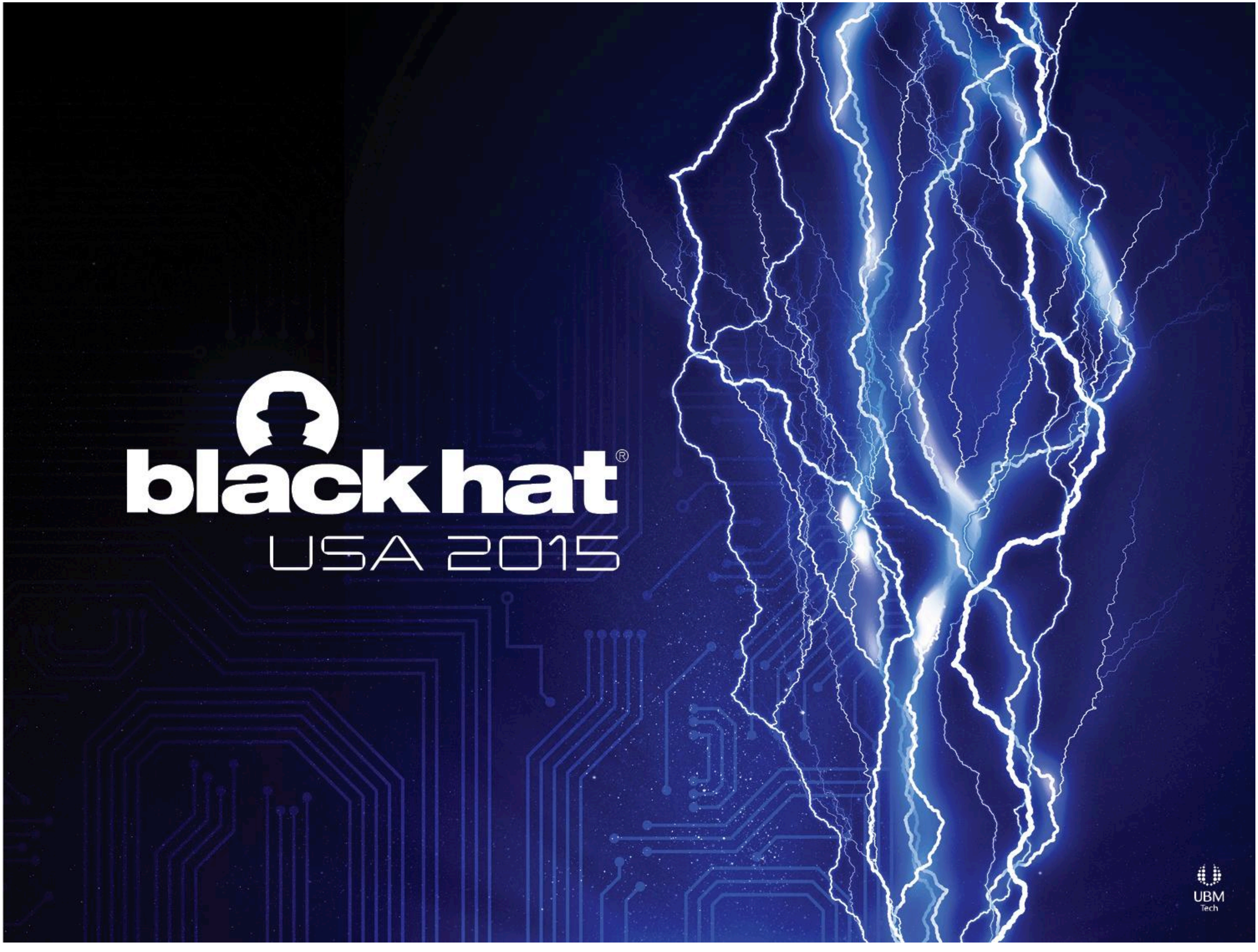

black hat®
USA 2015
UBM
Tech

Who are we?

- Co-founder and Chief Scientist at Lastline, Inc.
  – Lastline offers protection against zero-day threats and advanced malware

- Professor in Computer Science at UC Santa Barbara
  – many systems security papers in academic conferences

- Member of Shellphish

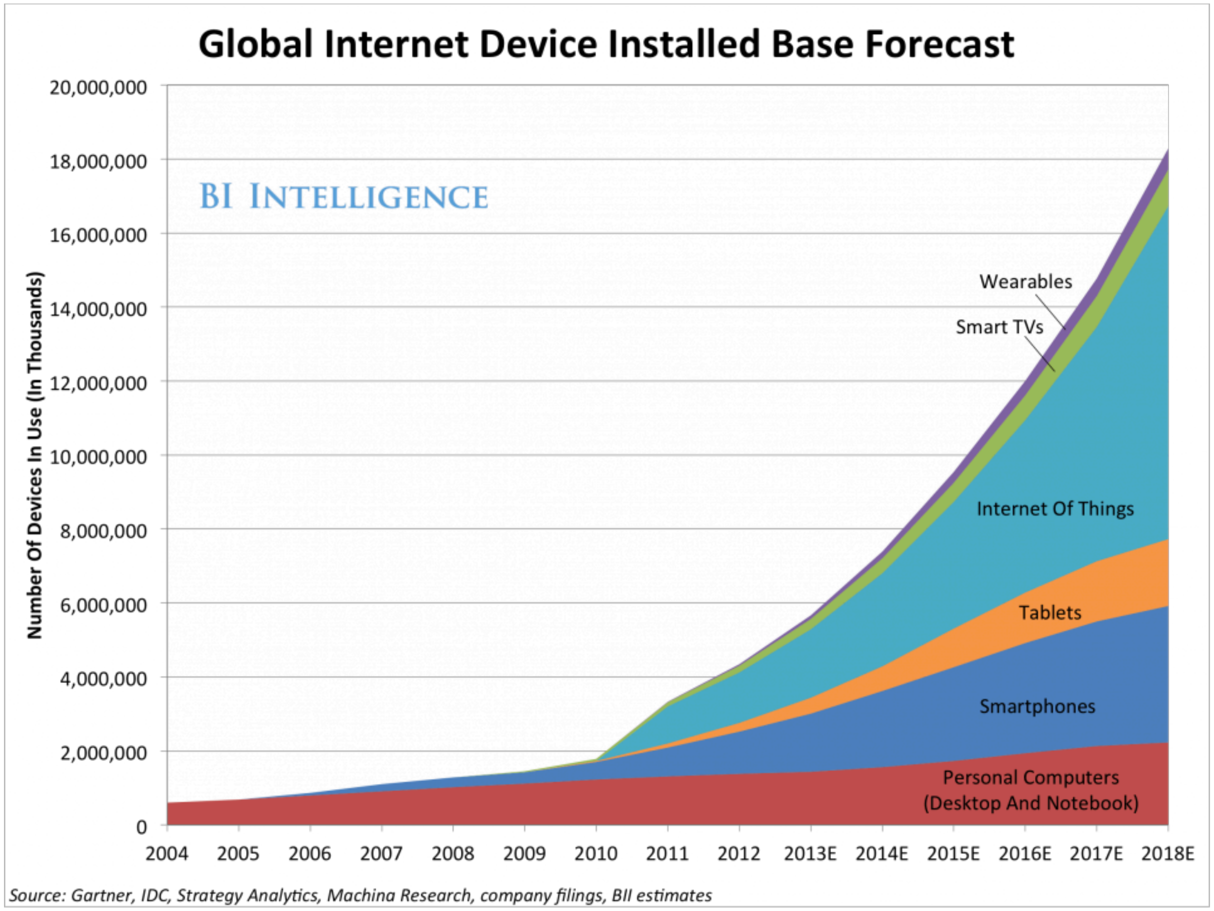
SHELLPHISH

Who are we?

- PhD Student at UC Santa Barbara
  - research focused primarily on binary security and embedded devices
- Member of Shellphish
  - team leader of Shellphish's CGC effort

What are we talking about?

The "Internet of Things"



Embedded software is everywhere

What is on embedded devices?

