Automating Exploitation

In the Cyber Grand Challenge and Beyond

Chris Salls salls@cs.ucsb.edu

ASU / 2019

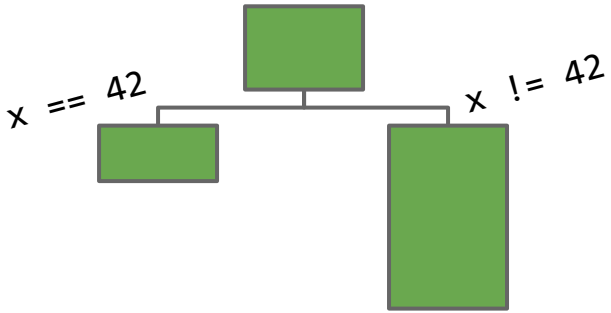
Bio

- PhD Student at UC Santa Barbara
- Shellphish member

Prerequisites

Basic Block

Constraints



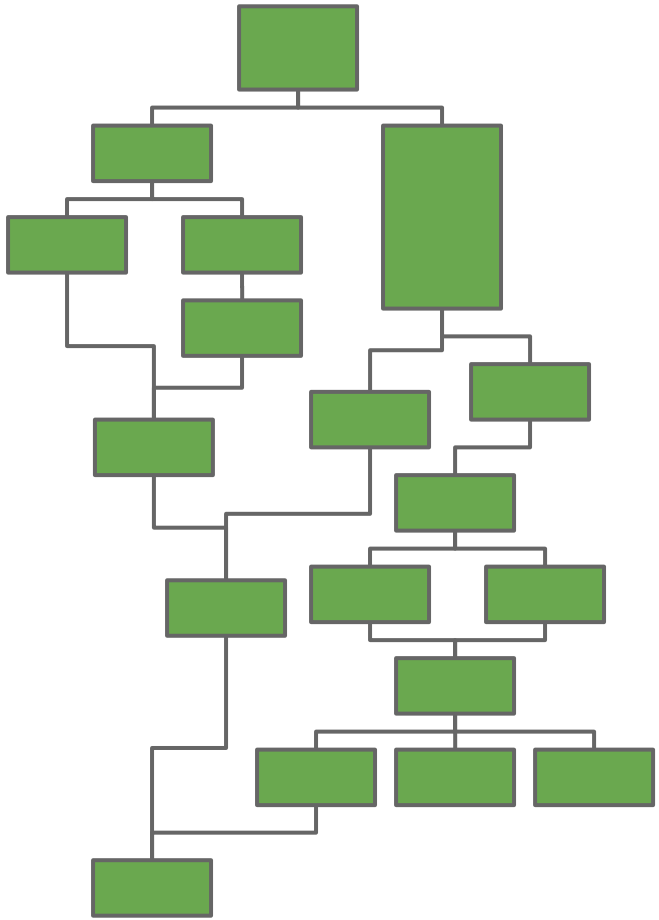

```
x = input()
if x == 42:
     print "Correct"
else:
     print "No"
```

Prerequisites

Basic Block

Constraints

Control Flow Graph



```
x = input()
if x == 42:
     print "Correct"
else:
     print "No"
     if x == 1337:
          print "Fine"
........
........
........
........
........
........
........
........
........
........
........
```

Prerequisites

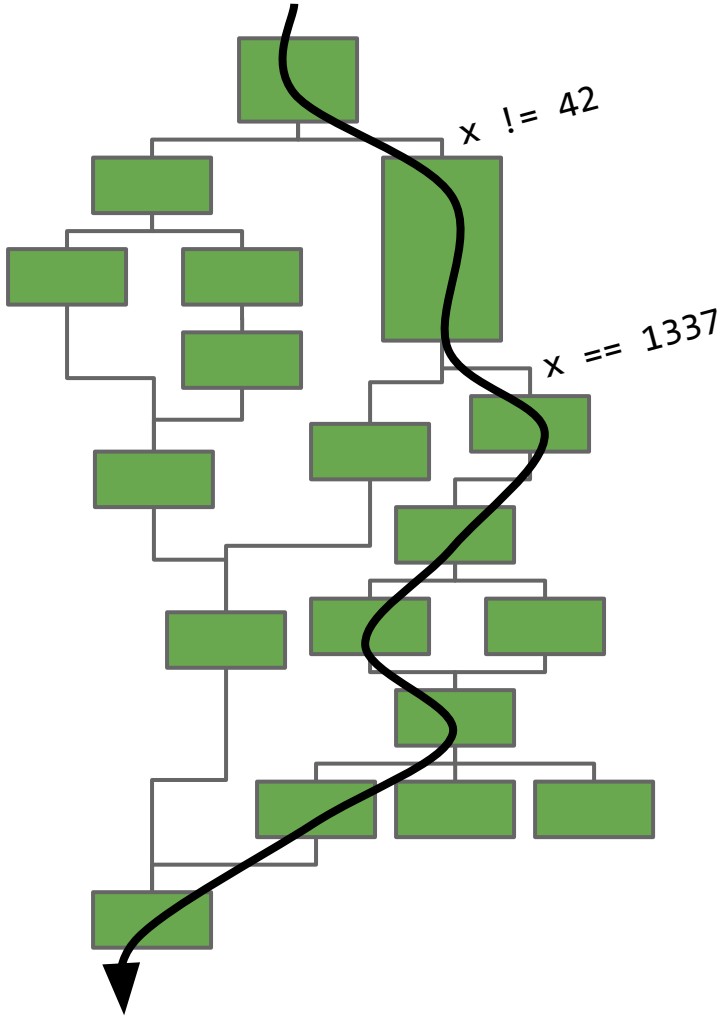
Basic Block

Constraints

Control Flow Graph

Path

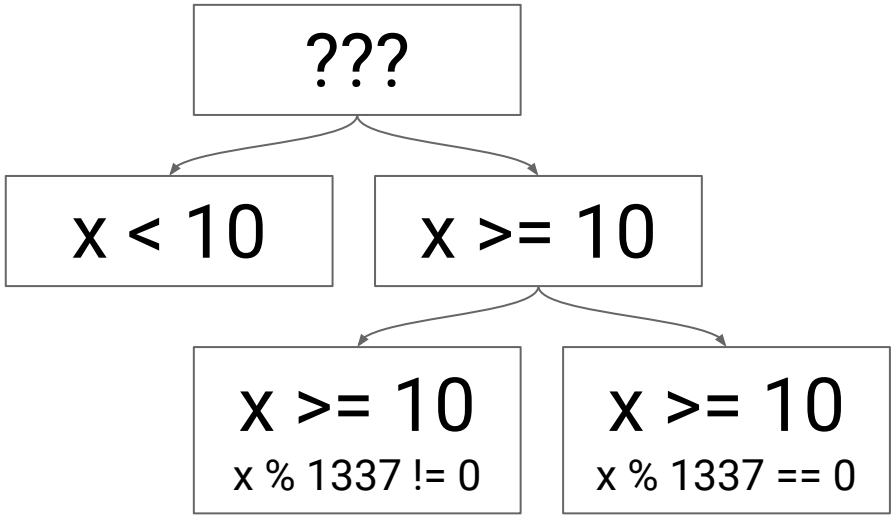
Path Predicates




```
x = input()
if x == 42:
     print "Correct"
else:
     print "No"
     if x == 1337:
          print "Fine"
........
........
........
........
........
........
........
........
........
........
........
```

