Finding Vulnerabilities in Embedded Software

Christopher Kruegel
UC Santa Barbara

What are we talking about?

UC Santa Barbara

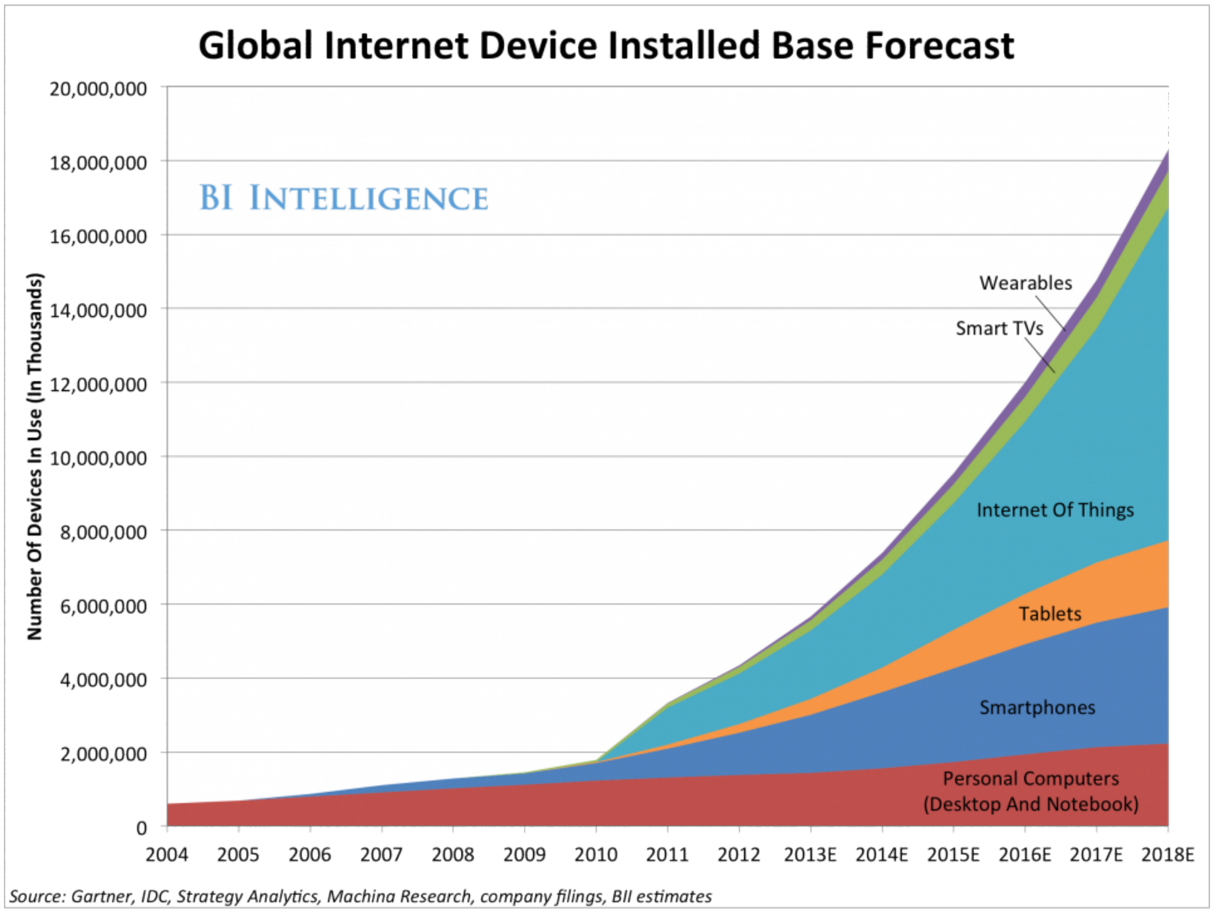
1. firmware and security
2. binary vulnerability analysis
3. vulnerability models
4. automation

Blend between real and virtual worlds

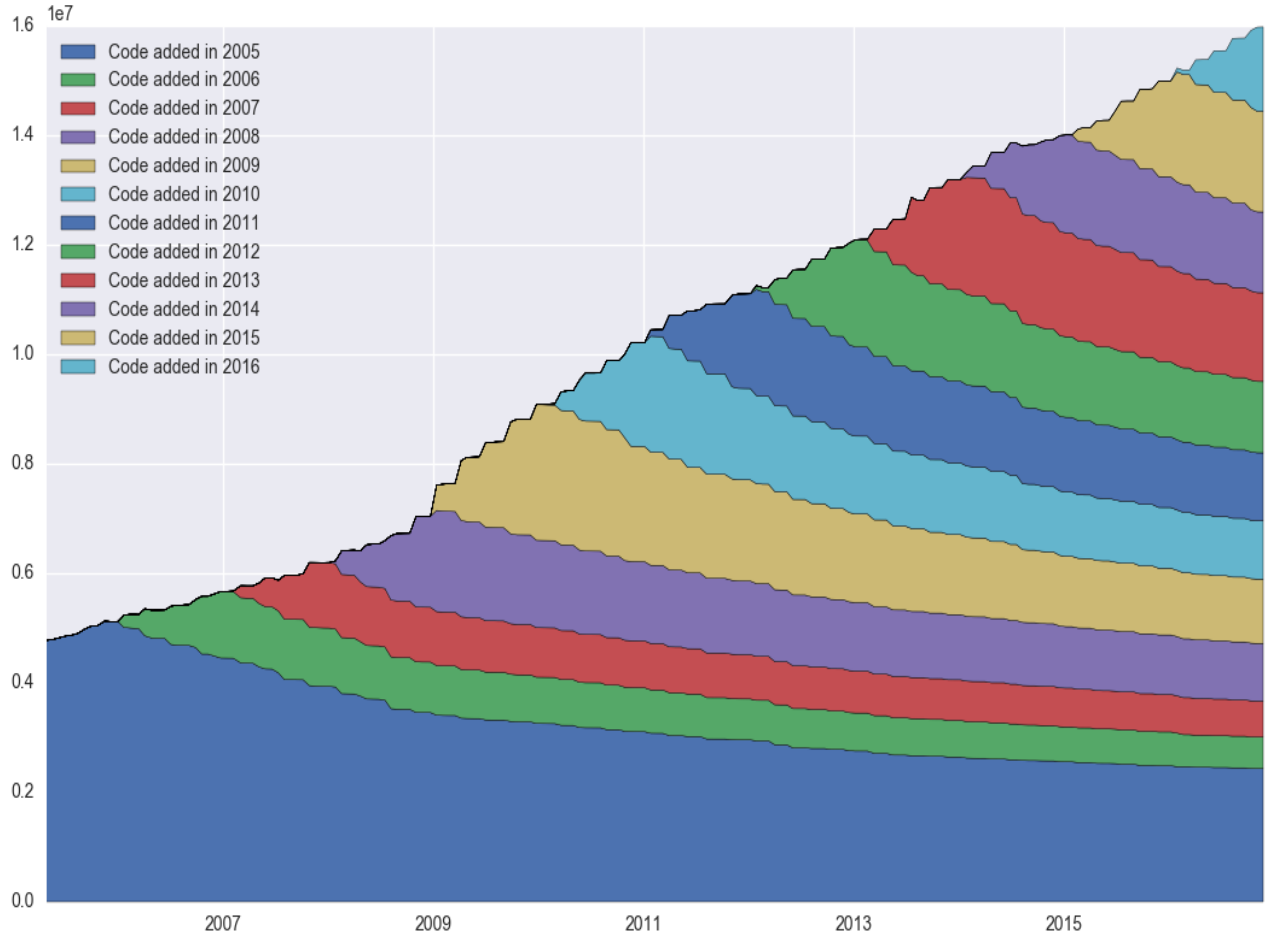
- Embedded software is everywhere
  - captured through many buzzwords
    - pervasive, ubiquitous computing
    - Internet of Things (IoT)
  - sensors and actuators

The "Internet of Things"

UC Santa Barbara



Increase in Lines of Code



Security Challenges

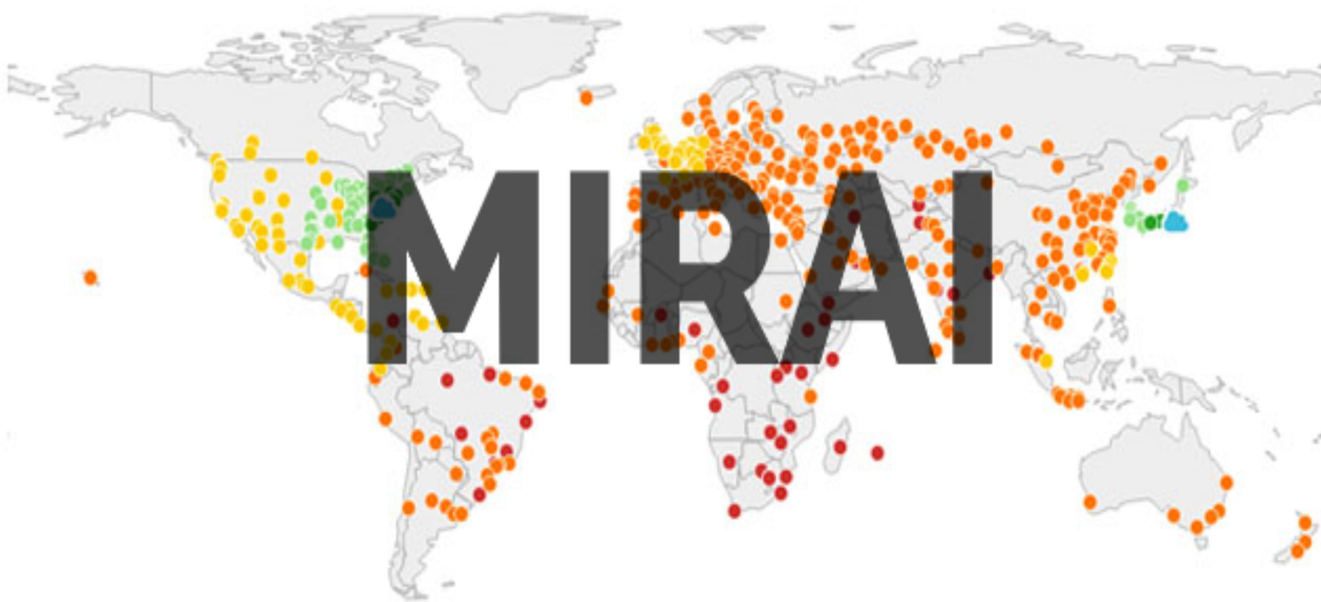
- Quantity has a quality all its own

- Vulnerability analysis
  - binary blobs (binary only, no OS or library abstractions)
  - software deeply connected with hardware

- Patch management
  - devices must be cheap
  - vendors might be long gone

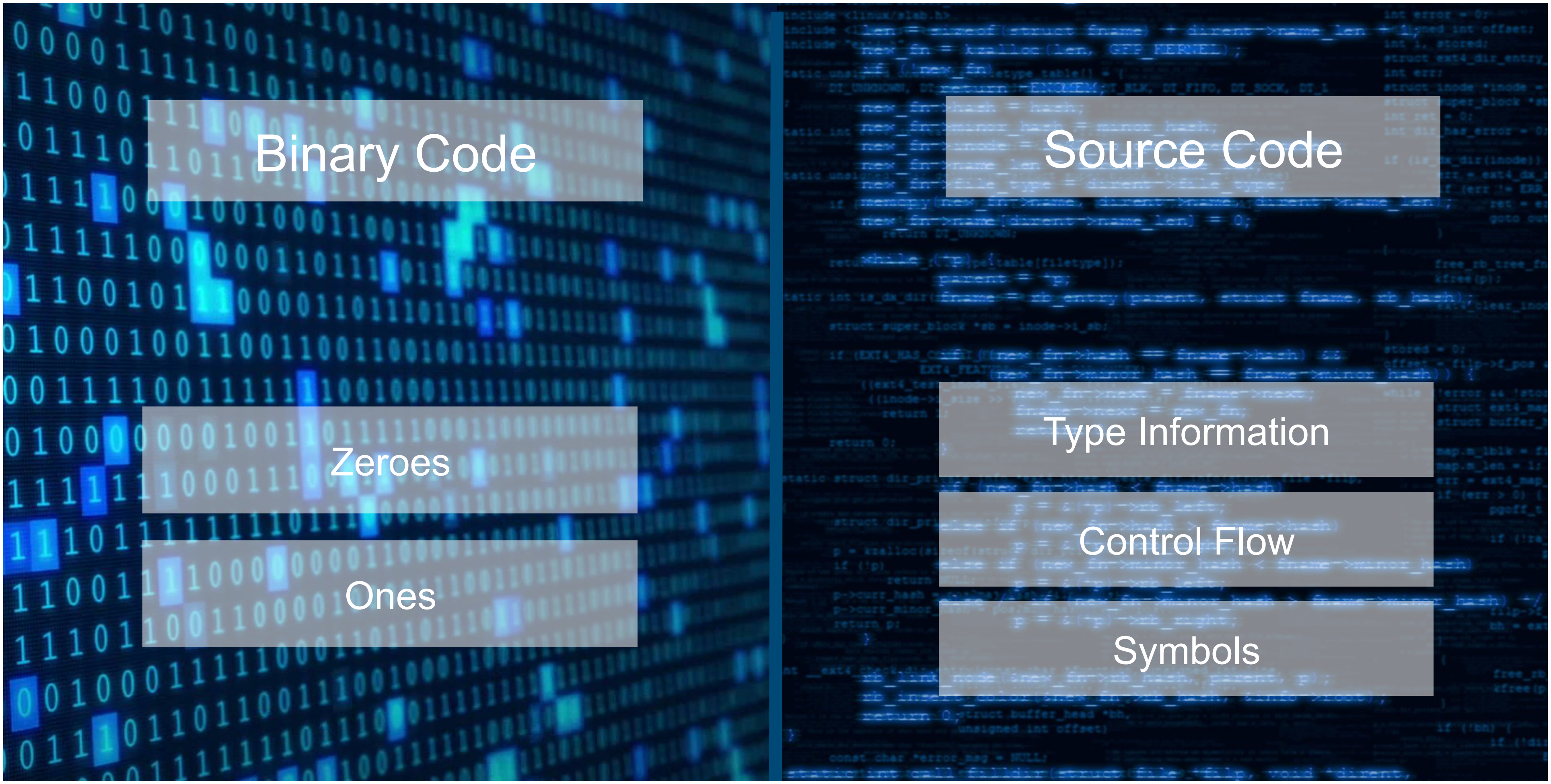
Security Challenges

- Remote accessibility
  - device authentication
  - access control (pacemaker during emergency)
  - stepping stone into inside of perimeter
- Additional vulnerability surface
  - attacks launched from physical world
  - supply chain attacks
- Getting access to the firmware


MIRAI

BINARY VULNERABILITY ANALYSIS

Binary Analysis



Binary Analysis

- Binary code is the worst-case, common denominator scenario

Symbolic Execution

"How do I trigger path X or condition Y?"

- Dynamic analysis
  - Input A? No. Input B? No. Input C? …
  - Based on concrete inputs to application
- (Concrete) static analysis
  - "You can't" / "You might be able to"
  - based on various static techniques
- We need something slightly different

Symbolic Execution

"How do I trigger path X or condition Y?"