- Embedded Linux and user-space programs

- Custom OS and custom programs combined together in a *binary blob*
  - typically, the binary is all that you get
  - and, sometimes, it is not easy to get this off the device

Binary Analysis

Binary analysis

noun | bi·na·ry anal·y·sis | \ˈbī-nə-rē ə-ˈna-lə-səs\

1. The process of automatically deriving properties about the behavior of binary programs
2. Including static binary analysis and dynamic binary analysis

Goals of Binary Analysis

- Program verification
- Program testing
- Vulnerability excavation
- Vulnerability signature generation

- Reverse engineering
- Vulnerability excavation
- Exploit generation

Static Binary Analysis

- reason over multiple (all) execution paths
- can achieve excellent coverage
- precision versus scalability trade-off
  - very precise analysis can be slow and not scalable
  - too much approximation leads to wrong results (false positives)
- often works on abstract program model
  - for example, binary code is lifted to an intermediate representation

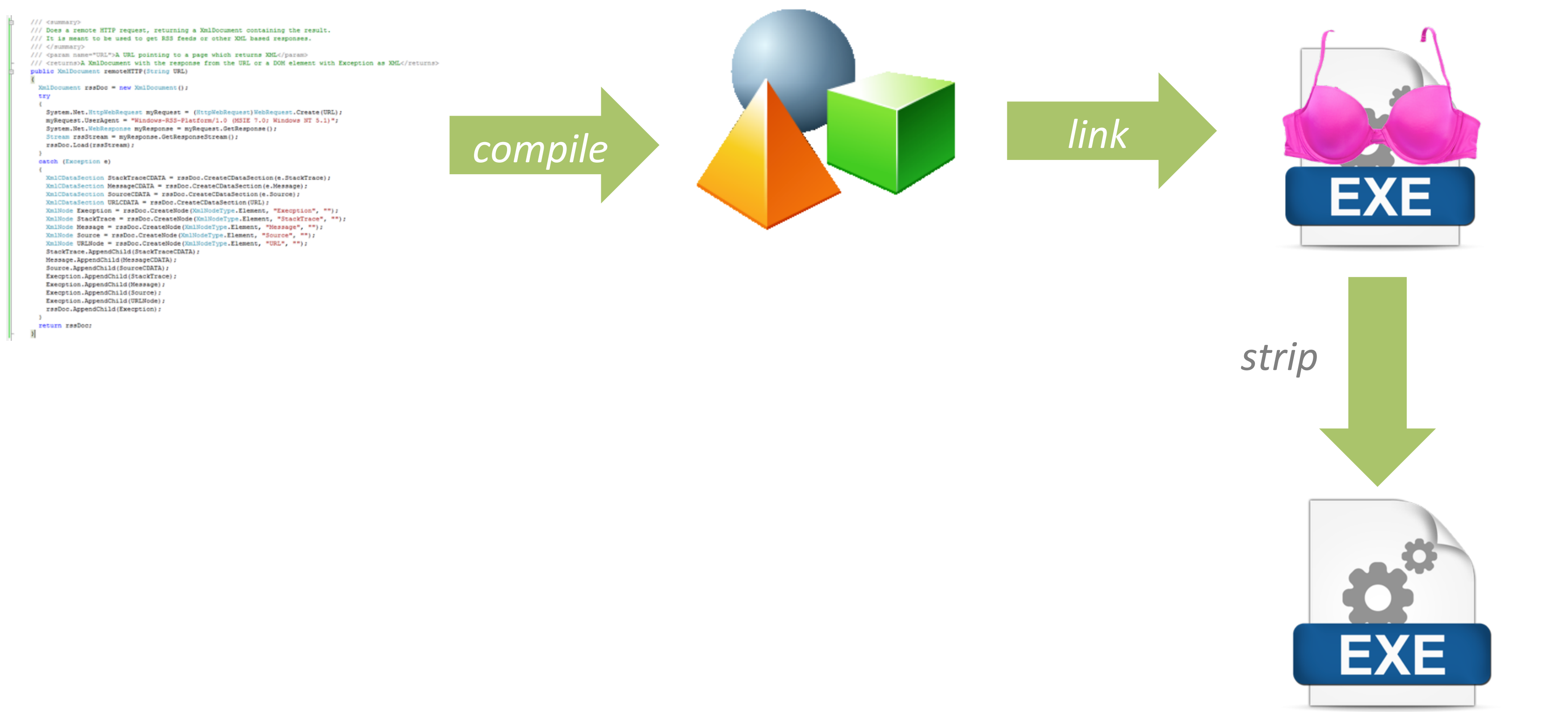
Dynamic Binary Analysis

- examine individual program paths
- very precise
- coverage is (very) limited
- sometimes hard to properly run program
  - hard to attach debugger to embedded system
  - when code is extracted and emulated, what happens with calls to peripherals?

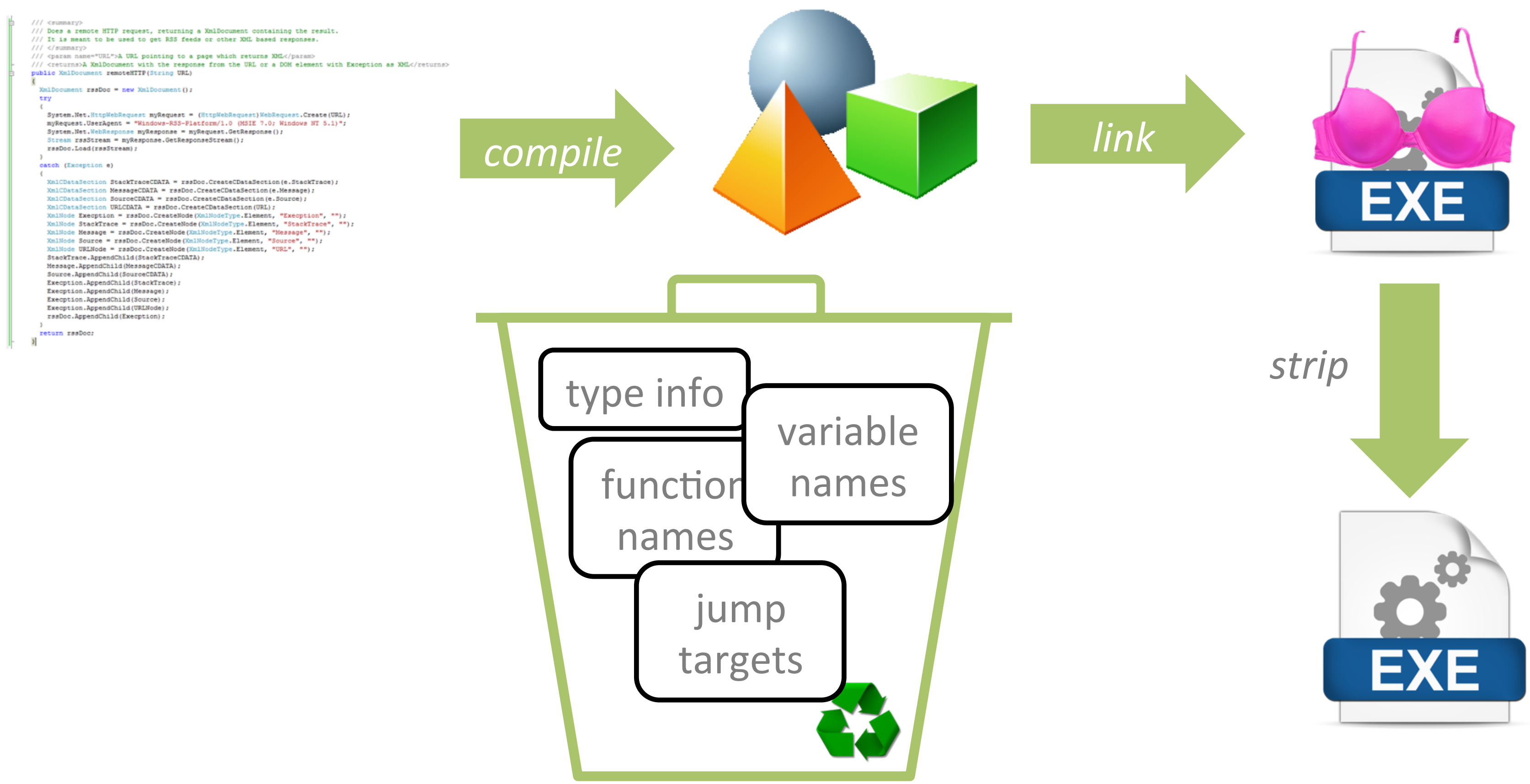
Challenges of Static Binary Analysis

- Get the binary code
- Binaries lack significant information present in source
- Often no clear library or operating system abstractions
  - where to start the analysis from?
  - hard to handle environment interactions

From Source to Binary Code



From Source to Binary Code



Missing OS and Library Abstractions

- (Linux) system call interface is great
  - you know what the I/O routines are
    - important to understand what user can influence
  - you have typed parameters and return values
  - let's the analysis focus on (much smaller) main program
- OS is not there or embedded in binary blob
  - heuristics to find I/O routines
  - open challenge to find mostly independent components