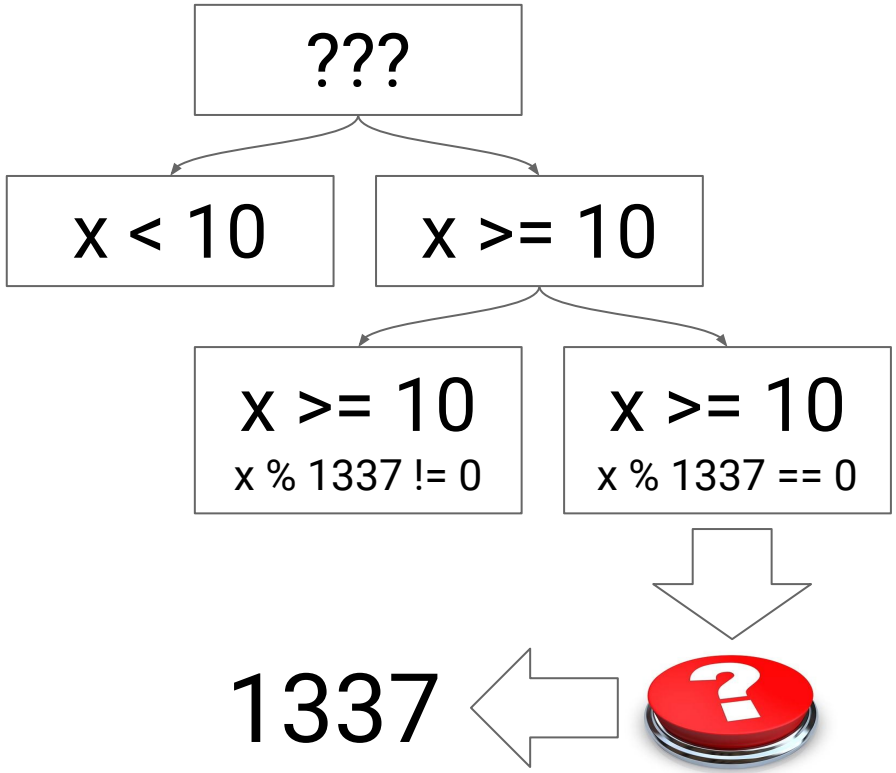
Symbolic Example

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



Symbolic Example

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



Roadmap

- **The CGC**
  Differences in CGC vs the “real” world
- Finding Bugs
  Fuzzing + symbolic execution
- From Crash to Exploit
  Symbolic tracing of crashes
- Real World Challenges
  Limitations and unsolved problems

The DARPA Cyber Grand Challenge

Completely autonomous system

- Patch
- Crash
- Exploit

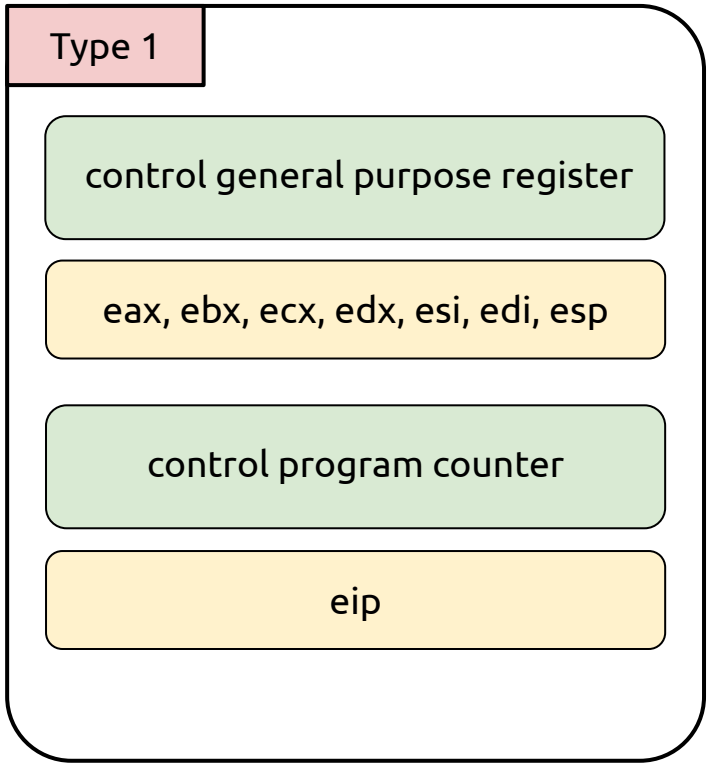
Simplified Environment

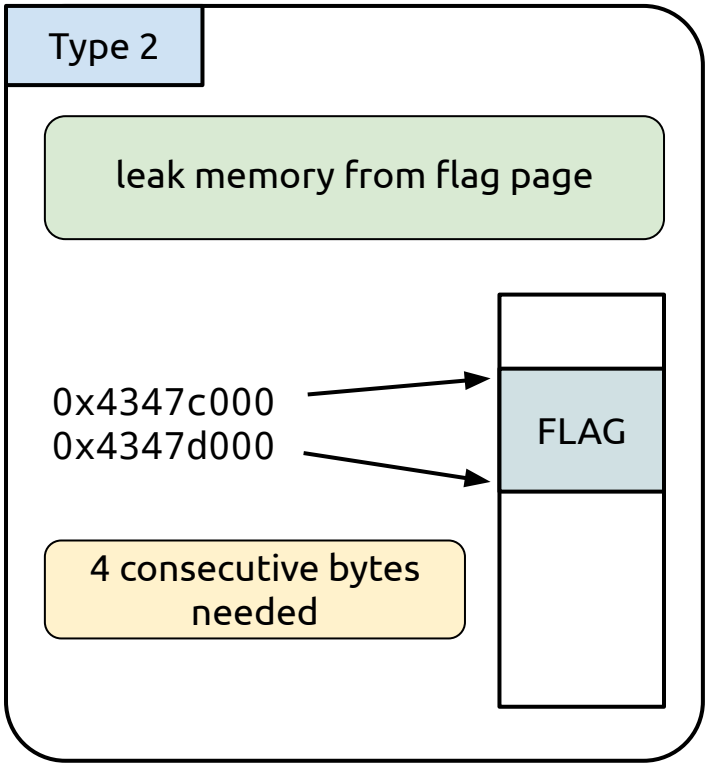
- Linux-inspired environment, with only 7 syscalls
  - receive, transmit, fdwait
  - allocate, dealloc
  - random
  - terminate
- No need to model the POSIX API!
- Otherwise real(istic) programs.

Memory Protections?