- Interpret the application, keeping input values abstract (symbolic)
- Track "constraints" on variables
- When a condition is triggered, "concretize" to obtain a possible input

Symbolic Execution - Example


```
x = int(input())
if x >= 10:
  if x < 100:
    vulnerable_code()
  else:
    func_a()
else:
  func_b()
```

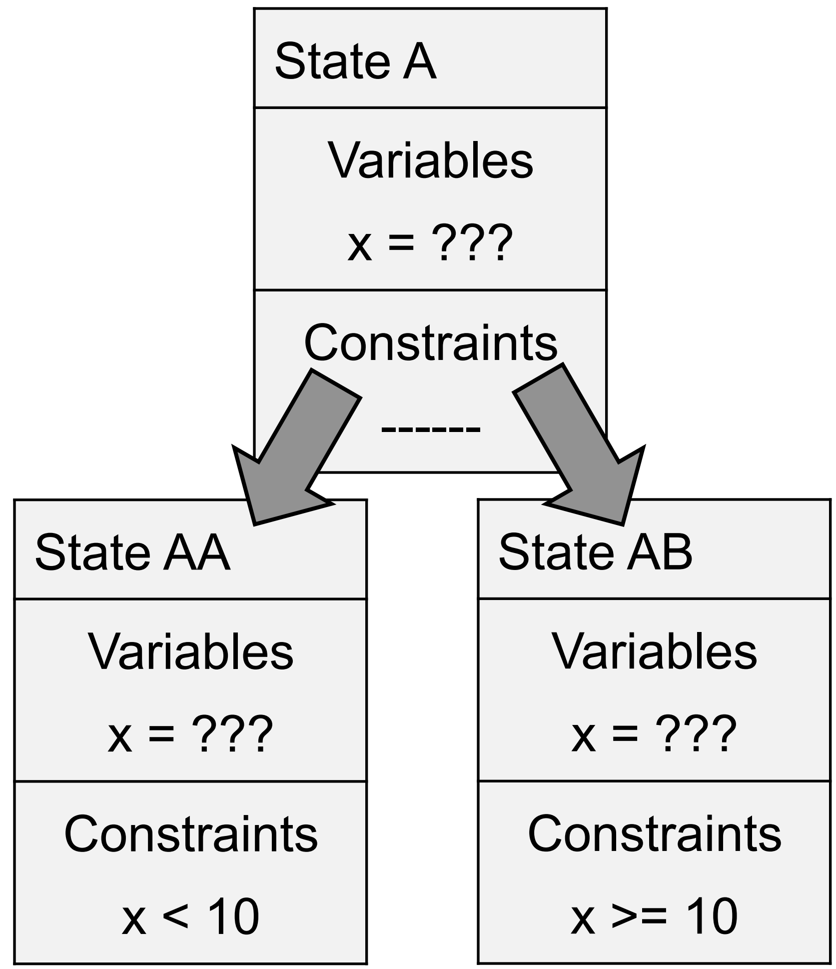
Symbolic Execution - Example


```
x = int(input())
if x >= 10:
  if x < 100:
    vulnerable_code()
  else:
    func_a()
else:
  func_b()
```

State A
Variables x = ???
Constraints ------

Symbolic Execution - Example


```
x = int(input())
if x >= 10:
  if x < 100:
    vulnerable_code()
  else:
    func_a()
else:
  func_b()
```



Symbolic Execution - Example

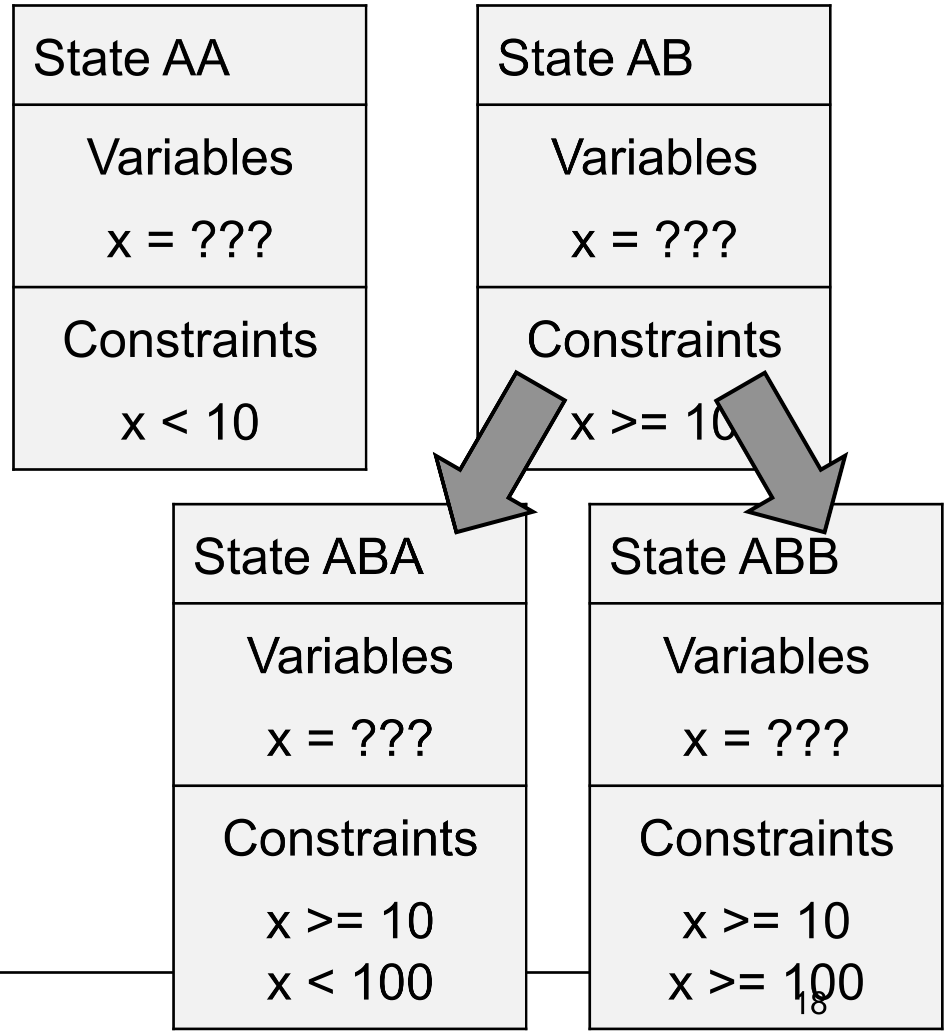

```
x = int(input())
if x >= 10:
  if x < 100:
    vulnerable_code()
  else:
    func_a()
else:
  func_b()
```

State AA
Variables x = ???
Constraints x < 10

State AB
Variables x = ???
Constraints x >= 10

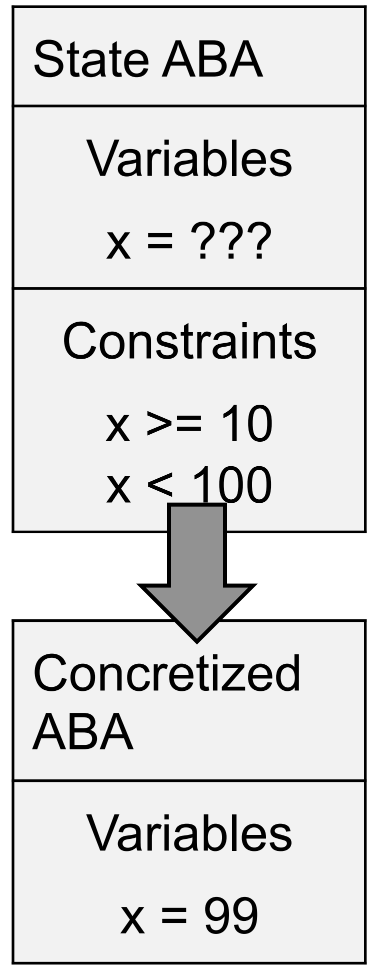
Symbolic Execution - Example


```
x = int(input())
if x >= 10:
  if x < 100:
    vulnerable_code()
  else:
    func_a()
else:
  func_b()
```



Symbolic Execution - Example


```
x = int(input())
if x >= 10:
  if x < 100:
    vulnerable_code()
  else:
    func_a()
else:
  func_b()
```



Symbolic Execution - Pros and Cons

Pros

- Precise
- No false positives
  - with correct environment model
- Produces directly-actionable inputs

Cons

- Not easily scalable
  - constraint solving is NP-complete
  - state and path explosion

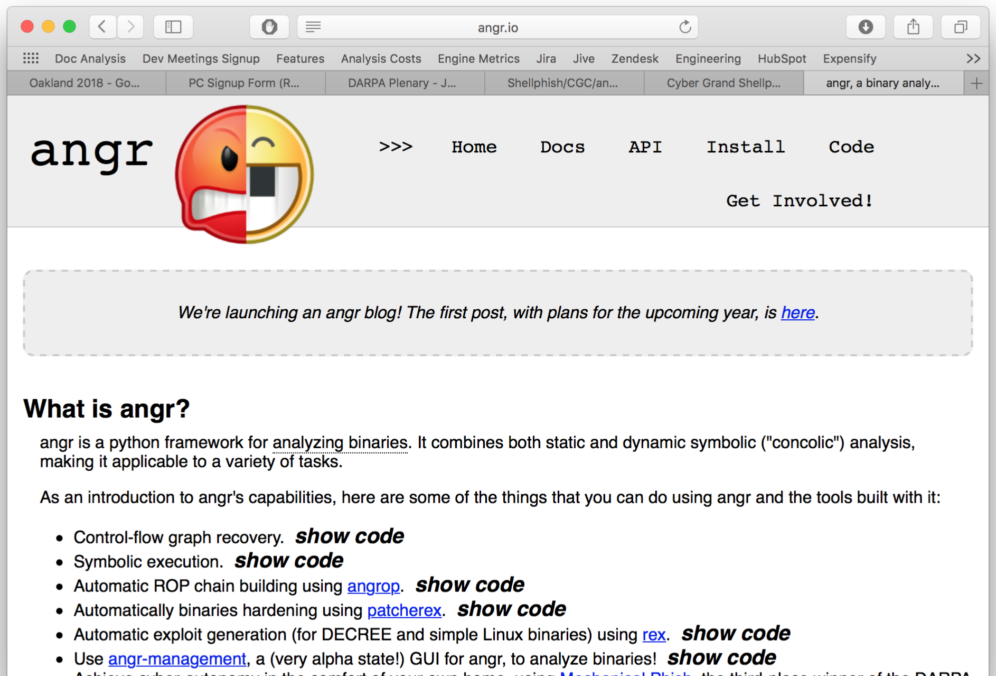
angr

Framework for the analysis of binaries, developed at UCSB

angr Components

- angr
  - Binary Loader
  - Static Analysis Routines
    - Control-Flow Graph
    - Data-Flow Analysis
    - Value-Set Analysis
  - Symbolic Execution Engine
    - Forward Symbolic Execution
    - Under-constrained SE

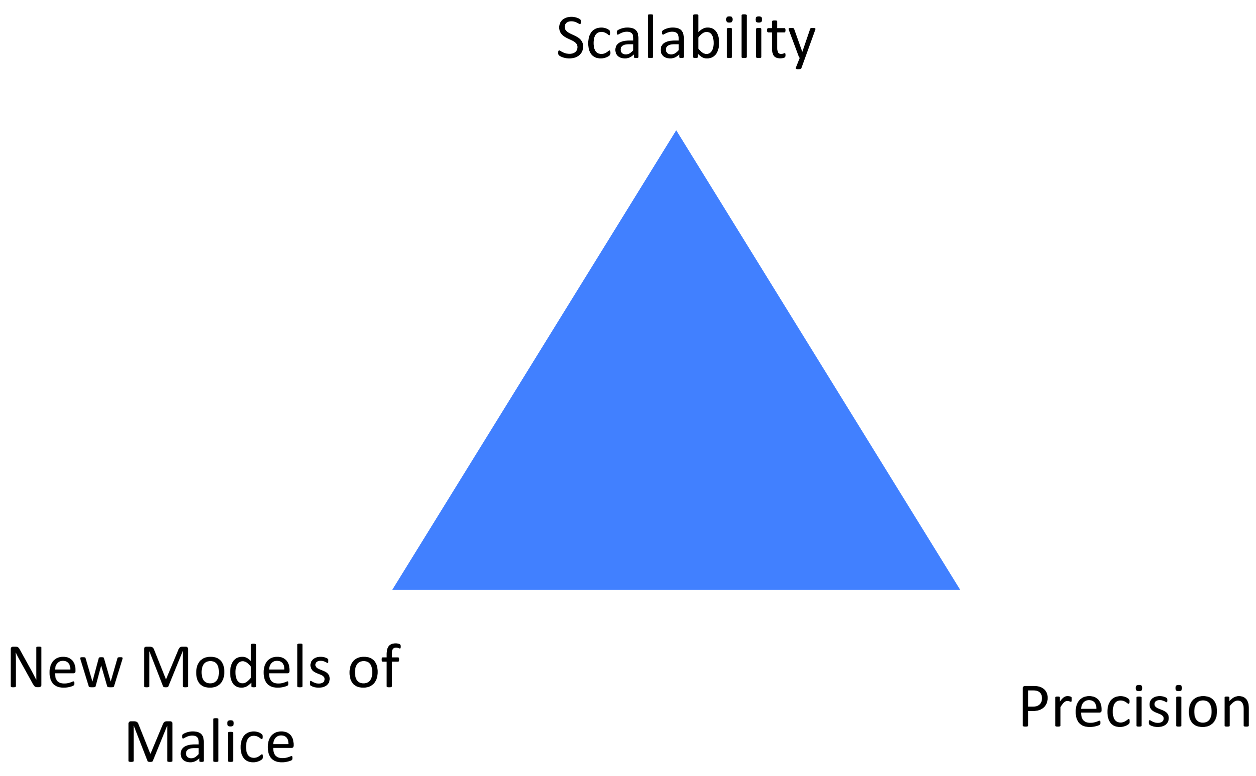
angr Platform



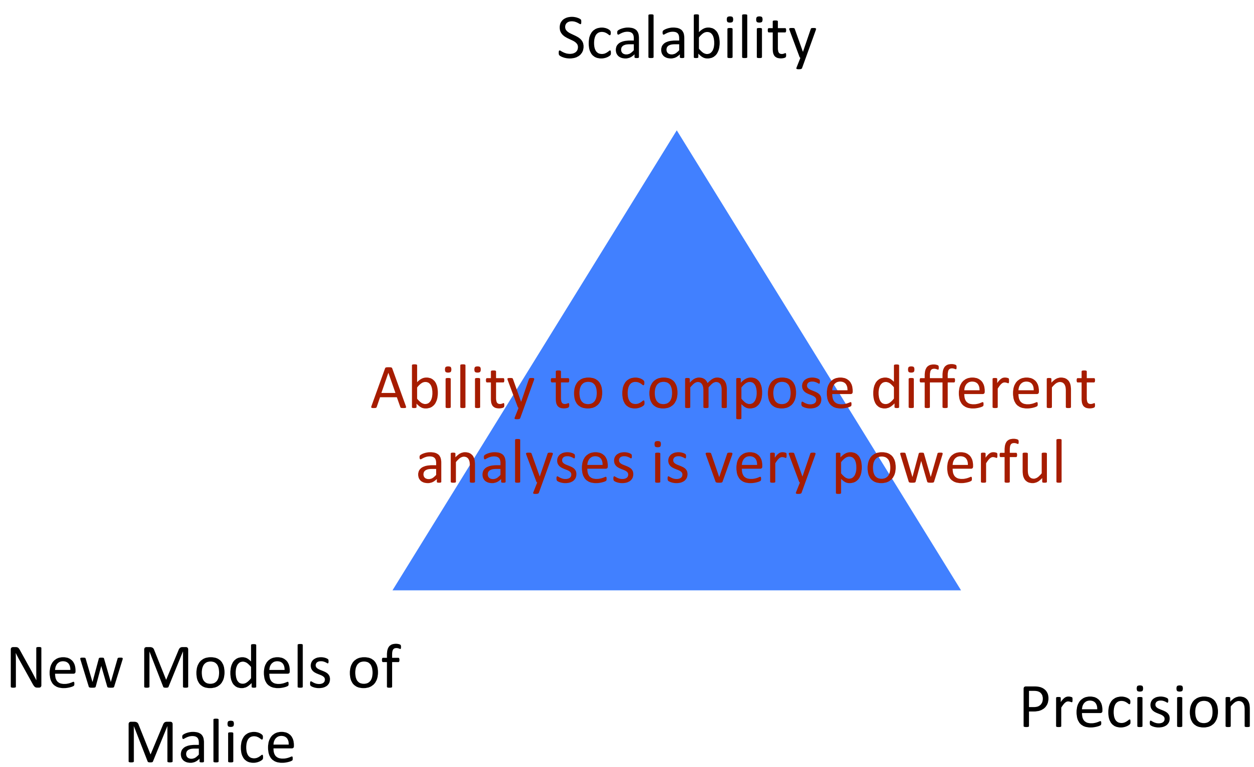
<u>Open Source Analysis Platform</u>

- More than 100 KLOC
- More than 10K commits
- More than 30K downloads in 2017
- 1,600+ stars on Github
- Users in industry, academia, government

