ANTIFUZZ: Impeding Fuzzing Audits of Binary Executables

Usenix Security 2019, Santa Clara
August 16, 2019

Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz

Chair for Systems Security
Ruhr-Universität Bochum
Motivation
But what about automated bug finding tools?
Motivation

Trusted Party
  • Can find bugs

Untrusted Party
  • Can’t find bugs

But what about automated bug finding tools?
Motivation

**Trusted Party**
- Can examine code $\implies$ Can find bugs

**Untrusted Party**
- Can’t examine code $\implies$ Can’t find bugs?
Motivation

**Trusted Party**
- Can examine code $\implies$ Can find bugs

**Untrusted Party**
- Can’t examine code $\implies$ Can’t find bugs?
- But what about automated bug finding tools?
Impeding Fuzzing Audits
Impeding Fuzzing Audits

- Analyze diverse set of fuzzers
Impeding Fuzzing Audits

- Analyze diverse set of fuzzers
- Find assumptions fuzzers need to make
Impeding Fuzzing Audits

- Analyze diverse set of fuzzers
- Find assumptions fuzzers need to make
- Invalidate those assumptions
### Assumptions

<table>
<thead>
<tr>
<th>Assumption</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>KAFL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>AFLFast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>CollAFL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>AFLGo</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>WinAFL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Steelix</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>RedQueen</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Honggfuzz</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Vuzzizer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Driller</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Klee</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>ZZUF</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Peach</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>QSym</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>T-Fuzz</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Angora</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Radamsa</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Libfuzzer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>
Demo Application

main

in == "ELF"

"not ELF" crash
Assumptions
(A) Coverage Yields Relevant Feedback
Coverage Yields Relevant Feedback

Blind Fuzzer

Coverage-guided Fuzzer
Coverage Yields Relevant Feedback

Blind Fuzzer

- Mutate input

Coverage-guided Fuzzer

- Mutate input
Coverage Yields Relevant Feedback

Blind Fuzzer
  - Mutate input
  - See if it crashes with given input

Coverage-guided Fuzzer
  - Mutate input
  - See if it crashes with given input

\[ \text{New Coverage} = \Rightarrow \text{New Behavior} \]
Coverage Yields Relevant Feedback

Blind Fuzzer
- Mutate input
- See if it crashes with given input

Coverage-guided Fuzzer
- Mutate input
- See if it crashes with given input
- ... or if new coverage is found

New Coverage $\Rightarrow$ New Behavior
Coverage Yields Relevant Feedback

Blind Fuzzer
  • Mutate input
  • See if it crashes with given input

Coverage-guided Fuzzer
  • Mutate input
  • See if it crashes with given input
  • ... or if new coverage is found
  • If so, add input to queue
Coverage Yields Relevant Feedback

Blind Fuzzer

• Mutate input
• See if it crashes with given input

Coverage-guided Fuzzer

• Mutate input
• See if it crashes with given input
• ... or if new coverage is found
• If so, add input to queue
• New Coverage $\implies$ New Behavior
Coverage Yields Relevant Feedback

main
in == "ELF"
"not ELF" crash
(B) Crashes Can Be Detected
Crashes Can Be Detected

```
main
in == "ELF"
"not ELF" crash
exit()
```
(C) Many Executions Per Second
Why is AFL so good?
Many Executions Per Second

Why is AFL so good?

- No ”human knowledge” about target necessary
Why is AFL so good?

- No "human knowledge" about target necessary
- Super fast implementation (thousands of executions per second)
Why is AFL so good?

- No "human knowledge" about target necessary
- Super fast implementation (thousands of executions per second)
- → As long as we are fast, we don’t need to be smart.
Bad approach
Bad approach

• Slow down application
Bad approach

- Slow down application
- But: real-world usage also slows down
Bad approach

- Slow down application
- But: real-world usage also slows down

Better approach
Bad approach
  • Slow down application
  • But: real-world usage also slows down

Better approach
  • Slow down application ...
Many Executions Per Second

Bad approach

• Slow down application
• But: real-world usage also slows down

Better approach

• Slow down application ...
• ... only when it’s being fuzzed?
What is the implication of being fuzzed?
What is the implication of being fuzzed?