- [ ] ASLR
- [ ] NX
- [ ] Canaries
- [x] None of the Above

Two Types of Exploits

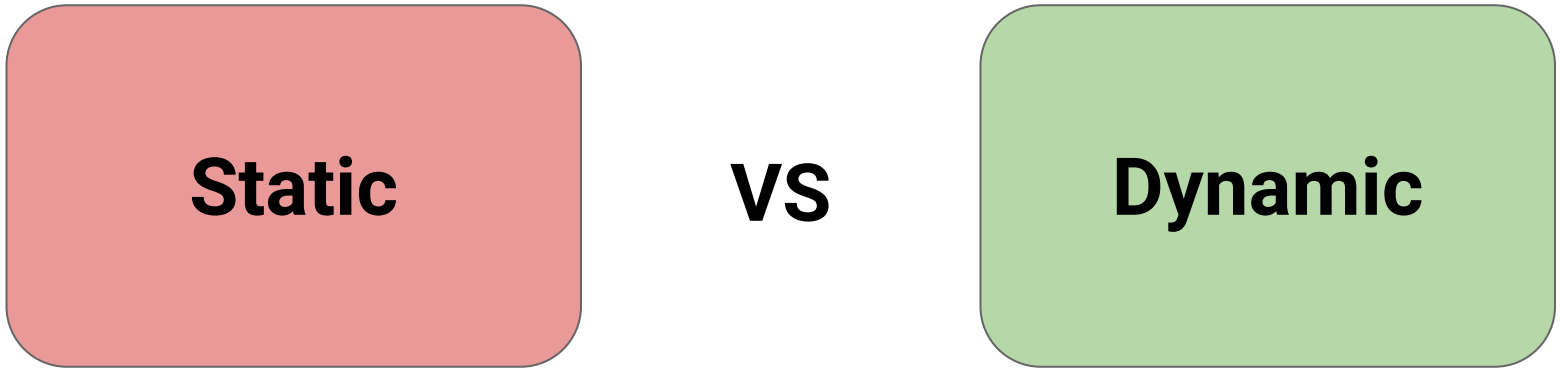




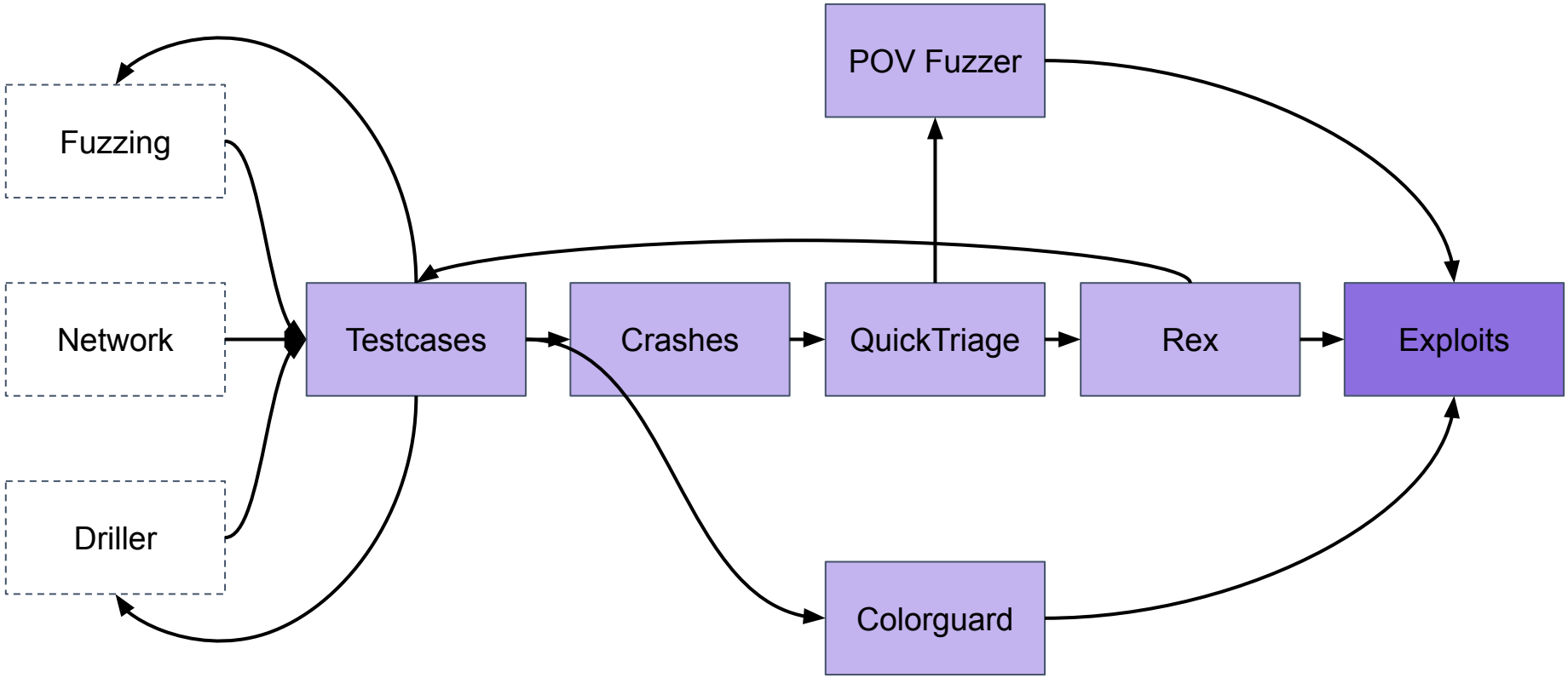
Roadmap

- Exploits in the CGC
  Differences in CGC vs the “real” world
- **Finding Bugs**
  Fuzzing + symbolic execution
- From Crash to Exploit
  Symbolic tracing of crashes
- Real World Challenges
  Limitations and unsolved problems

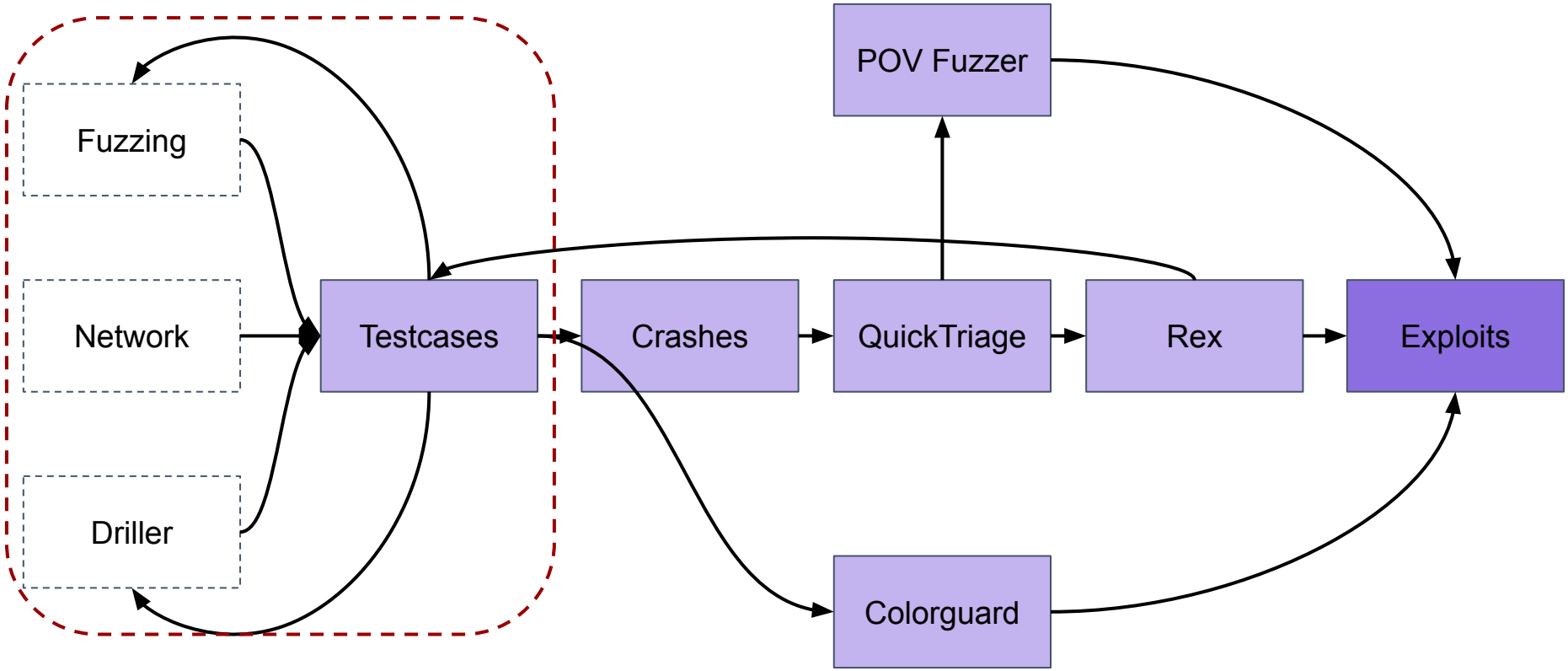
Finding Bugs



Exploitation Pipeline



Exploitation Pipeline



Testcase Generation

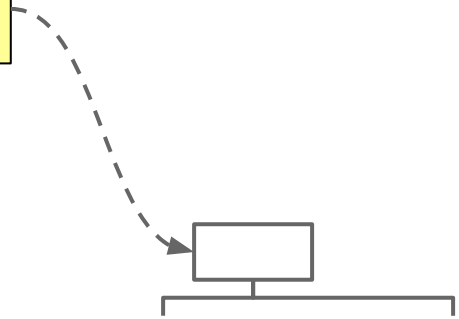
- Fuzzing with AFL
    - 8 fuzzers per challenge, 1 master, 7 slaves
    - Ability to add more slaves on-the-fly

- Network Tap
    - “Lazy” network syncing
    - Scheduled high-priority ‘sync’ jobs to read in pcaps
    - Uses AFL-showmap

- Driller...