angr - Challenges and Goals

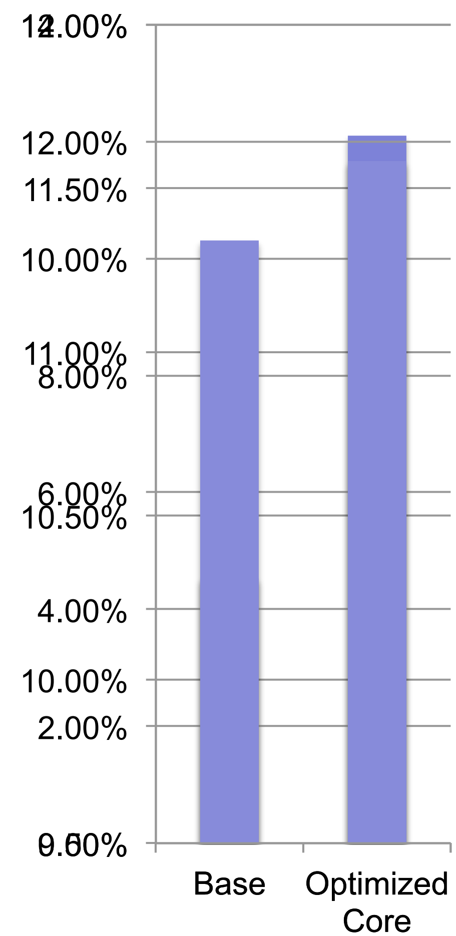


angr - Challenges and Goals



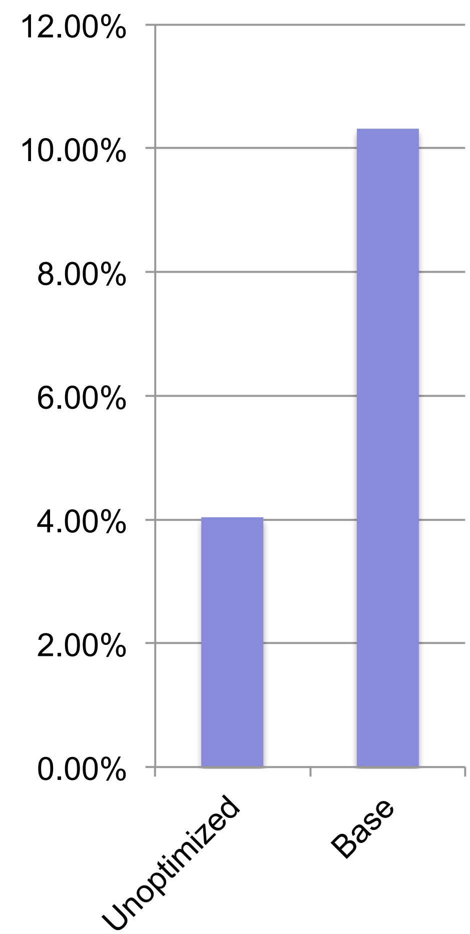
Symbolic Execution Improvements

- Fastpath and adaptive concretization
  - when possible, analyze parts of code non symbolically
- Peephole optimization
  - replace code snippets that blow up symbolic execution
- Lazy constraint solving
  - sometimes, waiting to add more constraints makes solving easier



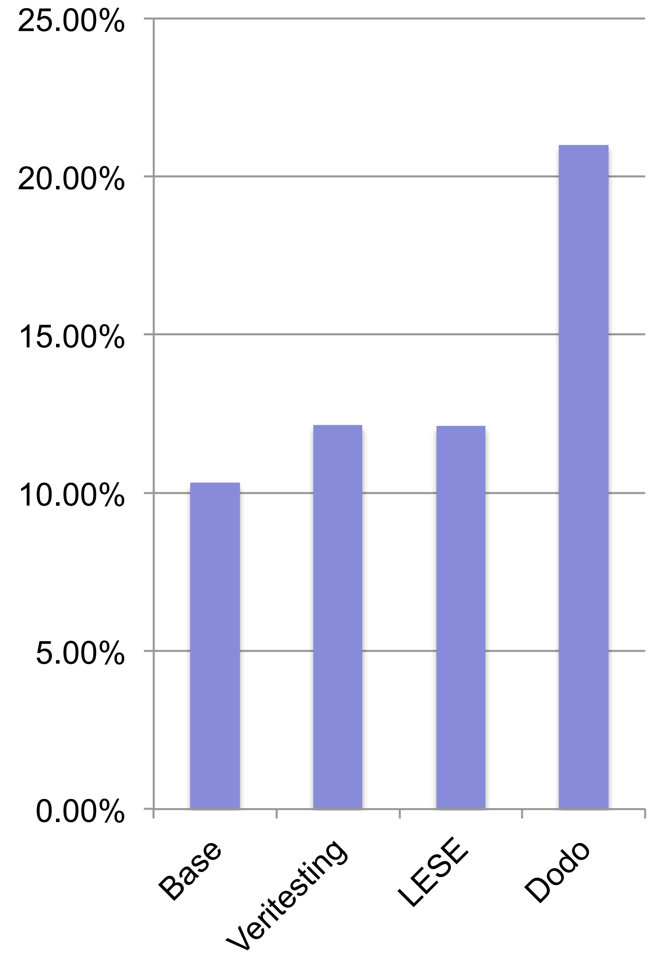
Constraint Solver Optimizations

- Solution caching
  - don't run solver on same constraints multiple times
- Constraint subset management
  - break up hard constraints into subparts and solve separately
- Expression simplification
  - before submitting constraints, simplify
- Expression rewriting



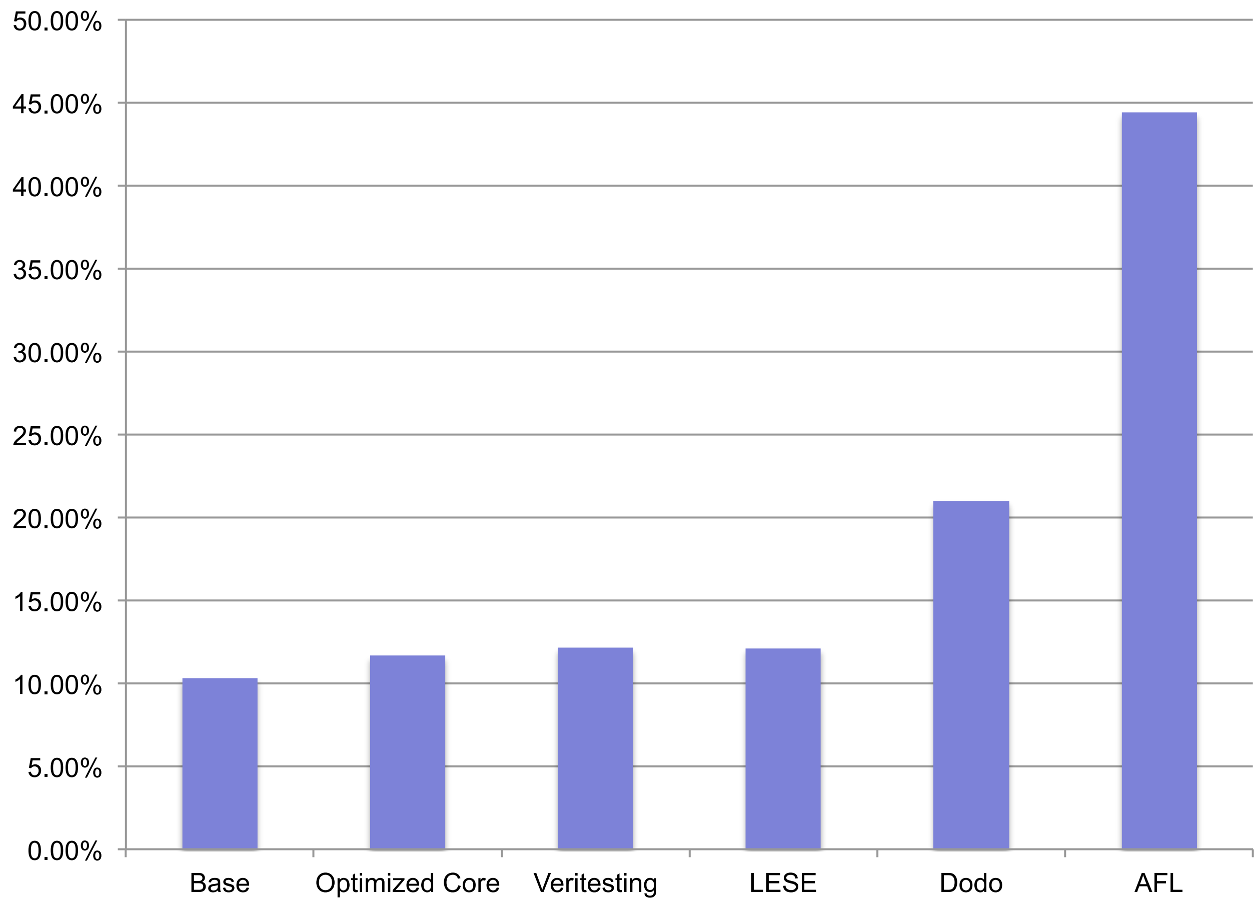
Static Analysis Support

- Veritesting
  - SSE to merge over multiple paths
- LESE - loop extended sym exec
  - intelligent loop unrolling
- Code summarization (Dodo)
  - automatically (and statically) summarize effect of loops / functions
- VSA - value set analysis
  - resolve ranges (and conditionals) without solving constraints



American Fuzzy Lop (AFL)

American Fuzzy Lop (AFL)



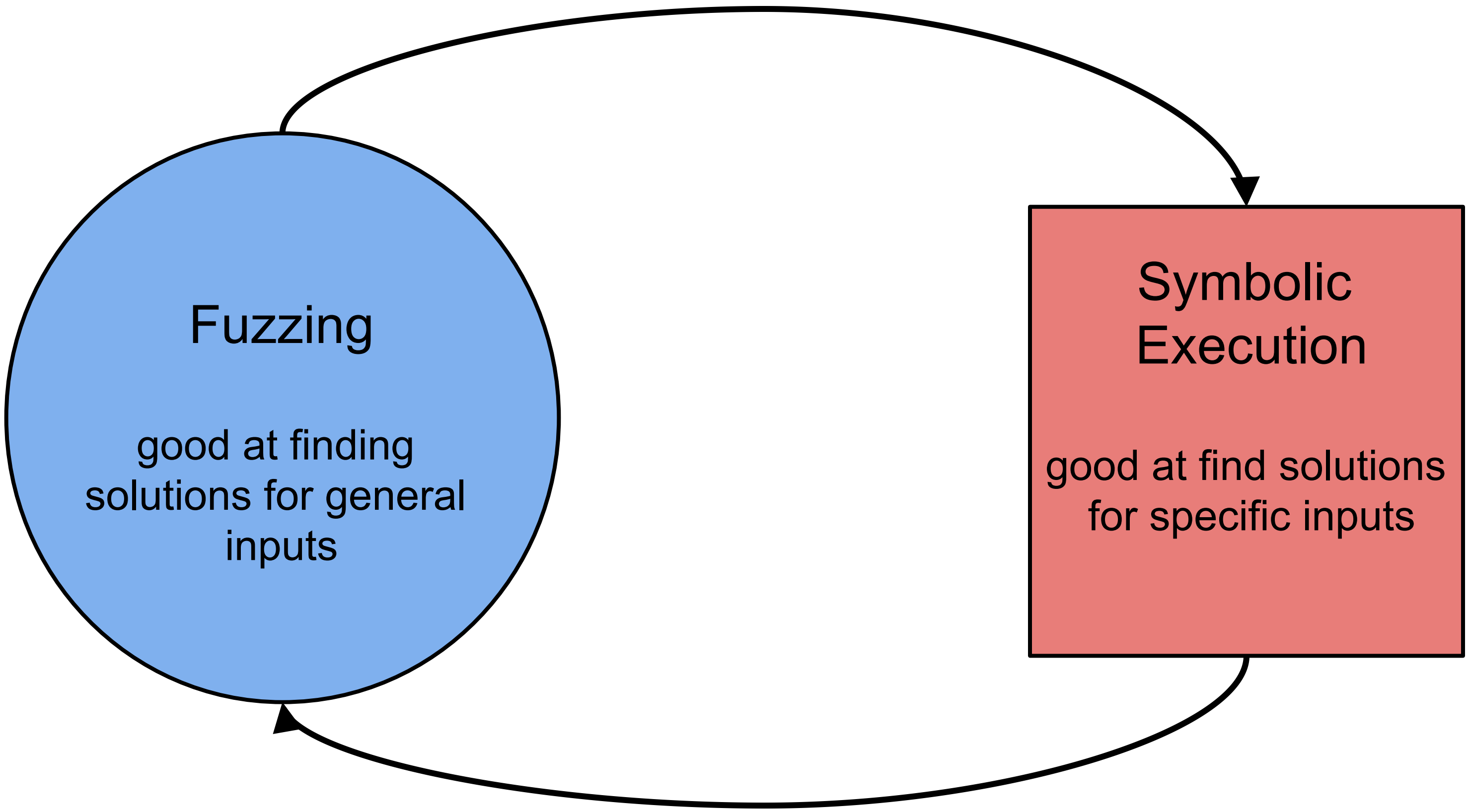
Combining Approaches

- angr can be used in combination with other tools

- Fuzzing excels at producing general inputs
- Symbolic execution is able to satisfy complex path predicates for specific inputs

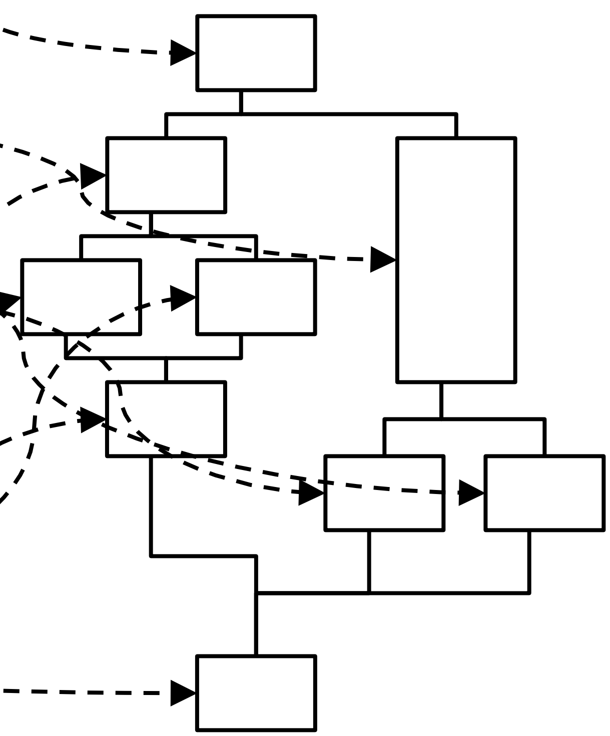
- Key Insight
  - combine both techniques to leverage their strengths and mitigate their weaknesses

Driller = AFL + angr



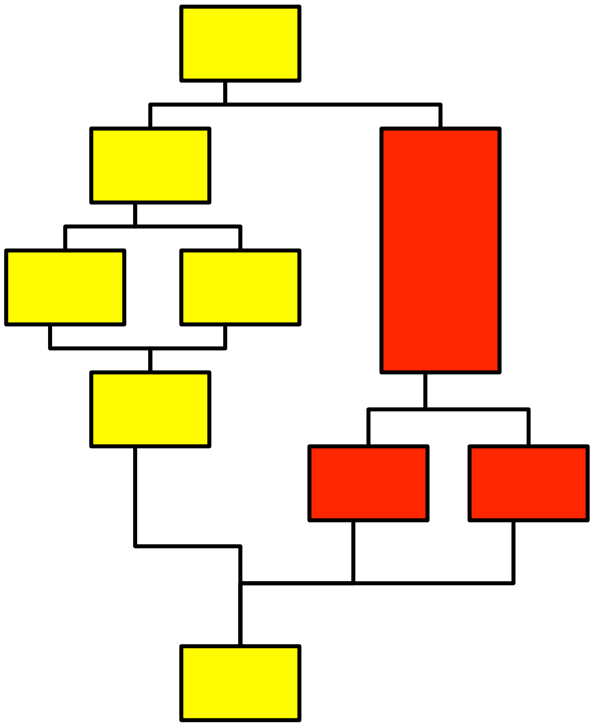
Driller Example


```
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C”:
        crash()
    else:
        print "Unknown command”
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!”
    else:
        print "Auth failure!”
    print "Try again..."
return
```

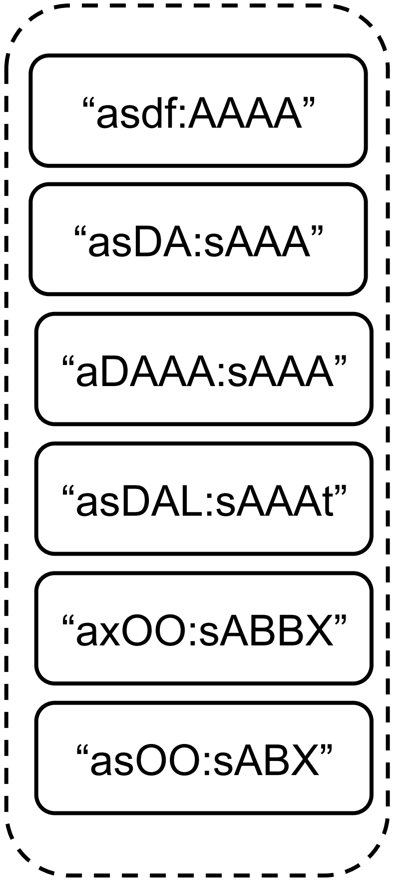


Driller Example


```
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C":
        crash()
    else:
        print "Unknown command"
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!"
    else:
        print "Auth failure!"
    print "Try again..."
return
```

