VUzzer: Application-aware Evolutionary Fuzzing

Presenter: Bin Xie
Overview

• Introduction
• Motivation
• Design and technique details
• Evaluation results
• Conclusions
Introduction

• Fuzzing, like AFL, is a dynamic analysis technique
  • Pros: high throughput, scalability
  • Cons: ineffective in deep execution and strict conditions due to blindly mutation

• Hybrid Fuzzing: concolic execution + AFL, like Driller
  • Pros: solve some strict conditions.
  • Cons: sacrifice scalability (key strength for Fuzzing) and time.
More limitations: Magic value

• 1. Because AFL is hard to detect if condition, AFL may focus on exploring else branch.

• 2. Even if concolic execution helps AFL to calculate these offsets in if branch, AFL may again mutate these offsets again, which waste processing power and time

```c
1...
2    read(fd, buf, size);
3 if (buf[5] == 0xD8 && buf[4] == 0xFF) // notice the order of CMPs
4      ... some useful code ...
5 else
6      EXIT_ERROR("Invalid file \n");
```
More limitations: Deeper path

1. AFL will give some prioritize efforts to the both first *if* and *else* branch, AFL will not be able to prioritize efforts to focus on the interesting path, such as any bugs inside the nested *if* code

```c
1 ...  
2 read(fd, buf, size);  
3 ...  
4 if (...) {  
5     if (...)  // nested IF  
6         ...  
7 } else {  
8     ...  
9 }
```
Motivation

- VUzzer is both scalable and fast to discover
  - Scalability: not use symbolic execution that hard to scale
  - Fast to discover: Utilize control-flow and data-flow features based on static and dynamic analysis to find more properties of the target application. And then use these to enable faster generation of interesting inputs.
Design Overview

1. The two main components of VUzzer are a static analyzer (shown on the left) and the main (dynamic) fuzzing loop (shown on the right).
2. VUzzer continuously pumps this information back into the evolutionary mutation and crossover operators to help generate better inputs in the next generation.
3. VUzzer uses static analysis to gain some information.
4. Execute evolutionary fuzzing loop with information from static analysis.
   1. Input test cases
   2. Use Dynamic taint analysis to monitor execution
   3. Evaluate input cases by fitness function.
   4. Mutate
   5. Repeat step 1

Fig. 1. A high-level overview of VUzzer. BB: basic block, CMP imm: cmp instruction with one immediate operand, DTA: dynamic taint analysis, LEA: load effective address instruction.
Data-flow features: Magic values

1. Data-flow features provide information about the relationship between input data and computations in the application.

2. VUzzer uses taint analysis to infer the structure of the input in terms of the types of data at certain offsets in the input, such as by instrumenting each instruction of the `cmp` family to determine which input bytes (offsets) it uses and against which values it compares them.

3. VUzzer can determine which offsets are interesting to mutate and what values to use at those offsets.
Control-flow features

- Two purposes by control-flow features:
  1. Infer the importance of certain execution paths: identifying such error-handling blocks may speed up the generation of interesting inputs
  2. Concerns the reachability of nested blocks: assigning weights to individual basic blocks
     - Basic blocks are part of error-handling code get a negative weight
     - Basic blocks in hard-to-reach code regions obtain a higher weight
  We use control-flow features to deprioritize and prioritize paths
The static analyzer

• At the beginning of the fuzzing process, we use a lightweight intraprocedural static analysis

  • Obtain immediate values of the cmp instructions by scanning the binary code of the application, a list $L_{imm}$

  • Compute the weights for the basic blocks of the application binary, a list of $L_{BB}$
The static analyzer

• A list $L_{imm}$ of byte sequences {0xEF, 0xFD, %, @, MAZE}

• Model the CFG of each function as a Markov model, and the weight $w_b$ of each basic block $b$ as $1/p_b$, which is the probability $p_b$ of reaching each basic block $b$ in a function
The main fuzzing loop: Evolutionary Algorithm

