SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities

Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana

Columbia University
COMPLEXITY VULNERABILITIES

- Difference between average and worst-case complexity
  - CPU, memory, space etc.
  - User-controlled
  - Exploitability & Denial of Service (DoS)

- Several instances seen in the wild
Difference between average and worst-case complexity
- CPU, memory, space etc.
- User-controlled
- Exploitability & Denial of Service (DoS)

Several instances seen in the wild
**COMPLEXITY VULNERABILITIES**

- *Difference between average and worst-case complexity*
  - CPU, memory, space etc.
  - User-controlled
  - Exploitability & Denial of Service (DoS)

- *Several instances seen in the wild*
COMPLEXITY VULNERABILITIES

- **Difference between average and worst-case complexity**
  - CPU, memory, space etc.
  - User-controlled
  - Exploitability & Denial of Service (DoS)

- **Several instances seen in the wild**
COMPLEXITY VULNERABILITIES

- Difference between average and worst-case complexity
  - CPU, memory, space etc.
  - User-controlled
  - Exploitability & Denial of Service (DoS)

- Several instances seen in the wild
Heavily dependent on application logic

Algorithmic worst-case vs implementation worst-case
- Minor changes often drastically change complexity (e.g., pivot selection in quicksort)

Reasoning about the problem in the generic case is hard:
- Theoretical analysis is often non-trivial
- Implementation varies
- Domain-specific tools predominantly require expert knowledge
EXAMPLE: QUICKSORT

- Average $O(n \log n)$ vs worst-case $O(n^2)$ complexity
- Implementation largely affects performance
- How do we reason on the effectiveness of a given implementation?
- How to test in a domain-agnostic manner?
Domain-independent test input generation

Known to perform well in grey-box settings

Very effective in modern fuzzers targeting crash/memory corruption bugs
  - No expert knowledge
  - Production tools compete with domain-specific engines
Can we steer evolutionary testing towards complexity bugs?

Coverage is irrelevant in this scenario

Re-use fuzzing infrastructure
SLOWFUZZ PROTOTYPE

- *SlowFuzz prototype*

- *Maintain and evolve an input corpus towards slower executions*
Maintain and evolve an input corpus towards slower executions
Maintain and evolve an input corpus towards slower executions
- Maintain and evolve an input corpus towards slower executions
Maintain and evolve an input corpus towards slower executions
Maintain and evolve an input corpus towards slower executions
SLOWFUZZ KEY IDEAS

- Three key controls:
  - Instrumentation, Fitness Function, Mutations

- Fitness Function should favor inputs that introduce slowdowns

- Mutation operations with locality in mind

- Avoid getting stuck!
SLOWFUZZ KEY IDEAS

- Three key controls:
  - Instrumentation, Fitness Function, Mutations

- Fitness Function should favor inputs that introduce slowdowns

- Mutation operations with locality in mind

- Avoid getting stuck!
Three key controls:
- Instrumentation, Fitness Function, Mutations

Fitness Function should favor inputs that introduce slowdowns

Mutation operations with locality in mind

Avoid getting stuck!
- **Three key controls:**
  - **Instrumentation, Fitness Function, Mutations**

- **Fitness Function** should favor inputs that introduce slowdowns

- **Mutation operations with locality in mind**

- **Avoid getting stuck!**
SLOWFUZZ KEY IDEAS

- **Fitness function maximizes CPU instructions**

- **Mutation Strategies:**
  - Random
  - Offset Priority
  - Mutation Priority
  - Hybrid
USECASE: SORTING

- Insertion sort & quicksort implementations
- Quadratic worst-case performance
- How close do we get to the theoretical worst slowdown?
- Slowdowns of 84.97% and 83.74% of theoretical worst-case
USECASE: SORTING / REAL WORLD EXAMPLES

- Apple: 3.34x
- OpenBSD: 3.3x
- GNU: 26.36%
- NetBSD: 8.7%
ENGINE PROPERTIES

- **Fitness function:**
  - CPU instructions vs Code Coverage vs Time-based tracing

