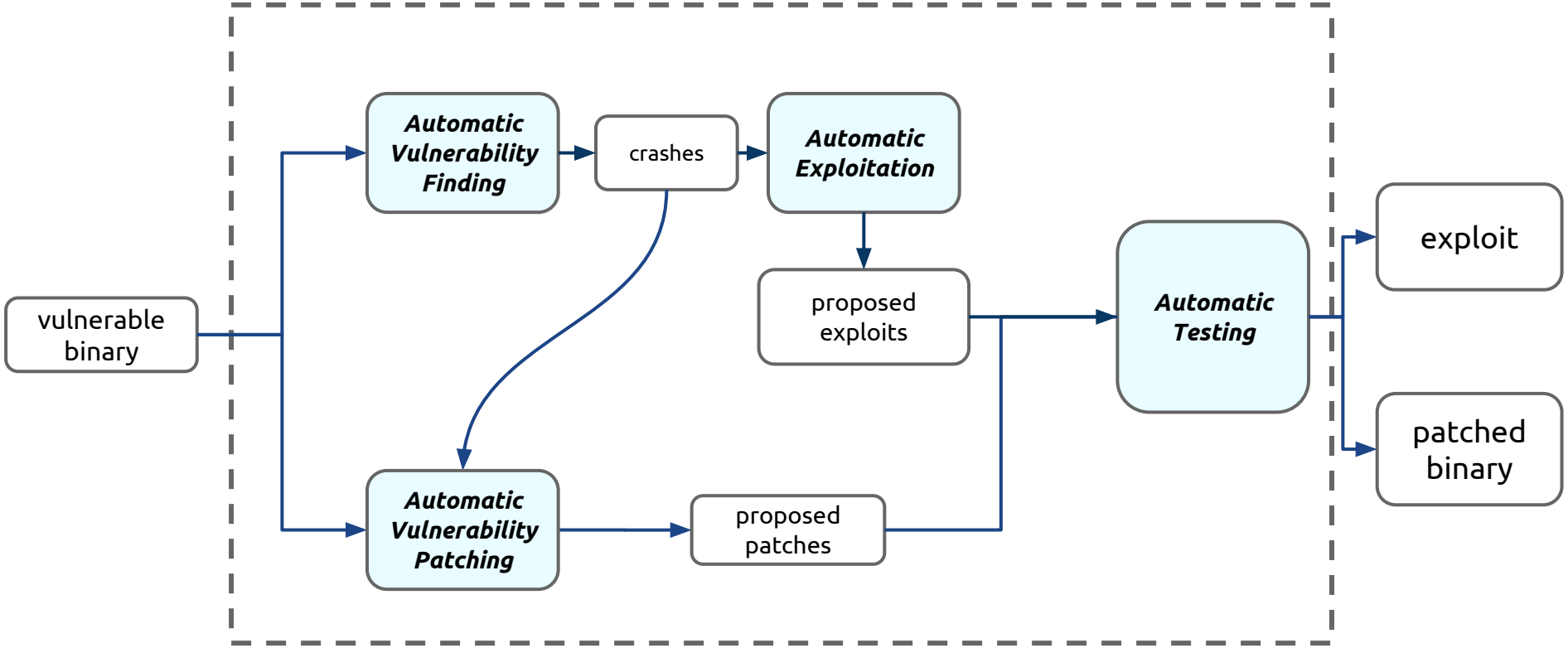
Roadmap

- DARPA Cyber Grand Challenge (CGC)
  The (almost-)Million Dollar Baby
- **Our Cyber Reasoning System (CRS)**
  Fancy term for auto-playing a CTF
- Automated Vulnerability Discovery
  Driller
- Automated Vulnerability Exploitation
  How it works
- Auto-exploitation demo using **angr**
  Open-source binary analysis framework

Shellphish CRS



Shellphish CRS



Roadmap

- DARPA Cyber Grand Challenge (CGC)
  The (almost-)Million Dollar Baby
- Our Cyber Reasoning System (CRS)
  Fancy term for auto-playing a CTF
- **Automated Vulnerability Discovery**
  Driller
- Automated Vulnerability Exploitation
  How it works
- Auto-exploitation demo using **angr**
  Open-source binary analysis framework

Quals Approach to Finding Bugs

Fuzz a binary

… and …

Symbolically explore a binary

Finding bugs

Fuzzing

```
x = int(input())
if x > 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

48 ⇒ "You win!"

Catching Bugs

- Monitors program for crashes

```
x = int(input())
if x > 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

42 ⇒ "You lose!"

3 ⇒ "You lose!"