Missing OS and Library Abstractions

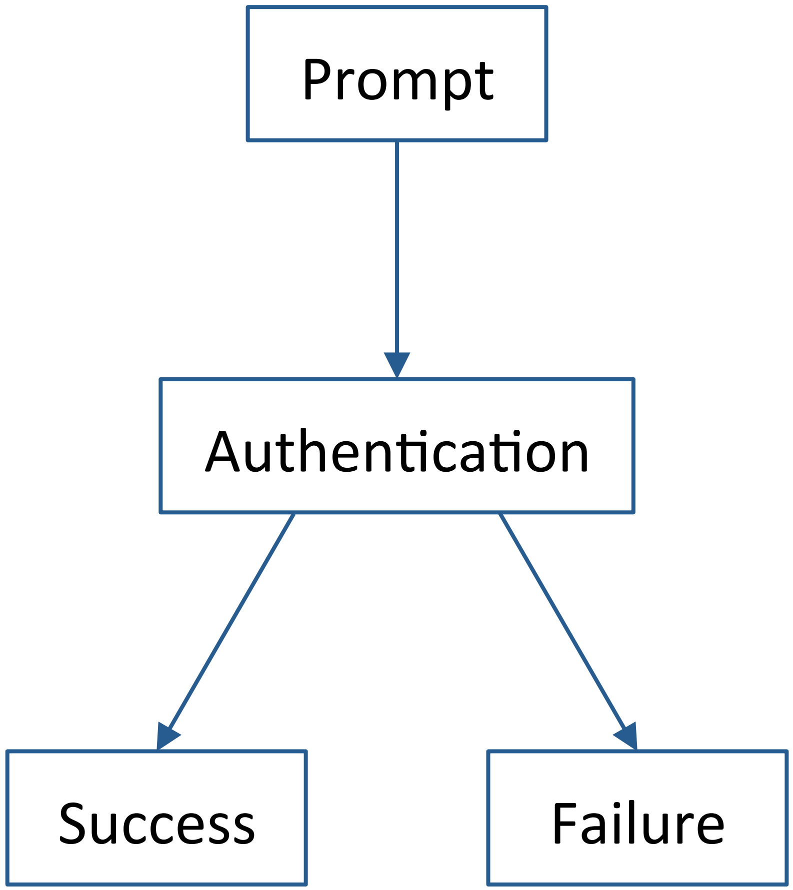
- Library functions are great
  - you know what they do and can write a "function summary"
  - you have typed parameters and return values
  - let's the analysis focus on (much smaller) main program

- Library functions are embedded (like static linking)
  - need heuristics to rediscover library functions
  - IDA FLIRT (Fast Library Identification and Recognition Technology)
  - more robustness based on looking for control flow similarity

