FUZZIFICATION: Anti-Fuzzing Techniques

Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee*, Taesoo Kim
Fuzzing Discovers Many Vulnerabilities

50 CVEs in 50 Days: Fuzzing Adobe Reader

December 12, 2018
Research By: Yoav Alon, Netanel Ben-Simon
Fuzzing Discovers Many Vulnerabilities

50 CVEs in 50 Days: Fuzzing Adobe Reader

December 12, 2018
Research By: Yoav Alf

Google’s automated fuzz bot has found over 9,000 bugs in the past two years
Google improves OSS-Fuzz service, plans to invite new open source projects to join.
Testers Find Bugs with Fuzzing

Source → Compilation → Released binary

Normal users → Detected bugs

Testers → Detected bugs

Compilation Distribution Fuzzing
But Attackers Also Find Bugs 😞

- Source
- Compilation
- Released binary
- Attackers
- Normal users
- Testers
- Detected bugs

Compilation | Distribution | Fuzzing
Our work: Make the Fuzzing Only Effective to the Testers

Source → Fuzzification → Fortified binary

Compilation → Binary

Detected bugs

Normal users → Attackers

Testers

Compilation → Distribution → Fuzzing
Threat Model

Source ➔ Fuzzification ➔ Fortified binary ➔ Normal users ➔ Detected bugs

Source ➔ Compilation ➔ Binary ➔ Attackers ➔ Detected bugs

Source ➔ Compilation ➔ Binary ➔ Testers ➔ Detected bugs

Compilation ➔ Distribution ➔ Fuzzing
Threat Model

Adversaries try to find vulnerabilities from fuzzing.
Adversaries only have a copy of fortified binary
Threat Model

Adversaries know Fuzzification and try to nullify
Research Goals

- Source
- Fuzzification
- Compilation
- Fortified binary
- Detected bugs
- Normal users
- Attackers
- Testers
- Compilation
- Distribution
- Fuzzing
Research Goals

Hinder Fuzzing
Reduce the number of detected bugs
Research Goals

- **Fuzzification**
- **Compilation**
- **Fortified binary**
- **Binary**
- **Normal users**
- **Attackers**
- **Testers**
- **Detected bugs**
- **AFL**
- **HonggFuzz**
- **QSym**
- **VUzzer**
- **…**

**Generic**

**Affect most of the fuzzers**
Research Goals

Overhead

Low overhead to normal user

High overhead to attackers
Research Goals

Resiliency

Resilient to the adversarial analysis
Why Existing Methods Are Not Applicable?

<table>
<thead>
<tr>
<th>Method</th>
<th>Generic to most fuzzers</th>
<th>Low overhead</th>
<th>Resilient to adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packing or obfuscation</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
</tbody>
</table>
## Why Existing Methods Are Not Applicable?

<table>
<thead>
<tr>
<th>Method</th>
<th>Generic to most fuzzers</th>
<th>Low overhead</th>
<th>Resilient to adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packing or obfuscation</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>Bug injection</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
</tbody>
</table>
## Why Existing Methods Are Not Applicable?

<table>
<thead>
<tr>
<th>Method</th>
<th>Generic to most fuzzers</th>
<th>Low overhead</th>
<th>Resilient to adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packing or obfuscation</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>Bug injection</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Fuzzer detection</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
</tbody>
</table>
## Why Existing Methods Are Not Applicable?

<table>
<thead>
<tr>
<th>Method</th>
<th>Generic to most fuzzers</th>
<th>Low overhead</th>
<th>Resilient to adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packing or obfuscation</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>Bug injection</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Fuzzer detection</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Emulator detection</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
</tbody>
</table>
## Why Existing Methods Are Not Applicable?

<table>
<thead>
<tr>
<th>Method</th>
<th>Generic to most fuzzers</th>
<th>Low overhead</th>
<th>Resilient to adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packing or obfuscation</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>Bug injection</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Fuzzer detection</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Emulator detection</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Fuzzification</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
Fuzzification Hinders Advanced Features

• Fast execution

• Coverage-guidance

• Hybrid approach
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- Hybrid approach
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- Hybrid approach
Fuzzification Hinders Advanced Features

- Fast execution
- **Coverage-guidance**
- Hybrid approach

![Diagram](image-url)
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- Hybrid approach
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- Hybrid approach
SpeedBump: Selective Delay Injection

Basic block
SpeedBump: Selective Delay Injection

- Identify *frequently* and *rarely* visited paths

Diagram:
- Basic block
- Rarely visited path
- Frequently visited path
**SpeedBump: Selective Delay Injection**

- Identify **frequently** and **rarely** visited paths

- Inject delays from the most **rarely** visited edges
SpeedBump: Selective Delay Injection

- Why this is effective?
  - User: follows common paths
  - Attacker: searches for new paths

→ Impact of delay is more significant to attackers
SpeedBump: How to delay?

