RSAConference2016

San Francisco | February 29 – March 4 | Moscone Center

Connect to Protect

SESSION ID: HT-W02

Autonomous Hacking: The New Frontiers of Attack and Defense

Giovanni Vigna

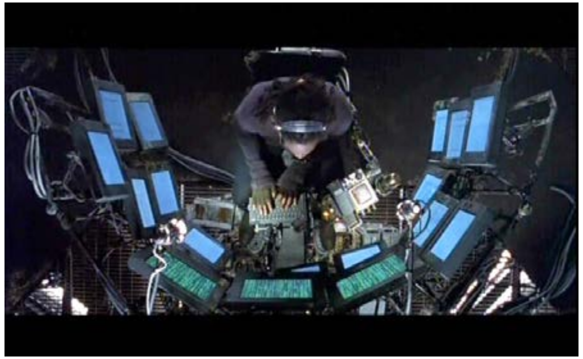
CTO

Lastline, Inc.

@lastlinelabs

#RSAC

Hacking



?

Hacking Teams

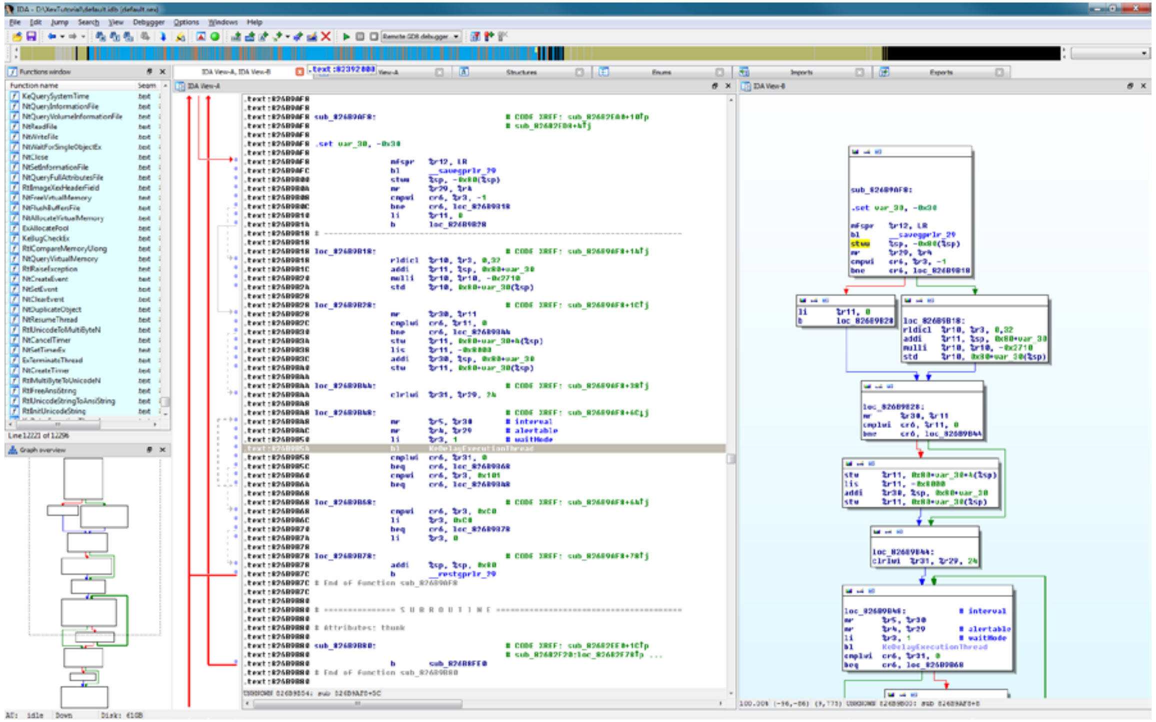
Hacking What?

- Security compromises can be achieved through different routes

Hacking the User

Hacking the Process

Hacking the Code



Functions window

Function name	Segm
KeQuerySystemTime	.text
NtQueryInformationFile	.text
NtQueryVolumeInformationFile	.text
NtReadFile	.text
NtWriteFile	.text
NtWaitForSingleObjectEx	.text
NtClose	.text
NtSetInformationFile	.text
NtQueryFullAttributesFile	.text
RtlImageXexHeaderField	.text
NtFreeVirtualMemory	.text
NtFlushBuffersFile	.text
NtAllocateVirtualMemory	.text
ExAllocatePool	.text
KeBugCheckEx	.text
RtlCompareMemoryUlong	.text
NtQueryVirtualMemory	.text
RtlRaiseException	.text
NtCreateEvent	.text
NtSetEvent	.text
NtClearEvent	.text
NtDuplicateObject	.text
NtResumeThread	.text
RtlUnicodeToMultiByteN	.text
NtCancelTimer	.text
NtSetTimerEx	.text
ExTerminateThread	.text
NtCreateTimer	.text
RtlMultiByteToUnicodeN	.text
RtlFreeAnsiString	.text
RtlUnicodeStringToAnsiString	.text
RtlInitUnicodeString	.text

Line 12221 of 12296

```
.text:826B9AF8
.text:826B9AF8
.text:826B9AF8 sub_826B9AF8:                           # CODE XREF: sub_82682FA0+10↑p
.text:826B9AF8                                         # sub_82682FD8+4↑j
.text:826B9AF8
.text:826B9AF8 .set var_30, -0x30
.text:826B9AF8
.text:826B9AF8                 mfspr   %r12, LR
.text:826B9AFC                 bl      __savegprlr_29
.text:826B9B00                 stwu    %sp, -0x80(%sp)
.text:826B9B04                 mr      %r29, %r4
.text:826B9B08                 cmpwi   cr6, %r3, -1
.text:826B9B0C                 bne     cr6, loc_826B9B18
.text:826B9B10                 li      %r11, 0
.text:826B9B14                 b       loc_826B9B28
.text:826B9B18 # ---------------------------------------------------------
.text:826B9B18
.text:826B9B18 loc_826B9B18:                           # CODE XREF: sub_826B9AF8+14↑j
.text:826B9B18                 rldicl  %r10, %r3, 0,32
.text:826B9B1C                 addi    %r11, %sp, 0x80+var_30
.text:826B9B20                 mulli   %r10, %r10, -0x2710
.text:826B9B24                 std     %r10, 0x80+var_30(%sp)
.text:826B9B28
.text:826B9B28 loc_826B9B28:                           # CODE XREF: sub_826B9AF8+1C↑j
.text:826B9B28                 mr      %r30, %r11
.text:826B9B2C                 cmplwi  cr6, %r11, 0
.text:826B9B30                 bne     cr6, loc_826B9B44
.text:826B9B34                 stw     %r11, 0x80+var_30+4(%sp)
.text:826B9B38                 lis     %r11, -0x8000
.text:826B9B3C                 addi    %r30, %sp, 0x80+var_30
.text:826B9B40                 stw     %r11, 0x80+var_30(%sp)
.text:826B9B44
.text:826B9B44 loc_826B9B44:                           # CODE XREF: sub_826B9AF8+38↑j
.text:826B9B44                 clrlwi  %r31, %r29, 24
.text:826B9B48
.text:826B9B48 loc_826B9B48:                           # CODE XREF: sub_826B9AF8+6C↓j
.text:826B9B48                 mr      %r5, %r30       # interval
.text:826B9B4C                 mr      %r4, %r29       # alertable
.text:826B9B50                 li      %r3, 1          # waitMode
.text:826B9B54                 bl      KeDelayExecutionThread
.text:826B9B58                 cmplwi  cr6, %r31, 0
.text:826B9B5C                 beq     cr6, loc_826B9B68
.text:826B9B60                 cmpwi   cr6, %r3, 0x101
.text:826B9B64                 beq     cr6, loc_826B9B48
.text:826B9B68
.text:826B9B68 loc_826B9B68:                           # CODE XREF: sub_826B9AF8+64↑j
.text:826B9B68                 cmpwi   cr6, %r3, 0xC0
.text:826B9B6C                 li      %r3, 0xC0
.text:826B9B70                 beq     cr6, loc_826B9B78
.text:826B9B74                 li      %r3, 0
.text:826B9B78
.text:826B9B78 loc_826B9B78:                           # CODE XREF: sub_826B9AF8+78↑j
.text:826B9B78                 addi    %sp, %sp, 0x80
.text:826B9B7C                 b       __restgprlr_29
.text:826B9B7C # End of function sub_826B9AF8
.text:826B9B7C
.text:826B9B80
.text:826B9B80 # =============== S U B R O U T I N E =======================================
.text:826B9B80
.text:826B9B80 # Attributes: thunk
.text:826B9B80
.text:826B9B80 sub_826B9B80:                           # CODE XREF: sub_82682FE0+1C↑p
.text:826B9B80                                         # sub_82682F20:loc_82682F78↑p ...
.text:826B9B80                 b       sub_826B8FF0
.text:826B9B80 # End of function sub_826B9B80
.text:826B9B80
```

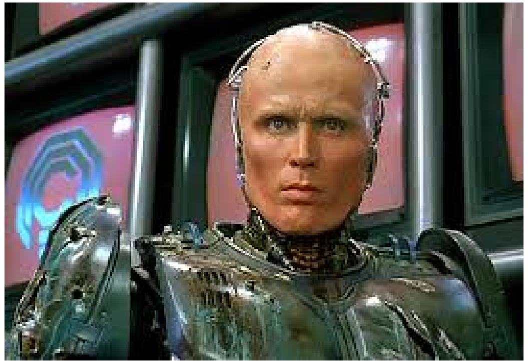
RSAConference2016

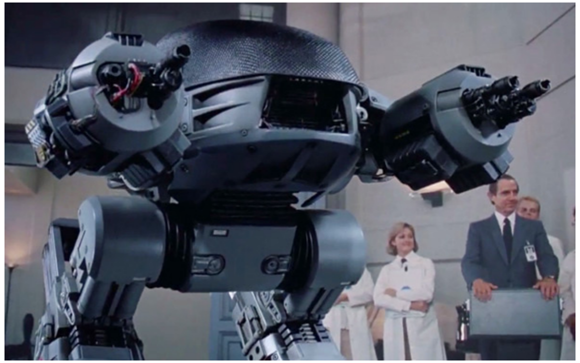
Hacking Binary Code

- Low abstraction level
- No structured types
- No modules or clearly defined functions
- Compiler optimization and other artifacts can make the code more complex to analyze
- WYSIWYE: What you see is what you execute

Finding Vulnerabilities

Human


Semi-Automated


Fully Automated

Manual Analysis

- “Look at the code and see what you can find”
- Requires substantial expertise
  - The analysis is as good as the person performing it
- Allows for the identification of complex vulnerabilities (e.g., logic-based)
- Expensive, does not scale

Tool-Assisted Analysis

- “Run these tools and verify/expand the results”
- Tools help in identifying areas of interest
  - By ruling out known code
  - By identifying potential vulnerabilities
- Since a human is involved, expertise and scale are still issues

Automated Analysis

- “Run this tool and find the vulnerability”
  - … and possibly generate an exploit...
  - ...and possibly generate a patch
- Requires well-defined models for the vulnerabilities
- Can only detect the vulnerabilities that are modeled
- Can scale (not always!)