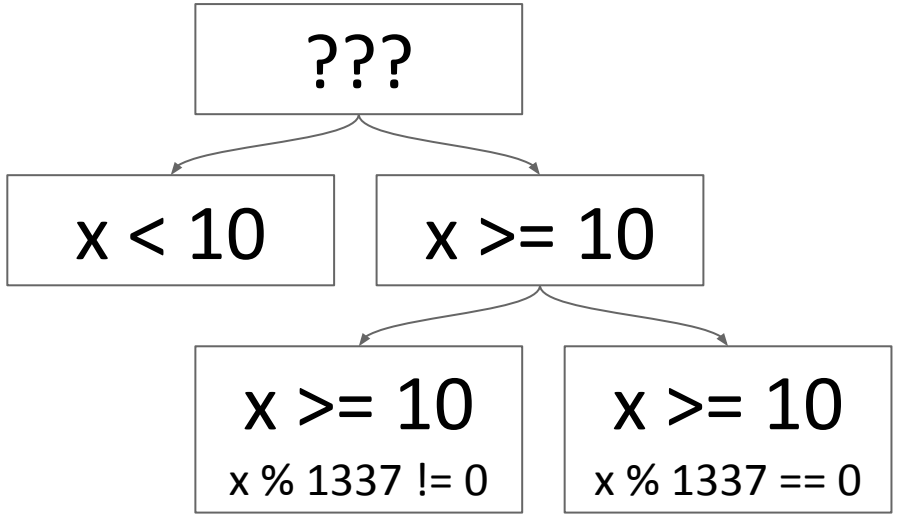
.....

57 ⇒ "You lose!"

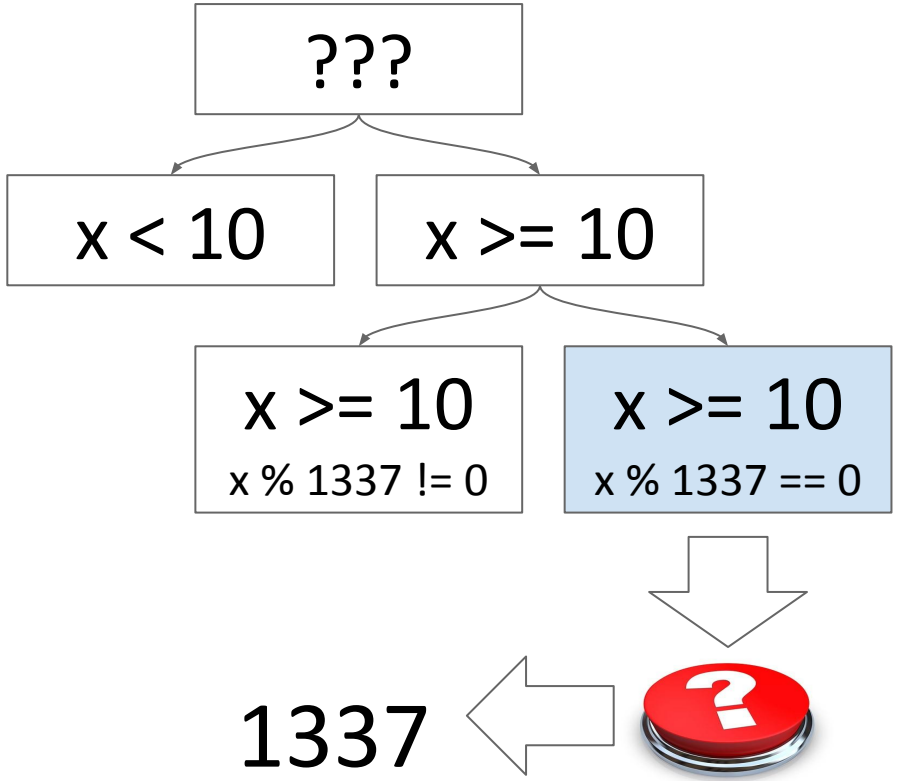
Finding bugs

Symbolic Execution

```
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



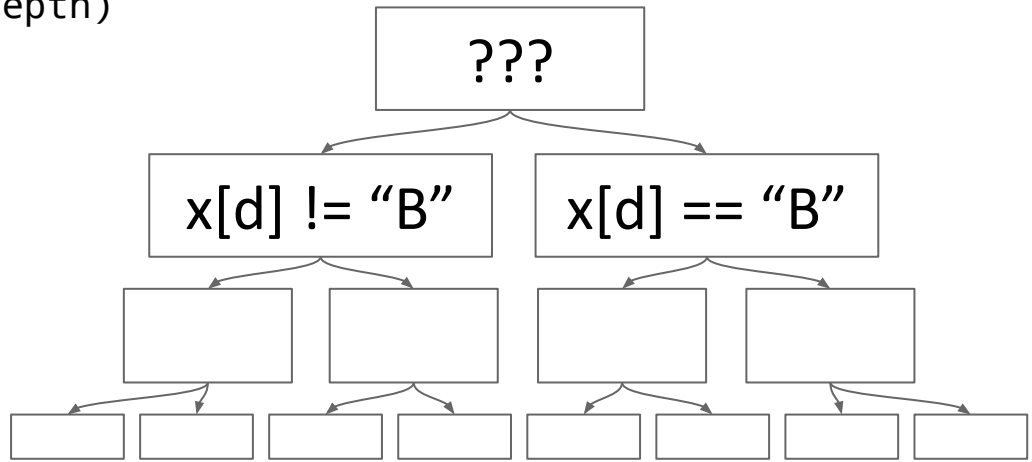
```
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



Catching Bugs

- Checks each state for safety violations
  - symbolic program counter
  - writes/reads from symbolic address