• Strawman: using sleep()
  ➔ trivially removed by adversary
SpeedBump: How to delay?

• Strawman: using sleep()
  ➔ trivially removed by adversary

• Counter to advanced adversary
  ▫ Use randomly generated code
    ➔ avoid static-pattern
SpeedBump: How to delay?

• Strawman: using sleep()
  ➔ trivially removed by adversary

• Counter to advanced adversary
  ▫ Use randomly generated code
    ➔ avoid static-pattern
  ▫ Impose control-flow and data-flow dependency
    ➔ avoid automated analysis
SpeedBump: Selective Delay Injection

```c
int rarely_executed_code ()
{
    return 0;
}
```
SpeedBump: Selective Delay Injection

```c
int rarely_executed_code ()
{
    return 0;
}

//define global variables
int global1 = 1;
int global2 = 2;

int rarely_executed_code ()
{
    //inject delay function
    int pass = 20;
    global2 = func(pass);
    return 0;
}
```
int rarely_executed_code ()
{
    return 0;
}

// define global variables
int global1 = 1;
int global2 = 2;

int rarely_executed_code ()
{
    // inject delay function
    int pass = 20;
    global2 = func(pass);
    return 0;
}

int func(int p6) {
    int local1[10];

    // affect global1 variable
    global1 = 45;
    int local2 = global1;
    for (int i = 0; i < 1000; i++)
        // affect local1 variable
        local1[i] = p6 + local2 + i;

    // affect global2 variable
    return local1[5];
}
BranchTrap Hinders Coverage Management

• Fast execution
• Coverage-guidance
• Hybrid approach
BranchTrap#1: Fabricates Input-sensitive Paths

Coverage #1

“AAAA”
BranchTrap#1: Fabricates Input-sensitive Paths

Coverage #1

Coverage #2
BranchTrap#1: Fabricates Input-sensitive Paths

Coverage #1

Coverage #2

Coverage #1
BranchTrap#1: Fabricates Input-sensitive Paths

"AAAA"  "AAAB"

Coverage #1  Coverage #2

BranchTrap

"AAAA"  "AAAB"

Coverage #1  Coverage #2
BranchTrap#1: ROP-based Fake Paths Generation

```
Func1 (arg1, arg2)

Caller

<table>
<thead>
<tr>
<th>call Func1</th>
</tr>
</thead>
<tbody>
<tr>
<td>next inst</td>
</tr>
</tbody>
</table>

Original epilogue

<table>
<thead>
<tr>
<th>pop rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop r15</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>
```
BranchTrap#1: ROP-based Fake Paths Generation

Func1 (arg1, arg2)

Original epilogue
pop rbp
pop r15
ret

caller

call Func1
next inst

Code snippet 1
pop rbp
pop r15
ret

Code snippet 2
pop rbp
pop r15
ret

…

Code snippet N
…
BranchTrap#1: ROP-based Fake Paths Generation

Code snippet 1
pop rbp
pop r15
ret

Code snippet 2
pop rbp
pop r15
ret

Code snippet N
...

Original epilogue
pop rbp
pop r15
ret

func1 (arg1, arg2)

index = arg1 ^ arg2

caller

next inst

① call Func1

②
BranchTrap#1: ROP-based Fake Paths Generation

```plaintext
Code snippet 1
pop rbp
pop r15
ret

Code snippet 2
pop rbp
pop r15
ret

…

Call Func1
next inst

Func1 (arg1, arg2)

index = arg1 ^ arg2

jmp table [index]

Original epilogue
pop rbp
pop r15
ret

Code snippet 1
pop rbp
pop r15
ret

Code snippet 2
pop rbp
pop r15
ret

…

Code snippet N
...
```
BranchTrap#1: ROP-based Fake Paths Generation

caller

\textbf{Func1} \((\text{arg1, arg2})\)

1. \textit{call Func1}
2. \textit{next inst}
3. \textit{index = arg1 \^ arg2}
4. \textit{jmp table [index]}
5. \textit{Original epilogue}

\textbf{Code snippet 1}
- \textit{pop rbp}
- \textit{pop r15}
- \textit{ret}

\textbf{Code snippet 2}
- \textit{pop rbp}
- \textit{pop r15}
- \textit{ret}

\ldots

\textbf{Code snippet N}
- \textit{\ldots}
BranchTrap#2: Saturate Feedback State

- One-time visit makes effect

- **BranchTrap:**
  - Saturates bitmap data
  - Prevents coverage recording
AntiHybrid Hinders Hybrid Fuzzing

- Fast execution
- Coverage-guidance
- Hybrid approach
Challenge of Hybrid Fuzzing

• Dynamic taint analysis
  ▫ Expensive implicit flow

Transform *explicit* data-flow ➔ *implicit* data-flow
Challenge of Hybrid Fuzzing