Automated Vulnerability Analysis

- An algorithm that takes as input a code artifact (source code, byte-code, binary code) and identifies potential vulnerabilities
- The Halting Problem: “the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.”
  - https://en.wikipedia.org/wiki/Halting_problem
- Alan Turing proved that a general algorithm does not exist

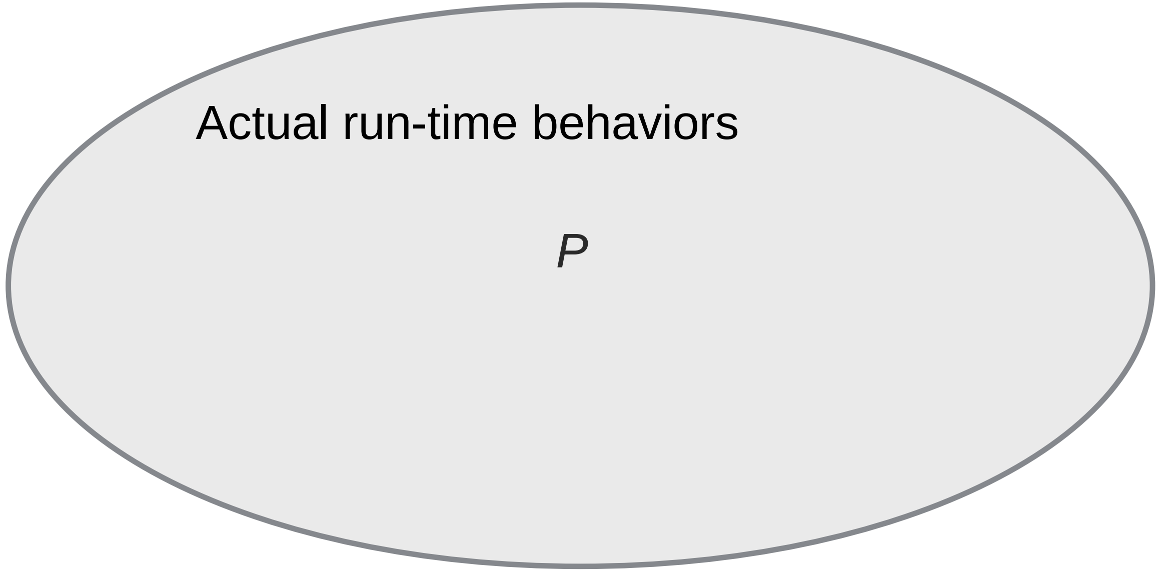
Types of Vulnerability Analysis

- Static Analysis
  - A form of abstract interpretation
  - Does not execute the code
- Dynamic Analysis
  - A form of concrete interpretation
  - Executes the code (or a model of it)

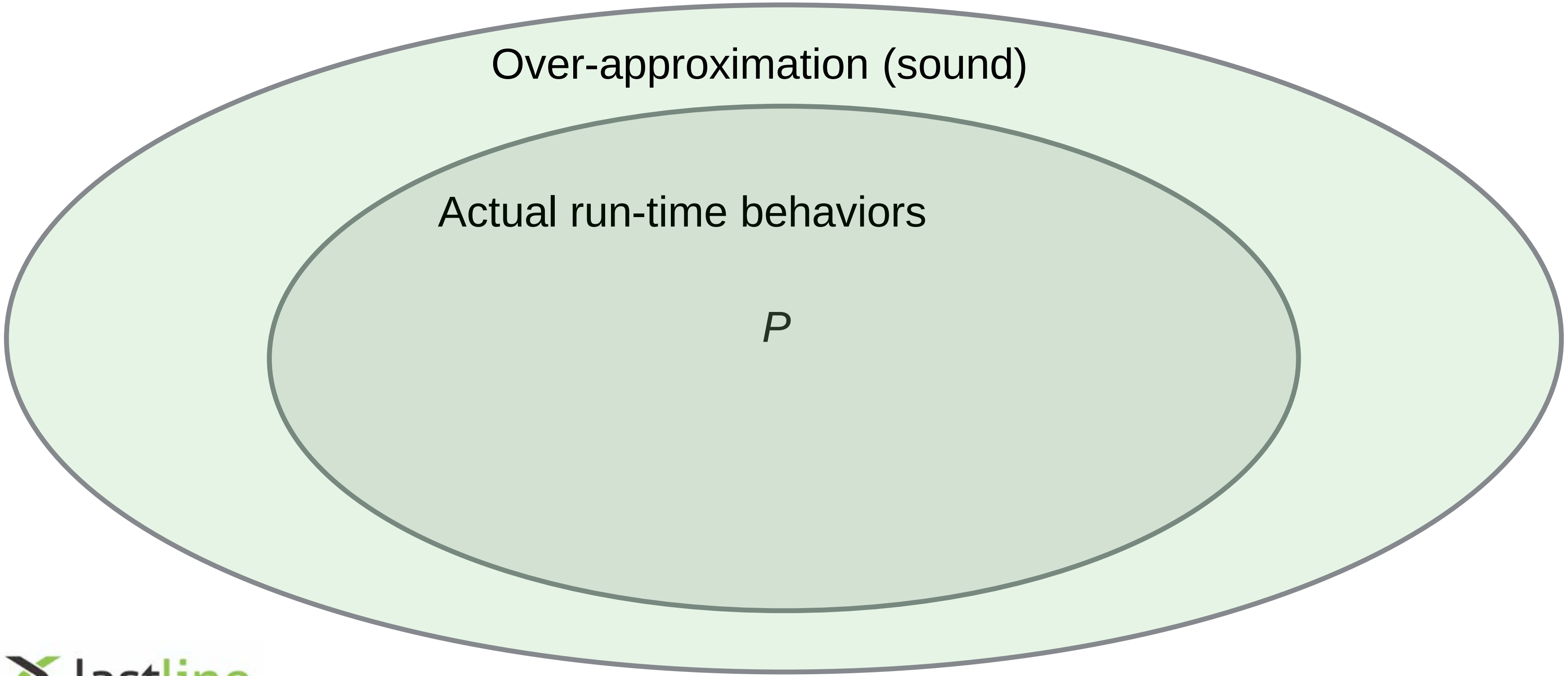
Static Analysis

- The goal of static analysis techniques is to characterize all possible run-time behaviors over all possible inputs without actually running the program
- Find possible bugs, or prove absence of certain kinds of vulnerabilities
- Static analysis has been around for a long while
  - Type checkers, compilers
  - Formal verification
- Challenges: soundness, precision, and scalability