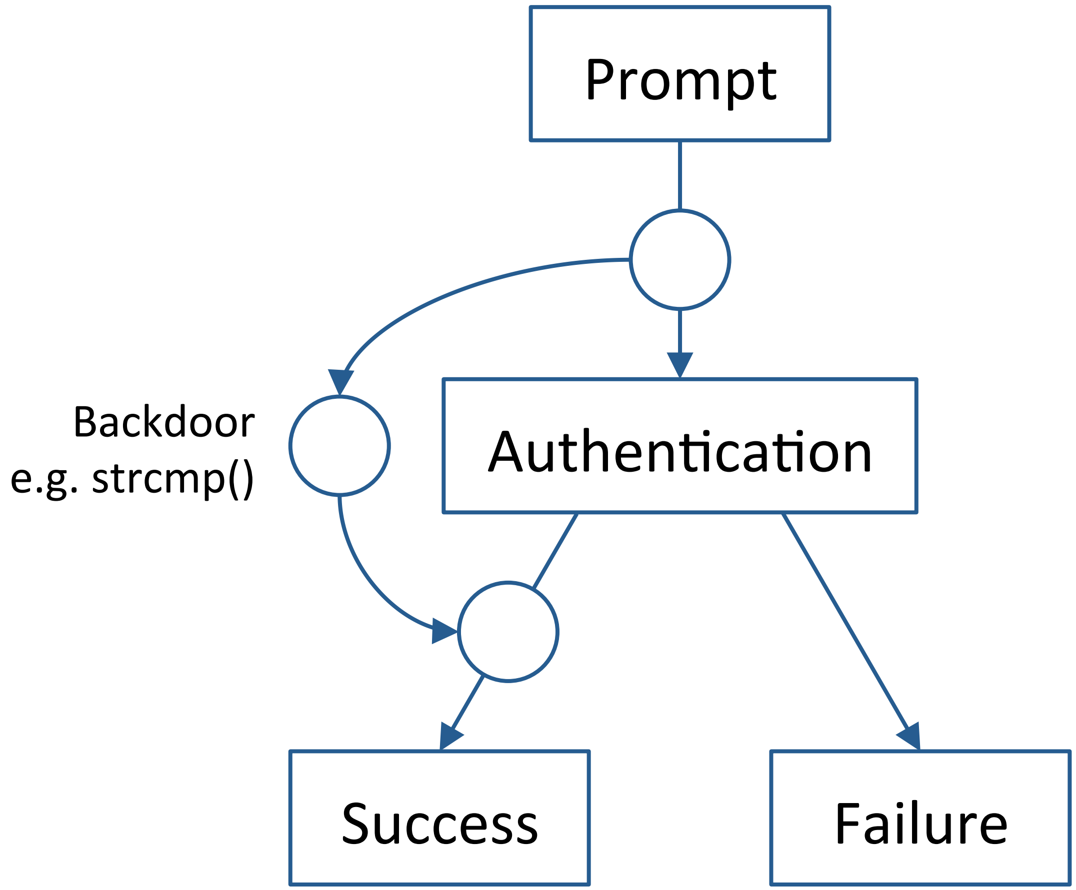
Types of Vulnerabilities

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

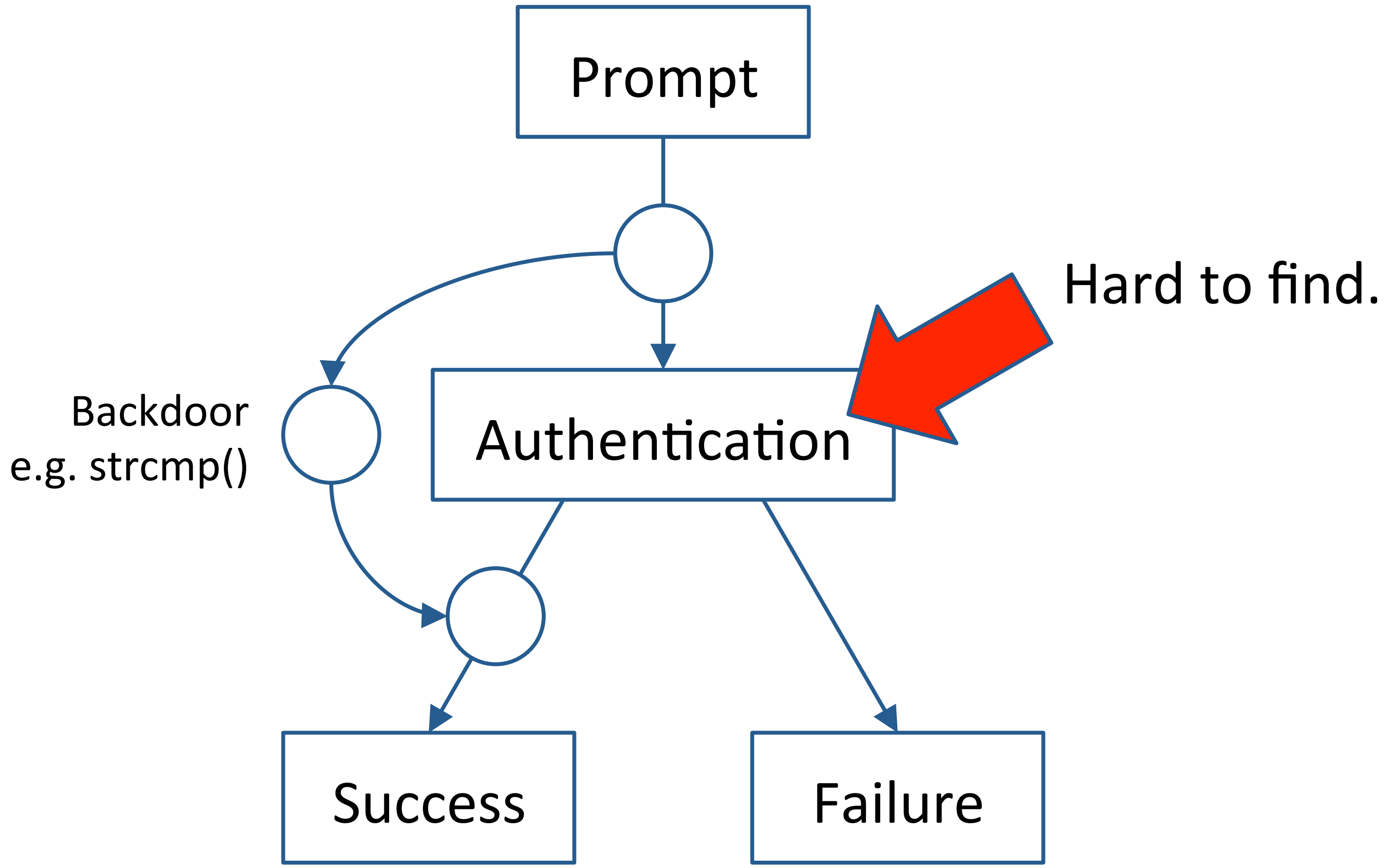
Authentication Bypass



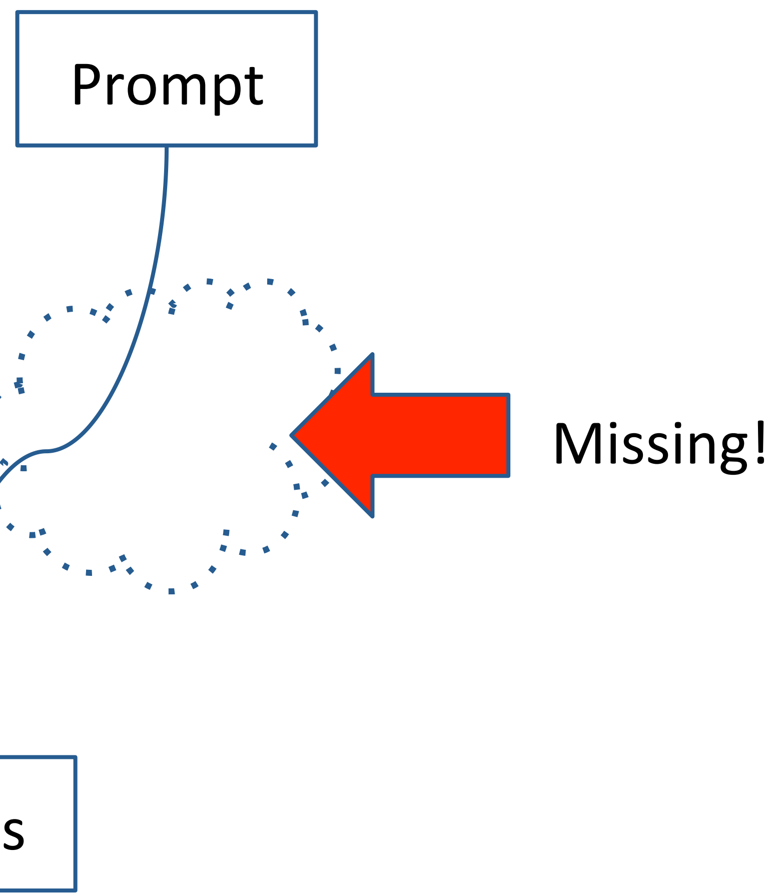
Authentication Bypass



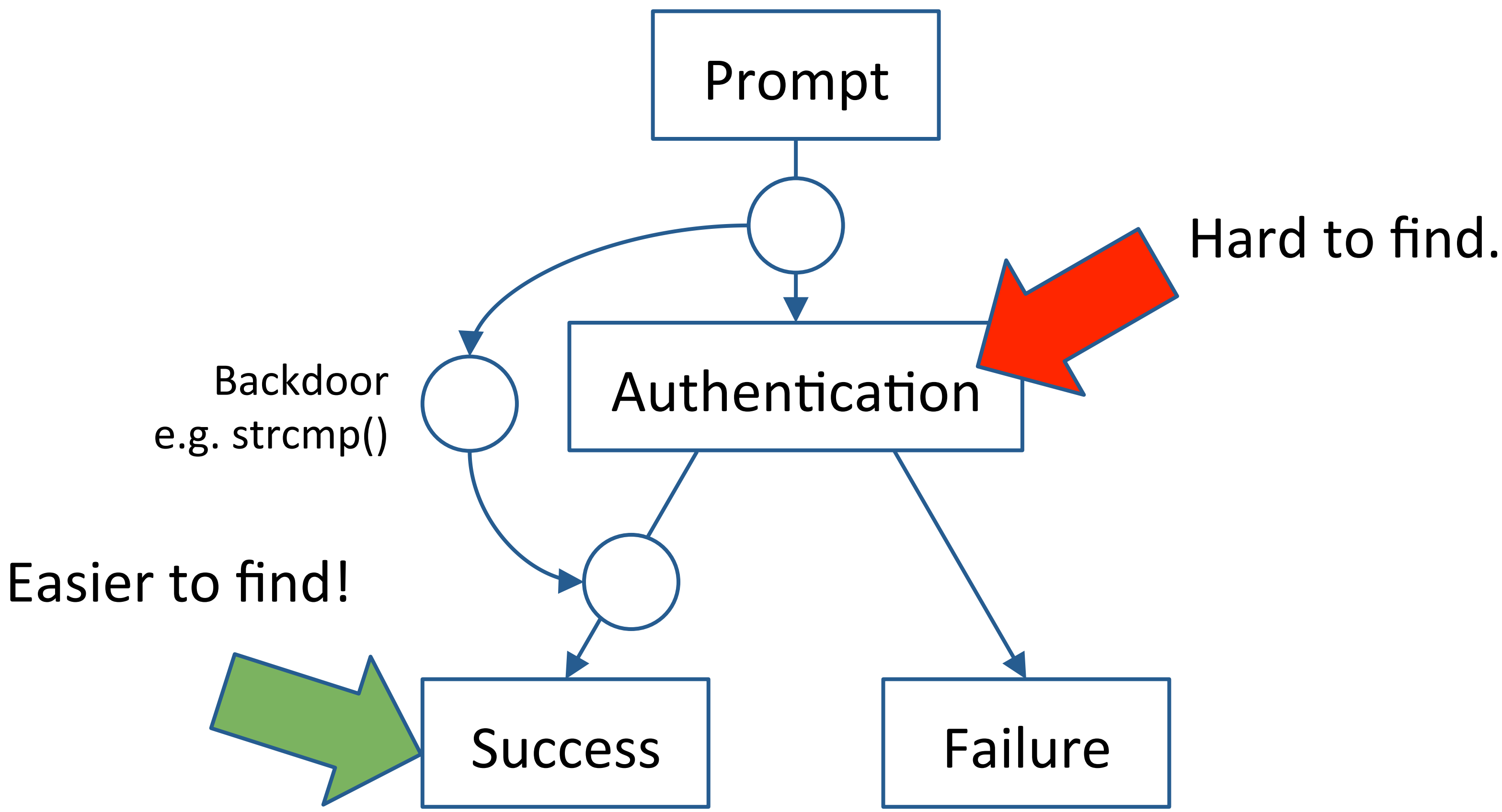
Authentication Bypass



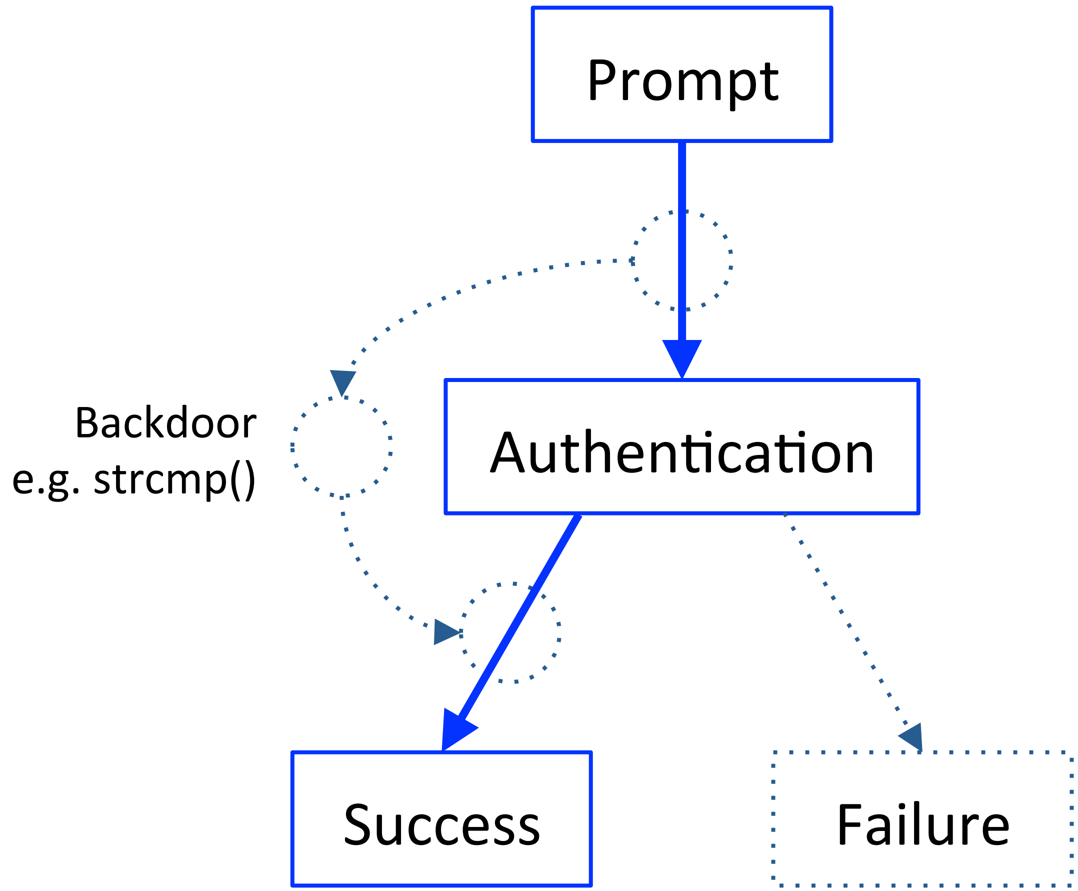
Authentication Bypass



Modeling Authentication Bypass



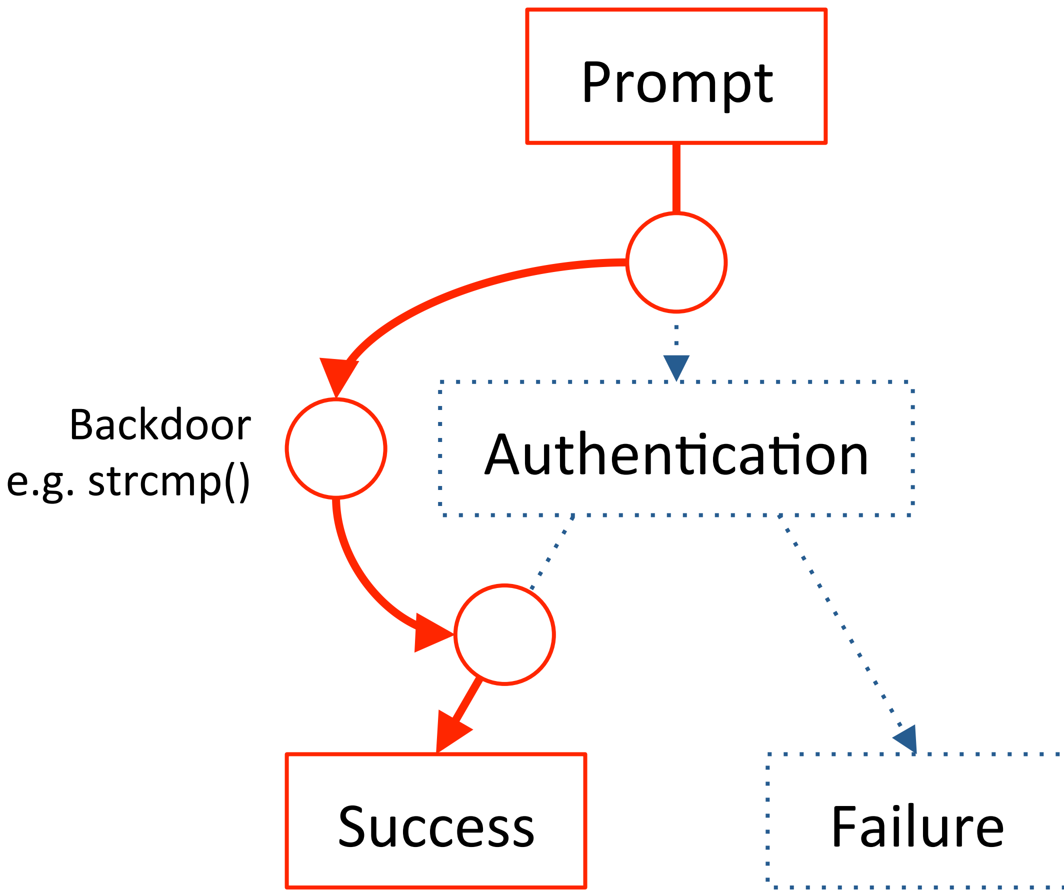
Input Determinism



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

The answer is NO

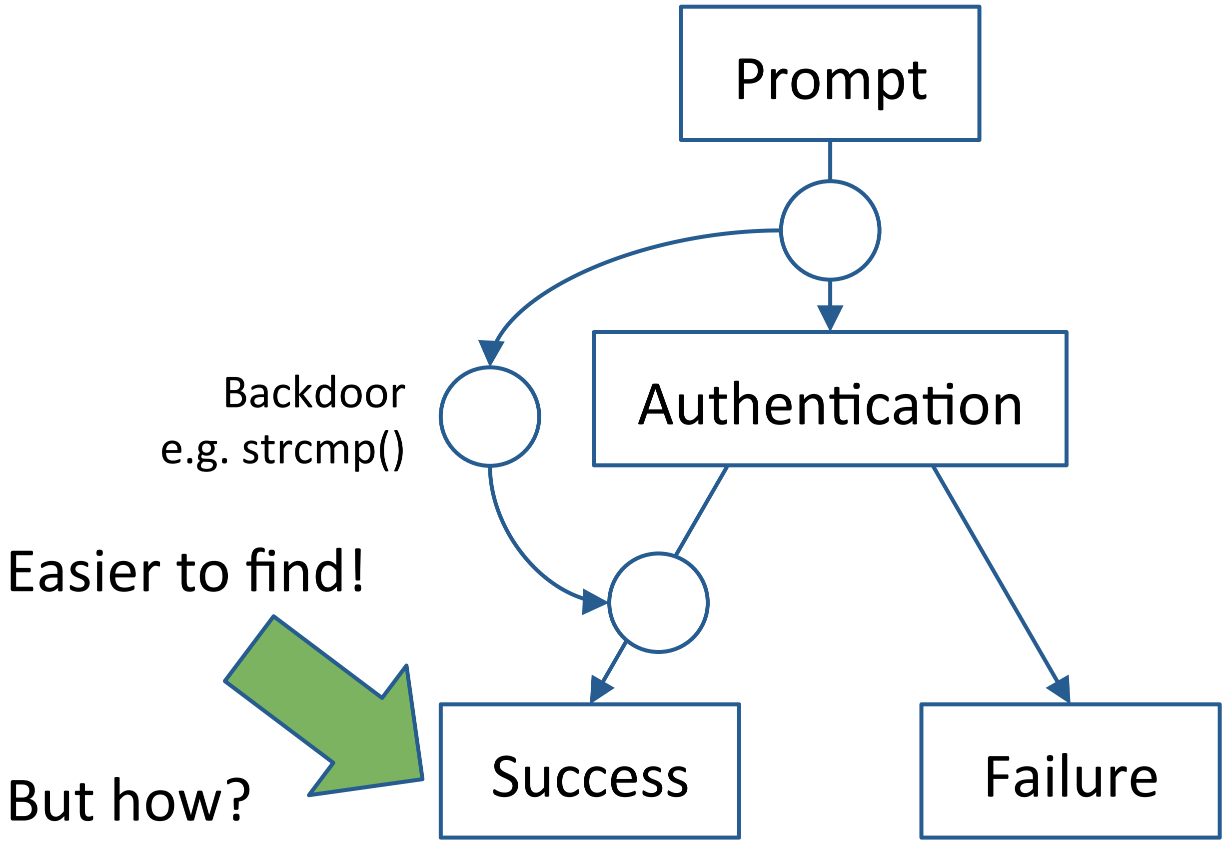
Input Determinism



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

The answer is YES

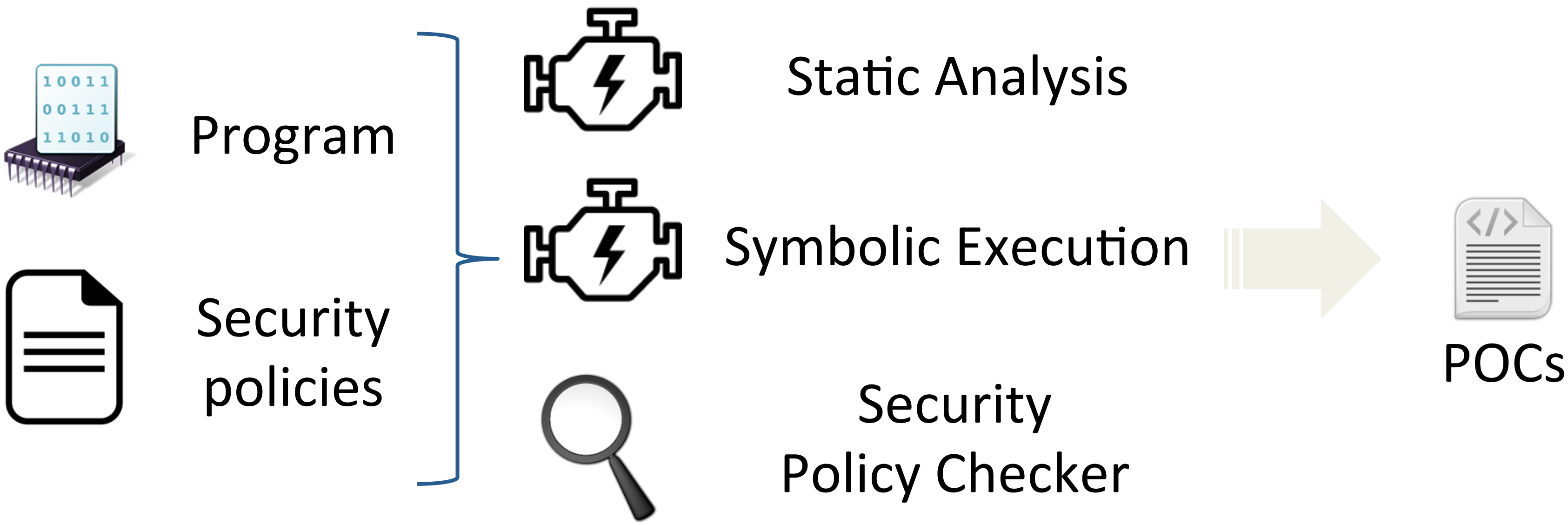
Modeling Authentication Bypass



Finding "Authenticated Point"

- Without OS/ABI information:
  - Manual reverse engineering
  - Program outputs/references certain strings (like "welcome admin")
  - Program accesses sensitive memory regions
- With ABI information:
  - Program calls sensitive syscalls
  - Program accesses sensitive resources/files

Using angr to Hunt for Vulnerabilities



angr: A Binary Analysis Framework

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

angr: A Binary Analysis Framework

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

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?"

1. Interpret the application.
2. Track "constraints" on variables.
3. When the required condition is triggered, "concretize" to obtain a possible input.

Symbolic Execution

Constraints
$x >= 10$ $x < 100$

Concretize →

$x = 42$

Constraint solving:

- Conversion from set of constraints to set of concrete values that satisfy them.
- NP-complete, in general.

Symbolic Execution

```
x = int(input())
if x >= 10:
  if x < 100:
      print "Two!"
  else:
      print "Lots!"
else:
  print "One!"
```

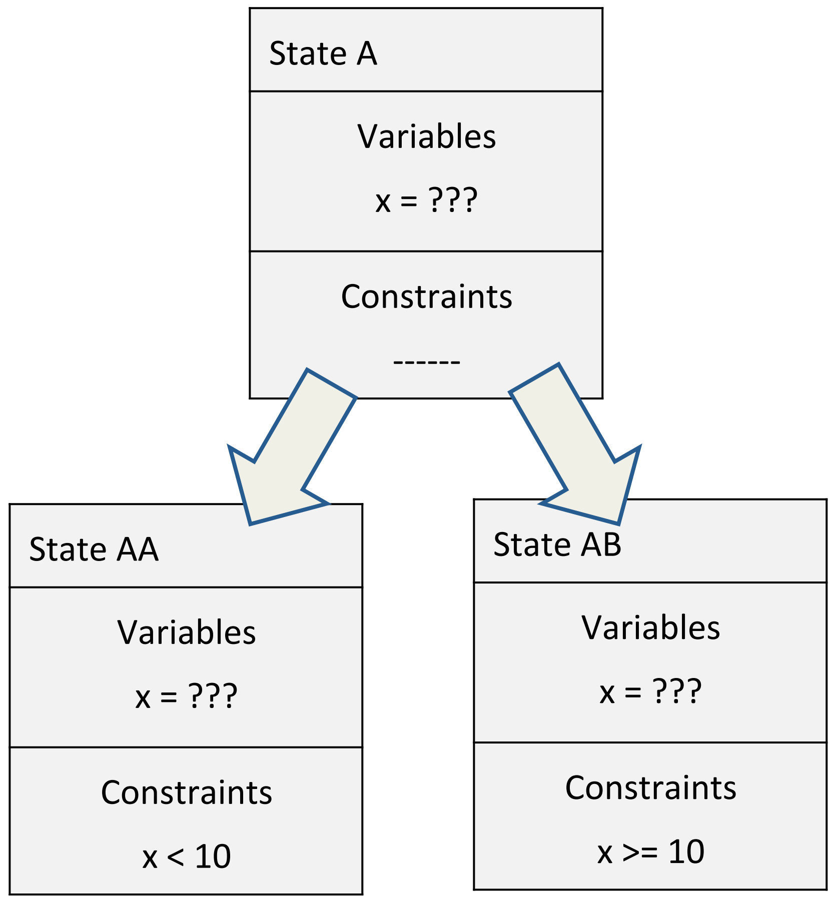
Symbolic Execution

```
x = int(input())
if x >= 10:
        if x < 100:
                print "Two!"
        else:
                print "Lots!"
else:
        print "One!"
```

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

Symbolic Execution

```
x = int(input())
if x >= 10:
        if x < 100:
                print "Two!"
        else:
                print "Lots!"
else:
        print "One!"
```



Symbolic Execution

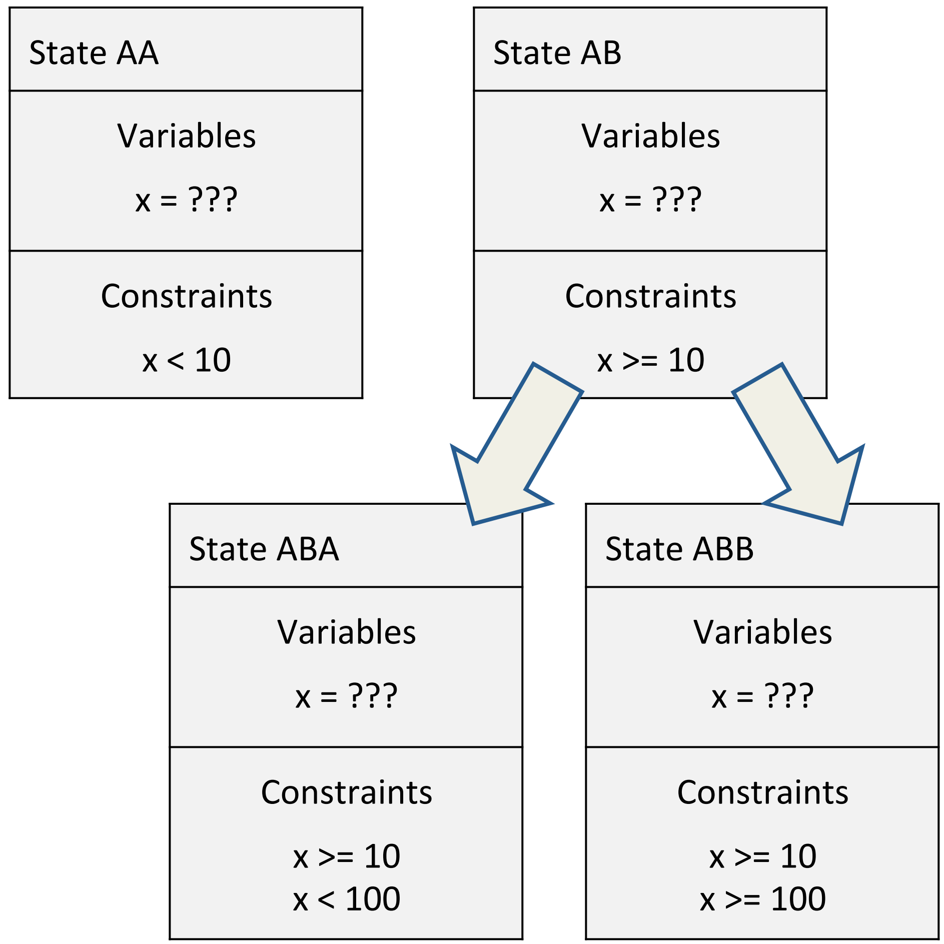
```
x = int(input())
if x >= 10:
        if x < 100:
                print "Two!"
        else:
                print "Lots!"
else:
        print "One!"
```

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

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

Symbolic Execution

```
x = int(input())
if x >= 10:
        if x < 100:
                print "Two!"
        else:
                print "Lots!"
else:
        print "One!"
```



Symbolic Execution

```
x = int(input())
if x >= 10:
        if x < 100:
                print "Two!"
        else:
                print "Lots!"
else:
        print "One!"
```

State ABA

Variables

x = ???