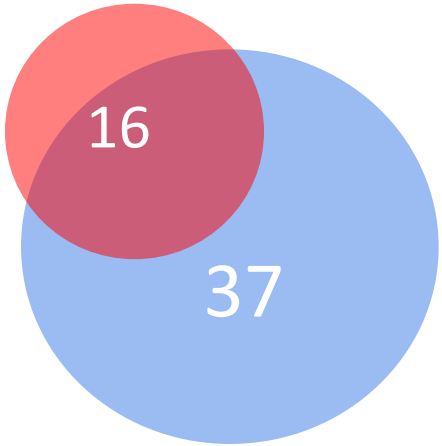
```
x = input()

def recurse(x, depth):
  if depth == 2000:
    return 0
  else:
    r = 0;
    if x[depth] == “B”:
      r = 1
    return r + recurse(x[depth], depth)

if recurse(x, 0) == 1:
  print “You win!”
else:
  print “You lose!”
```



Quals Performance

- Symbolic Execution (angr): 16
- Fuzzing (AFL): 37

Total: 44 / 128



… need something better

Driller: Augmenting Fuzzing through Selective Symbolic Execution

NDSS 2016

Different Approaches

Fuzzing

- Good at finding solutions for general conditions
- Bad at finding solutions for specific conditions

Symbolic Execution

- Good at finding solutions for specific conditions
- Spends too much time iterating over general conditions

Fuzzing vs. Symbolic Execution

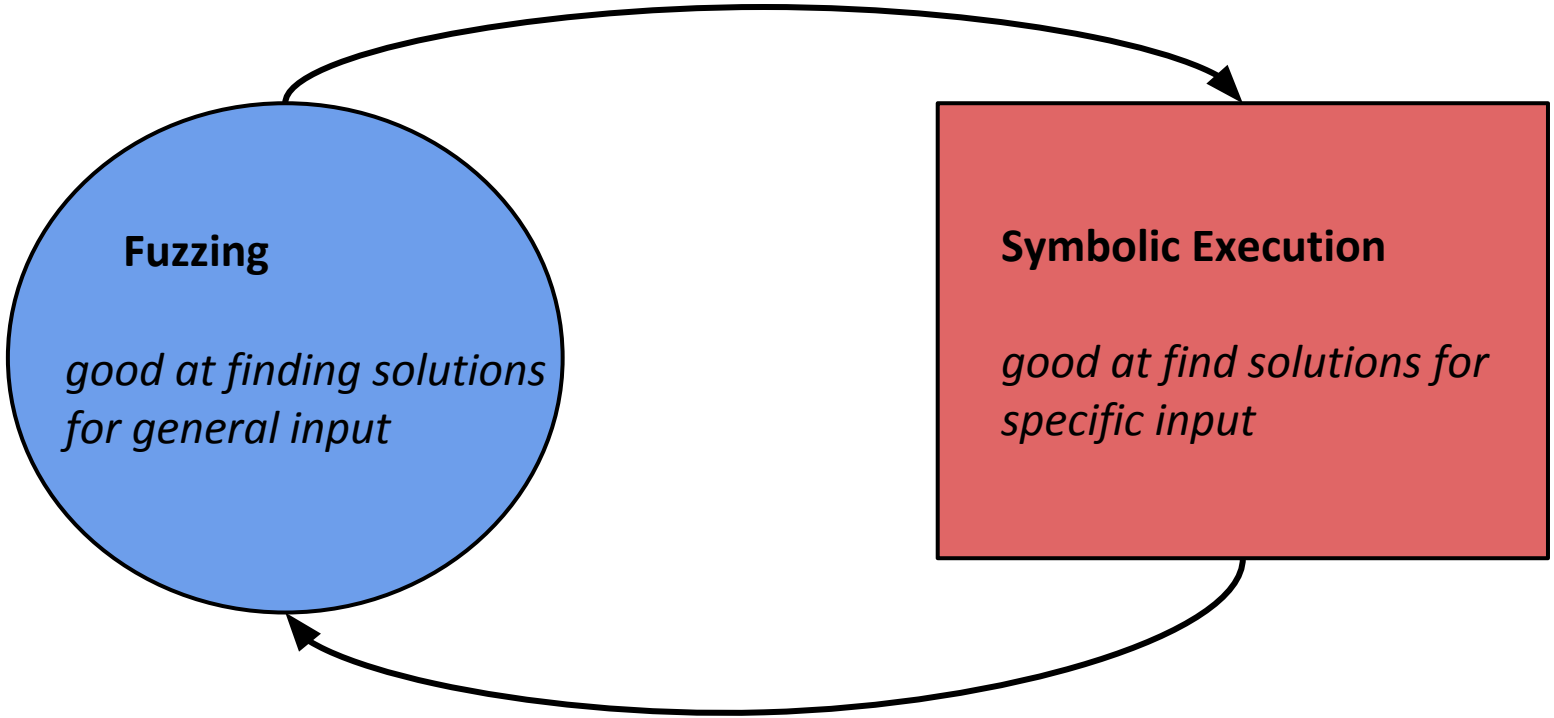
```
x = input()

def recurse(x, depth):
  if depth == 2000:
    return 0
  else:
    r = 0;
    if x[depth] == “B”:
      r = 1
    return r + recurse(x[depth], depth)

if recurse(x, 0) == 1:
  print “You win!”
```

Fuzzing Wins

```
x = int(input())
if x >= 10:
     if x^2 == 152399025:
          print "You win!"
     else:
          print "You lose!"
else:
     print "You lose!"
```