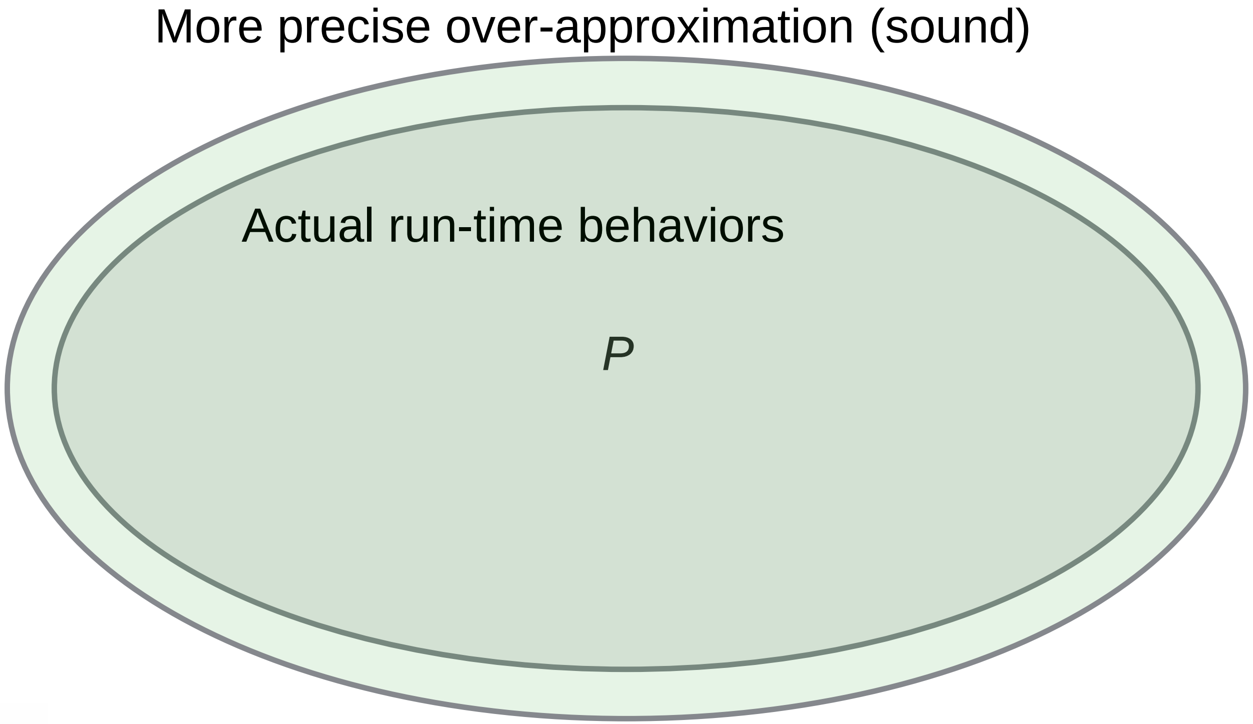
Soundness and Completeness



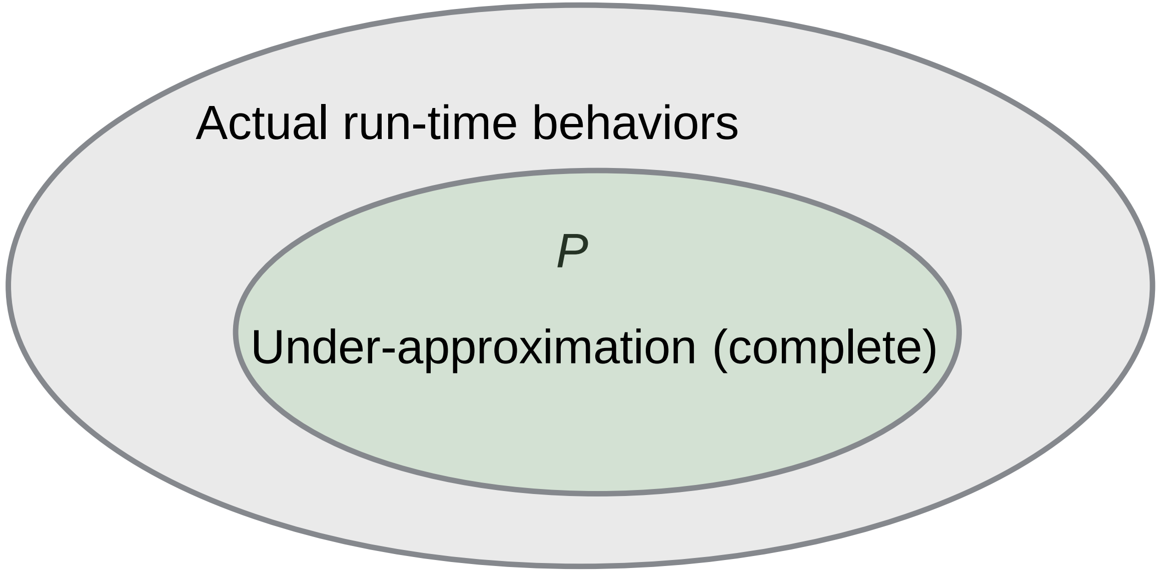
Soundness and Completeness



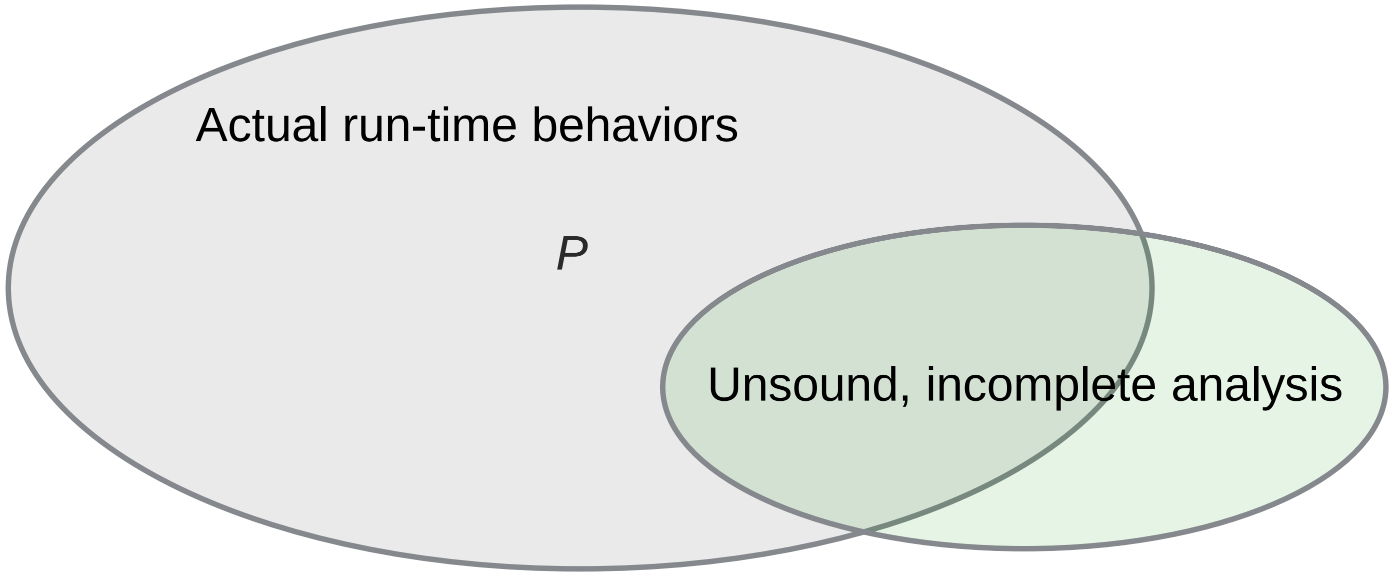
Soundness and Completeness


More precise over-approximation (sound)
Actual run-time behaviors
P

Soundness and Completeness



Soundness and Completeness



Example Analyses

- Control-flow analysis: Find and reason about all possible control-flow transfers (sources and destinations)
- Data-flow analysis: Reason about how data flows at run-time (from sources to sinks)
- Data dependency analysis: Reason about how data influences other data
- Points-to analysis: Reason about what values can pointers take
- Alias analysis: Determine if two pointers might point to the same address
- Value -set analysis: Reason about what are all the possible values that variables can hold

Dynamic Analysis

- Dynamic approaches are very precise for particular environment and inputs
  - You execute the code!
- However they provide no guarantee of coverage
  - You evaluate only the part of a program that you exercise!

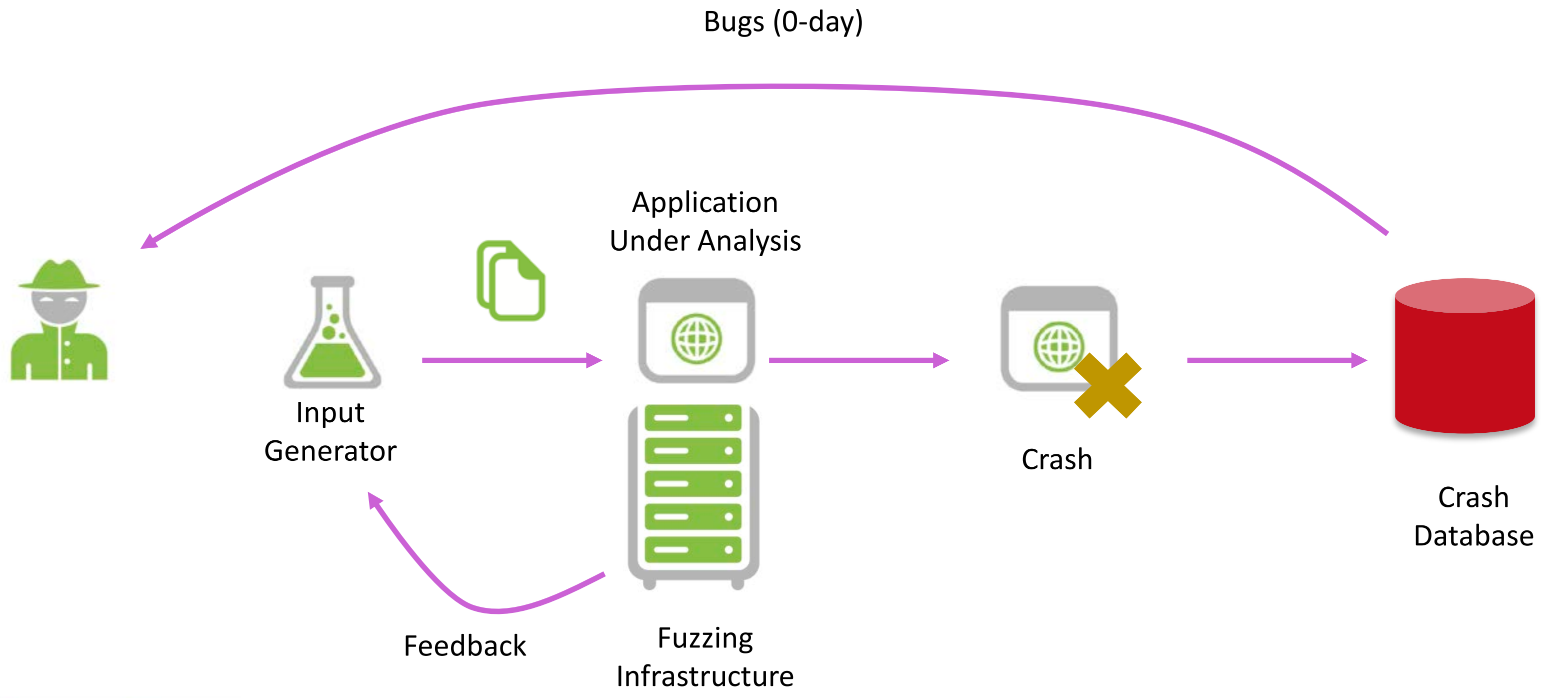
Example Analyses

- Taint analysis
- Fuzzing
- Forward symbolic execution
- Concolic execution

Fuzzing

- Fuzzing is an automated procedure to send inputs and record safety condition violations as crashes
  - Assumption: crashes are potentially exploitable
- Several dimensions in the fuzzing space
  - How to supply inputs to the program under test?
  - How to generate inputs?
  - How to generate more “relevant” crashes?
  - How to change inputs between runs?
- Goal: maximized effectiveness of the process

Gray/White-box Fuzzing



Fuzzing: American Fuzzy Lop

Fuzzing: American Fuzzy Lop

- Instrumentation-guided genetic fuzzer developed by Michael Zalewski
- The instrumentation collects information at branch points
  - Supports the generation of inputs that improve coverage
- Inputs that bring new paths are considered more interesting and queued for further exploration
- Inputs are chosen and mutated
- Unique crashes are identified using branch analysis (instead of stack summaries)

Symbolic Execution: angr

- Framework for the analysis of binaries
- Supports a number of architectures
  - x86 (32 and 64), MIPS, ARM, PPC, etc.
- http://angr.io
- https://github.com/angr
- angr@lists.cs.ucsb.edu