Constraints

x >= 10
x < 100

Concretized ABA

Variables

x = 99

Symbolic Execution - Pros and Cons

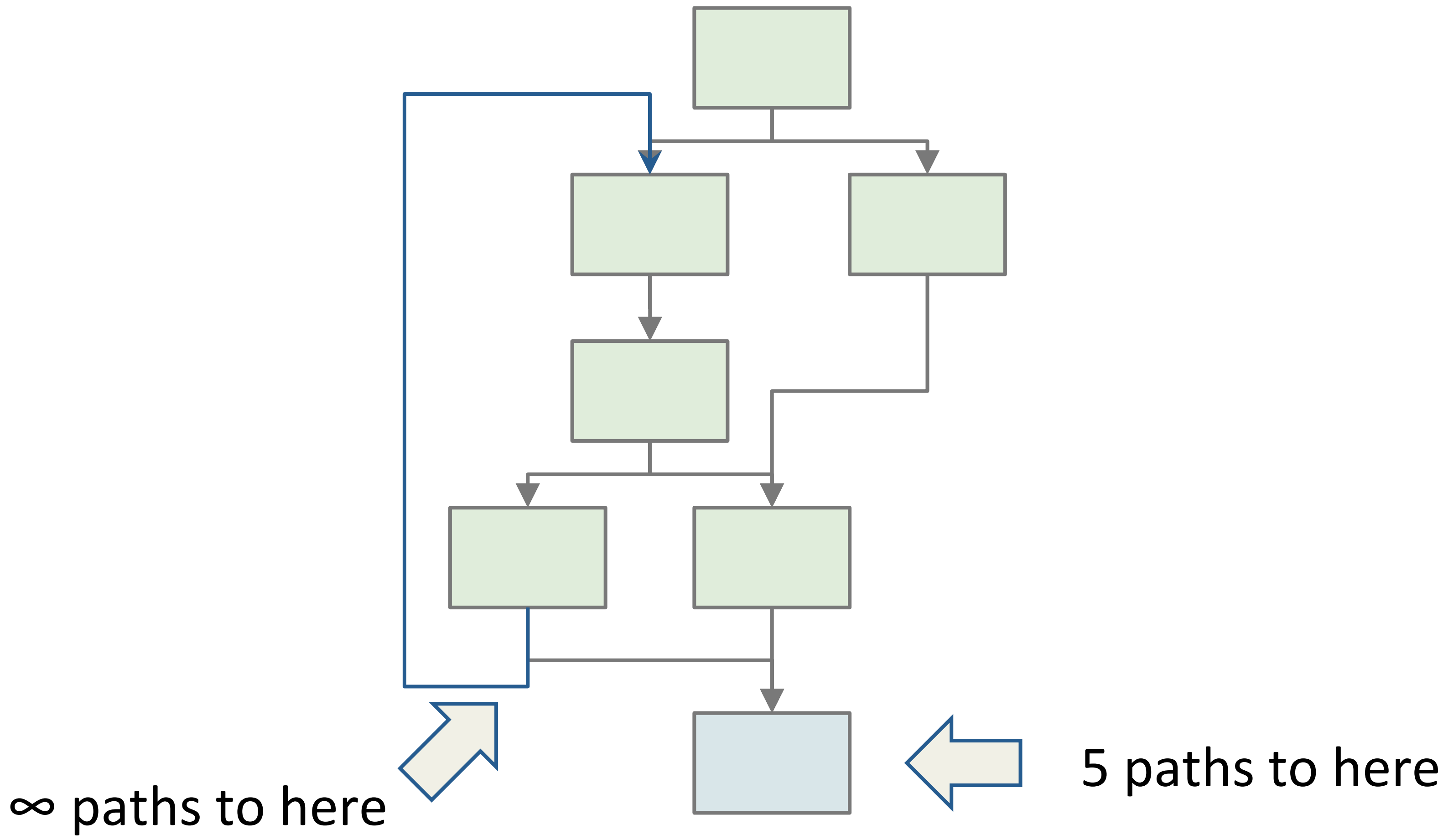
Pros

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

Cons

- Not scalable
  - constraint solving is np-complete
  - path explosion

Path Explosion



angr: A Binary Analysis Framework

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

angr: A Binary Analysis Framework

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

angr: A Binary Analysis Framework

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

angr: A Binary Analysis Framework

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

Value Set Analysis

Memory access checks	*Type inference*
Variable recovery	**Range recovery**
Wrapped-interval analysis	
Value-set analysis	
Abstract interpretation	

Value Set Analysis

```
{
        ( global,                    (4[0x601000, 0x602000],
32) ),
        ( stack_0x400957, (8[-0xc, -0x4],
        32) )
}
```

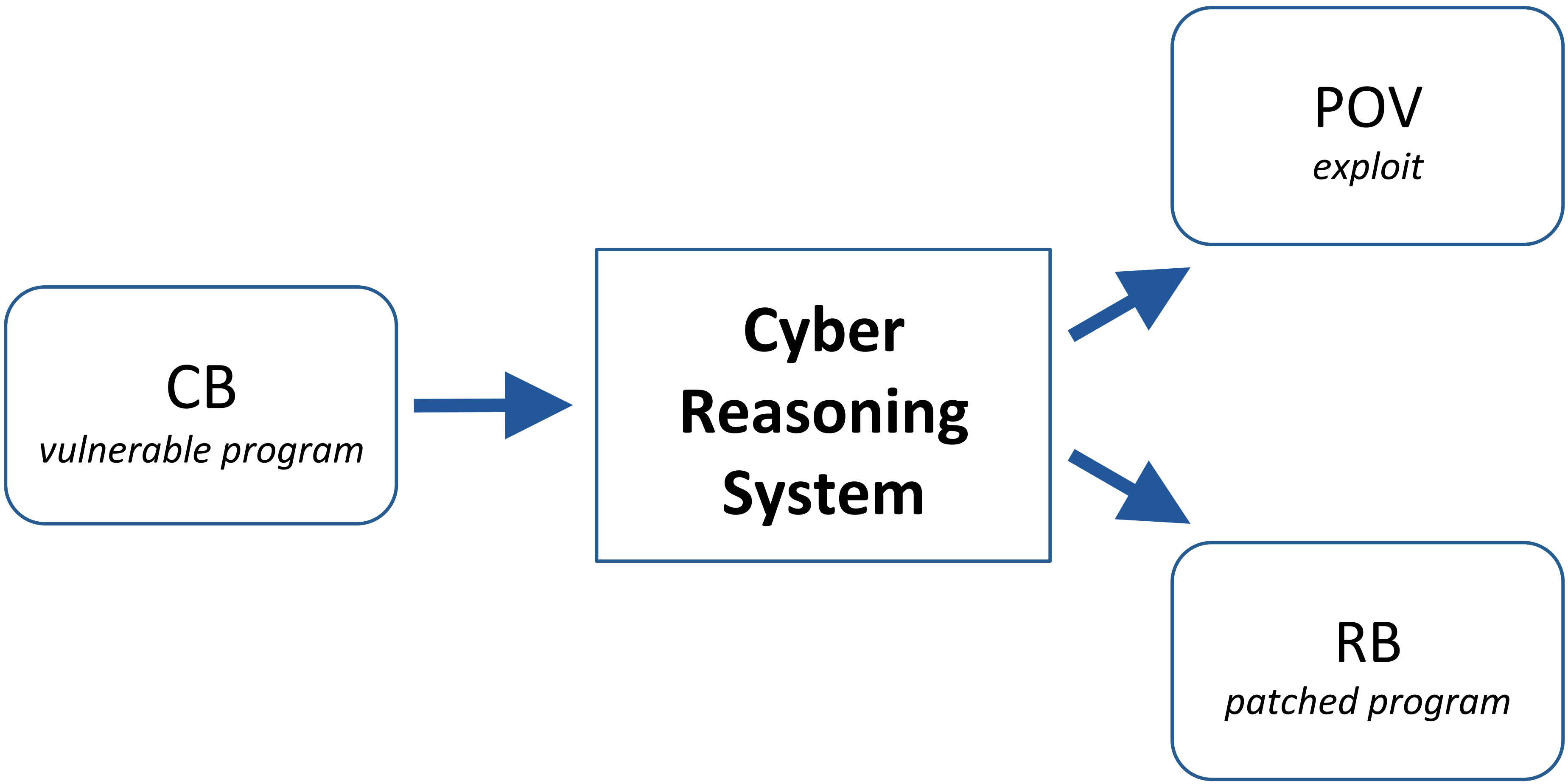
global
...
0x601000, 0x601004
0x601008, 0x60100c
...

stack_0x400957
...
- 0xc
- 0x4
...