Symbolic Execution Wins

The Goal


Fuzzing
good at finding solutions for general input
Symbolic Execution
good at find solutions for specific input

American Fuzzy Lop + angr

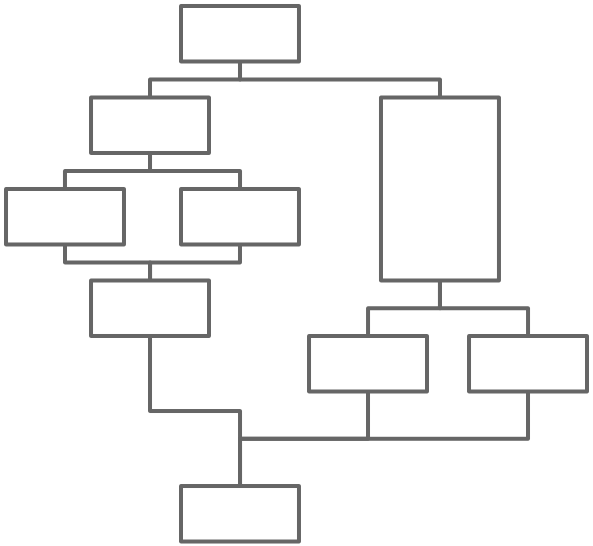
AFL

- state-of-the-art instrumented fuzzer
- path uniqueness tracking
- genetic mutations
- open source

angr

- binary analysis platform
- implements symbolic execution engine
- works on binary code
- available on github

Combining the Two (High-level)