- **Mutation Strategies:**
  - Random
  - Offset Priority
  - Mutation Priority
  - Hybrid
ENGINE EVALUATION / MUTATION STRATEGIES - OPENBSD QUICKSORT

Normalized slowdown over best performing input

- Random
- Mutation Priority
- Offset Priority
- Hybrid

![Graph showing normalized slowdown over generation](image)
ENGINE EVALUATION / FITNESS FUNCTIONS - OPENBSD QUICKSORT

Normalized slowdown over best performing input

Slowdown

Generation

Time
Coverage Edge
Counters
Evolutionary testing for complexity bugs is promising

Testcases: common instances of complexity vulnerabilities
- Hashtables
- Regular Expression Parsers
- Compression/decompression routines
USECASE: PHP’S DJBX33A HASH

- Hash used for string keys in PHP
- Known worst-case performance
- Has been exploited in the wild
- For ‘ab’, ‘cd’ to collide it must hold
  \[ c = a + n \land d = b - 33 \times n, n \in \mathbb{Z} \]
- If if two equal-length strings A and B collide, then strings xAy, xBy also collide

```c
1 /*
2  * @arKey is the array key to be hashed
3  * @nKeyLength is the length of arKey
4  */
5 static inline ulong
6 zend_inline_hash_func(const char *arKey, uint
7     nKeyLength)
8 {
9     register ulong hash = 5381;
10    for (uint i = 0; i < nKeyLength; ++i) {
11        hash = ((hash << 5) + hash) + arKey[i];
12    }
13    return hash;
14 }
```
USE CASE: PHP’S DJBX33A HASH

- 64 hashtable entries & 64 insertions
- Slowfuzz generated inputs causing monotonically increasing collisions
- No knowledge of the internals of the hash function
Multiple instances of ReDoS in the wild

Backtracking can be catastrophic

Handling of both regexes and inputs
  - Evil Regexes
  - Slowdowns on given inputs

Identifying evil regexes is a hard problem
  - Widely varying complexity: linear to exponential
  - Focus on super-linear & exponential matching

```c
regex_match(regex, string)
```
Can SlowFuzz find evil regexes given a fixed input?
Can SlowFuzz find evil regexes given a fixed input?

- Yes! Without any knowledge of the regex logic
USECASE: REGEX PARSERS / PCRE

- Can SlowFuzz find evil regexes given a fixed input?
  - Yes! *Without any knowledge of the regex logic*

<table>
<thead>
<tr>
<th>Super-linear</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>c<em>ca</em>b<em>a</em>b</td>
<td>(b+)+c</td>
</tr>
<tr>
<td>a+b+b+b+a+</td>
<td>c*(b+b)+c</td>
</tr>
<tr>
<td>c*c+ccbc+</td>
<td>a(ala*)+a</td>
</tr>
</tbody>
</table>
Can SlowFuzz find evil regexes given a fixed input?
- Yes! Without any knowledge of the regex logic

**Example:** \((b^+)+c\)

**USECASE: REGEX PARSERS / PCRE**

- Example: \((b^+)+c\)
USECASE: REGEX PARSERS / PCRE

- 100 runs / 1 million generation each
- Regexes of 10 characters or less
- At least 31 regexes causing a slowdown with 90% probability
- At least 2 regexes with super-linear matching with 90% probability
- At least 1 regex with exponential matching with 45.45% probability
- Can SlowFuzz find inputs causing a slowdown on a fixed regex?
  - Regexes from production WAFs
  - 8 - 25% slowdowns
USECASE: DECOMPRESSION / BZIP

- bzip2
- 250-byte inputs
- 300x slowdown on fixed input size

Average Slowdown per Fuzzing Hour
CONCLUSION

- SlowFuzz: automated detection of complexity bugs through fuzzing

- Found non-trivial issues involving high performant code
  - PHP’s hashtable implementation
  - PCRE regular expression library
  - bzip2

- Evolutionary fuzzing as a generic means of code exploration
  - Different objectives for different bug types
  - Beyond code coverage maximization
  - Objective vs Controls: Instrumentation, Fitness Functions, Mutations