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

Symbolic Execution

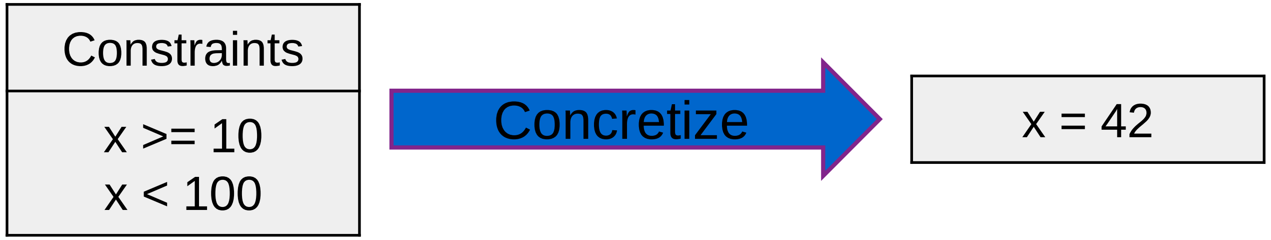
- "How do I trigger path X or condition Y?”
- Dynamic analysis
  - Input A? No. Input B? No. Input C? …
  - Based on concrete inputs to application
- (Concrete) static analysis
  - You can’t/You might be able to
  - Based on various static techniques

Symbolic Execution

- "How do I trigger path X or condition Y?"
- Interpret the application
- Track "constraints" on variables
- When the required condition is triggered, "concretize" to obtain a possible input

Concretization

- Constraint solving
- Conversion from set of constraints to set of concrete values that satisfy them