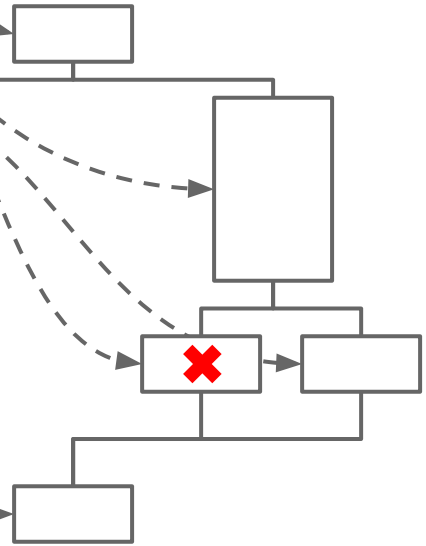
Driller - Motivating Example

```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```



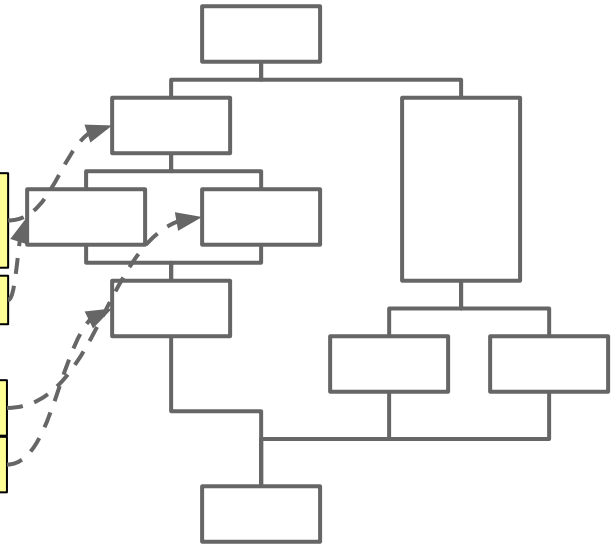
Driller - Motivating Example

```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```



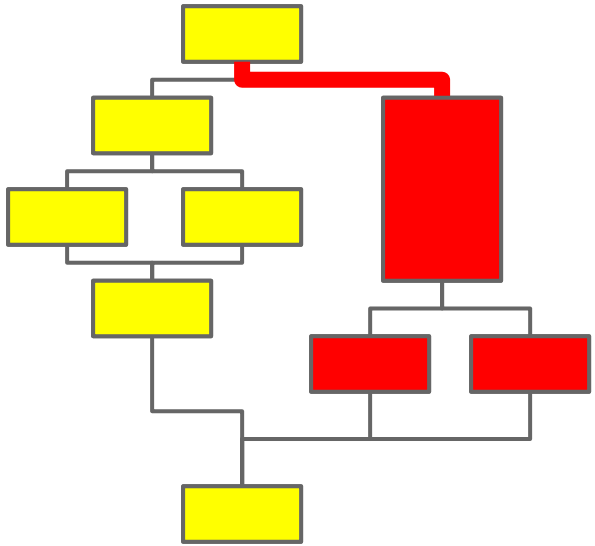
Driller - Motivating Example

```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```



Fuzzing

```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```

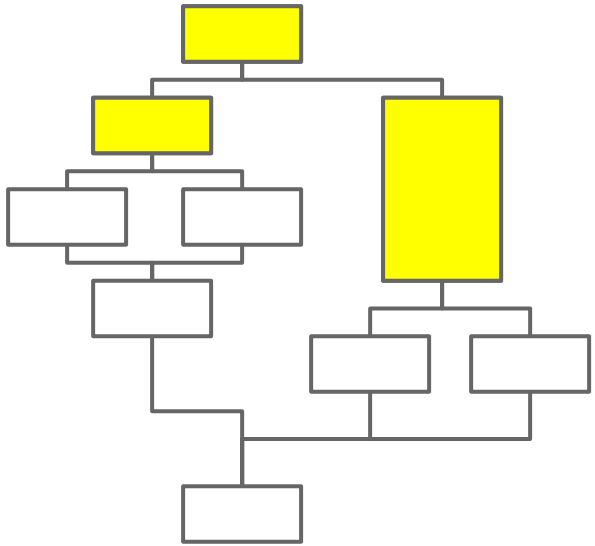


Test Cases

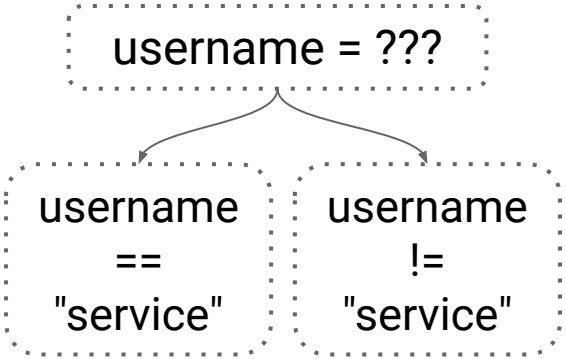
- “asdf:111”
- “asDA:111”
- “asDAAA:1111”
- “asDALA:11121”
- “axDOO:15129”
- “asFOO:75129”

Symbolic Execution

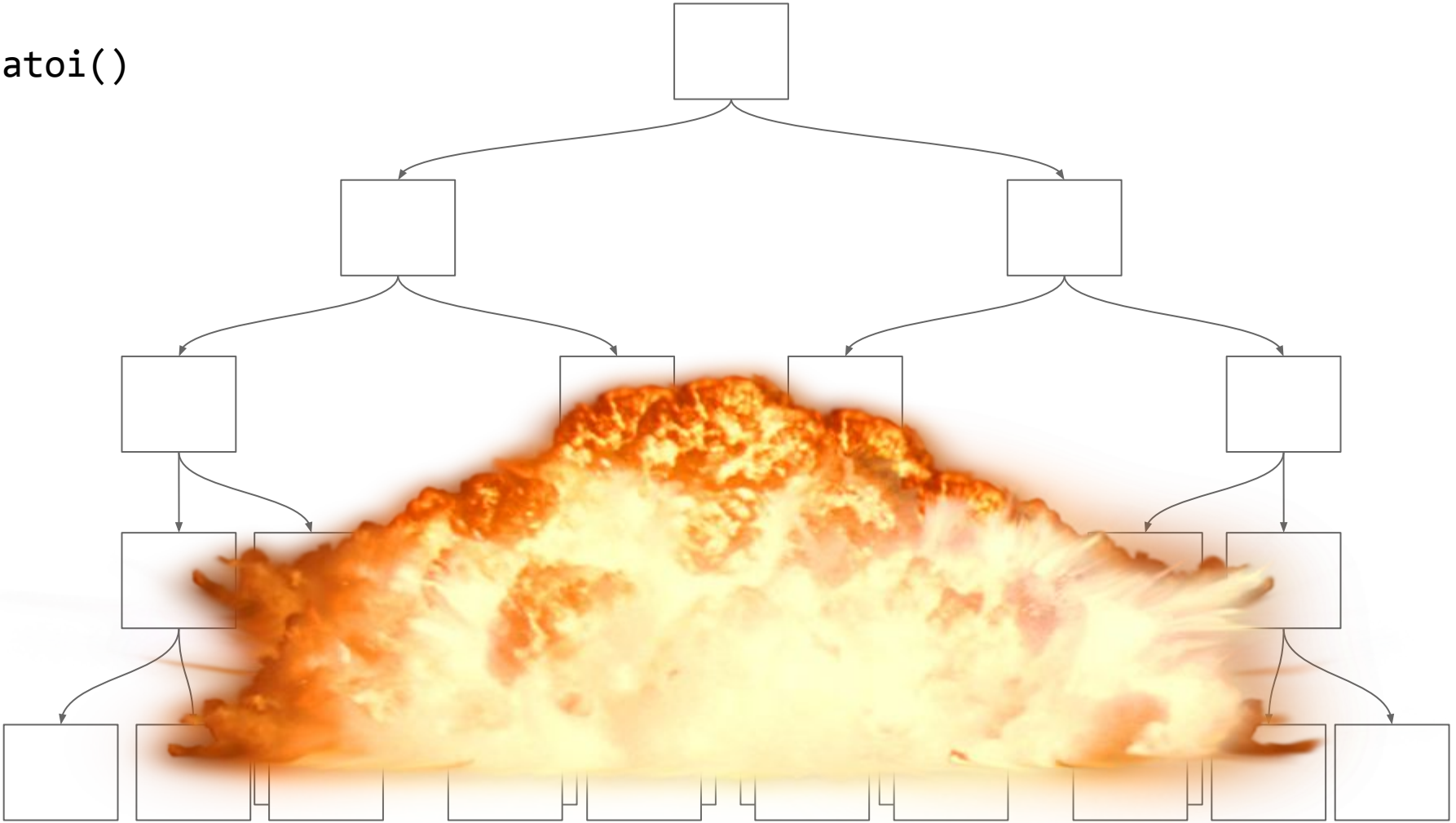
```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```



Constraints

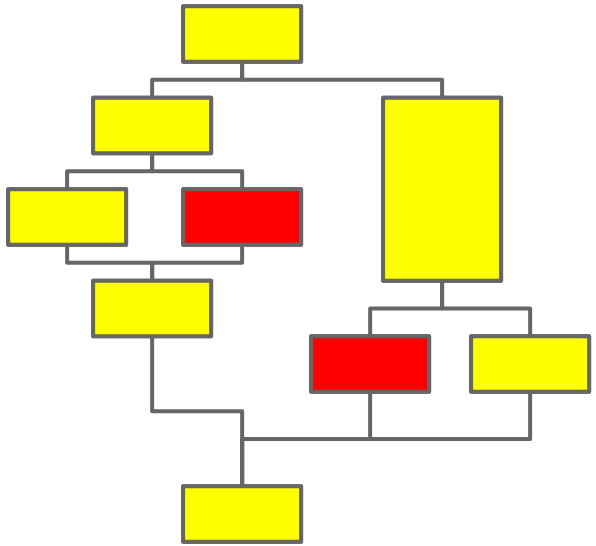


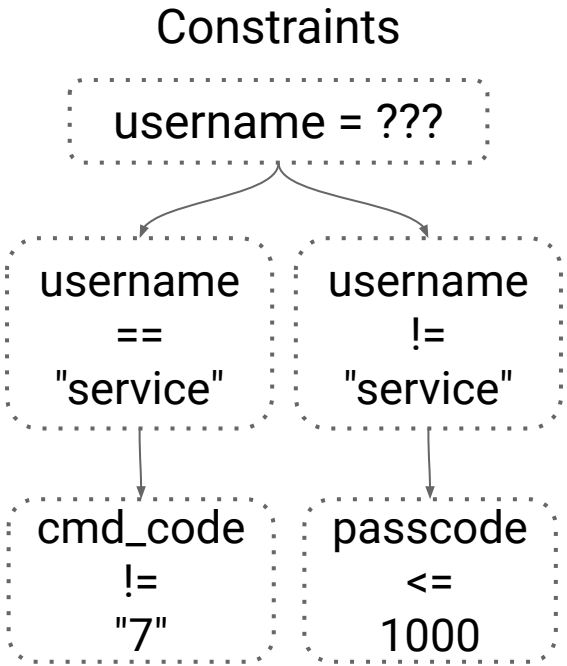
atoi()



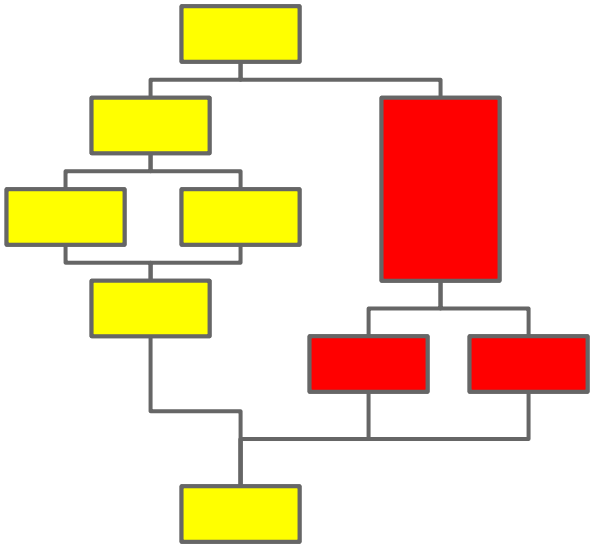
Symbolic Execution