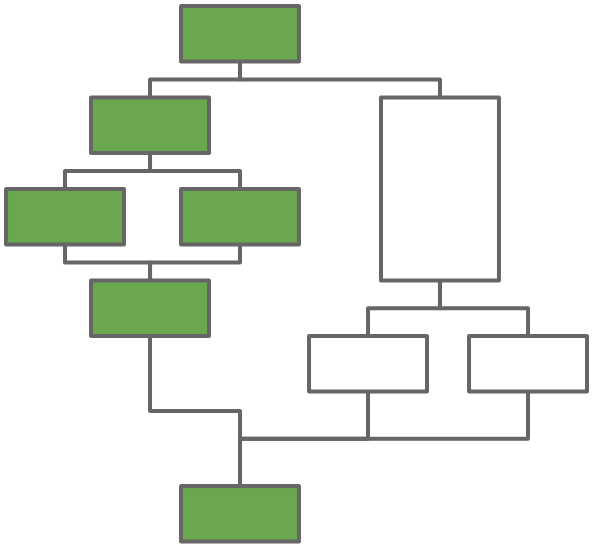


Control Flow Graph

Test Cases

Combining the Two

"Cheap" fuzzing coverage



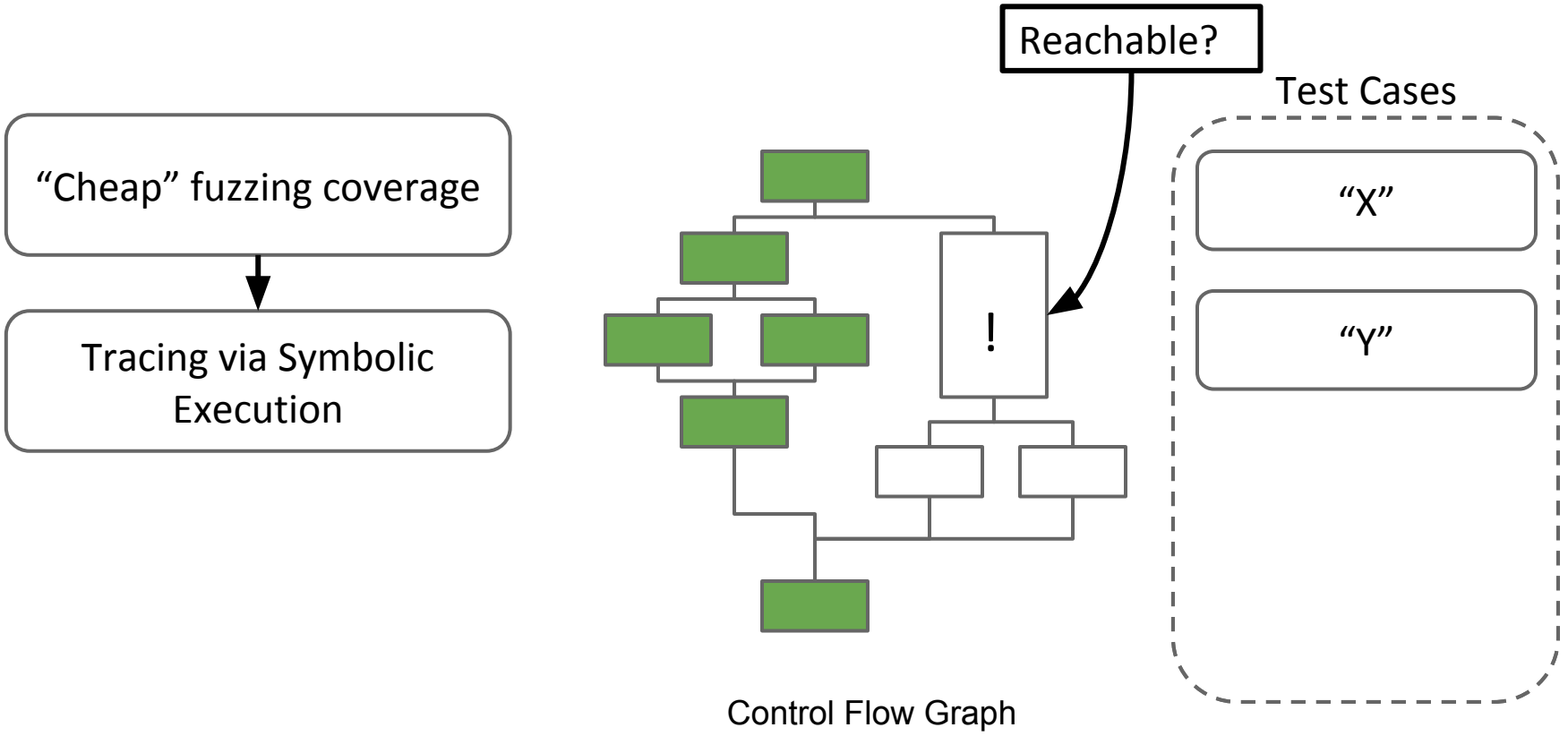
Control Flow Graph

Test Cases

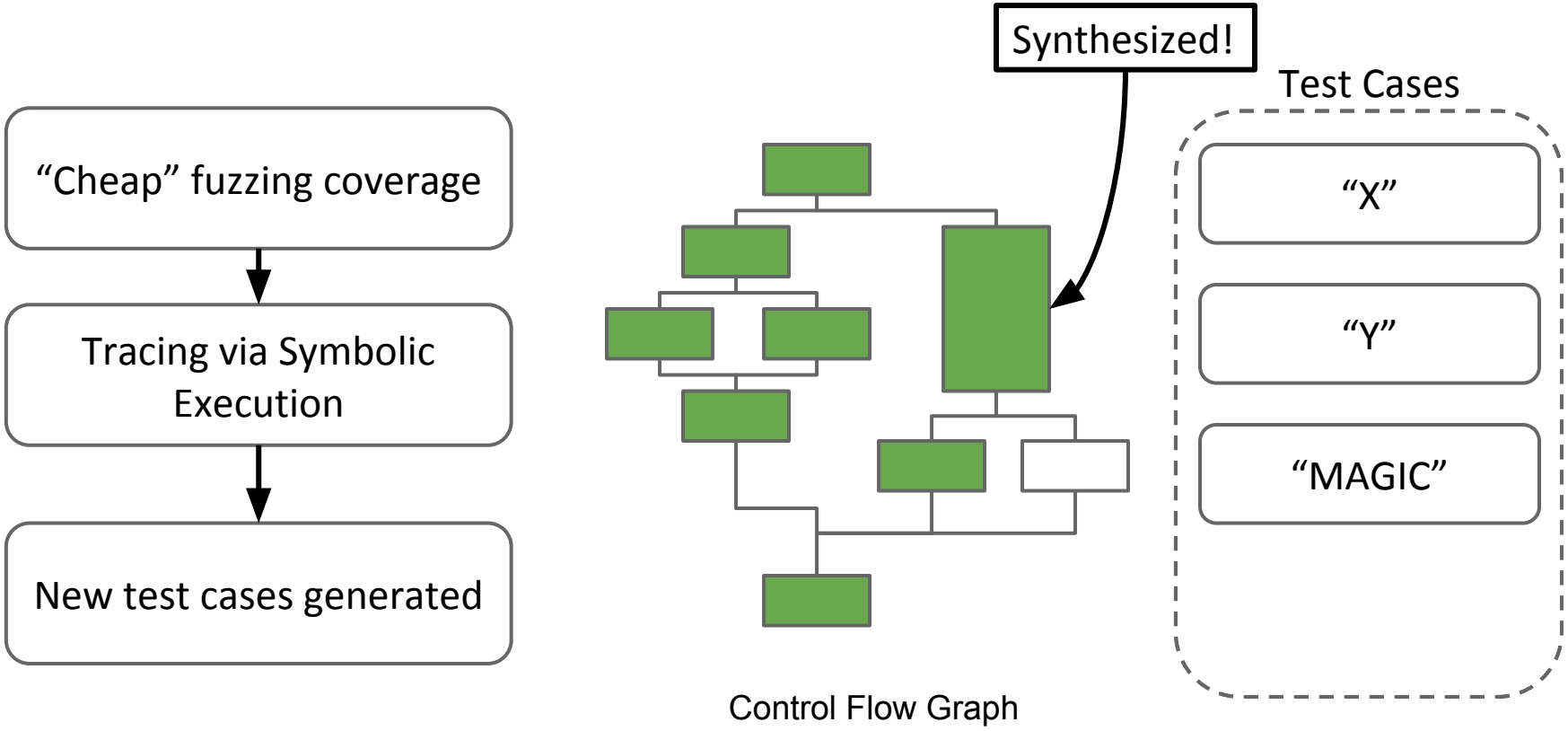
"X"

"Y"