Example

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

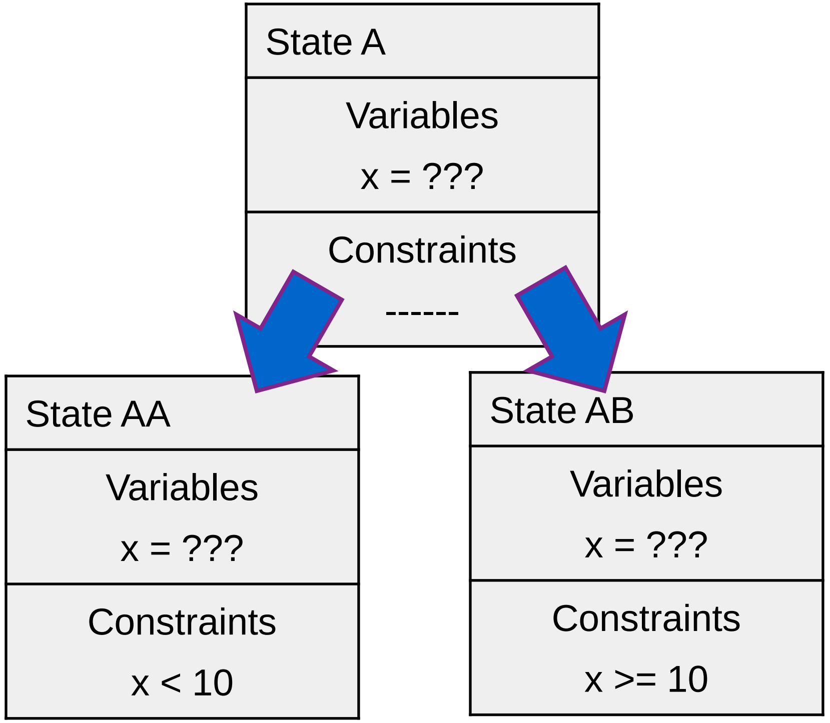
Example

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

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

Example

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



Example

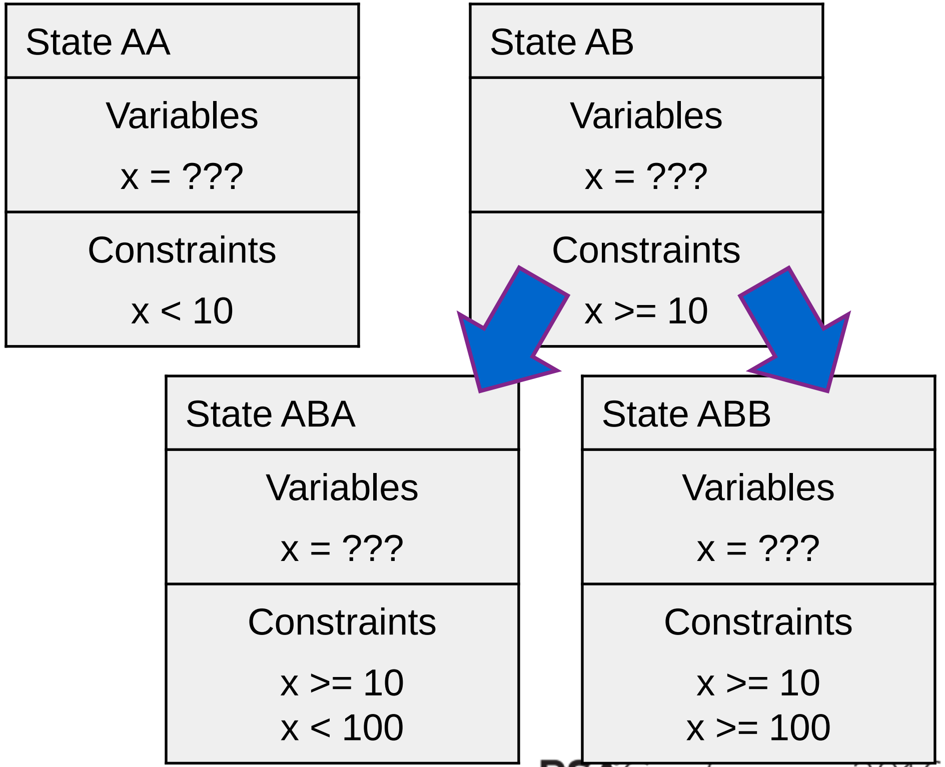
```
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```

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

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

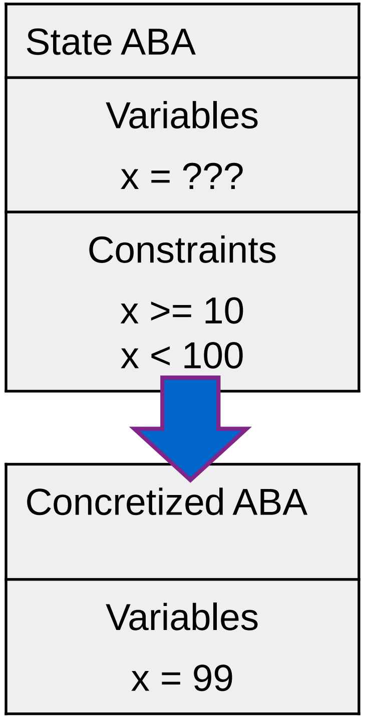
Example

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



Example

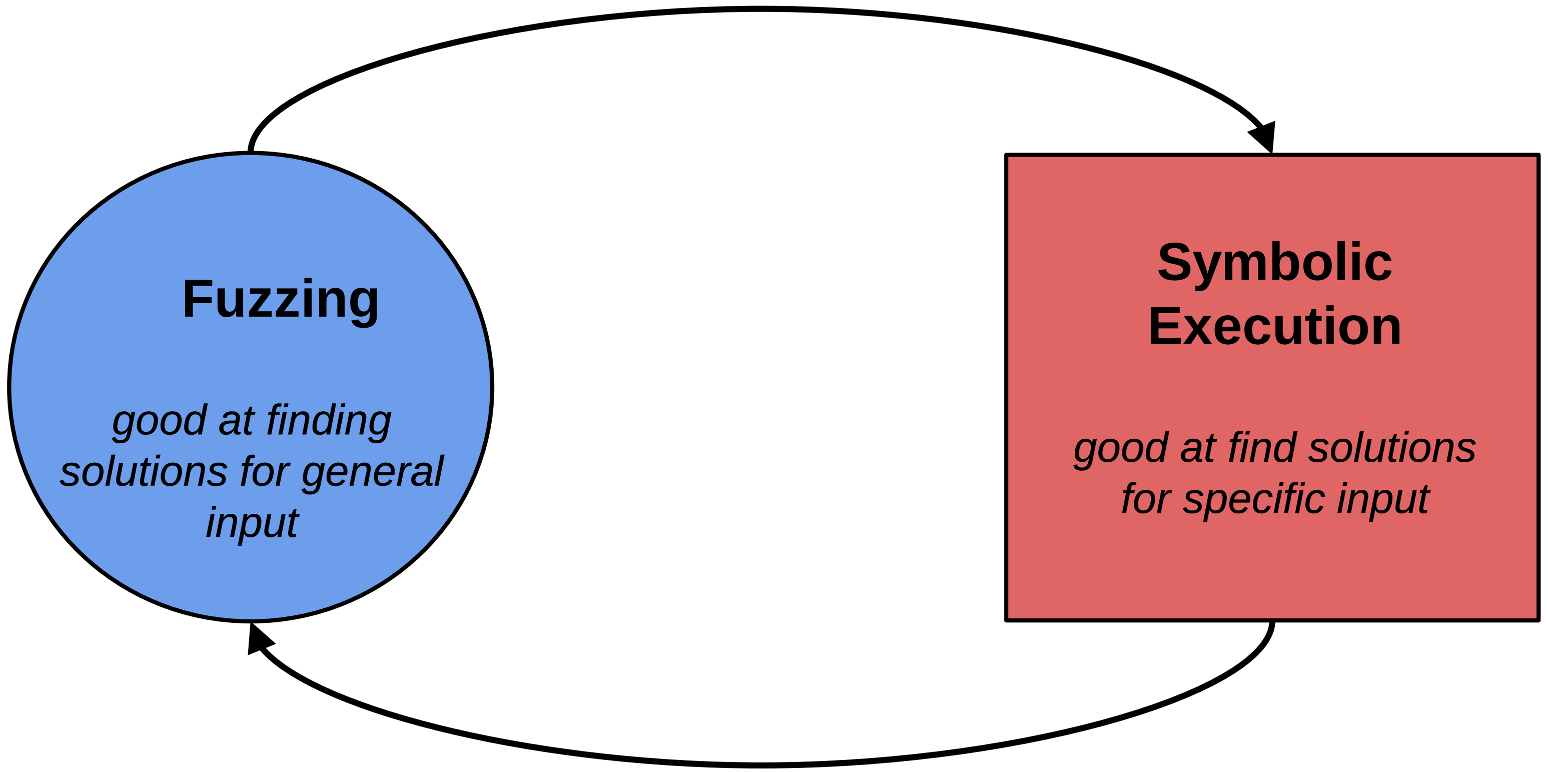
```
x = int(input())
if x >= 10:
    if x < 100:
      vulnerable_code()
    else:
      func_a()
else:
    func_b()
```



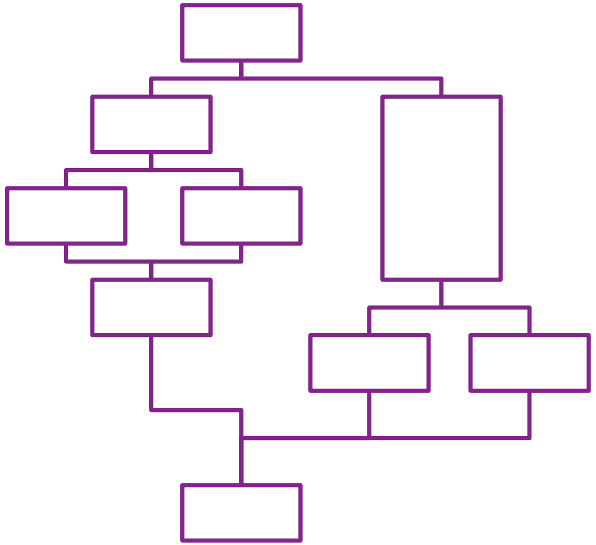
Putting It All Together

- Fuzzing excels at producing general input
- Symbolic execution is able to satisfy complex path predicates for specific input
- Key insight: Combine both techniques to leverage their strengths and mitigate their weaknesses

Assisting Fuzzing with Symbolic Execution

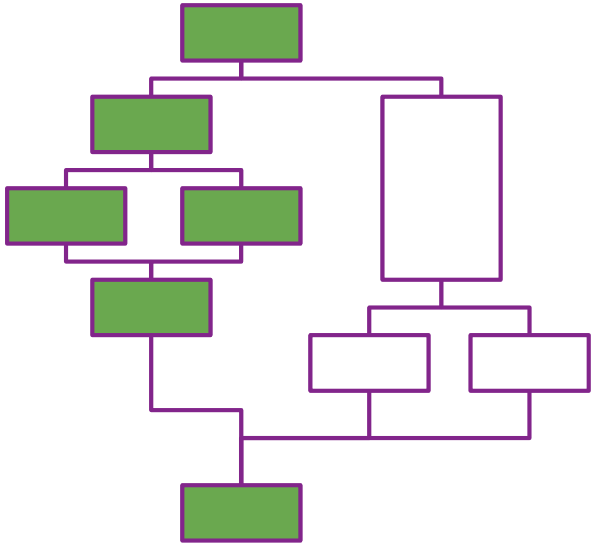

Fuzzing
good at finding solutions for general input
Symbolic Execution
good at find solutions for specific input

Driller



Driller

"Cheap" fuzzing coverage



Test Cases

"X"

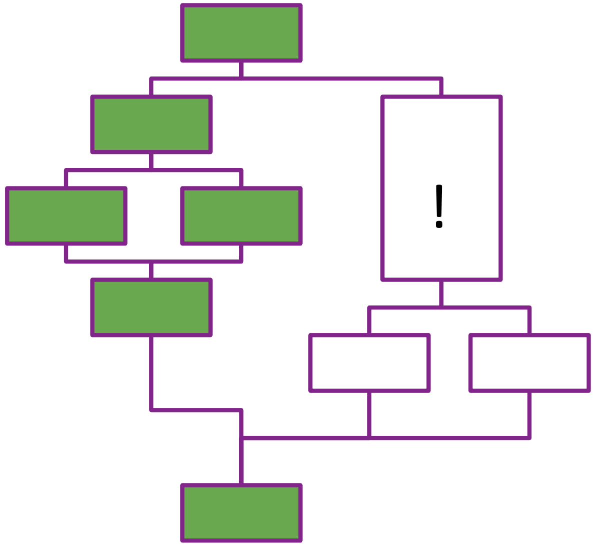
"Y"

Driller

“Cheap” fuzzing coverage

↓

Dynamic Symbolic Execution



Test Cases

“X”

“Y”

Driller

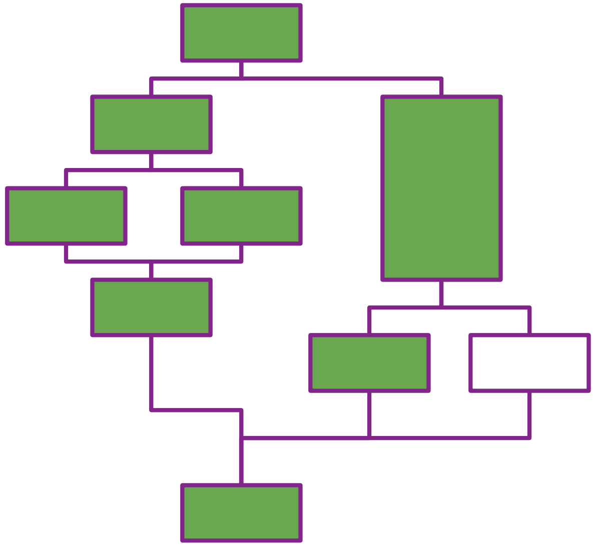
"Cheap" fuzzing coverage

↓

Dynamic Symbolic Execution

↓

New test cases generated



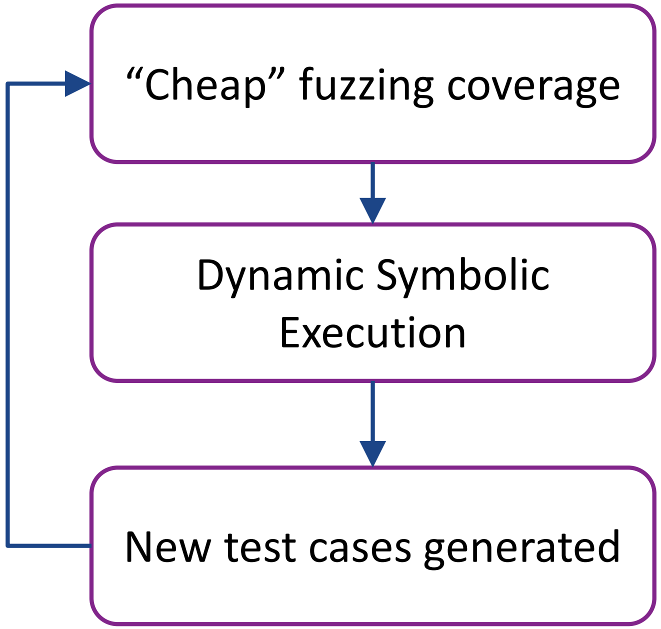
Test Cases

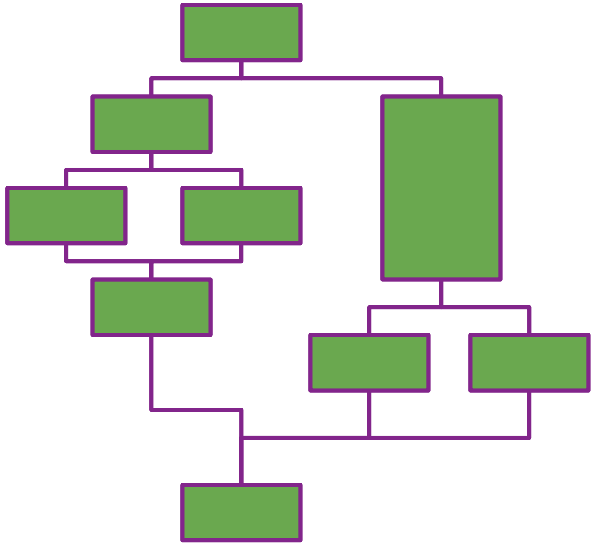
"X"

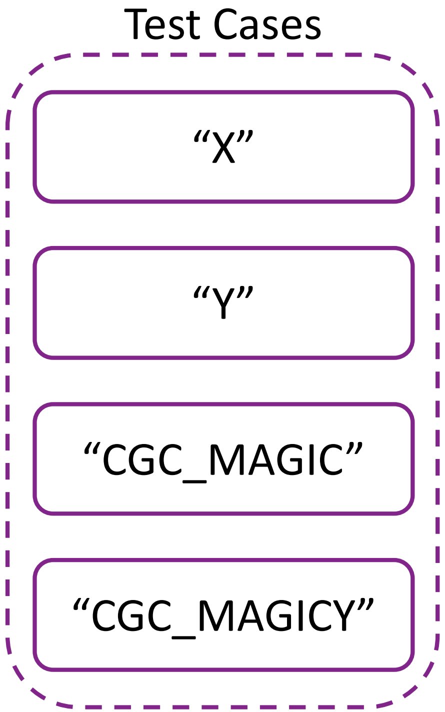
"Y"

"CGC_MAGIC"

Driller


"Cheap" fuzzing coverage
Dynamic Symbolic Execution
New test cases generated




Test Cases
"X"
"Y"
"CGC_MAGIC"
"CGC_MAGICY"

Why Hacking?

- Vulnerability analysis can be used
  - Offensively
  - Defensively
  - For fun (and profit)
- Hacking competitions have become a popular venue for the application of breakthrough techniques in vulnerability analysis
  - DefCon CTF
  - Pwn20wn

Many Competition Styles

Challenge-based

- Should not be called “CTF”!
- Easy to organize
- Easy to scale
- Exclusively focused on attacking
- No real-time component

Interactive, online CTFs

- Very difficult to organize
- Require substantial infrastructure
- Difficult to scale
- Focused on both attacking and defending in real time

Current Interactive, Online CTFs

- From ctftime.org: 100+ events listed
- Online attack-defense competitions:
  - UCSB iCTF 13 editions
  - RuCTF 5 editions
  - FAUST 1 edition

The iCTF Framework