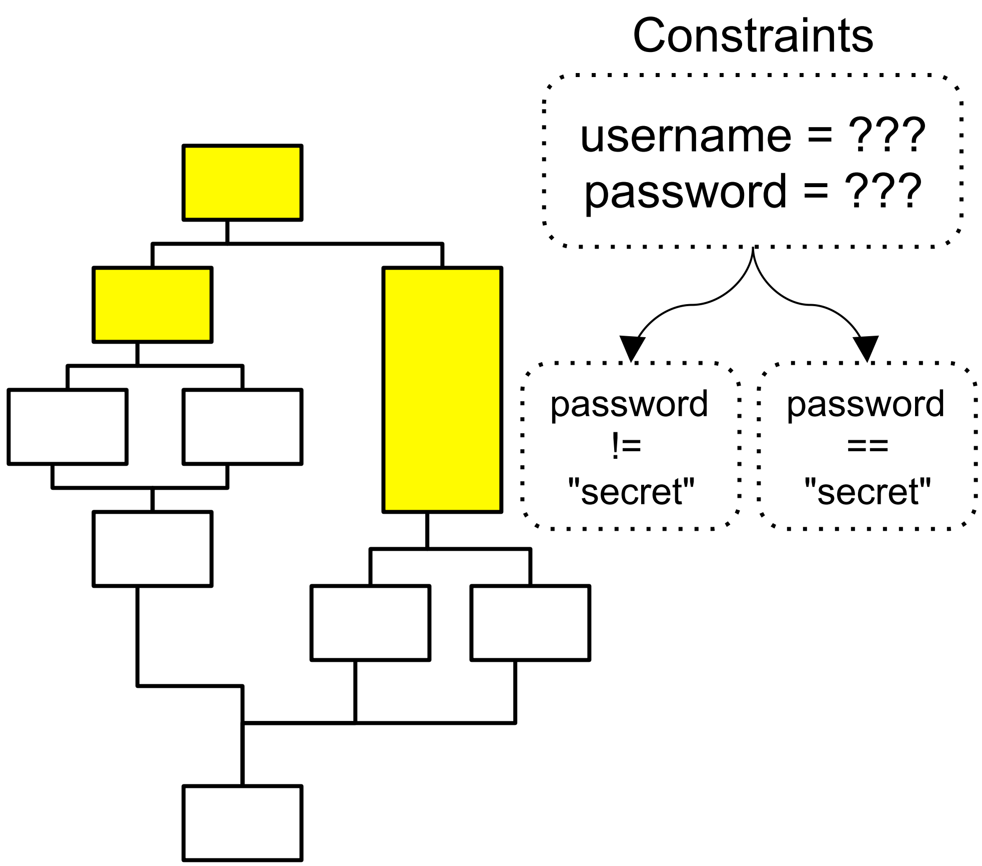


Test Cases



Driller Example

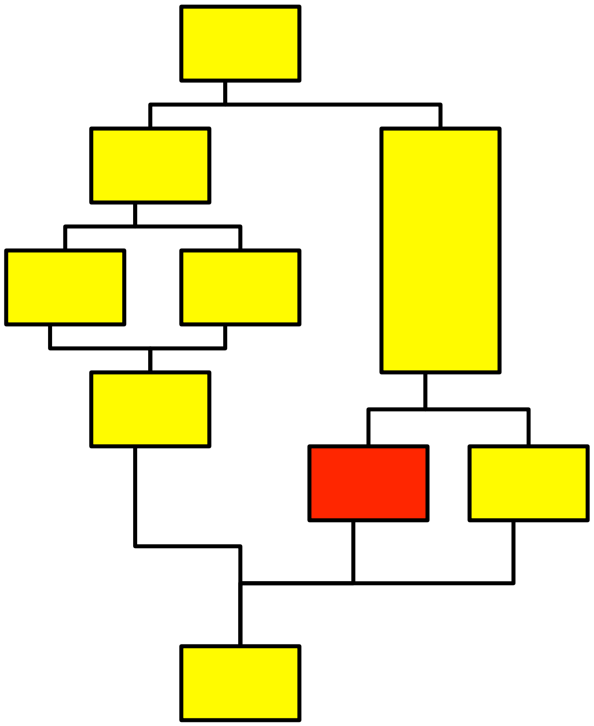

```
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C”:
        crash()
    else:
        print "Unknown command”
else:
    complex_function()
    if len(username) < 5:
       print "Invalid username!”
    else:
        print "Auth failure!”
    print "Try again..."
return
```



Driller Example

UC Santa Barbara

```
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C”:
        crash()
    else:
        print "Unknown command”
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!”
    else:
        print "Auth failure!”
    print "Try again..."
return
```

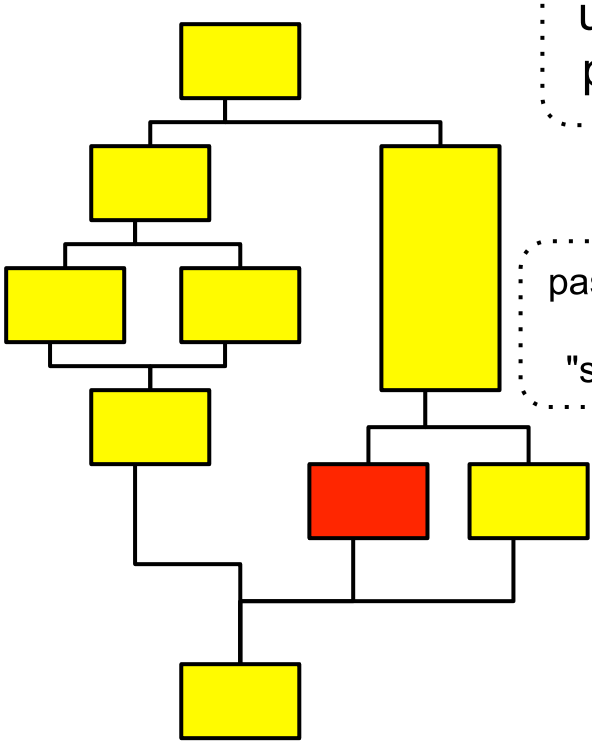


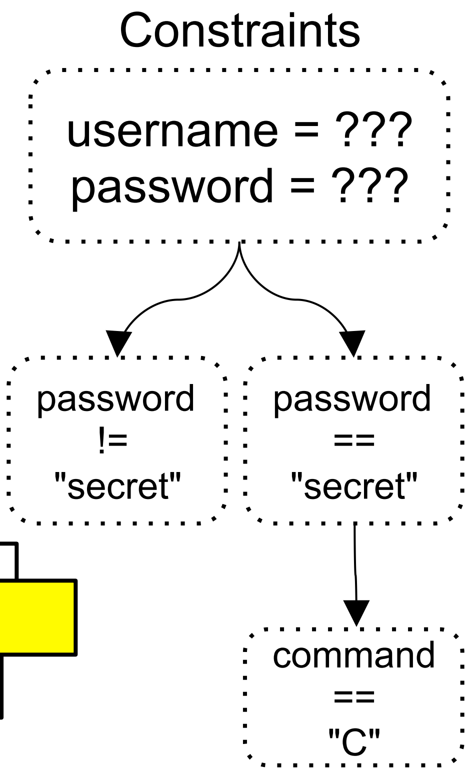
Test Cases

- “asdf:secret”
- “asdf:ljafe”
- “aDAA:secret”
- “aaDAA:etsf”

Driller Example


```
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C”:
        crash()
    else:
        print "Unknown command”
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!”
    else:
        print "Auth failure!”
    print "Try again..."
return
```

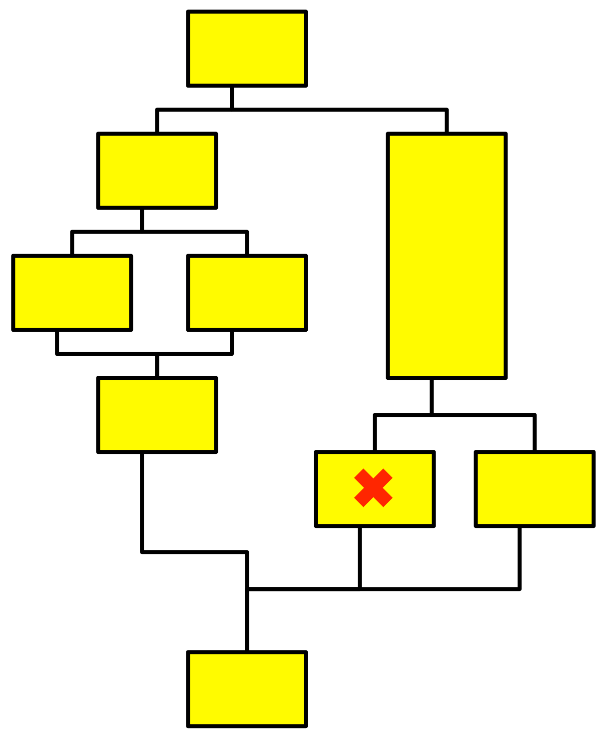




Driller Example

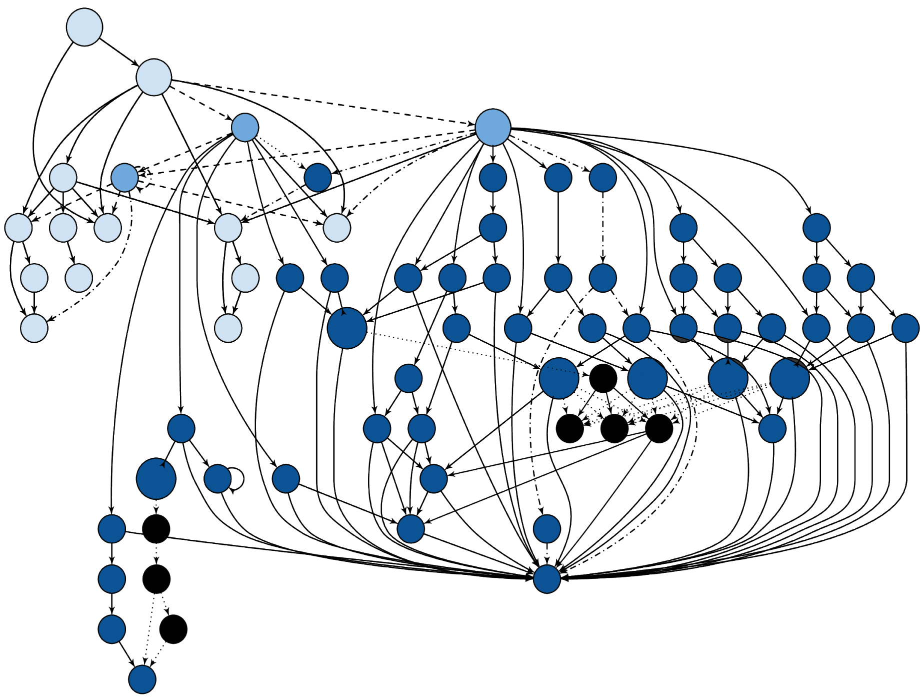
UC Santa Barbara

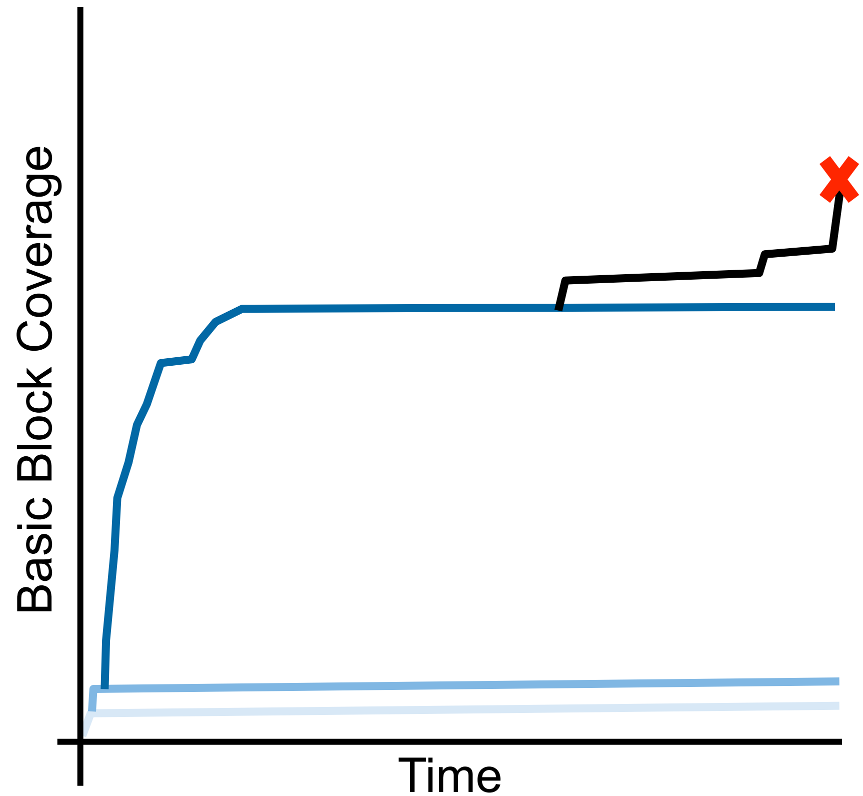
```
username = input()
password = input()
if password == "secret":
    complex_function()
    command = input()
    if command == "C”:
        crash()
    else:
        print "Unknown command”
else:
    complex_function()
    if len(username) < 5:
        print "Invalid username!”
    else:
        print "Auth failure!”
    print "Try again..."
return
```



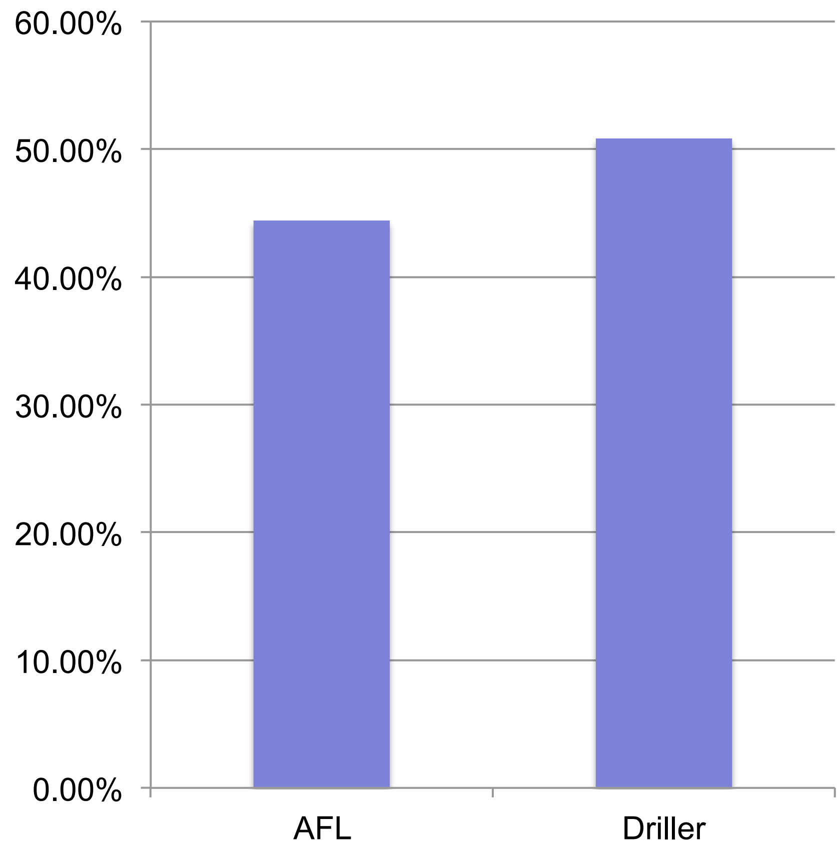
Impact of Driller

Applicability varies by program. Where it was needed, Driller increased block coverage by an average of 71%.