Combining the Two

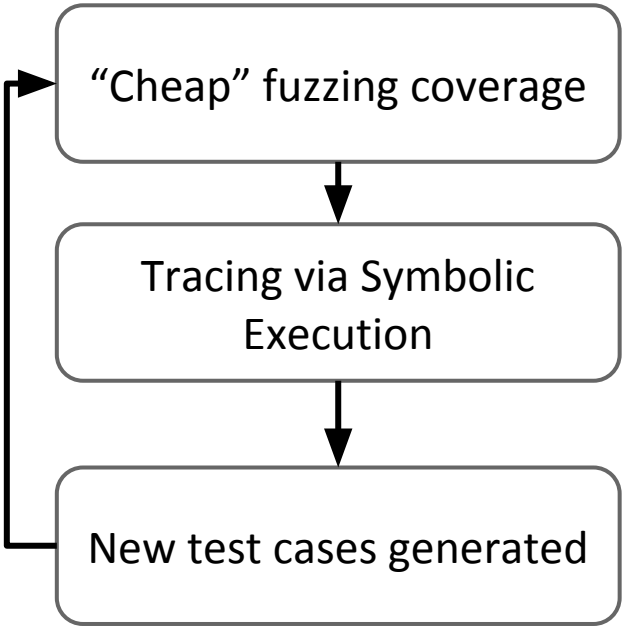


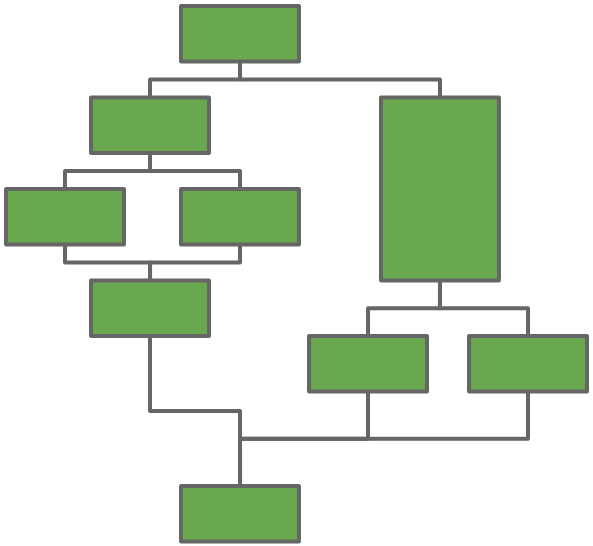
Combining the Two



Combining the Two

Towards completer code coverage!





Control Flow Graph

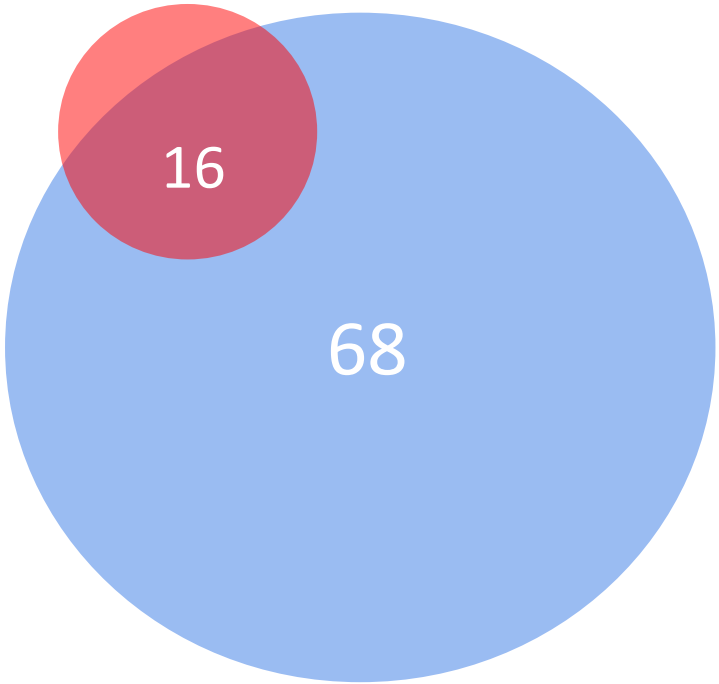
Test Cases

- “X”
- “Y”
- “MAGIC”
- “MAGICY”

CGC Fuzzer Optimizations

- Symbolic Execution (angr) - 16
- Fuzzing (AFL) - 68 total
- S & F Shared - 13 total