- Lessons learned from running iCTFs were the basis for building a framework
- The framework formalizes the structure of services and allows for the reuse of the infrastructure
- Available at:
  - http://github.com/ucsb-seclab/ictf-framework
  - http://ictf.cs.ucsb.edu/framework

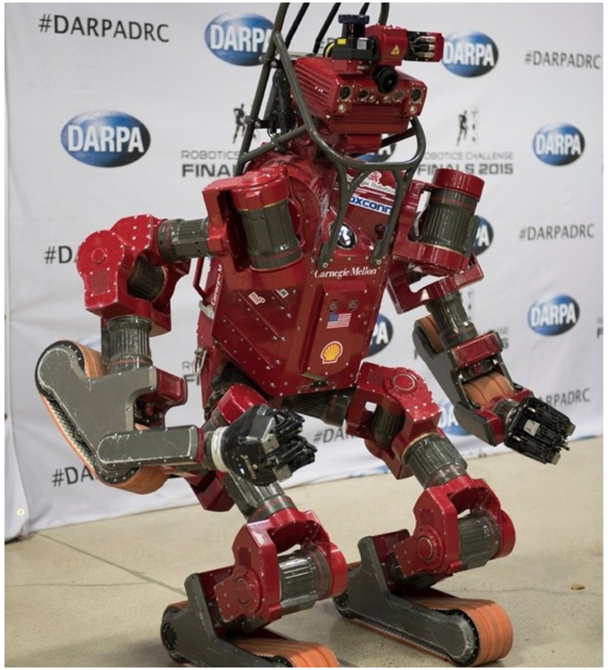
CTFs Are Playgrounds…

- For people (hackers)
- For tools (attack, defense)
- But can they be used to advance science?

DARPA Competitions

Self-driving Cars

Robots



The DARPA Cyber Grand Challenge

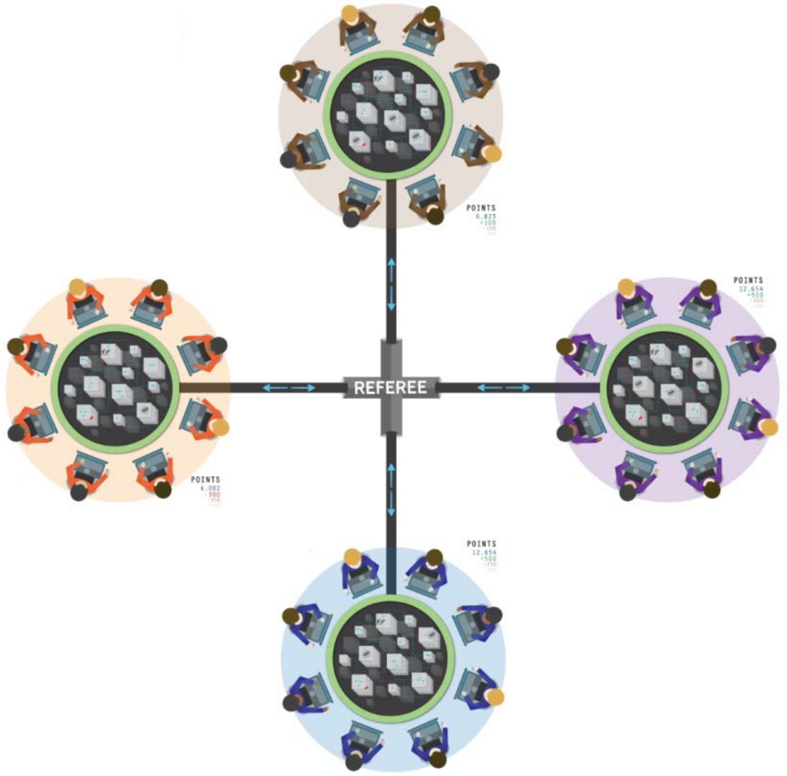
Programs!

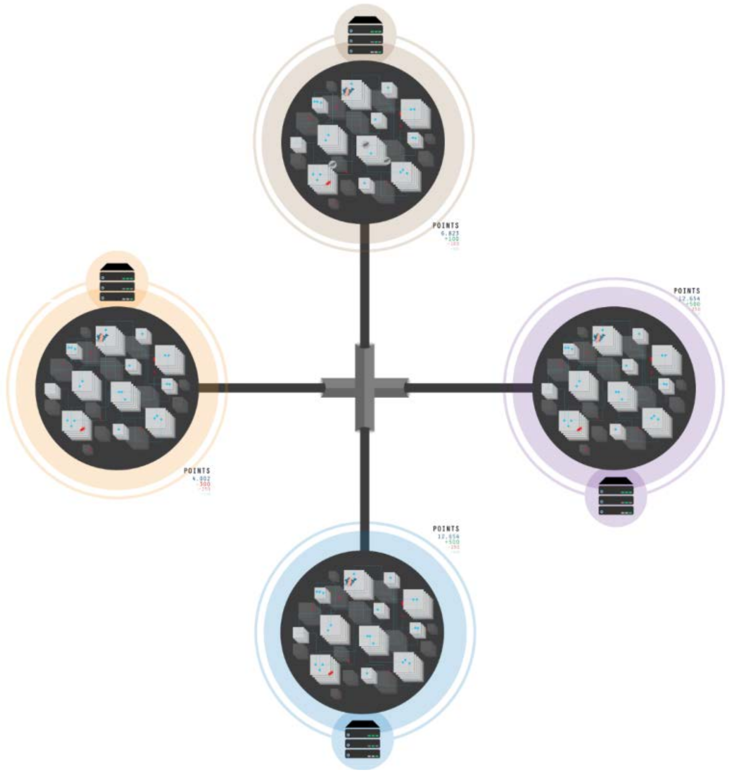
The DARPA Cyber Grand Challenge

CGC

CYBER
GRAND_CHALLENGE

The DARPA Cyber Grand Challenge





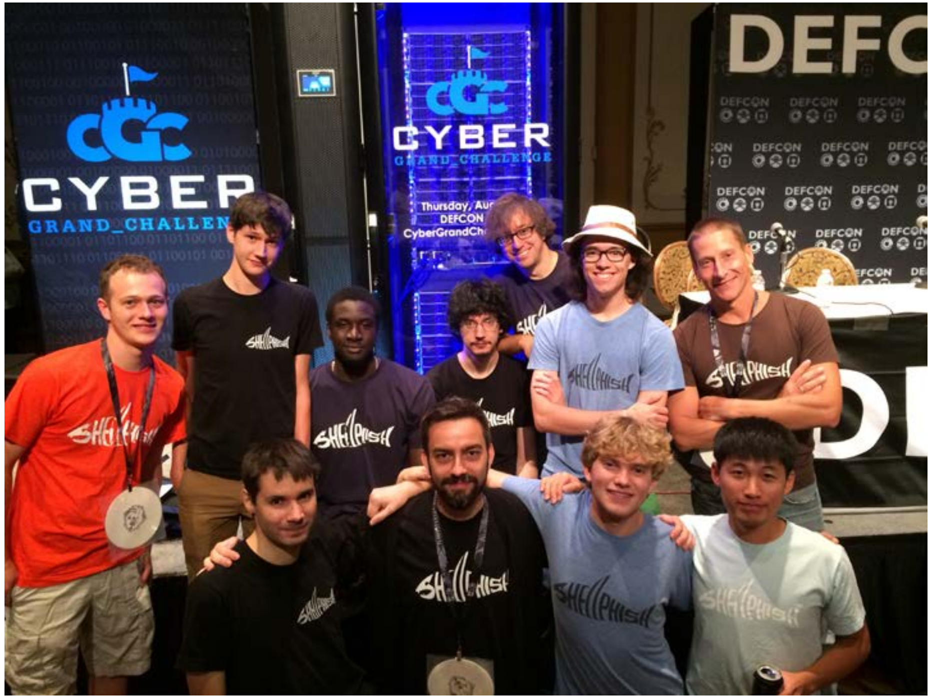
The DARPA Cyber Grand Challenge

- CTF-style competition
- Autonomous Cyber-Reasoning Systems (CRS) attack and defend a number of services (binary programs)
- NO HUMAN IN THE LOOP
- A first qualification round decided who the 7 finalists are
  - Qualification comes with a $750,000 cash prize
- The final event is scheduled for August 4, 2016 during DefCon
  - The top team will receive a $2,000,000 cash prize

Shellphish CGC Team

SHELLPHISH

SHELLPHISH



CGC Other Finalists

CodeJitsu

CSDS

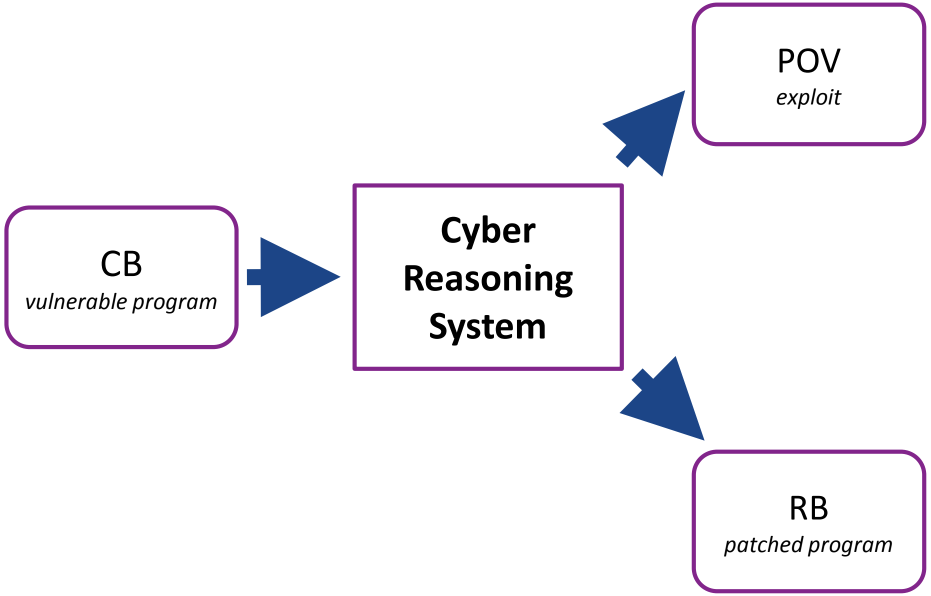
DeepRed

disekt

ForAllSecure

TECHx

CGC Participant Systems

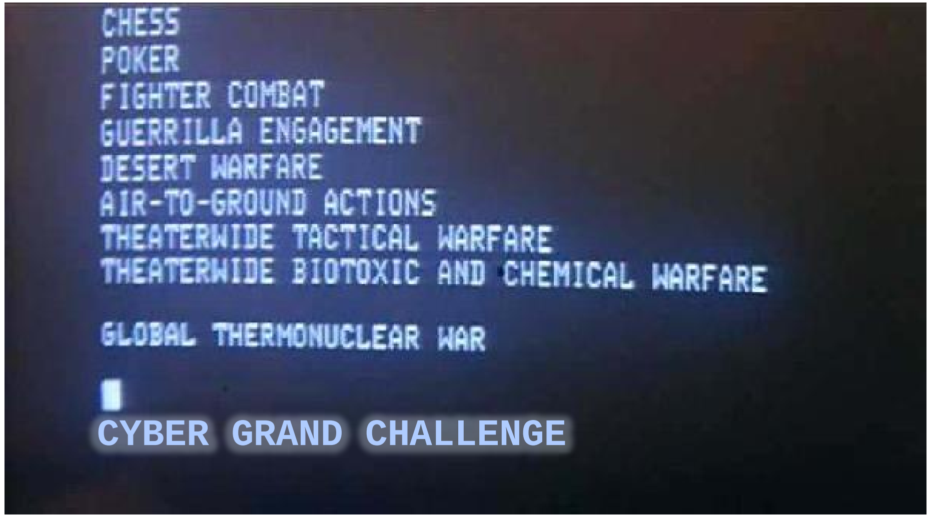


The CGC Environment

- Binaries run on a custom OS, called DECREE
  - Limited number of system calls
- A POV has to demonstrate the ability:
  - To read a specific value from memory
  - To set a register to a specific value
- Not all rules have been finalized

The Shellphish CRS: ShellWePlayAGame?

SHALL WE PLAY A GAME?

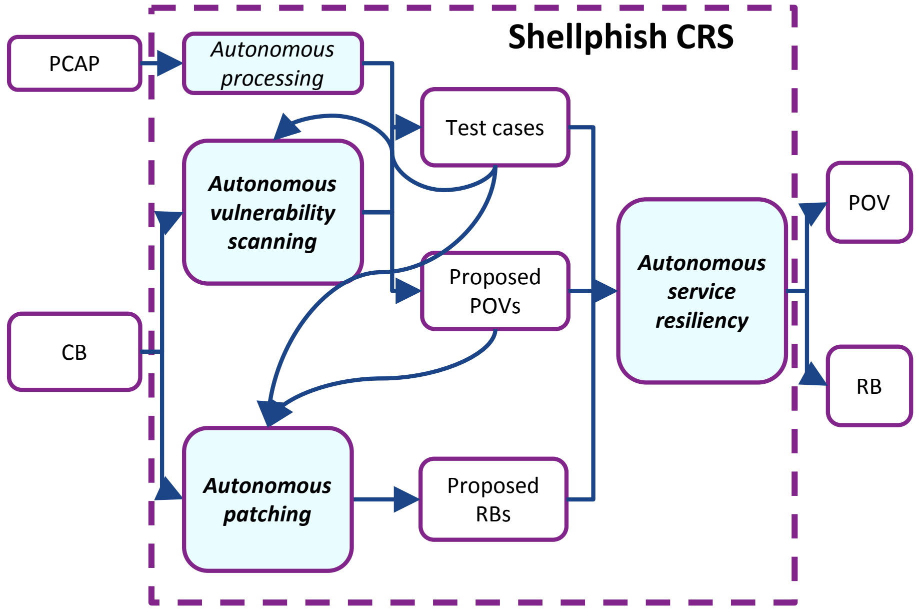


CHESS
POKER
FIGHTER COMBAT
GUERRILLA ENGAGEMENT
DESERT WARFARE
AIR-TO-GROUND ACTIONS
THEATERWIDE TACTICAL WARFARE
THEATERWIDE BIOTOXIC AND CHEMICAL WARFARE

GLOBAL THERMONUCLEAR WAR

CYBER GRAND CHALLENGE

The Shellphish CRS: ShellWePlayAGame?



May the Best CRS Win!

$$\sum_{i=0}^{\#CB} Availability \times Security \times Evaluation$$

- Patching cannot affect performance
- Patching cannot affect functionality
- When you are shooting blindfolded automatic weapons, it’s easy to shoot yourself in the foot...

Fostering Research in Automated Hacking

- The goal of the CGC is to foster the development of new attack and defense techniques that…
  - Automatically identify and exploit vulnerabilities in binary programs
  - Automatically patch vulnerability and provide functionally-equivalent yet secure versions of a vulnerable binary

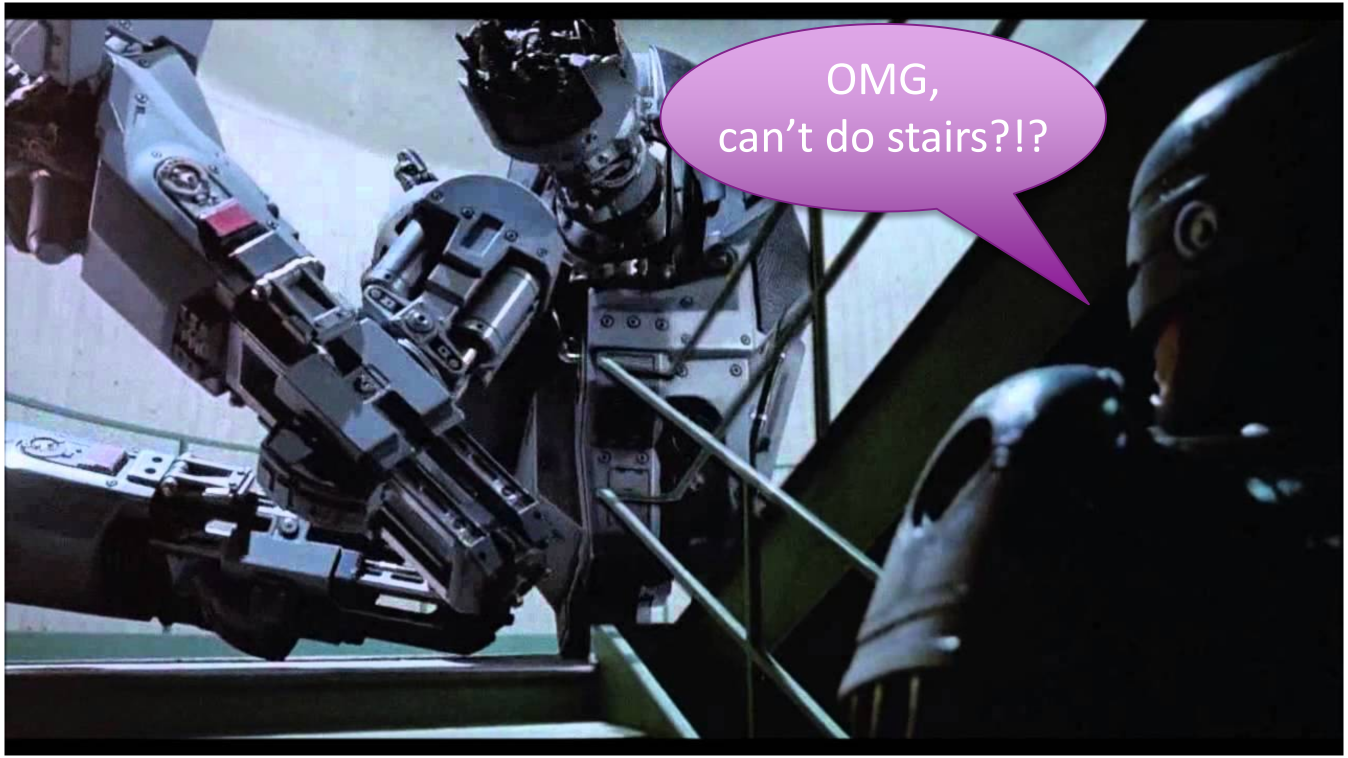
What Does All This Mean to YOU?

- Novel automated analysis techniques will allow for