• Dynamic taint analysis
  ▫ Expensive implicit flow

  Transform *explicit* data-flow ➔ *implicit* data-flow

• Symbolic execution
  ▫ Path explosion

  Introduce an arbitrary path explosions
AntiHybrid Avoids Dynamic Taint Analysis

• Transform explicit data-flow to implicit data-flow

```
char input = 'a';
if (!strcmp(input, 'a'))
{
    ... }
```

```
char input = 'a';

char anti_dta; 
if (input == 97)
    anti_dta = 'a';

if (!strcmp(anti_dta, 'a'))
{
    ... }
```
AntiHybrid Incurs Path Explosions

- Inject hash calculations into branches

```c
if(a == 30)
{
    ...
}

if(Hash(a) == 0x300df11)
{
    ...
}
```

Path Explosion
Fuzzification Work-flow

1. Source
2. Valid/invalid inputs
3. Binary
4. Run
5. Profile
Fuzzification Work-flow

1. Run
   - Profile

2. Inject component
   - LLVM IR
     - SpeedBump
     - BranchTrap
     - AntiHybrid

Valid/invalid inputs

Source
Fuzzification Work-flow

1. Run binary
2. Inject component
3. Measure overhead & inject more component
Fuzzification Work-flow

1. Run

2. Inject component

   - LLVM IR
   - SpeedBump
   - BranchTrap
   - AntiHybrid

3. Measure Overhead & Inject More Component

4. Release fortified binary

Valid/invalid inputs

Source
Evaluation Summary

• Implementation
  ▫ 6,599 lines of Python and 758 lines of C++

• Evaluation questions:
  ▫ Effective in “Reducing discovered paths and bugs?”
  ▫ Effective on “Various fuzzers?”
  ▫ Impose “Low overhead” to the normal user?
Reduced the Discovered Coverage By 71%

* Fuzzing result on AFL-QEMU
Reduced the Discovered Coverage By 71%

Other binaries

* Fuzzing result on AFL-QEMU
Fuzzification is Effective on Various Fuzzers

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL (QEMU)</td>
<td>74%</td>
</tr>
<tr>
<td>HonggFuzz (PT)</td>
<td>61%</td>
</tr>
<tr>
<td>QSym (AFL-QEMU)</td>
<td>80%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>71%</strong></td>
</tr>
</tbody>
</table>

Reduced code coverage
### Reduced the Discovered Bugs

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>binutils v2.3.0</strong></td>
<td></td>
</tr>
<tr>
<td>AFL (QEMU)</td>
<td>88%</td>
</tr>
<tr>
<td>HonggFuzz (PT)</td>
<td>98%</td>
</tr>
<tr>
<td>QSym (AFL-QEMU)</td>
<td>94%</td>
</tr>
<tr>
<td>Average</td>
<td>93%</td>
</tr>
<tr>
<td><strong>LAVA-M dataset</strong></td>
<td></td>
</tr>
<tr>
<td>Vuzzer</td>
<td>56%</td>
</tr>
<tr>
<td>QSym (AFL-QEMU)</td>
<td>78%</td>
</tr>
<tr>
<td>Average</td>
<td>67%</td>
</tr>
</tbody>
</table>
## File size & CPU Overheads

### binutils v2.3.0

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Size</td>
<td>1.4MB (62.1%)</td>
</tr>
<tr>
<td>CPU Overhead</td>
<td>3.7%</td>
</tr>
</tbody>
</table>

### Real-world applications (e.g., GUI)

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Size</td>
<td>1.3MB (5.4%)</td>
</tr>
<tr>
<td>CPU Overhead</td>
<td>0.73%</td>
</tr>
</tbody>
</table>

* Both overheads are configurable
Discussion

• Best-effort protections against adversarial analysis

• Complementary to other defense techniques
  ▫ Not hiding all vulnerabilities
  ▫ But introducing significant cost on attacker
## Comparison: Fuzzification vs. AntiFuzz

<table>
<thead>
<tr>
<th>Component</th>
<th>Fuzzification</th>
<th>AntiFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay execution</td>
<td>● (+ cold path)</td>
<td>●</td>
</tr>
<tr>
<td>Fake coverage</td>
<td>● (randomized return)</td>
<td>● (fake code)</td>
</tr>
<tr>
<td>Saturate coverage</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Prevent crash</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Anti-hybrid</td>
<td>● (+ anti-DTA)</td>
<td>●</td>
</tr>
<tr>
<td>Countermeasures</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>
Conclusion

Make the fuzzing only effective to the testers

• **SpeedBump**: Inject delays and only affects attackers
• **BranchTrap**: Insert input-sensitive branches
• **AntiHybrid**: Hinder hybrid fuzzing techniques

[https://github.com/sslab-gatech/fuzzification](https://github.com/sslab-gatech/fuzzification)