71 / 128 binaries

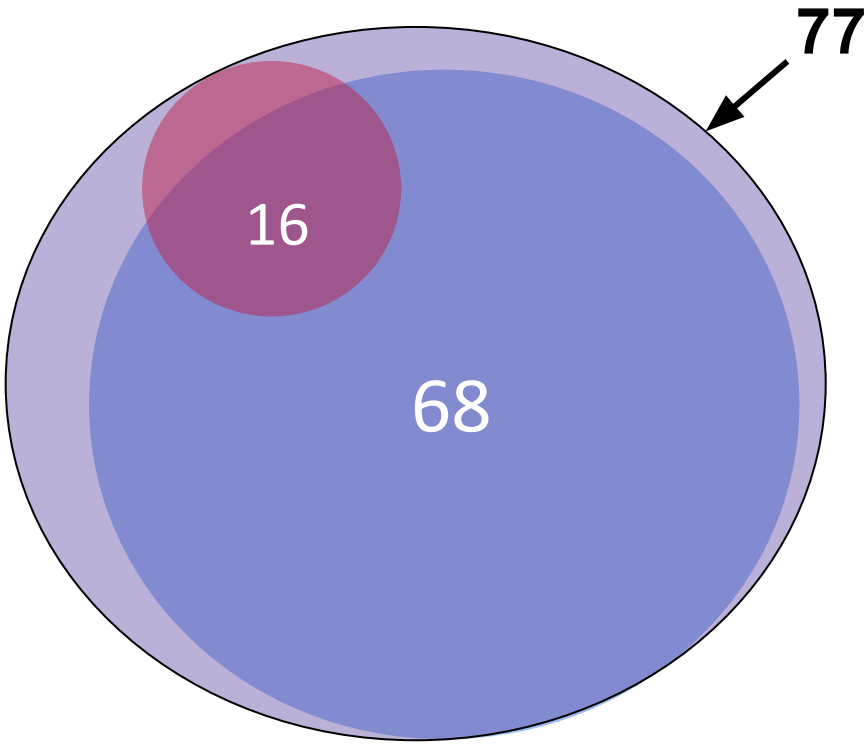


Binary Crashes per Technique

Driller Results

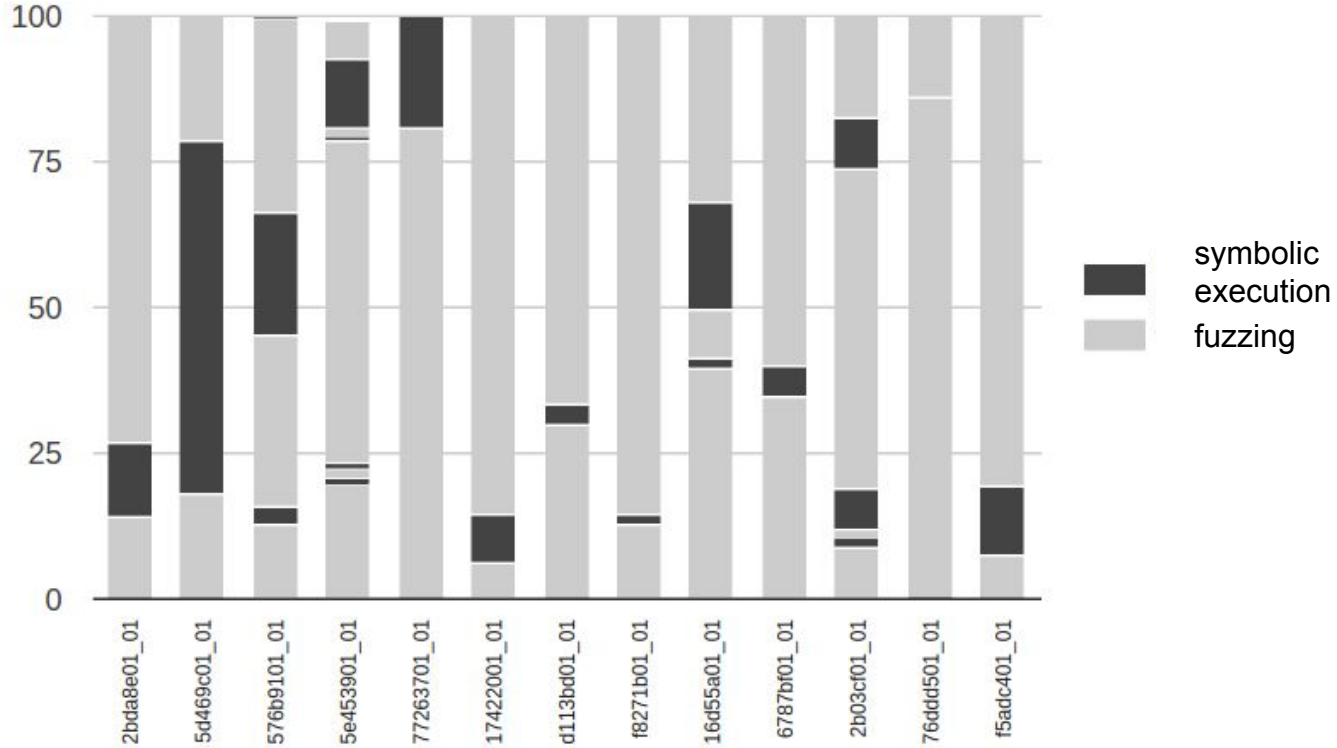
- Symbolic Execution (angr) - 16
- Fuzzing (AFL) - 68 total
- S & F Shared - 13 total
- Driller - 77 total

77 / 128 binaries



Binary Crashes per Technique

Distribution of Transitions Found as Iterations of Symbolic Execution and Fuzzing



Roadmap

- DARPA Cyber Grand Challenge (CGC)
  The (almost-)Million Dollar Baby
- Our Cyber Reasoning System (CRS)
  Fancy term for auto-playing a CTF
- Automated Vulnerability Discovery
  Driller
- **Automated Vulnerability Exploitation**
  How it works
- Auto-exploitation demo using **angr**
  Open-source binary analysis framework

