InputFinder: Reverse Engineering Closed Binaries using Hardware Performance Counters

Bogdan Copos [1] Praveen Murthy [2]

[1]University of California, Davis

[2]Fujitsu Laboratories of America

November 30, 2015

Introduction

- Most program analysis techniques necessitate test suites
- Test suite availability and completeness issues
- Analysis dependencies:
  - platform
  - architecture
  - binary format
  - source code availability
  - ...

Background

Fuzzing

Black-box fuzzing

- Randomly generated input
- Advantages
  - easy to implement
  - previous success
- Disadvantages
  - poor code coverage
  - slow

Background

Fuzzing

White-box fuzzing

- Systematically generated input
- Advantages
  - improved effectiveness
- Disadvantages
  - limitations of constraint sovler (if symbolic execution is used)
  - source code dependencies (if instrumentation is used, i.e. afl-fuzz)

Background

Symbolic Execution

- Execution of program with symbols as input
- Build path conditions
- Constraint-solver limitations
  - non-linear constraints
  - scalability (path explosion)
- *Concolic execution*

Problem Statement

Platform, architecture, format, source-code **independent** method for generating input for programs.

Programs will normally

- read input

Programs will normally

- read input
- validate input

Programs will normally

- read input
- validate input
- process

Programs will normally

- read input
- validate input
- process
- change state (possibly)

Programs will normally

- read input
- validate input
- process
- change state (possibly)
- provide output (possibly)

Validation

The processor may perform additional work during input validation.

E.g.: *strcmp()* stops at first character mismatch.

Validation

The processor may perform additional work during input validation.

E.g.: *strcmp()* stops at first character mismatch.

Use this observation to identify valid input

Proposed Approach

Leverage information exposed by the hardware [performance counters] during the program's execution

1. Analyze the number of instructions retired to incrementally build input strings
2. Use discovered input and dynamically generated traces to identify the expected protocol

Assumptions:

- Instructions retired vary based on the validity of the input.

Assumptions:

- Instructions retired vary based on the validity of the input.
- Validation occurs in user-space

Assumptions:

- Instructions retired vary based on the validity of the input.
- Validation occurs in user-space
- Most printable characters are not valid at a given input string index.

Assumptions:

- Instructions retired vary based on the validity of the input.
- Validation occurs in user-space
- Most printable characters are not valid at a given input string index.
- Input is validated in order.

Assumptions:

- Instructions retired vary based on the validity of the input.
- Validation occurs in user-space
- Most printable characters are not valid at a given input string index.
- Input is validated in order.
- No inter dependencies between fields of a single input.

Assumptions:

- Instructions retired vary based on the validity of the input.
- Validation occurs in user-space
- Most printable characters are not valid at a given input string index.
- Input is validated in order.
- No inter dependencies between fields of a single input.
- Input is not altered prior to validation.