```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```



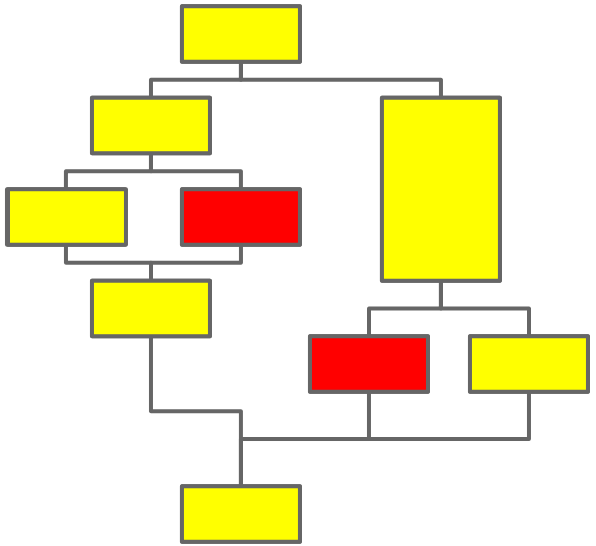


Fuzzing



fast, scalable, dumb

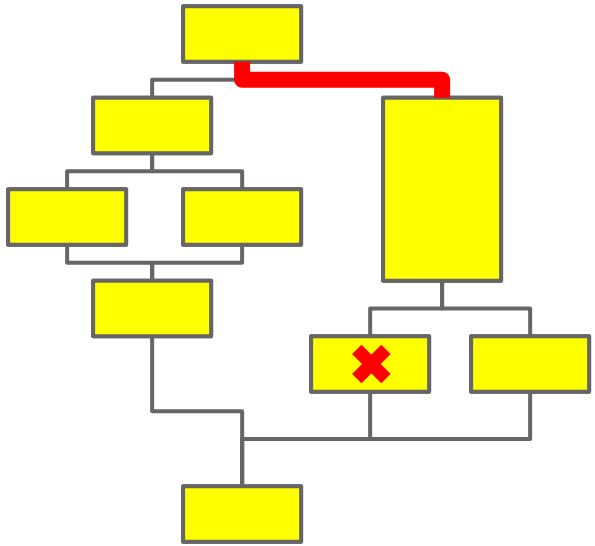
Symbolic Execution



smart, slow, not scalable

Drilling

```
username = input()
if username == "service":
      cmd_code = atoi(input())
      if cmd_code == 7:
            crash()
      else:
            print "Unknown command".
else:
      passcode = atoi(input())
      if 1000 <= passcode < 10000:
            print "Invalid passcode!"
      else:
            auth(username, passcode)
      print "Exiting..."
exit()
```

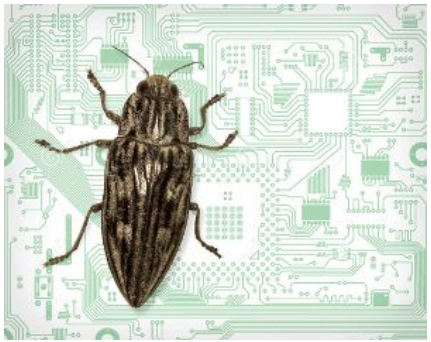


Test Cases

- “asdf:111”
- “asDAAA:1111”
- username == "service"
  cmd_code != "7"
- “service:5”
- “servic3:5”
- “service:7”