Previous Work

- Automatic Generation of Control Flow Hijacking Exploits …
  - Sean Heelan

- AEG: Automatic Exploit Generation
  - Thanassis Avgerinos, CMU

- Unleashing MAYHEM on Binary Code
  - Sang Kil Cha, CMU

- Q: Exploit Hardening Made Easy
  - Edward J. Schwartz, CMU

- More work being done by Julien Vanegue, as far as I know still largely theoretical at this point

Auto Exploitation - Simplified

- Find a path with safety violations

- Exploit it

Auto Exploitation - Simplified

```
typedef struct component {
      char name[32];
      int (*do_something)(int arg);
} comp_t;

comp_t *initialize_component(char *cmp_name) {
      int i = 0;
      struct component *cmp;

      cmp = malloc(sizeof(struct component));
      cmp->do_something = sample_func;

      while (*cmp_name)
            cmp->name[i++] = *cmp_name++;

      cmp->name[i] = '\0';
      return cmp;
}
x = get_input();
cmp = initialize_component(x);
cmp->do_something(1);
```

HEAP

Symbolic Byte[0] Symbolic Byte[1] Symbolic Byte[2] Symbolic Byte[3] Symbolic Byte[4] Symbolic Byte[5] Symbolic Byte[6] Symbolic Byte[7] ...
Symbolic Byte[32] … Symbolic Byte[36]
'\0'

call <symbolic byte[36:32]>

Auto Exploitation - Simplified

Turning the state into an **exploited** state

angr

```
assert state.se.symbolic(state.regs.pc)
```

Constrain **buffer** to contain our shellcode

angr

```
buf_addr = find_symbolic_buffer(state, len(shellcode))
mem = state.memory.load(buf_addr, len(shellcode))
state.add_constraints(mem == state.se.bvv(shellcode))
```

find_symbolic_buffer is not included in the angr project, but an implementation of this will be shown in the demo

Auto Exploitation - Simplified

Constrain **PC** to point to the buffer

angr

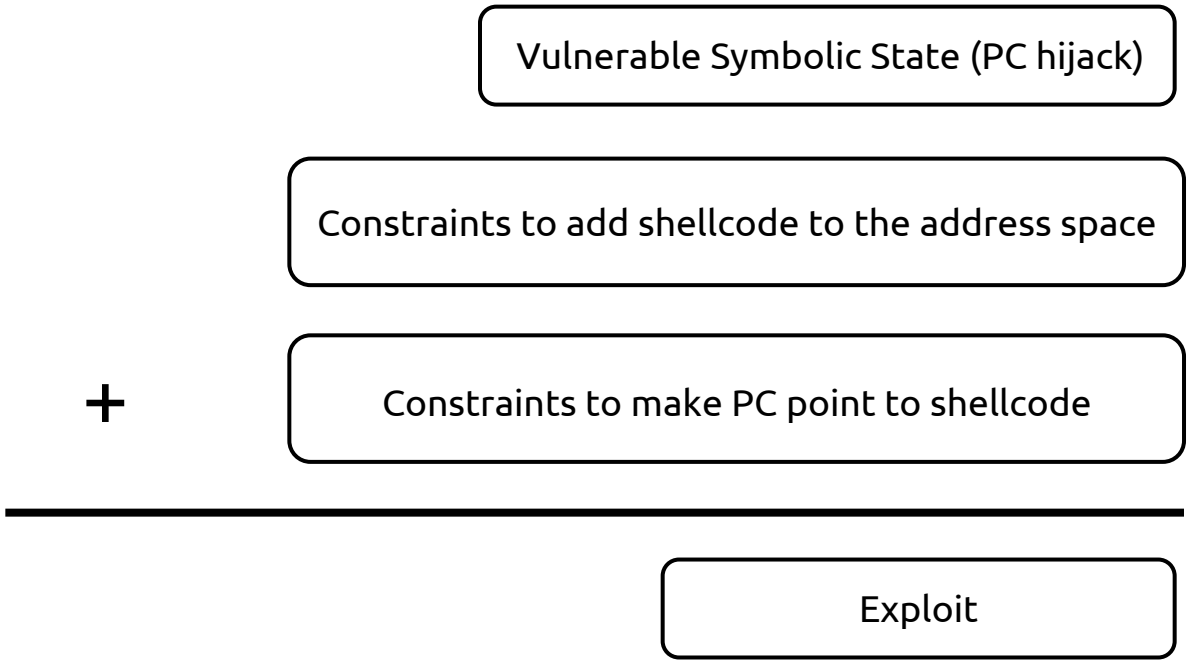
```
state.se.add_constraints(state.regs.pc == buf_addr)
```

Synthesize!

angr

```
exploit = state.posix.dumps(0)
```

Auto Exploitation - Simplified



Auto Exploitation - Advances

- ROP chain generation and ROP placement (**angrop**)
  - state-aware ROP
- Write primitives and Read primitives
- Forced leaking
- Shellcode distribution

Roadmap

- DARPA Cyber Grand Challenge (CGC)
  The (almost-)Million Dollar Baby
- Our Cyber Reasoning System (CRS)
  Fancy term for auto-playing a CTF
- Automated Vulnerability Discovery
  Driller
- Automated Vulnerability Exploitation
  How it works
- **Auto-exploitation demo using angr**
  Open-source binary analysis framework

Demo

Available at:

https://github.com/angr/angr-doc/

examples/insomnihack_aeg