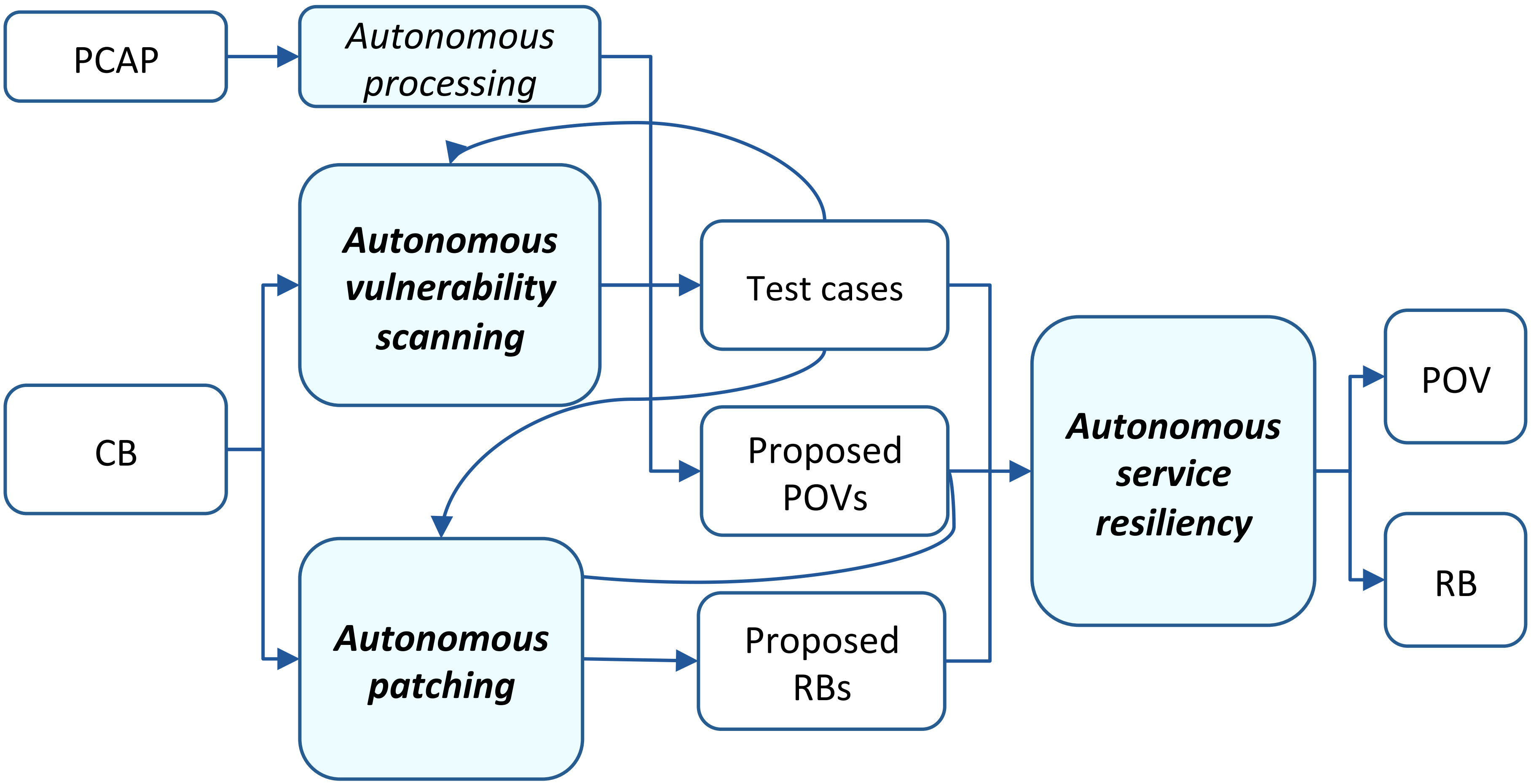
What have we used this for?

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

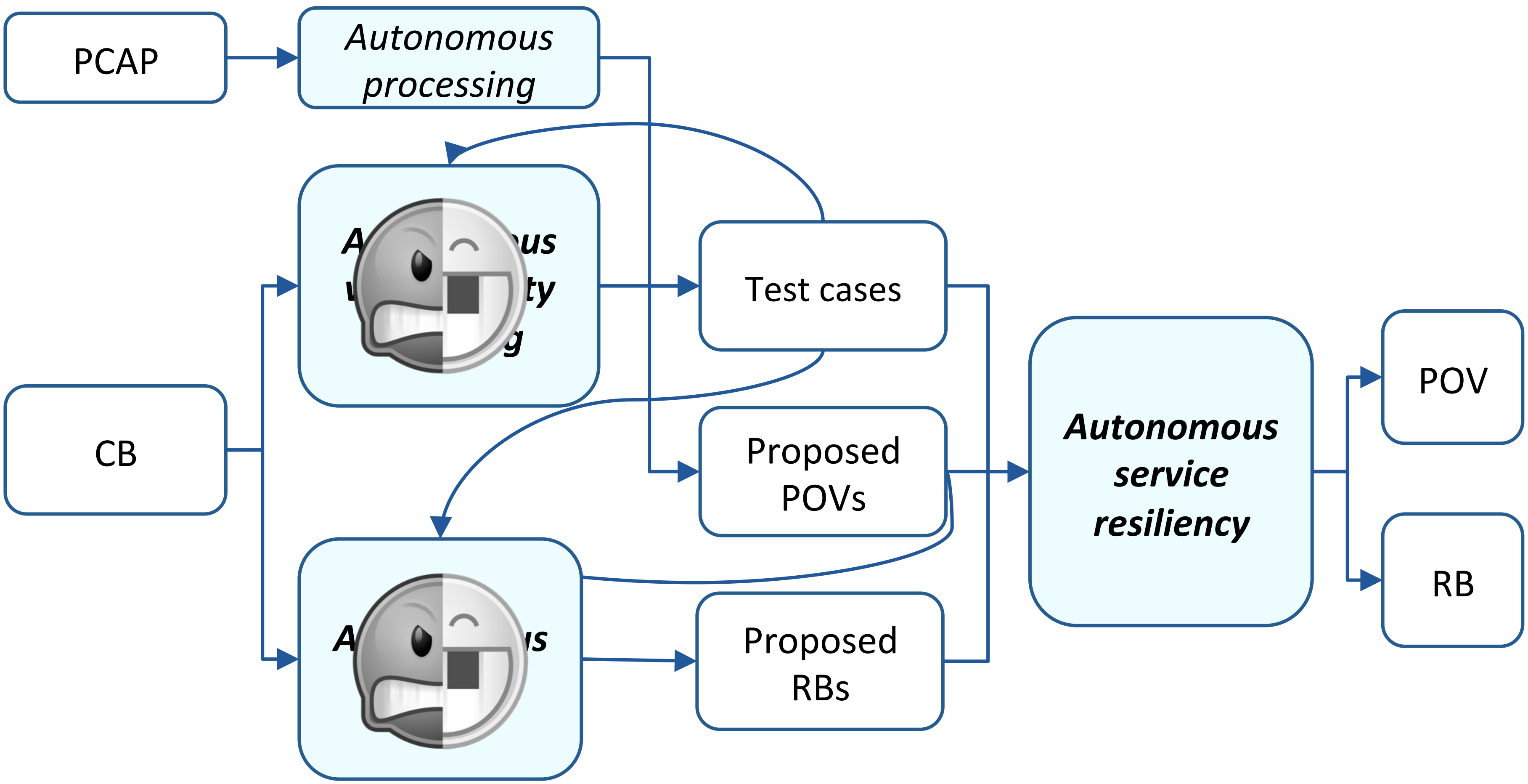
The Cyber Grand Challenge!



The Shellphish CRS



The Shellphish CRS



black hat USA 2015