Finding Input

1. Start with an empty input string

Finding Input

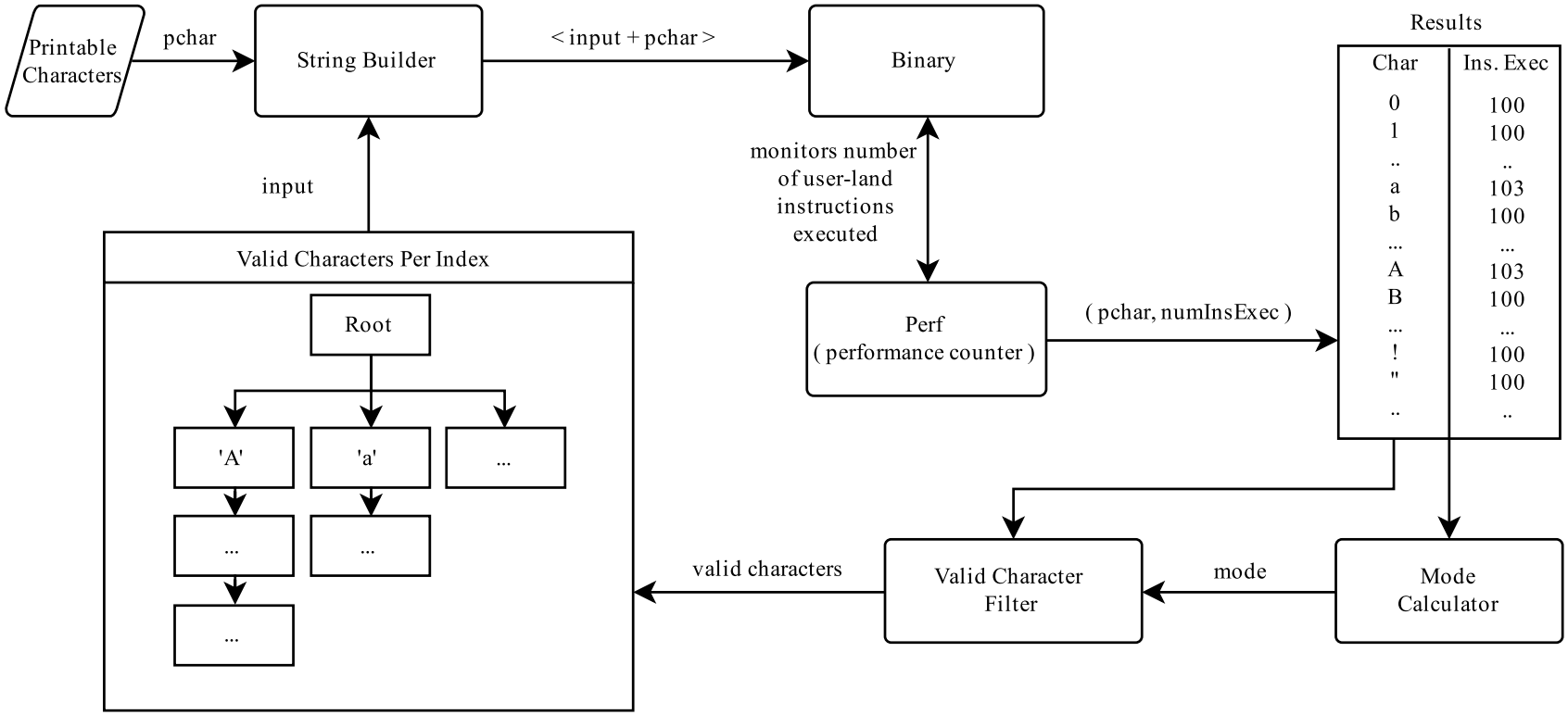
1. Start with an empty input string
2. Count the number of user-land instruction retired as the program is given as input each character from the set of printable characters

Finding Input

1. Start with an empty input string
2. Count the number of user-land instruction retired as the program is given as input each character from the set of printable characters
3. Calculate the mode number of user-land instructions

Finding Input

1. Start with an empty input string
2. Count the number of user-land instruction retired as the program is given as input each character from the set of printable characters
3. Calculate the mode number of user-land instructions
4. Valid characters for the current index of the input are characters for which the number of user-land instructions retired is not within the range of mode +/- threshold
5. Append one of the discovered valid characters to the current input string and repeat steps 2-5 (until no more valid characters are identified).


Printable Characters
pchar
String Builder
< input + pchar >
Binary
Results
Char
Ins. Exec
0 100
1 100
.. ..
a 103
b 100
... ...
A 103
B 100
... ...
! 100
" 100
.. ..
monitors number of user-land instructions executed
input
Valid Characters Per Index
Root
'A'
'a'
...
...
...
...
Perf (performance counter)
(pchar, numInsExec)
valid characters
Valid Character Filter
mode
Mode Calculator

Protocol State Machine Generation

To discover the expected protocol:

1. Generate every permutations of input strings
2. Discover arguments for input strings
3. Test permutations with instrumented binary
4. Compare execution traces

Execution Traces

Instrument binary using PIN to record every basic block (defined by its starting address) executed in order of execution

0x8049522
0x8049528
0x8049530
0x8049536
0x8049542
0x8049506
0x80480B3
0x904954D
0x80480BB
0x8049824
0x8049658
0x8049661
0x804966B
0x8049542
0x8049506
...