One Caveat

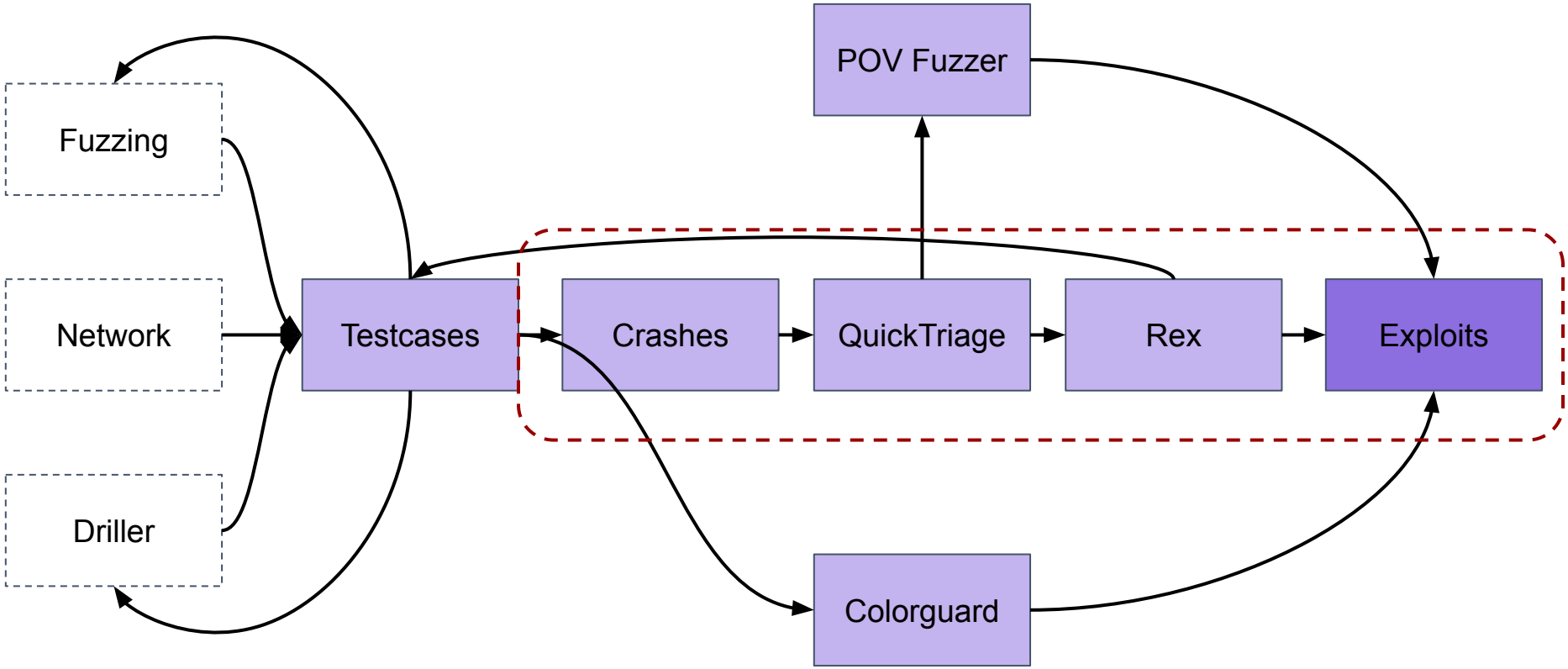
Crash vs Bug



Roadmap

- Exploits in the CGC
  Differences in CGC vs the “real” world
- Finding Bugs
  Fuzzing + symbolic execution
- **From Crash to Exploit**
  Symbolic tracing of crashes
- Real World Challenges
  Limitations and unsolved problems

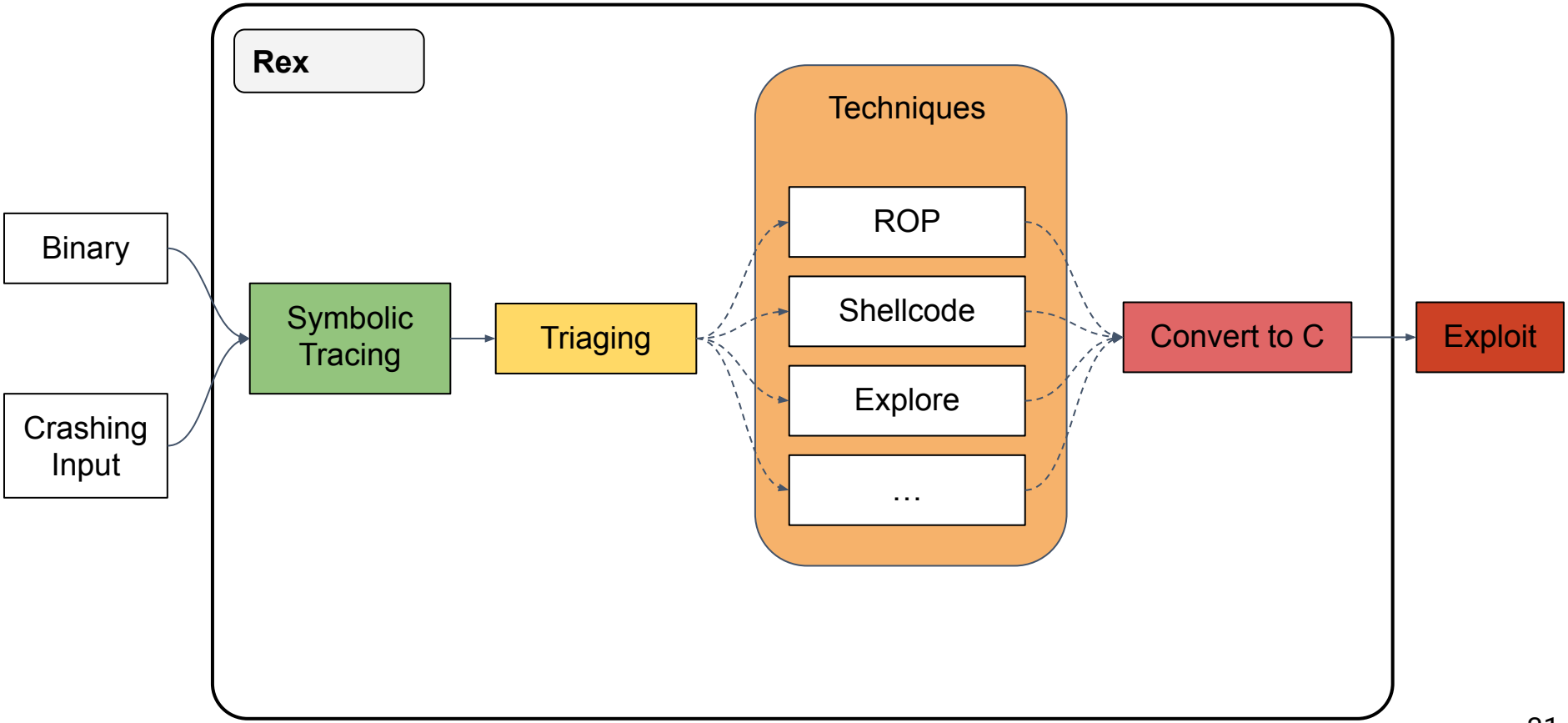
Exploitation Pipeline



Rex - From Crash to Exploit

- Memory corruption tracer and exploiter
- Relies heavily on angr
- Takes a crash and binary as input
- Produces an exploit as an output (in most cases)

Rex Workflow



Symbolic Tracing

```
void say_hello() {
  char name[0x20];
  read_string(name);
  printf("hello %s", name);
  return;
}
```

Stack

```
Symbolic Byte[0]
Symbolic Byte[1]
Symbolic Byte[2]
Symbolic Byte[3]
Symbolic Byte[4]
...
```