Impact of Driller



Failed Attempts (aka Future Research)

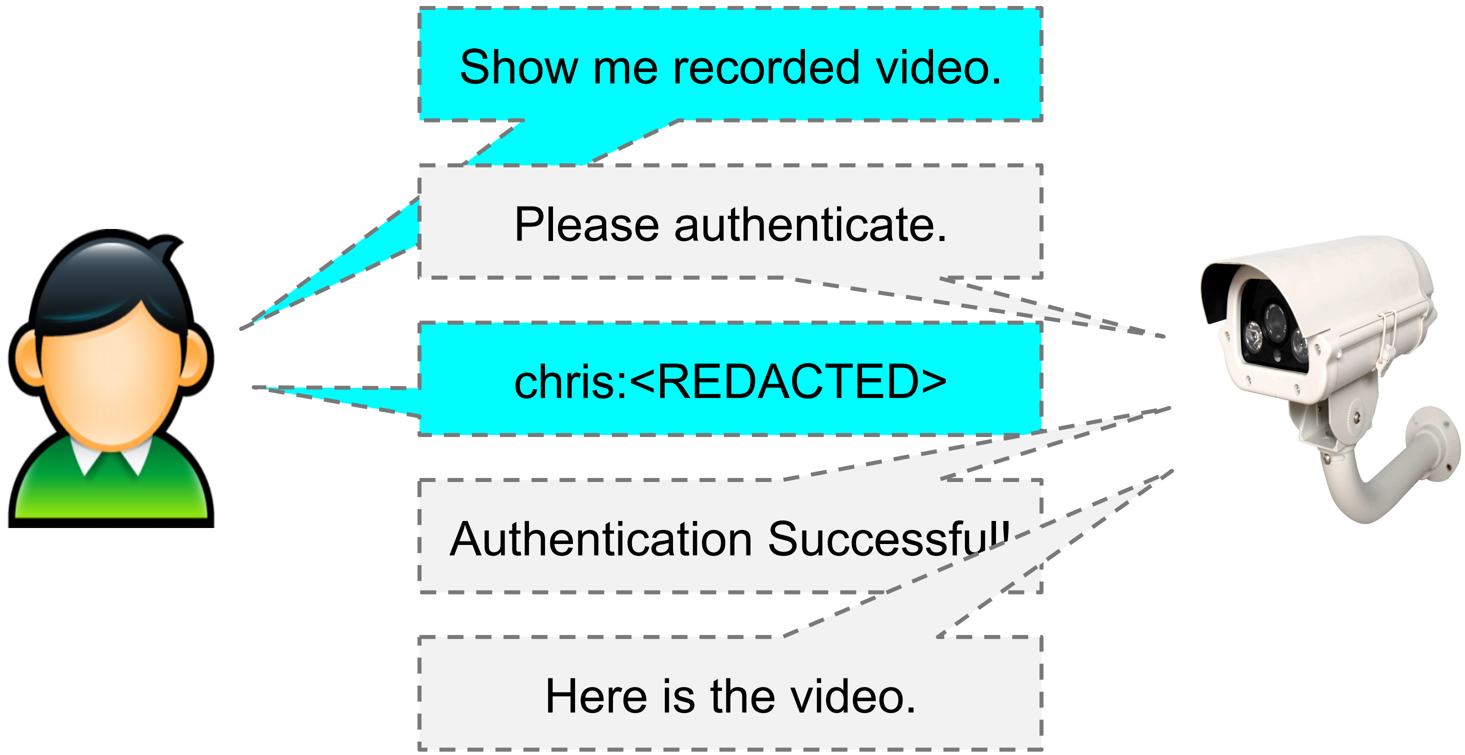
- State management
  - duplicate state detection
- Path selection to reach “promising” parts of the program
  - heuristics that guide analysis to areas that are more likely vulnerable

VULNERABILITY MODELS

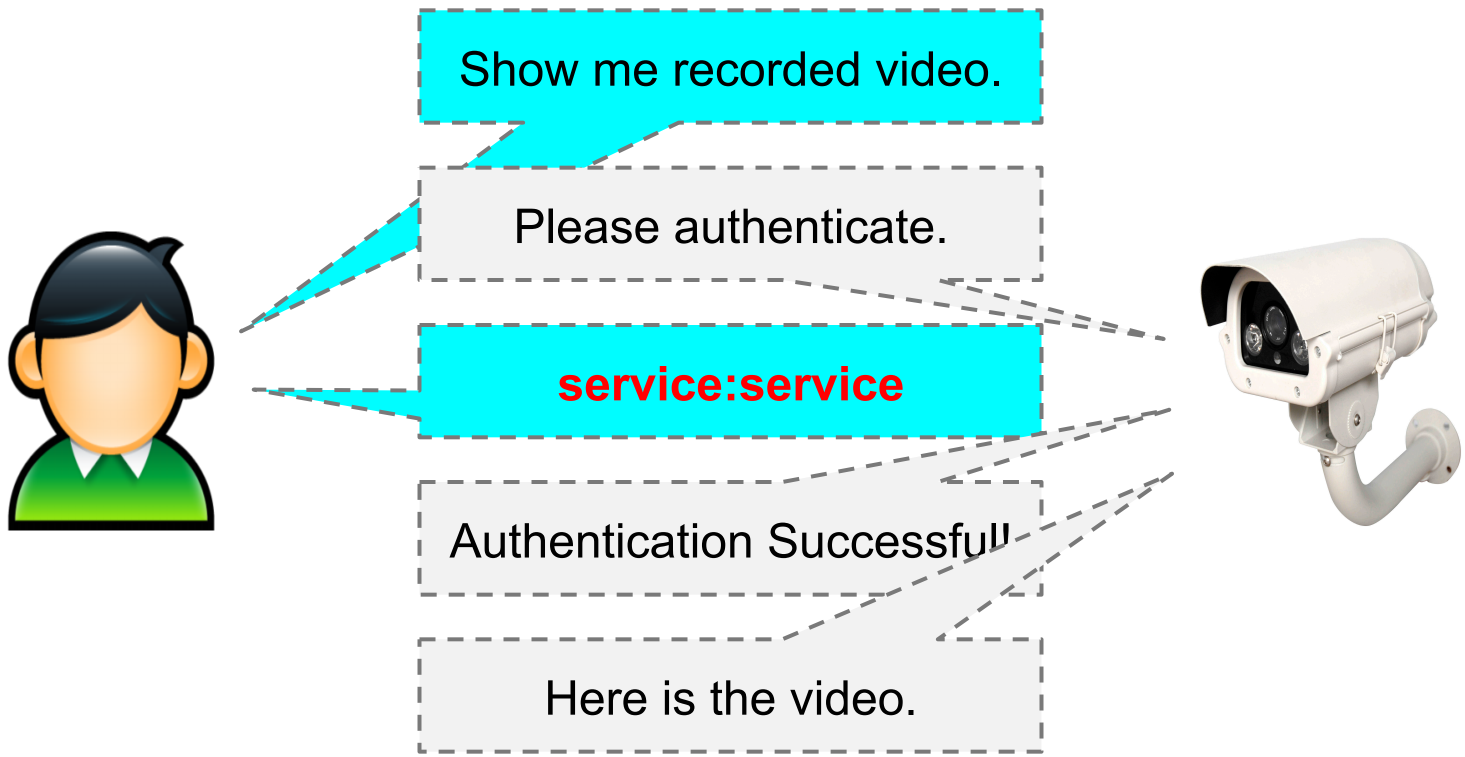
Interesting Vulnerabilities

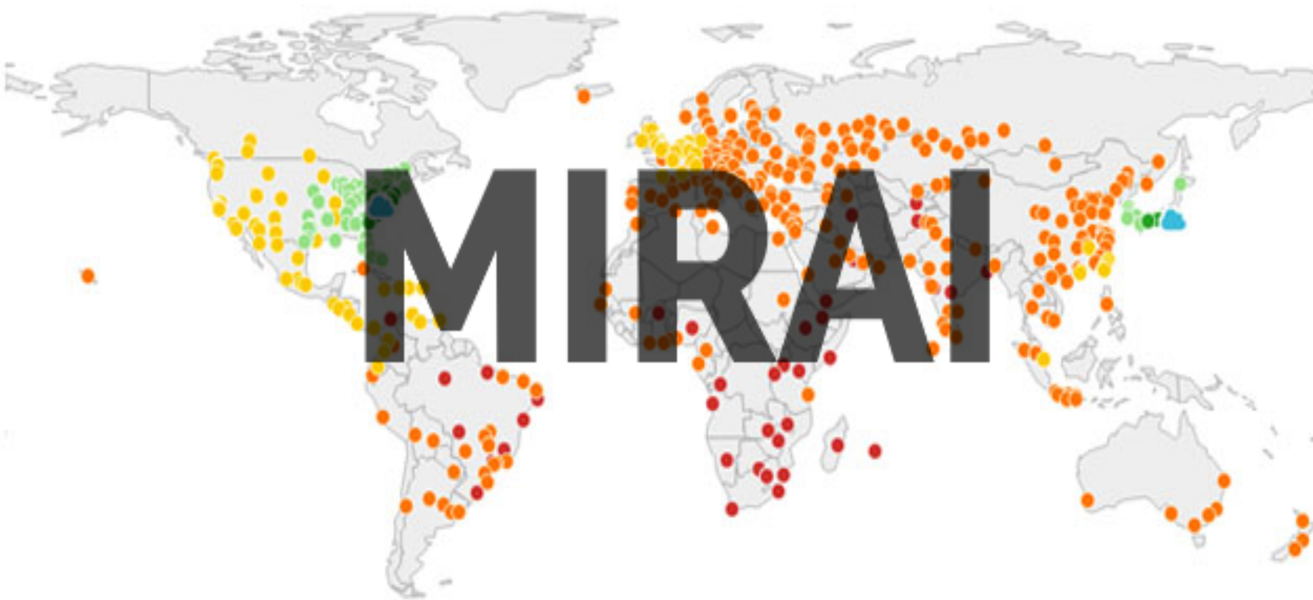
- Memory safety vulnerabilities
  - buffer overrun
  - out of bounds reads (heartbleed)
  - write-what-where
- Authentication bypass (backdoors)
- Actuator control