Example

Valid input strings: HELLO, AUTH, SET, CALL, BYE
Permutations: [[HELLO, AUTH], [HELLO, CALL], ... [HELLO, AUTH, SET, CALL, BYE]]
Arguments: check arguments for HELLO; send HELLO, check arguments for AUTH; send HELLO, check arguments for CALL ... send HELLO, AUTH, SET, CALL, check arguments for BYE

Evaluation

- 24 DARPA Cyber Grand Challenge (CGC) binaries
- 1 cyber security interview challenge (password cracking program)

DARPA Cyber Grand Challenge

- Architecture (32 bit, x86)
- Binary format
- System calls
  - *allocate()*
  - *deallocate()*
  - *terminate()*
  - *fdwait()*
  - *random()*
  - *receive()*
  - *transmit()*
- No standard library

Results

Binary	Input	Number of Inputs	Input Size	Crash Input	Protocol State Machine
06459301	yes (8 min)	8 out of 8	yes	no	-
06b71301	no	-	yes	no	-
07a9a901	-	-	yes	yes	-
0b32aa01	yes (2 min)	1 out of 1	yes	yes	-
11dc8e01	no	-	yes	no	-
1877a601	yes (11 min)	6 out of 6	yes	no	-
250d1101	yes (20 min)	5 out of 5	yes	no	yes
2eca0101	yes (42 min)	7 out of 7	no	no	-
37e97201	yes (38 min)	6 out of 6	yes	no	-
3dcf1a01	yes (6 min)	6 out of 6	yes	yes	-
48b9cf01	yes (50 min)	9 out of 9	yes	no	-
65884701	no	-	no	no	-
701b7301	-	-	yes	no	-
7262d006	yes (5 min)	4 out of 5	yes	no	-
7fa39f01	no	-	yes	yes	-
b8993403	yes (12 min)	1 out of 1	yes	no	-
badd9e01	no	-	yes	no	-
caea9c01	no	-	yes	no	-
cc366801	yes (60 min)	10 out of 10	no	no	-
d476cd01	no	-	yes	no	-
df9df201	no	-	yes	no	-
e7cd3901	-	-	no	yes	-
f14eb101	yes (53 min)	11 out of 11	no	no	yes
f658d801	yes (2 min)	1 out of 4	yes	no	-
interview challenge	yes (2 min)	1 out of 1	yes	-	-

Results

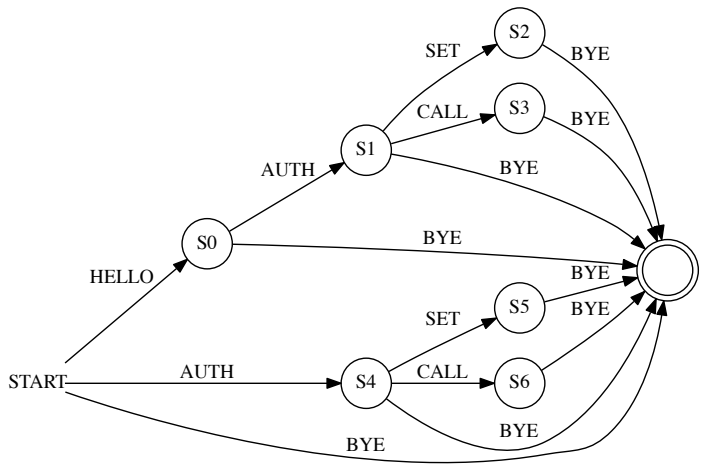
- CGC
  - Binaries with well defined input: 21 out of 24
  - Found some input strings 13 out of 21 (62%)
  - Found all input strings 11 out of 13 (85%)
- Interview challenge password cracked

Case Study

250d1101

Binary implements a simple protocol that allows a user perform *root64* encoding/decoding and utilize *parcour* schemes.

- 5/5 input strings discovered
- Correct protocol identified, including hidden authentication backdoor



Future Work

- Eliminate the "majority" constraint
- Use hardware counters for protocol state machine generation