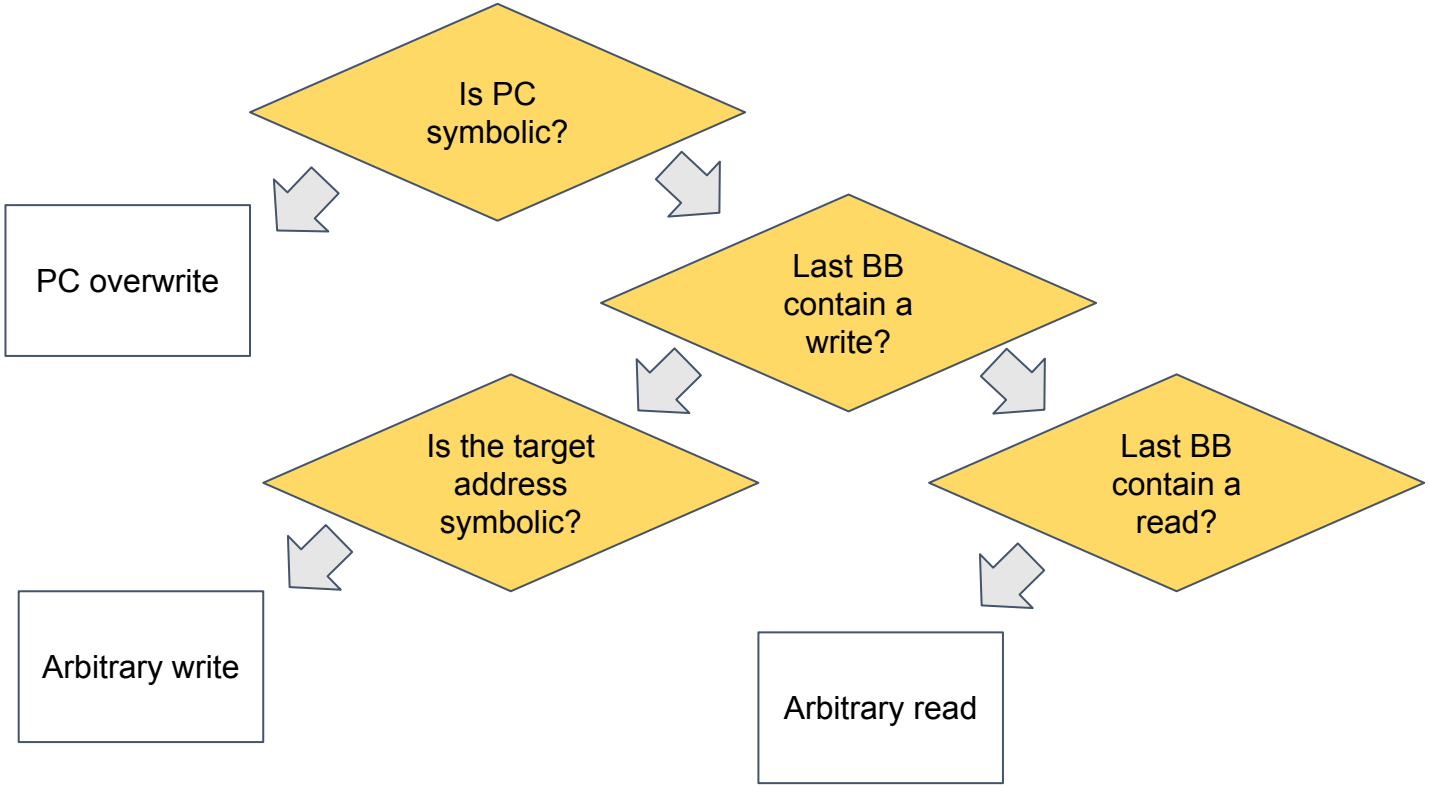
```
<symbolic byte[36:32]>
```

Constraints:

```
Symbolic Byte[0] != '\n'
Symbolic Byte[1] != '\n'
...
```

Return to **<symbolic byte[36:32]>**

Crash Triaging



Technique Application

- Technique = Exploitation Logic
- Input is a symbolic ‘crash’ state
- Output is an exploited state
- Triage result determines which techniques can be applied

Rex - Techniques

PC Overwrite

- Jump to Shellcode
- Pivot to ROP chain
- “Circumstantial”

Arbitrary Read

- Point-to-Flag
- Point-to-Data

Arbitrary Write

- Point-to-Data
- Explore for Exploit

Jump to Shellcode

Find buffer containing symbolic data

```
0xbaaaaf80: <symbolic byte[32:0]>
```

Constrain buffer to have shellcode

```
<symbolic byte[32:0]> == shellcode
```

Jump to Shellcode

Constrain **PC** to point to the buffer

```
PC:
<symbolic byte[36:32]> == 0xbaaaaf80
```

Ask the solver for an input

```
solver.any_str(stdin)
```

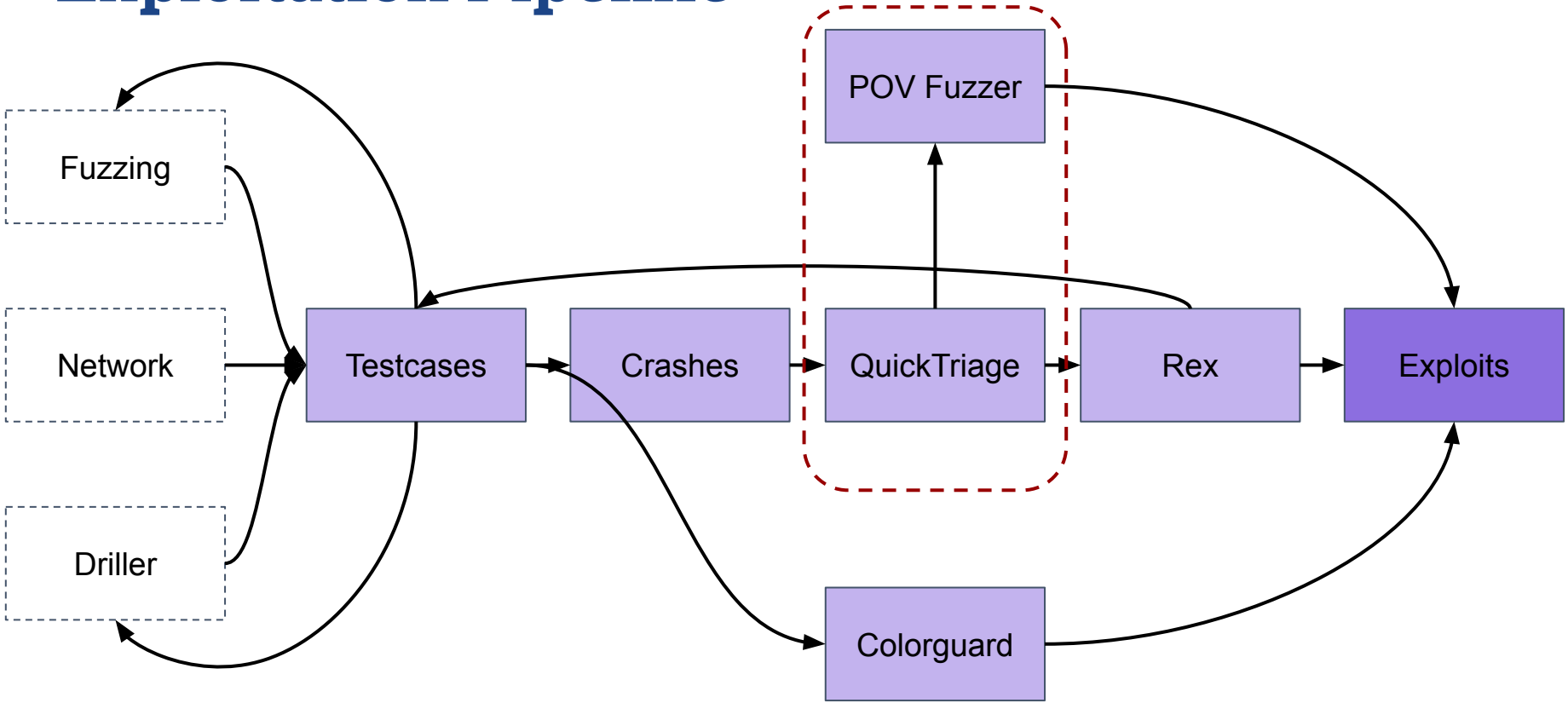
Rex - Pivot to ROP

- ROP is similar
- Find symbolic regions on the stack
- Find a gadget which results in SP pointing to one of these regions

Rex - Pivot to ROP

- Constrain stack to contain ROP payload
- In CGC might read from flag page
- For Linux constructs a `system('/bin/sh')` payload
- Uses angrop