**Algorithm 1** Pseudo-code of a typical evolutionary algorithm

INITIALIZE population with *seed* inputs

repeat
  SELECT1 parents
  RECOMBINE parents to generate children
  MUTATE parents/children
  EVALUATE new candidates with FITNESS function
  SELECT2 fittest candidates for the next population
until TERMINATION CONDITION is met
return BEST Solution
INITIALIZE step

• Purposes: gain an initial set of control-flow and data-flow features

• Implement: run dynamic taint analysis for all initial seed inputs to capture common characteristics, like magic-byte and error-handling code detection
SELECT1 parents

• Purposes: select two inputs (parents) for generating new inputs

• Implement:
  • VUzzer uses fitness functions to calculate fitness scores for all taint inputs and sorts it to gain a sorted lists $L_{fit}$
  • Fetch top n% of $L_{fit}$ and set as ROOT set
  • Select two inputs (parents) from ROOT
RECOMBINE

• Purposes: combine two inputs to generate two children for next step, MUTATE

• Implement: two inputs are combined by choosing an offset (cut-point) and exchanging the corresponding two parts to form two children
MUTATE

• Purposes: use a single parent input to form one child

• Implement:
  • Mutation uses several sub-operations, such as addition, deletion, replacement, and insertion of bytes at certain offsets in the given input.
  • The mutation operator makes use of the data-flow features to generate new values, for example, use characters from $L_{imm}$
EVALUATE

- Purposes: use a fitness function to assess the suitability of the input

- Implement:
  - Fitness calculation: sum the weights, list $L_{BB}$, of the frequencies of executed basic blocks
  - Provide high scores to inputs that execute basic blocks with higher weights and thereby prioritize the corresponding paths
SELECT2 and repeat

• Purposes: select fittest candidates for the next population

• Implement: choose the highest fitness score from the fitness function for next generation of inputs.
Evaluation results

- DARPA CGC binaries: VUzzer found crashes in 29 of the CGC binaries, whereas AFLPIN found only 23 crashes.

![Relative number of inputs executed for each of the CGC Binaries, wherein both VUzzer and AFLPIN find crashes. The numbers above the bars are the total number of inputs (in thousands) executed.](image)
Evaluation results

• LAVA Dataset
  • FUZZER: a coverage-based fuzzer
  • SES: symbolic execution and a SAT-based approach

<table>
<thead>
<tr>
<th>Program</th>
<th>Total bugs</th>
<th>FUZZER</th>
<th>SES</th>
<th>VUzzer (unique bugs, total inputs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>28</td>
<td>7</td>
<td>0</td>
<td>27 (27K)</td>
</tr>
<tr>
<td>base64</td>
<td>44</td>
<td>7</td>
<td>9</td>
<td>17 (14k)</td>
</tr>
<tr>
<td>md5sum</td>
<td>57</td>
<td>2</td>
<td>0</td>
<td>1*</td>
</tr>
<tr>
<td>who</td>
<td>2136</td>
<td>0</td>
<td>18</td>
<td>50 (5.8k)</td>
</tr>
</tbody>
</table>
Evaluation results

- Various Applications (VA) Dataset

<table>
<thead>
<tr>
<th>Application</th>
<th>VUzzer</th>
<th></th>
<th>#Unique crashes</th>
<th>#Inputs</th>
<th>#Unique crashes</th>
<th>#Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg321</td>
<td>337</td>
<td>23.6K</td>
<td>19</td>
<td>883K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gif2png+libpng</td>
<td>127</td>
<td>43.2K</td>
<td>7</td>
<td>1.84M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pdf2svg+libpoppler</td>
<td>13</td>
<td>5K</td>
<td>0</td>
<td>923K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tcpdump+libpcap</td>
<td>3</td>
<td>77.8K</td>
<td>0</td>
<td>2.89M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tcptrace+libpcap</td>
<td>403</td>
<td>30K</td>
<td>238</td>
<td>3.29M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>djpeg+libjpeg</td>
<td>1'</td>
<td>90K</td>
<td>0</td>
<td>35.9M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

• They leverage control-flow and data-flow features of the application to infer several interesting properties of the input, which enables much faster generation of interesting inputs.

• Scalable and faster
Thank you & Question