  - The identification of vulnerabilities (and, possibly, backdoors) in binaries before they are deployed
  - The patching of binaries on-the-fly without having to wait for vendors' fixes
  - Scale…

What Can I Do NOW?

- Use CTFs (or other security competitions) to foster computer education in your company
- The iCTF Framework is free, open, and can be used to create sophisticated attack-defense security competition within your organization
- Familiarize yourself with vulnerability analysis tool and learn how to use them as integral part of your development process
- After all…

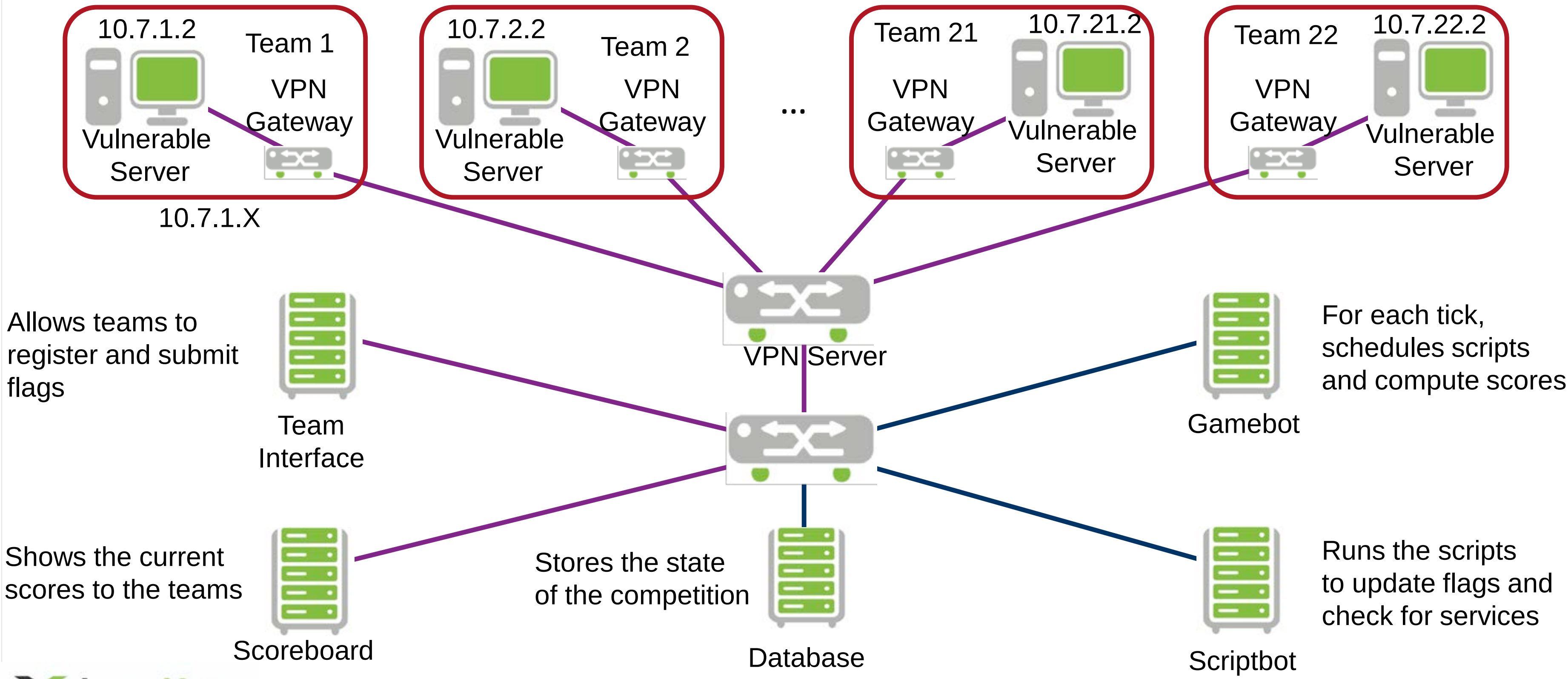
Human + Machine = WIN!



Q&A

Extra Slides

The iCTF Architecture

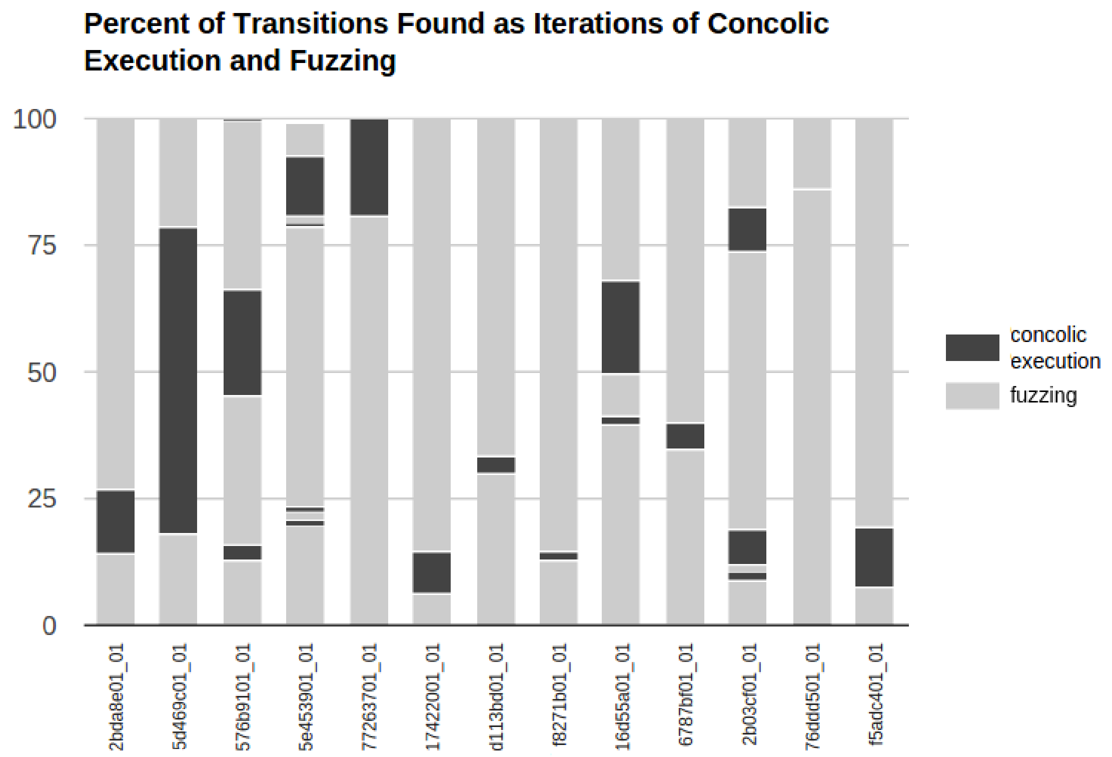


Example: Simple Overflows

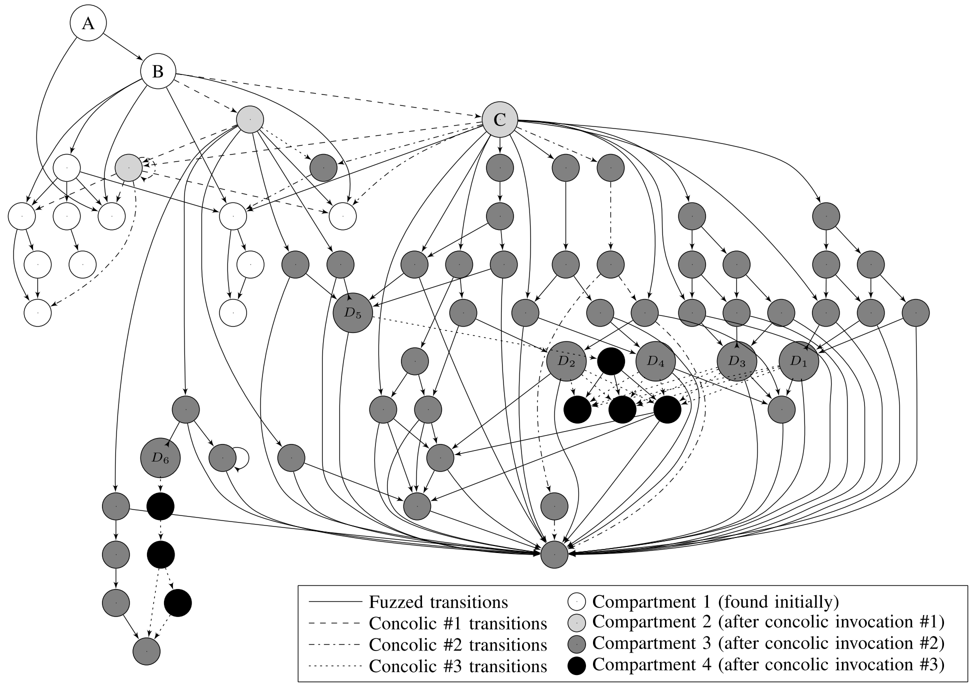
```
int main(int argc, char** argv) {
    char buf[256];
    strcpy(buf, argv[1]);
    return 0;
}
```

- Simplest detection approach: grep for strcpy
- More rigorous:
  - Determine data flow from command-line argument to strcpy's parameter
  - Determine size of source, destination buffers
  - Model semantics of strcpy
  - Check safety condition: len(argv[1]) < len(buf)

Fuzzing vs. Symbolic Execution



Exploration Process


A
B
C
D_1
D_2
D_3
D_4
D_5
D_6
Fuzzed transitions
Concolic #1 transitions
Concolic #2 transitions
Concolic #3 transitions
Compartment 1 (found initially)
Compartment 2 (after concolic invocation #1)
Compartment 3 (after concolic invocation #2)
Compartment 4 (after concolic invocation #3)

The UCSB iCTFs

Year	Theme	Teams
2003	Open-Source Windows	7
2004	UN Voting System	15
2004	Bass Tard Corporation	9
2005	Spam Museum	22
2006	Hillbilly Bank	25
2007	Copyright Mafia	36
2008	Softerror.com Terrorist Network	39
2009	Rise of the Botnet	57
2010	Rogue Nation of Litya	73
2011	Money Laundering	89
2012	SCADA Defense	92
2013	Nuclear Cyberwar	123
2014	Large-Scale Hacking	86
2015	Crowdsourced Evil	35

Branch Tracking

- The instrumentation collects information about which branches are taken
- The information is stored in a shared hash table. A branch from a previous location to the current location triggers the instrumentation code:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Branch Tracking

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

- Note that the index in the hash is a combination of the previous and current location
- The size of the shared memory is 64K
  - Big enough to avoid collisions
  - Small enough to be fast and fit in memory caches
- The shift of the marker for the current location allows for
  - Distinguishing A->B from B->A
  - Distinguishing A->A from B->B

Branch Tracking

- Branch tracking is a better metric for program exploration than plain basic block coverage