Exploitation Pipeline



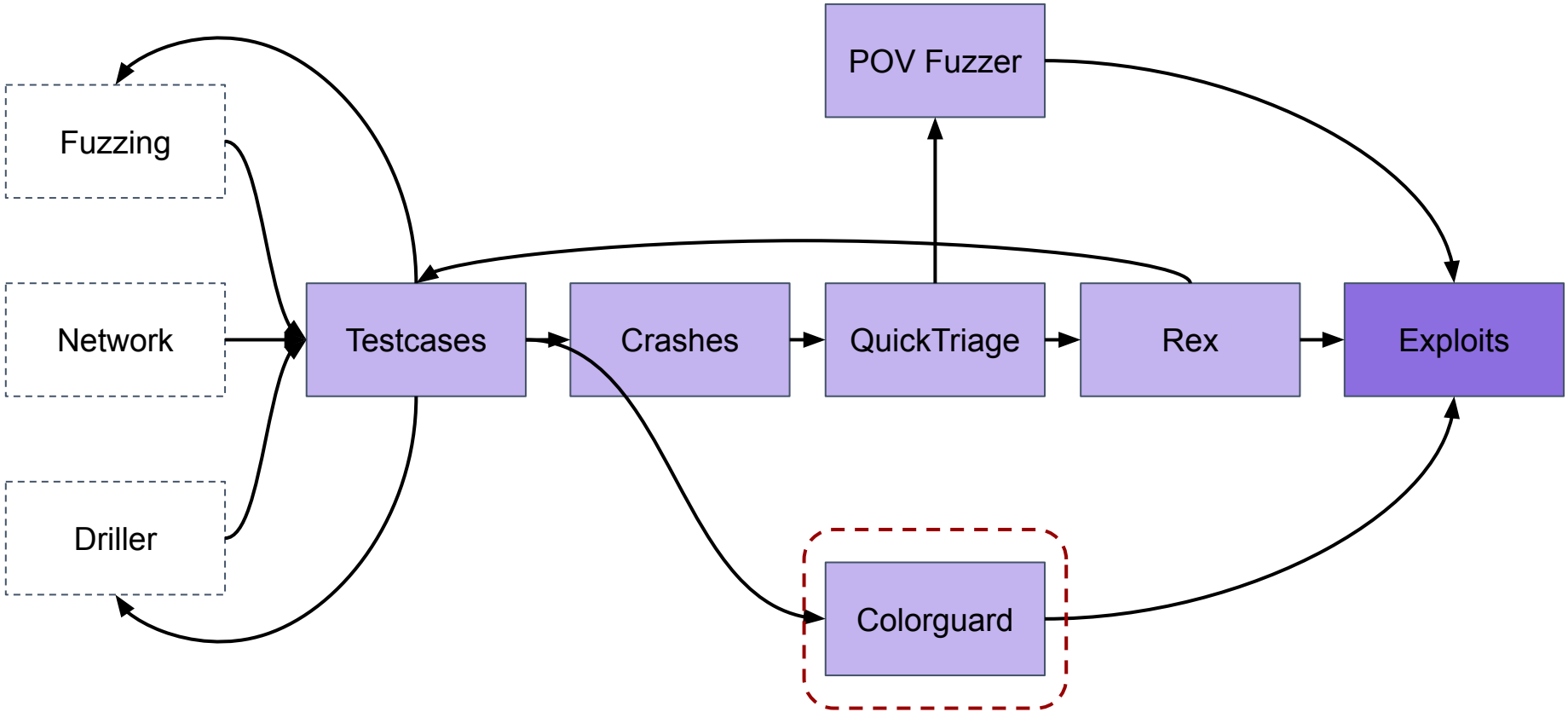
Rex - Scalability Issues

- Rex is SLOW
- AFL can find hundreds of “unique” crashes
- Scheduling Rex on a single challenge for thousands of similar “unique” crashes can consume all cloud resources
- Need a method of scheduling Rex intelligently

Rex - Quick Triage

- Load a dumped core in angr
- Use simple heuristics to determine crash type
- No need for expensive symbolic tracing
- Schedule exploitation of crashes based on type

Exploitation Pipeline

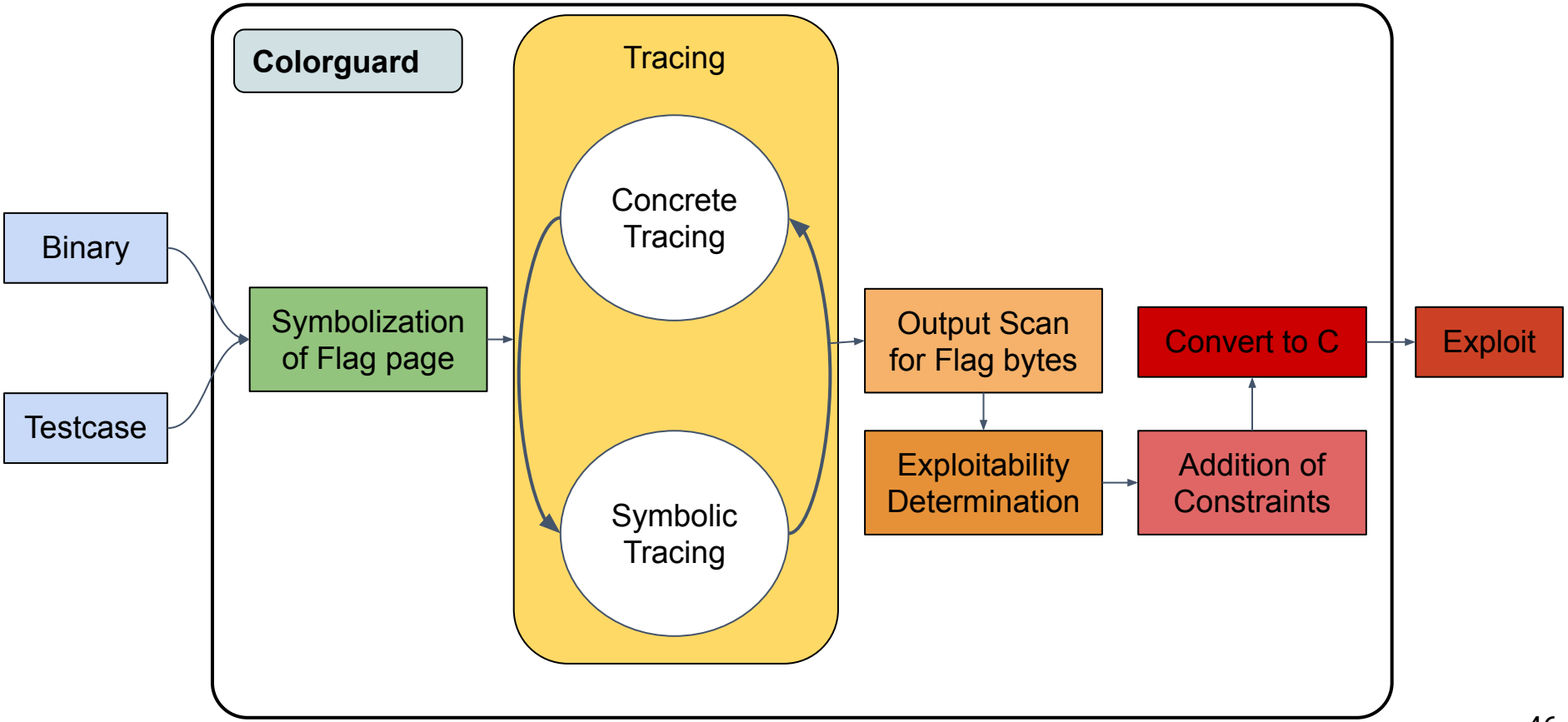


Colorguard - Exploiting Leaks

- Want to find leaks found by AFL and Rex’s point-to-flag method

- Problem: many leaks are operations performed on flag data

Colorguard


Colorguard
Tracing
Concrete Tracing
Symbolic Tracing
Binary
Testcase
Symbolization of Flag page
Output Scan for Flag bytes
Exploitability Determination
Addition of Constraints
Convert to C
Exploit

Roadmap

- Exploits in the CGC
  Differences in CGC vs the “real” world
- Finding Bugs
  Fuzzing + symbolic execution
- From Crash to Exploit
  Symbolic tracing of crashes
- **Real World Challenges**
  Limitations and unsolved problems

Limitations

- Only a bag of techniques + symbolic execution?

Human vs Computer

- Understand Program
- Identify Bug
- Trigger Bug
- Set Up State
- Exploit

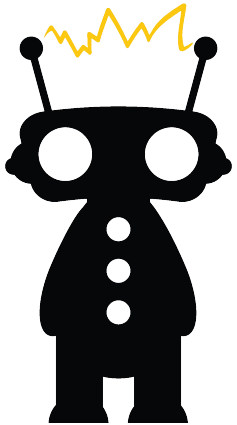
Human vs Computer

Find Crashes

Symbolically Trace

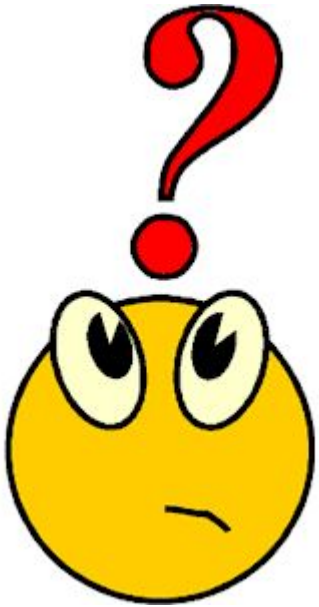
Mutate State

Exploit



Limitations

- Doesn’t understand bugs, only crashes

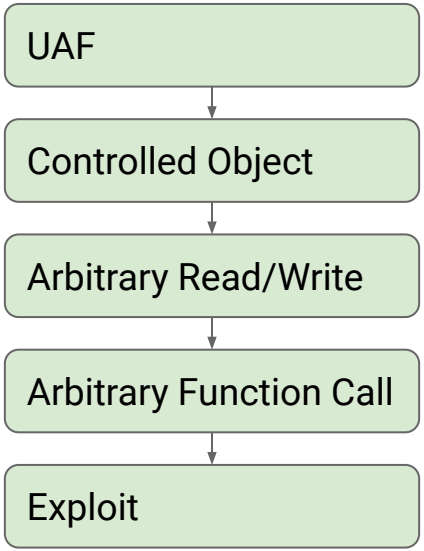


Limitations

- Often need to modify path to set up state
- Getting around mitigations
  - Leak + Crash

Limitations

- No way to iteratively gain control



Lots Of Room For Improvement

Thank You!

- https://github.com/shellphish/fuzzer
- https://github.com/shellphish/driller
- https://github.com/shellphish/rex
- https://github.com/shellphish/colorguard