- Most inputs will be malformed
What is the implication of being fuzzed?

- Most inputs will be malformed
- But in real-world scenarios, most inputs are well-formed
What is the implication of being fuzzed?

- Most inputs will be malformed
- But in real-world scenarios, most inputs are well-formed

Solution: slow down application if input is malformed
Many Executions Per Second

```
main
in == "ELF"
"not ELF"
crash
exit()
sleep()
```

17
(D) Constraints Are Solvable with Symbolic Execution
Why use symbolic execution?
Why use symbolic execution?

- Some constraints are too hard to solve via random mutations
Why use symbolic execution?

- Some constraints are too hard to solve via random mutations
- Let’s get help from symbolic execution
Why use symbolic execution?

- Some constraints are too hard to solve via random mutations
- Let’s get help from symbolic execution

Assumption: some constraints are too hard to be solved by random mutations alone, but could be solved by symbolic execution
How to break this assumption? Two techniques:
How to break this assumption? Two techniques:

- Replace constants comparisons by hash comparisons
Constraints Are Solvable with Symbolic Execution

How to break this assumption? Two techniques:

- Replace constants comparisons by hash comparisons
- Put input through encryption and decryption before using
Constraints Are Solvable with Symbolic Execution

```
main
in == "ELF"
"not ELF" crash
exit()
sleep()
h(in) == a27b...
dec(enc(in))
```
Evaluation
<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>Coverage</th>
<th>Crash</th>
<th>Speed</th>
<th>Symbolic Exec.</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Honggfuzz</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Vuzzer</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Driller</td>
<td>✗</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Klee</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓^a</td>
</tr>
<tr>
<td>zzuf</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Peach</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>QSYM</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓ = No crashes were found
Coverage Evaluation

**objdump**

![Box plots showing coverage evaluation for Plain and Protected modes for Vuzzer, AFL, Hongg, and QSYM.](image)

- **Plain** mode shows a significant variation in the number of branches covered for different tools.
- **Protected** mode does not show any branches covered, indicating that the protection mechanism is effective.

The tools display different levels of coverage, with Vuzzer and AFL showing higher coverage compared to Hongg and QSYM.
Conclusion

- Systematic analysis reveals: contemporary fuzzers rely on four core assumptions

- Coverage Yields Relevant Feedback
- Crashes Can Be Detected
- Many Executions Per Second
- Constraints Are Solvable With Symbolic Execution

AntiFuzz breaks these assumptions to impede fuzzing attempts

https://github.com/RUB-SysSec/antifuzz
Conclusion

- Systematic analysis reveals: contemporary fuzzers rely on four core assumptions
  - Coverage Yields Relevant Feedback
  - Crashes Can Be Detected
  - Many Executions Per Second
  - Constraints Are Solvable With Symbolic Execution
Conclusion

• Systematic analysis reveals: contemporary fuzzers rely on four core assumptions
  • Coverage Yields Relevant Feedback
  • Crashes Can Be Detected
  • Many Executions Per Second
  • Constraints Are Solvable With Symbolic Execution

• **AntiFuzz** breaks these assumptions to impede fuzzing attempts
Conclusion

- Systematic analysis reveals: contemporary fuzzers rely on four core assumptions
  - Coverage Yields Relevant Feedback
  - Crashes Can Be Detected
  - Many Executions Per Second
  - Constraints Are Solvable With Symbolic Execution
- **AntiFuzz** breaks these assumptions to impede fuzzing attempts
- [https://github.com/RUB-SysSec/antifuzz](https://github.com/RUB-SysSec/antifuzz)
Q & A