Authentication Bypass



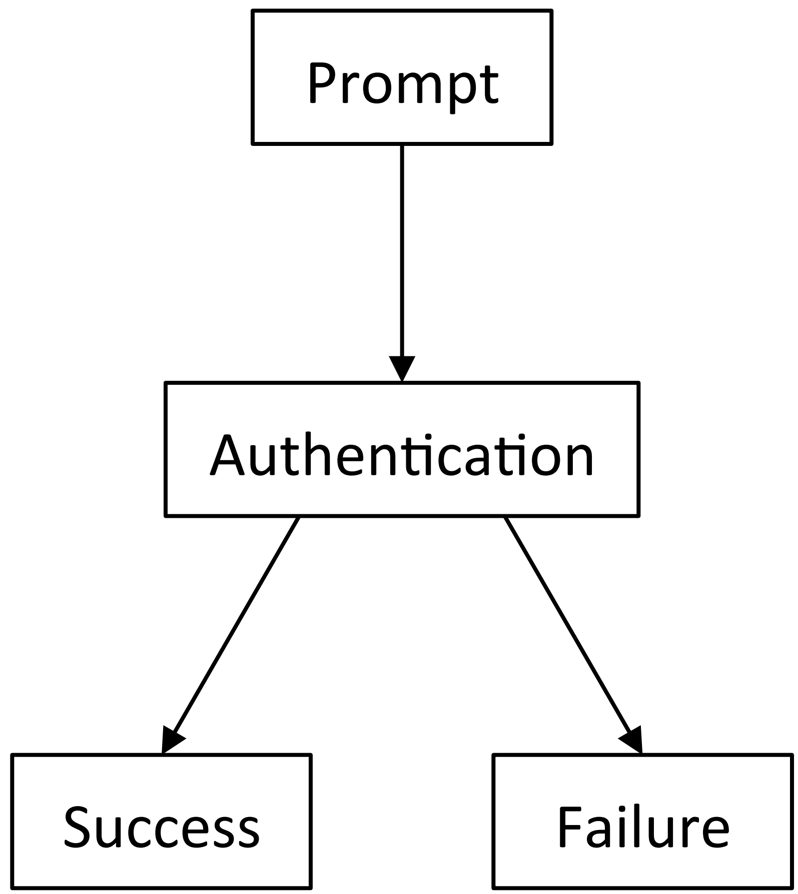
Authentication Bypass



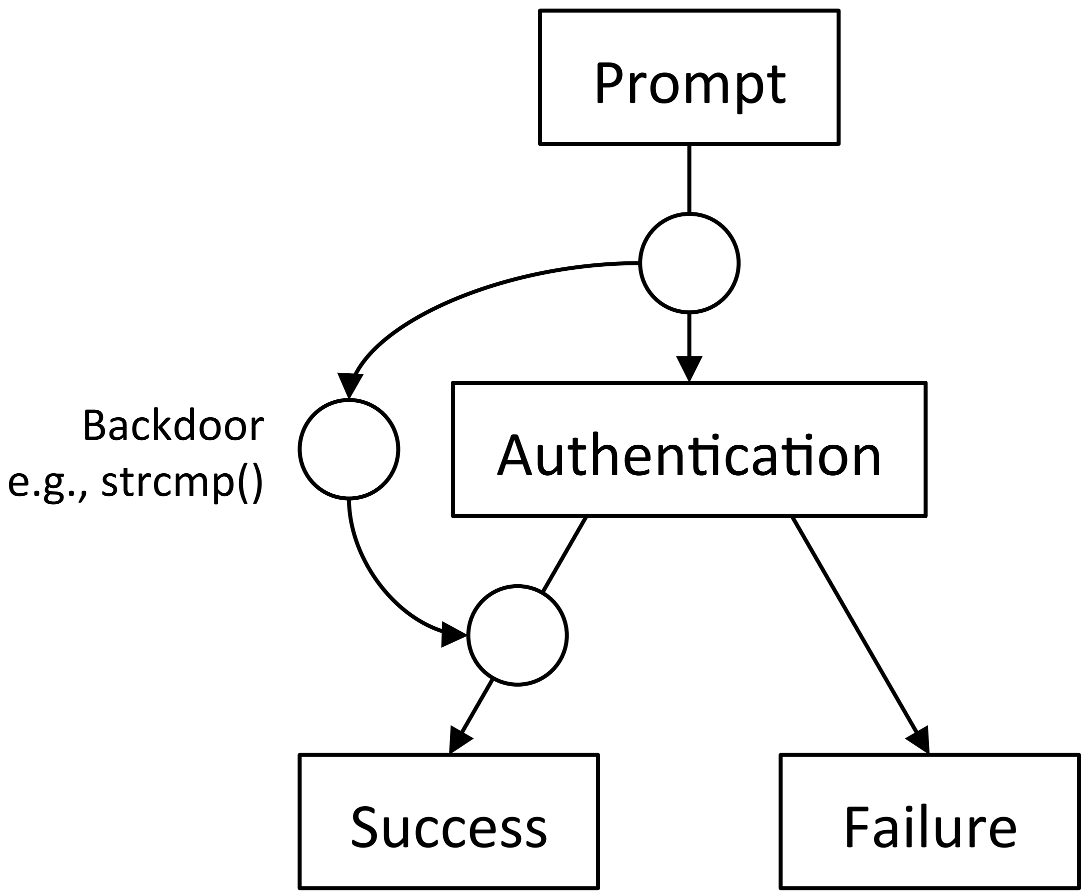


service:service

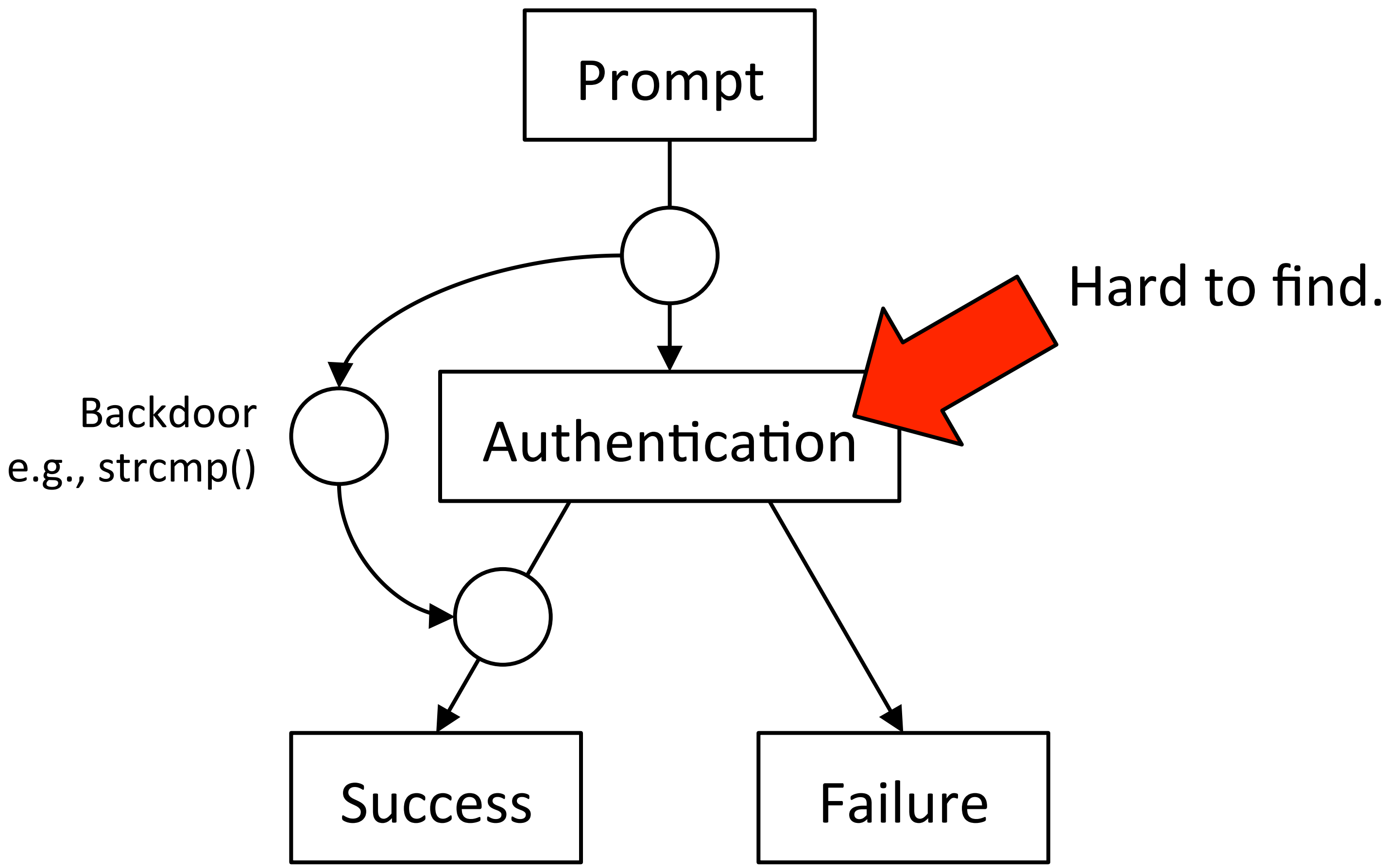
Authentication Bypass



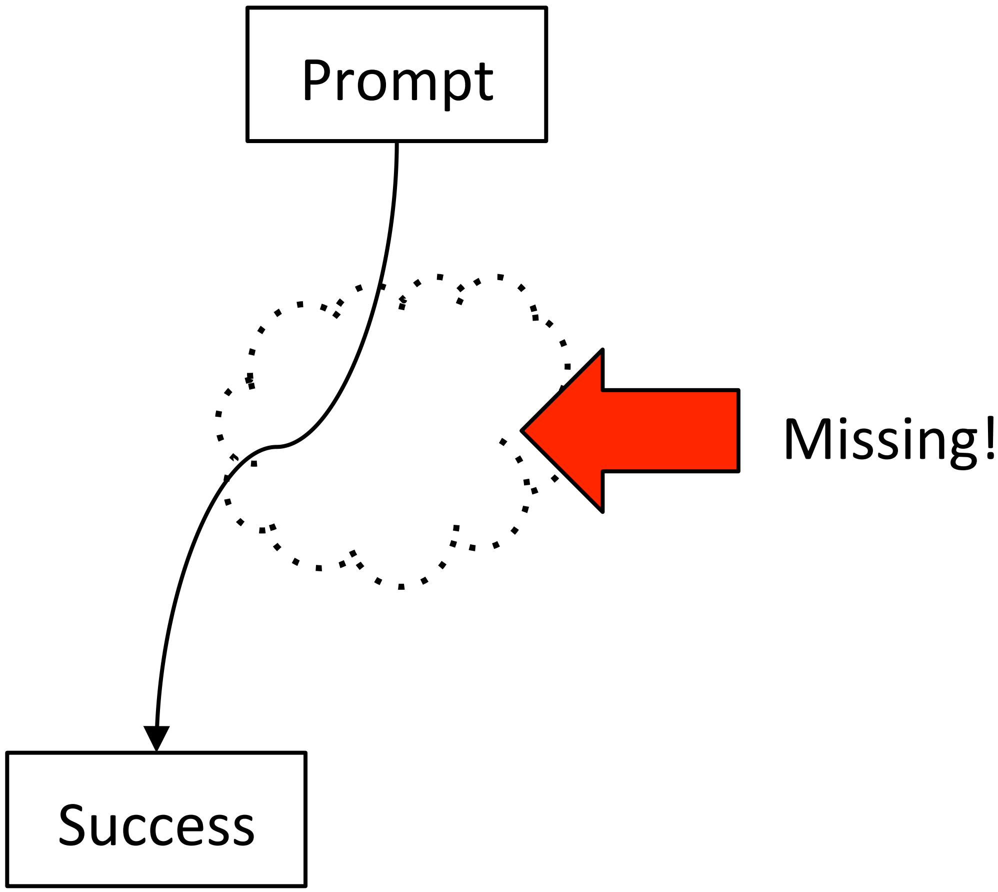
Authentication Bypass



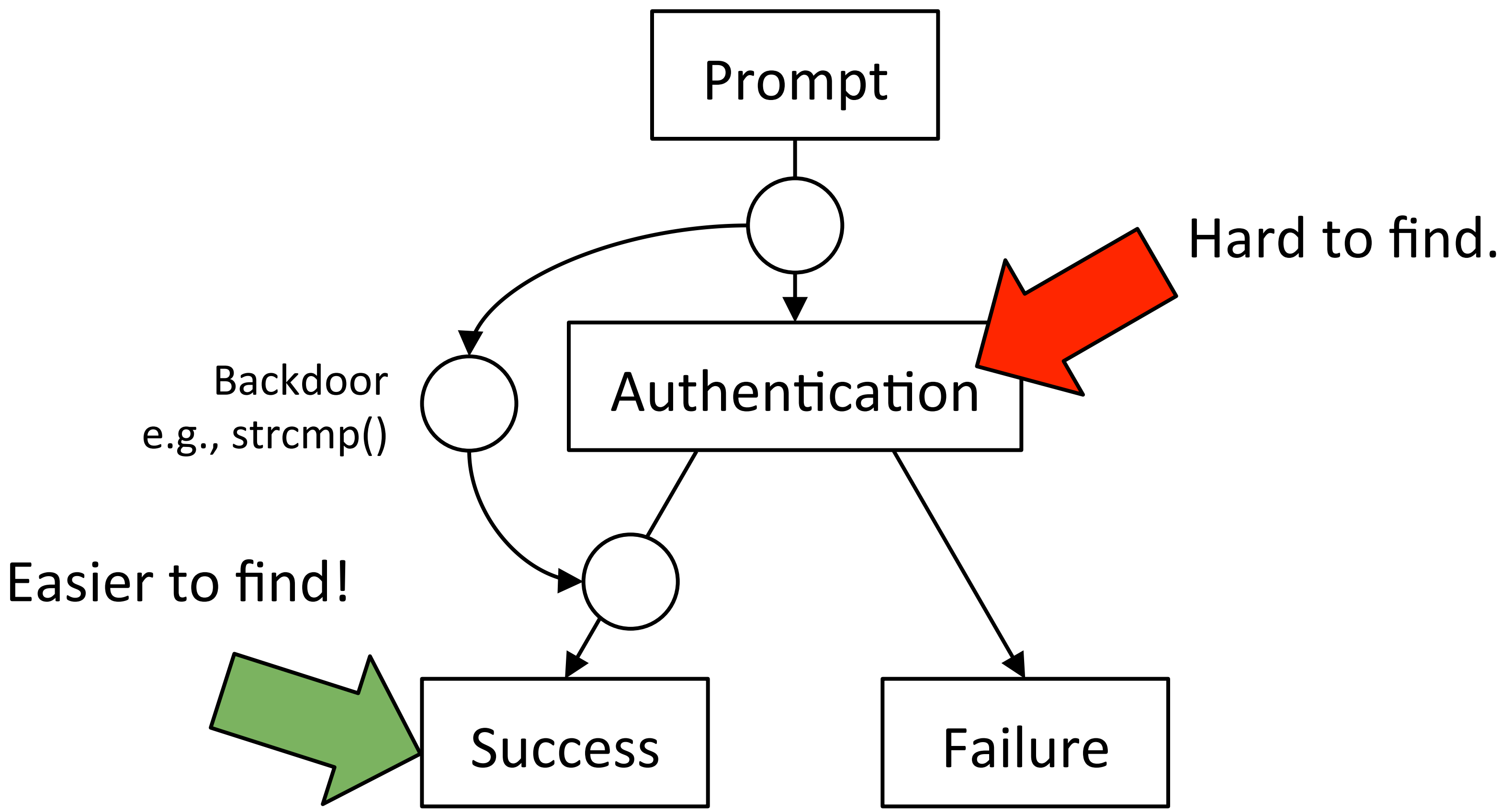
Authentication Bypass



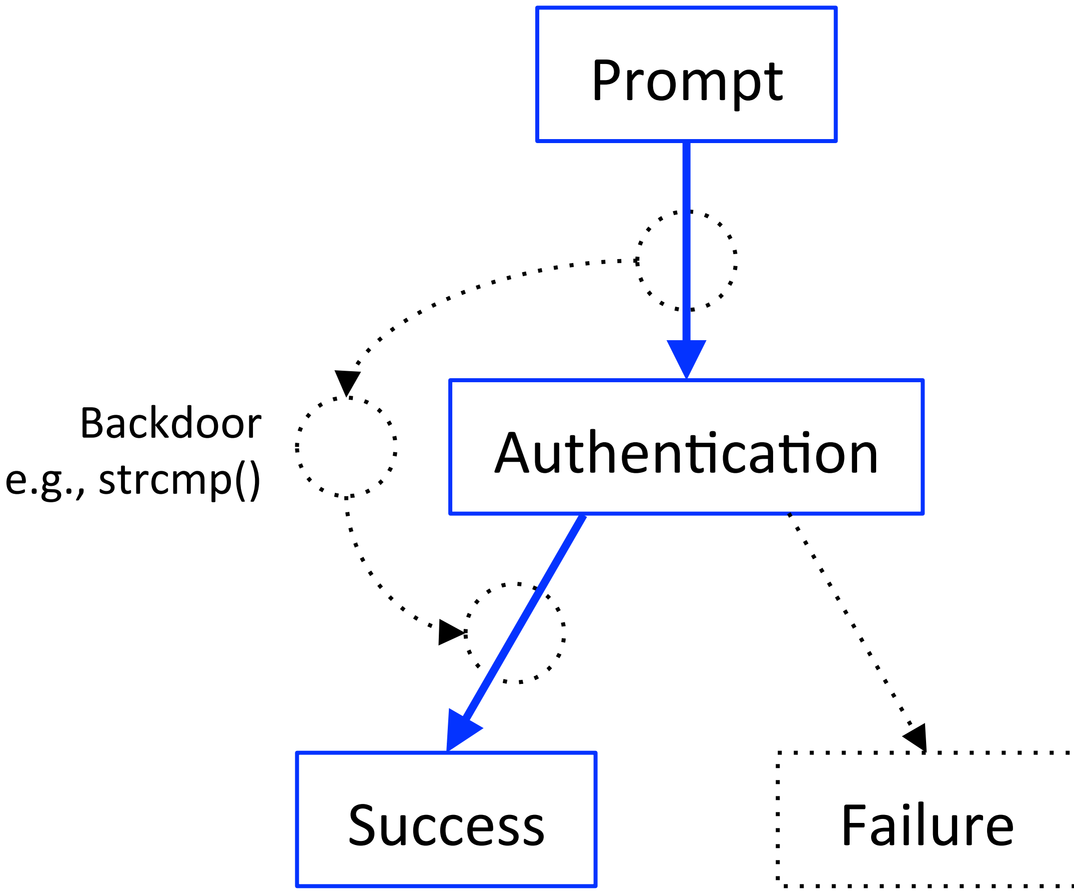
Authentication Bypass



Modeling Authentication Bypass



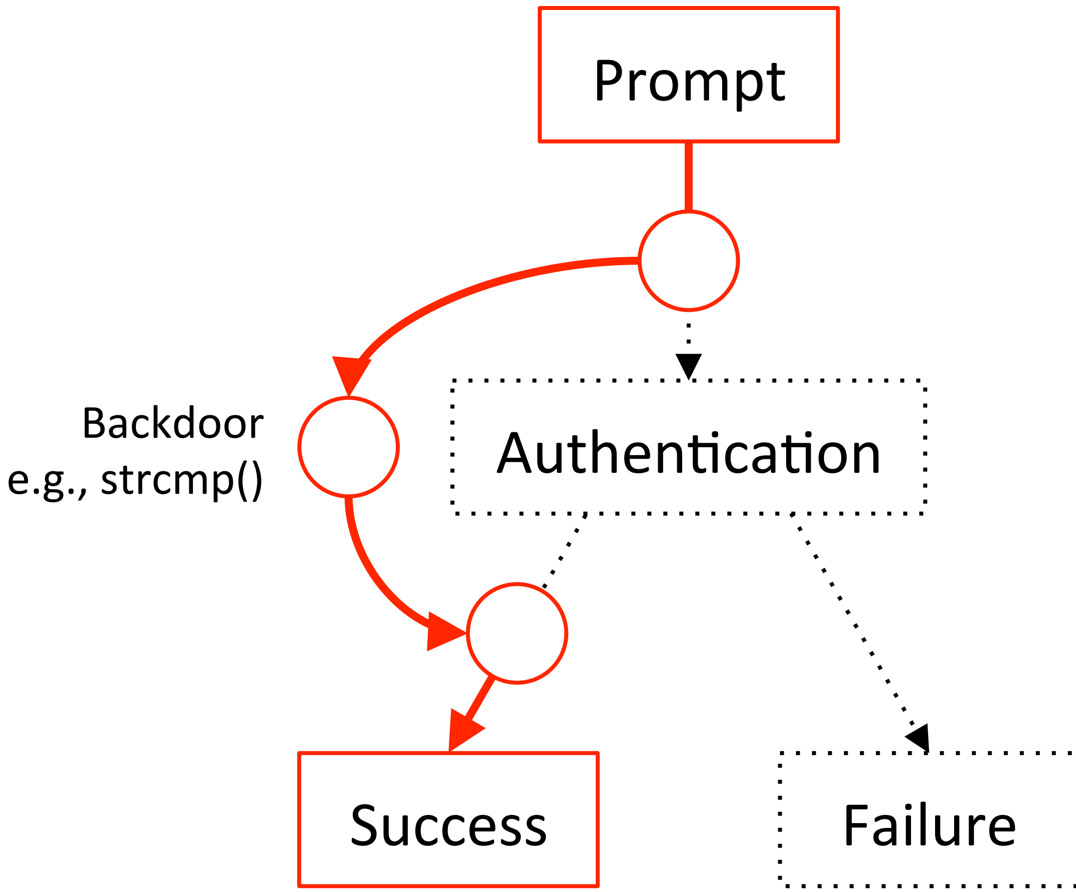
Input Determinism



Can we determine the input needed to reach the success function, just by analyzing the code?