- Consider the following cases, where A, B, C, D, E are code blocks:
  - A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)
  - A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)
- While the same amount of code is covered, but different paths are taken

Guiding Exploration

- AFL maintains a global map of all the paths observed in all the executions up to the current one
- When a mutated input file introduces tuple that were not observed before, the input file is queued for further processing
- Inputs that do not generate new transitions, are discarded (even if the sequence has not been seen before)

Example

- #1: A -> B -> C -> D -> E
- #2: A -> B -> C -> A -> E (C->A, A->E are new)
- #3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E (no new tuples)

Counting Branches

- AFL keeps track of how many times a certain transition happens for each run
  - Buckets: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
- If a particular input causes a transition to move between buckets, then the input is deemed interesting and queues for processing
  - Buckets allow for emphasizing small changes (1 to 2) vs. not-so-relevant changes (67 to 70)

Processing Input

- The interesting file (thousands) are added to the input queue
  - Usually 10-30% from the discovery of new transitions
  - The rest from changes in the hit count
- The input queue is analyzed so that a subset of the (best) files is marked as “favorite”
  - The files cover all the tuples
  - The files have lowest latency and size

Prioritizing the Inputs

1. Choose a tuple from the ones observed so far and put it in a set
2. Select the input that caused the shortest execution and has the smallest size
3. Add all the transitions observed for that execution to the set
4. If the set does not covered all the previously observed transitions, goto 1

Prioritizing the Inputs

- If there are new, yet-to-be-fuzzed favorites present in the queue, 99% of non-favored entries will be skipped to get to the favored ones
- If there are no new favorites:
  - If the current non-favored entry was fuzzed before, it will be skipped 95% of the time
  - If the current non-favored entry was not fuzzed before, the odds of skipping it are 75%
- These values are chosen to balance queue cycling and diversity

Fuzzing Strategies

- Sequential bit flips with varying lengths and stepovers
- Sequential addition and subtraction of small integers
- Sequential insertion of known interesting integers (0, 1, INT_MAX, etc.)
- Stacked bit flips, insertions, deletions, arithmetic operations, and splicing of different test cases
- It is also possible to provide dictionaries of known keywords to help in the fuzzing process