The answer is NO

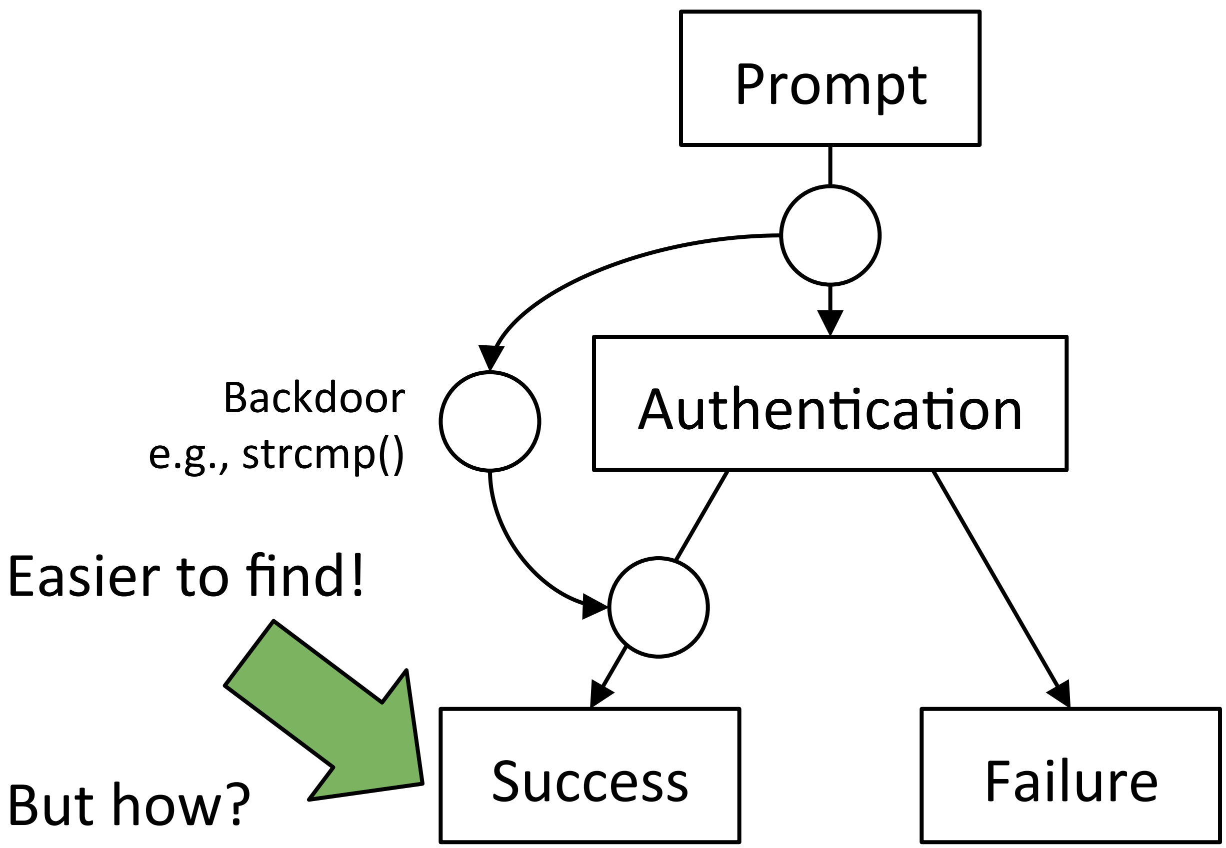
Input Determinism



Can we determine the input needed to reach the success function, just by analyzing the code?

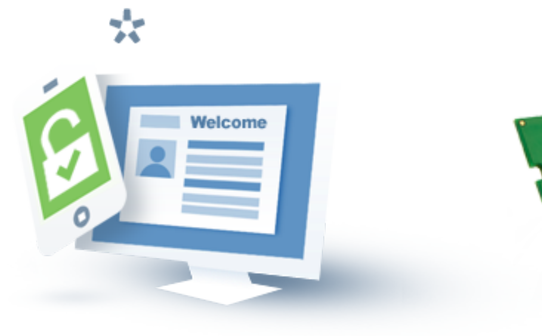
The answer is YES

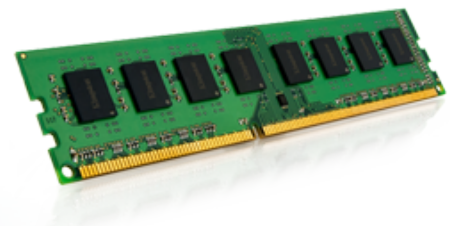
Modeling Authentication Bypass



Finding "Authenticated Point"

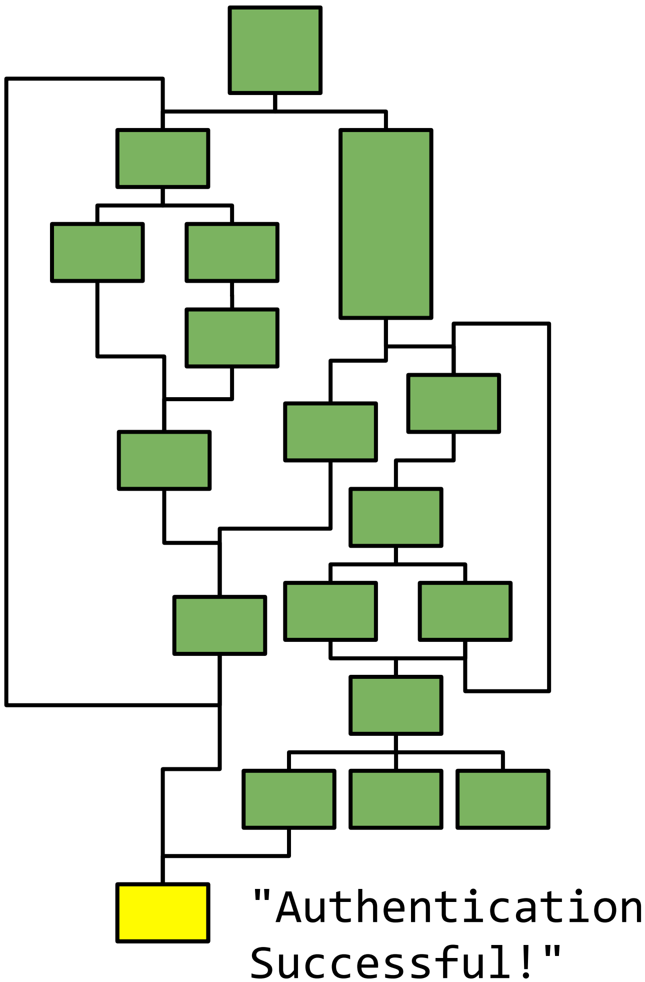
- Without OS/ABI information





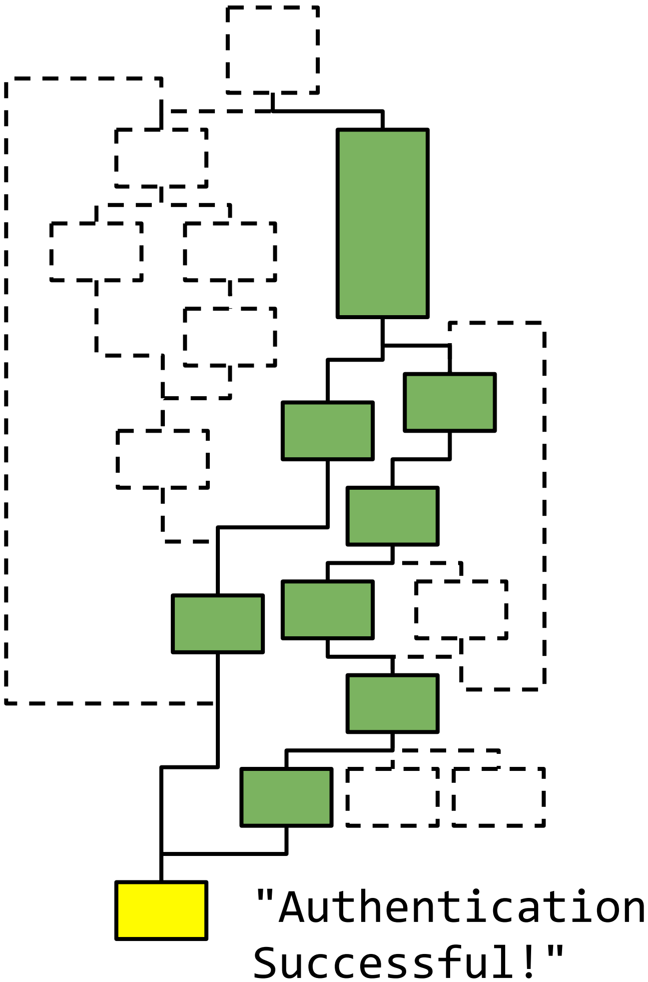
- With ABI information

Identify Authenticated Point



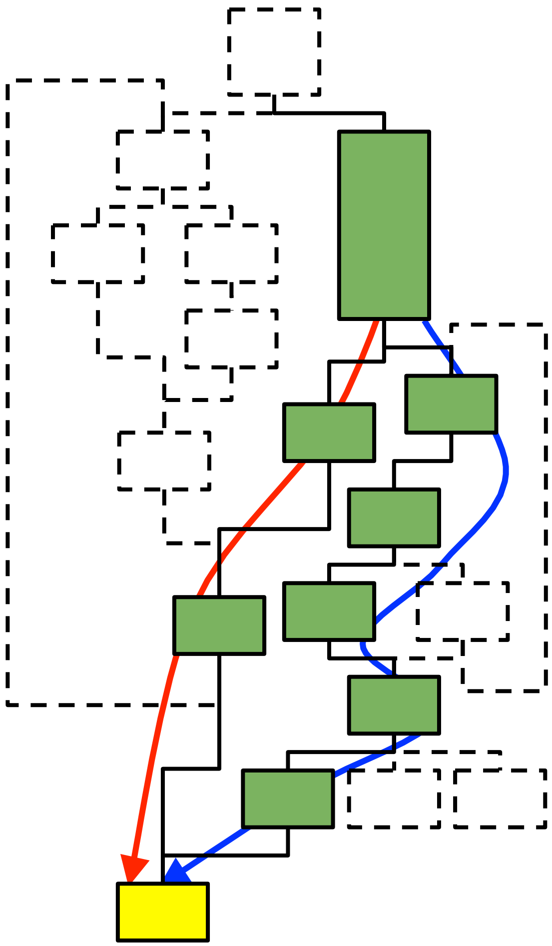
- static analysis (data references, system calls)
- human analyst fallback

Compute Authentication Slice



- static analysis (program slicing)

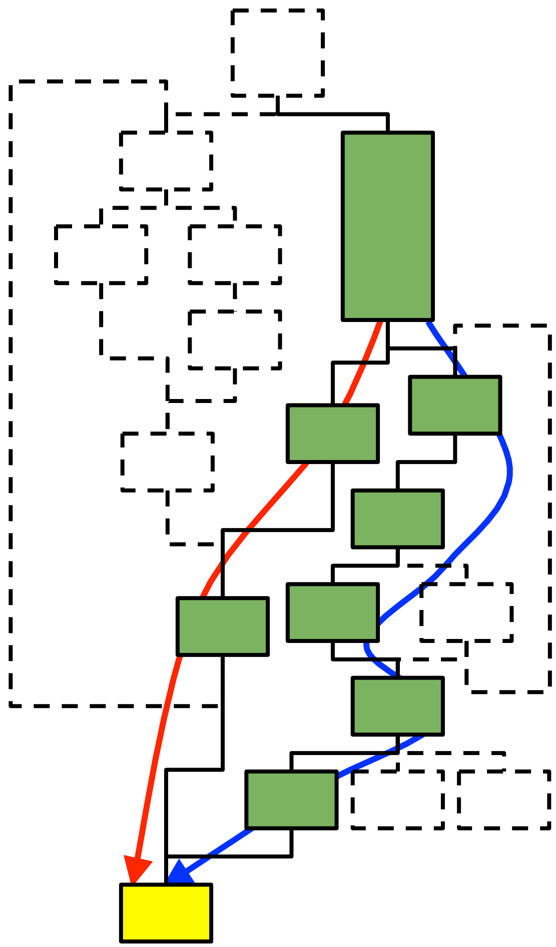
Path Collection



authenticated path

authenticated path

Vulnerability Detection

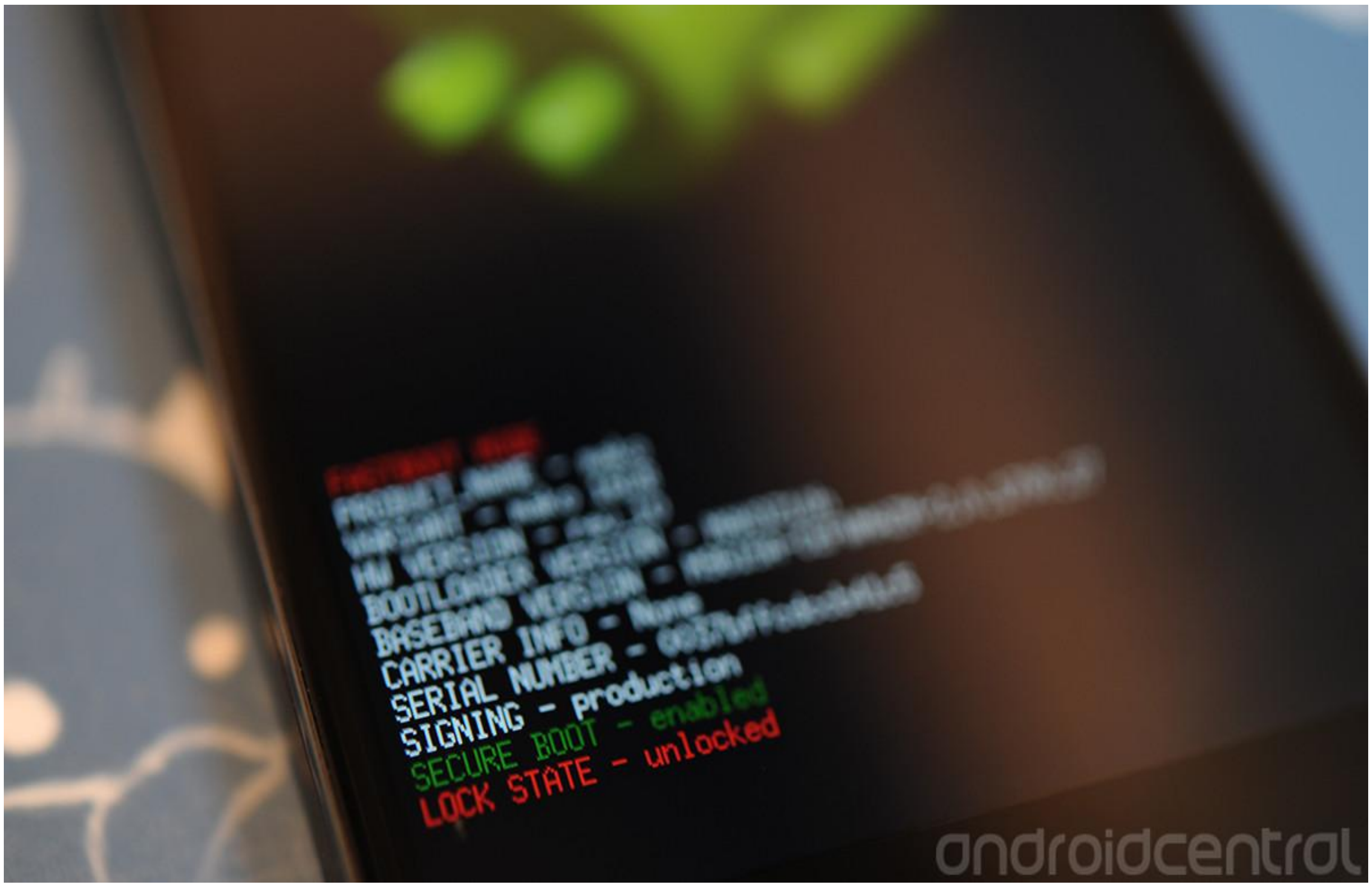


- can the attacker determine a concrete authenticating input via program analysis?

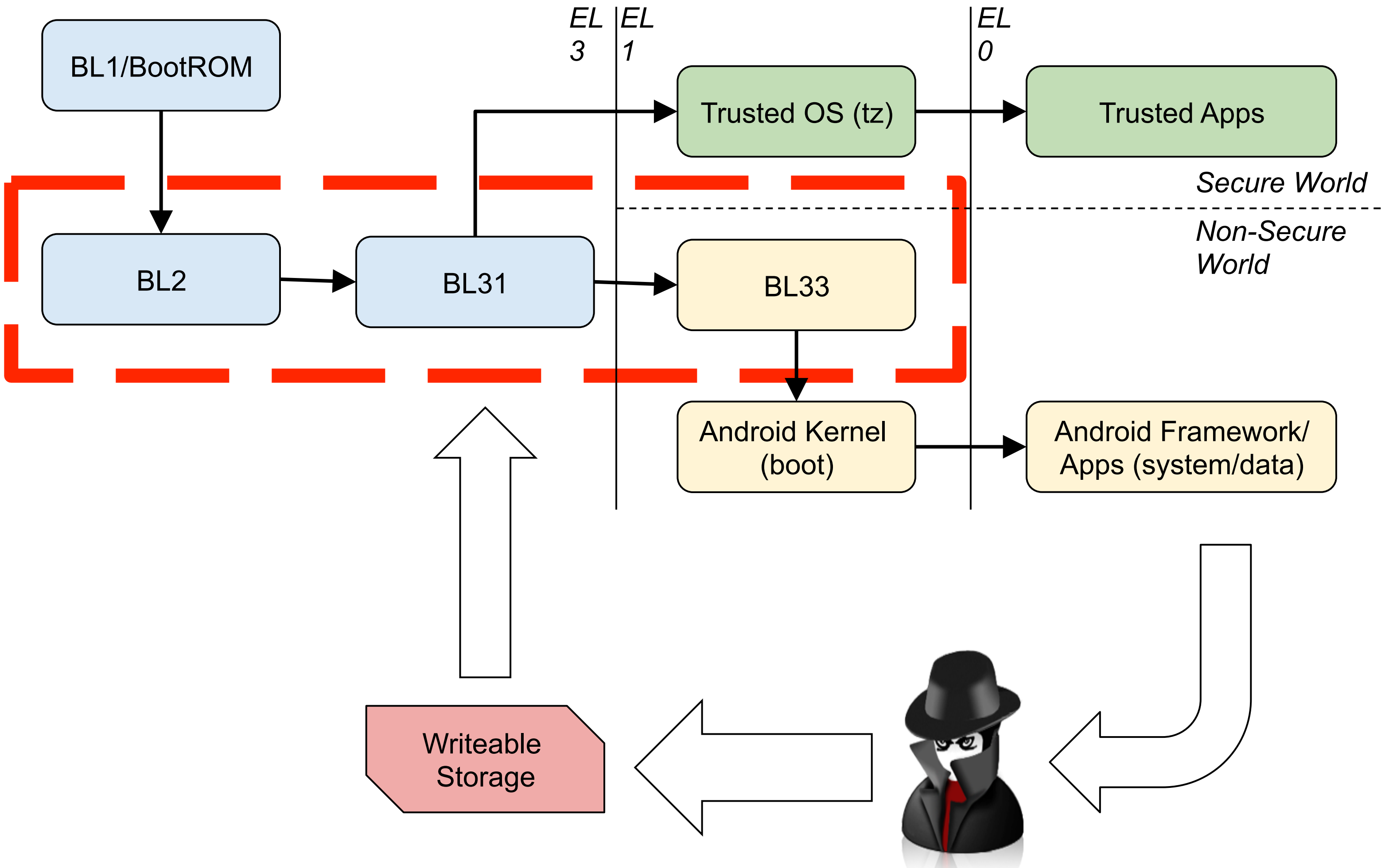
"AAA:
XXX"
"BBB:
YYY"
"CCC:
ZZZ"
...

"service:
service"

Bootloader Vulnerabilities



Bootloader Vulnerabilities



Two Malice Models

Memory Corruption

"Is data, read from writeable storage, used unsafely in memory operations?"

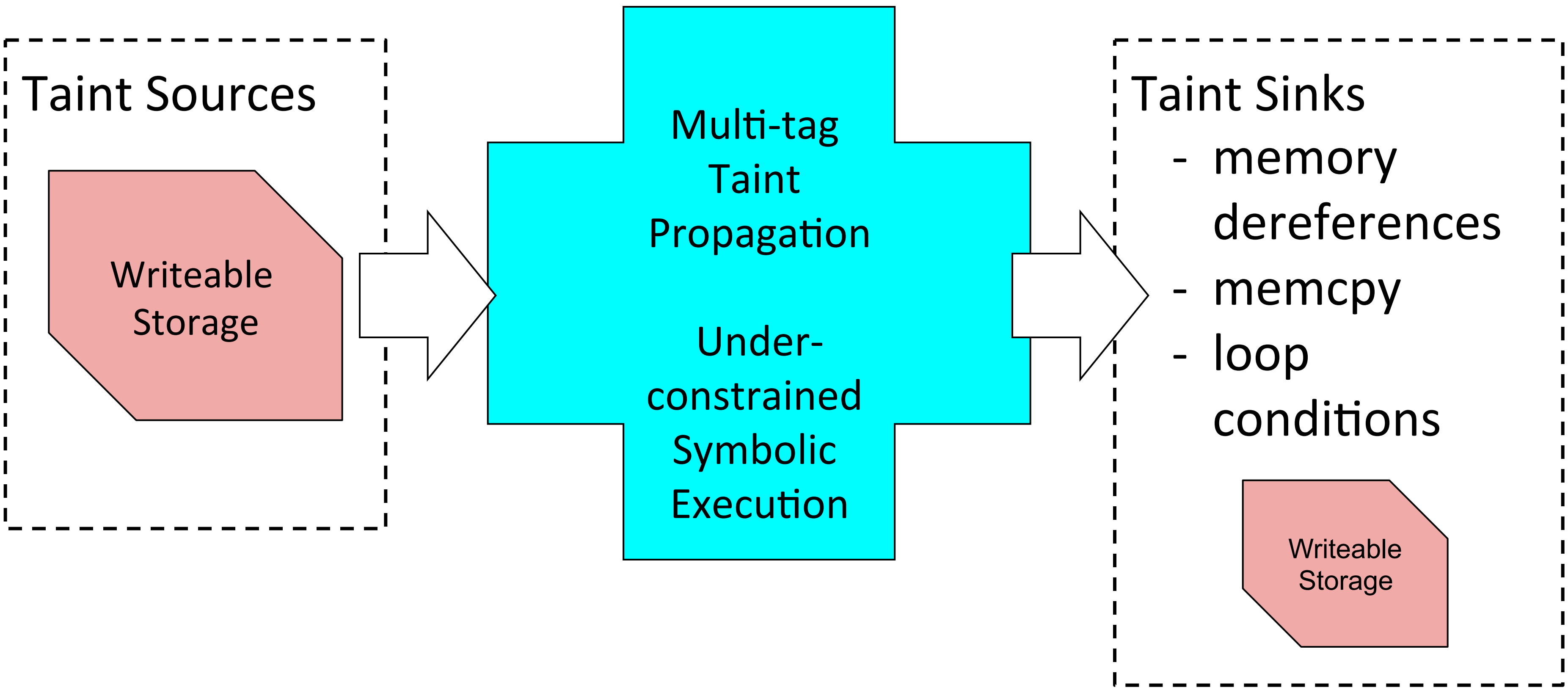
(can result in bricking, device compromise, and even TrustZone compromise!)

Unsafe Unlock

"Can the device be unlocked without triggering a user data wipe?"

(can result in data compromise)

Symbolic Taint Propagation

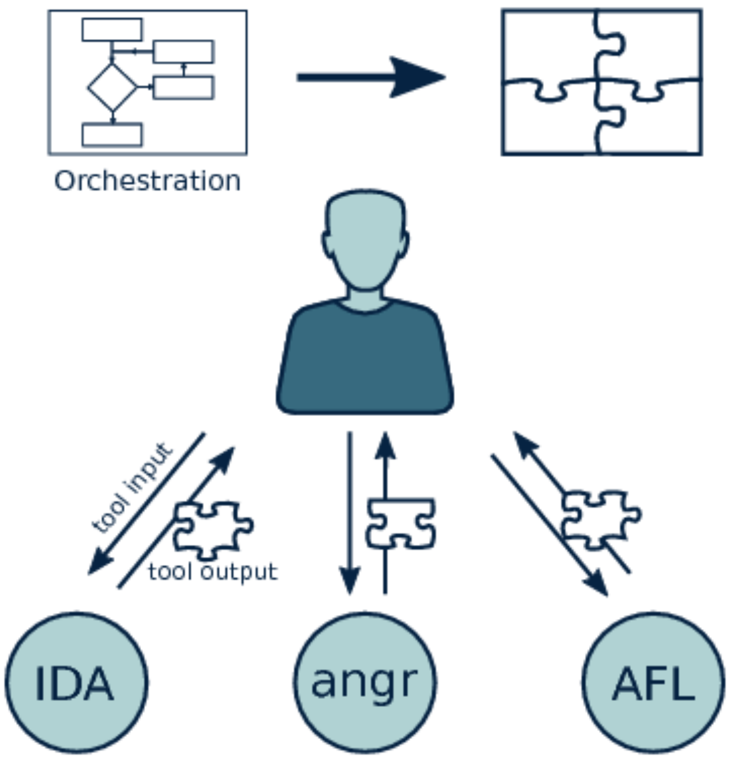


Results

Bootloader	Sources	Sinks	Alerts	Memory Bugs	Unsafe Unlock
Qualcomm (Latest)	2	1	0	0	1
Qualcomm (Old)	3	1	4	1	1
NVIDIA	6	1	1	1	0
HiSilicon/Huawei	20	4	15	5	1
MediaTek	2	2	-	-	-
Total	**33**	**9**	**20**	**7**	**3**

AUTOMATING VULNERABILITY ANALYSIS

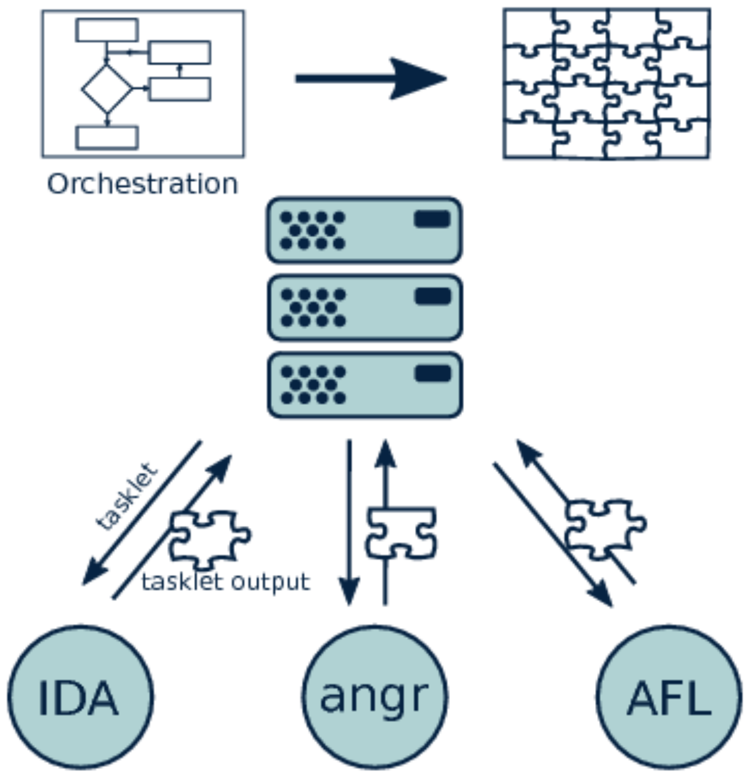
From Tools Supporting Humans …



High effectiveness

Low scalability

… To Fully Automated Analysis

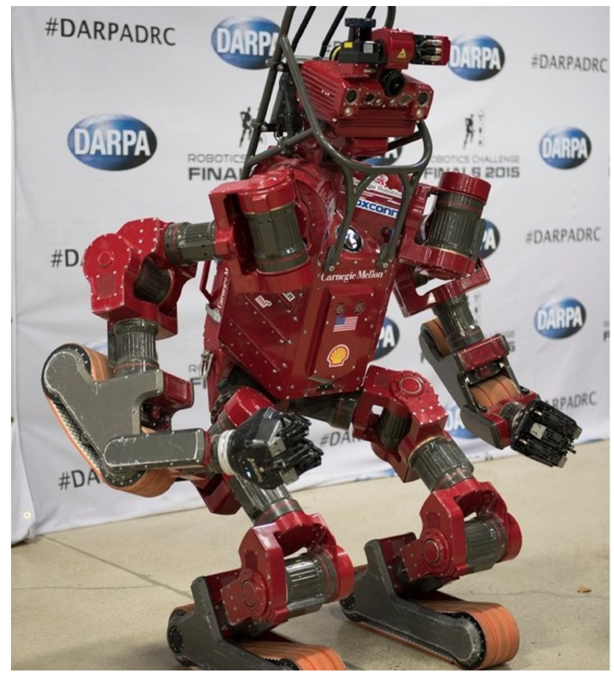


High scalability

DARPA Grand Challenges

Self-driving Cars

Robots



DARPA Cyber Grand Challenge

Programs!

DARPA Cyber Grand Challenge (CGC)

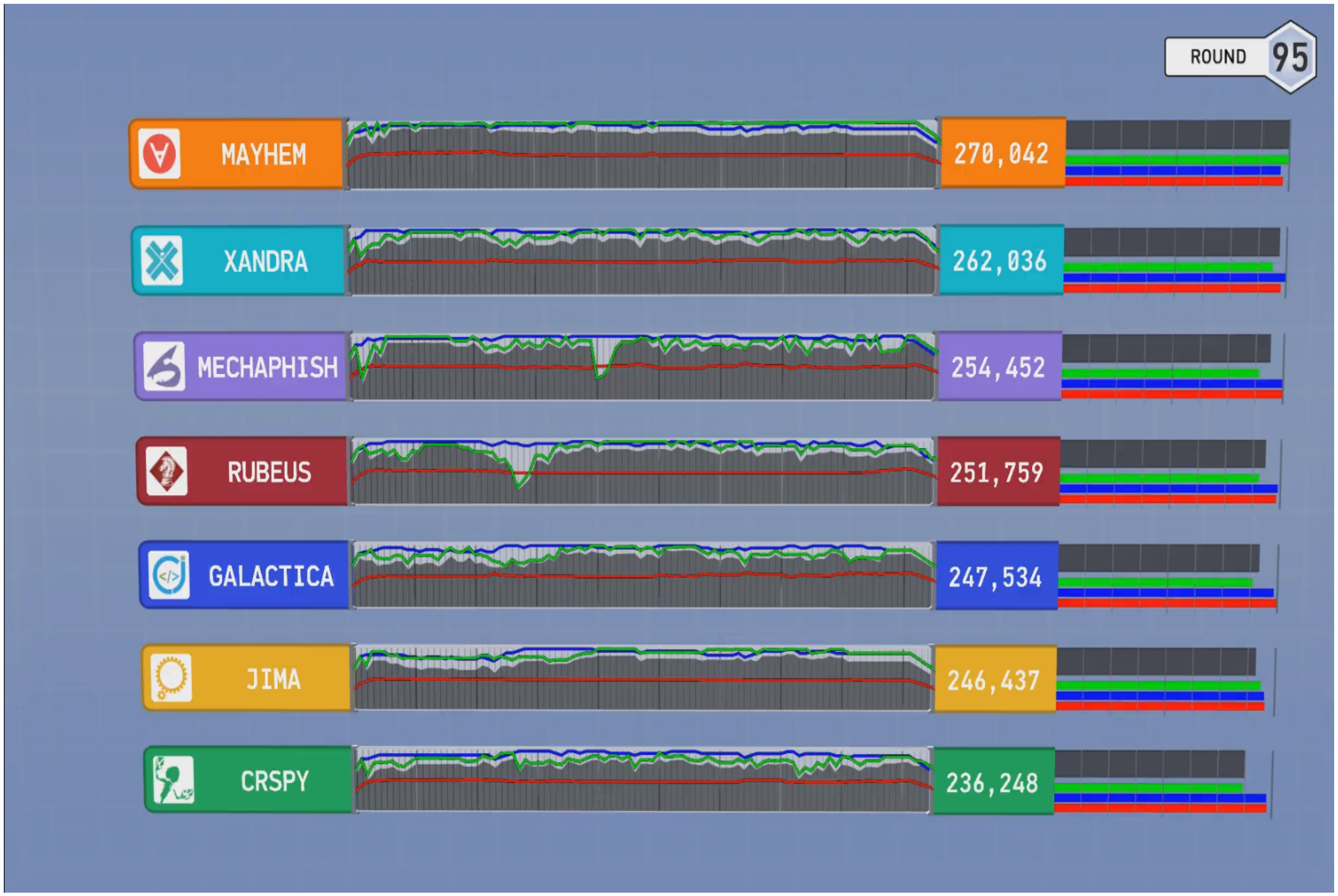
CGC
CYBER
GRAND_CHALLENGE

DARPA Cyber Grand Challenge

- CTF-style competition
- Autonomous Cyber-Reasoning Systems (CRSs) attack and defend a number of services (binaries)
- No human in the loop
- A first qualification round decided the 7 finalists
- Final event was on August 4, 2016 during DefCon
  - Shellphish came in 3rd place
- Significant cash prizes
  - 750K for qualification, 2M for win (750K for 3rd place)

CGC Results

UC Santa Barbara



Summary

- Internet of Things
  - explosive growth of devices with embedded software
  - many interesting security challenges
- Binary analysis
  - key tool to hunt for IOT vulnerabilities
  - delivers powerful results, but faces scalability issues
  - promising approach is to combine analysis techniques (e.g., fuzzing and symbolic execution)
- angr
  - UCSB open-source binary analysis software

Thank You!