V-Fuzz: Evolutionary Fuzzing Assisted By Vulnerability Prediction

Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen
Qinchen Gu, Chunming Wu, Raheem Beyah
Fuzzing is an automatic, dynamic vulnerability detection technique.

Fuzzing detects vulnerabilities by iteratively and randomly feeding inputs to the target programs.

Fuzzer is a tool that implements fuzzing process.

AFL

Angora

Google OSS-Fuzz
### Types of fuzzers

#### Exploration Strategy
- Directed
  - AFLGo
  - Hawkeye
- Coverage-based
  - AFL
  - AFLFast
  - Angora

#### Input Generation
- Mutation-based
  - AFL
  - AFLFast
- Grammar-based
  - Quickfuzz
  - Peach

#### Target Program
- Whitebox fuzzer
  - Driller
- Greybox fuzzer
  - AFL
  - AFLFast
- Blackbox fuzzer
  - zzuf
Coverage-based fuzzer is one of the most popular fuzzer.
  - AFL, VUzzer, Angora, honggfuzz, t-fuzz, Driller, ...

Its goal is to cover as much as coverage of the target program as possible.
Coverage-based fuzzers may not so efficient

Coverage-based fuzzer is one of the most popular fuzzer.
- AFL, VUzzer, Angora, honggfuzz, t-fuzz, Driller, ...

Its goal is to cover as much as coverage of the target program as possible.

However, it is not efficient for detecting vulnerabilities:
1. Vulnerable code only takes a tiny fraction of the entire code.
   e.g., Only 3% of the source code files in Mozilla Firefox has vulnerabilities.

2. Achieving high coverage is still very difficult.
   e.g., Driller can only generate valid inputs for 13 out of 41 CGC binaries.
Coverage-based fuzzers may not be so efficient.

```c
#define SIZE 1000
int main(int argc, char **argv) {
    unsigned char source[SIZE];
    unsigned char dest[SIZE];
    char *input=ReadData(argv[1]);
    int i;
    i=argv[2];
    /*magic byte check*/
    if (input[0] != 'x')
        return ERROR;
    if (input[1] == 0xAB && input[2] == 0xCD) {
        printf("Pass_1st_check!\n");
        /*some nested conditions*/
        if (strncmp(&input[6], "abcde", 5) == 0) {
            printf("Pass_2nd_check!\n");
            /* some common codes without vulnerabilities*/
            ...
        } else {
            printf("Not_pass_the_2nd_check!\n");
        }
    } else {
        printf("Not_pass_the_1st_check!\n");
    }
}
/*A buffer overflow vulnerability*/
func_v(input, source, dest);
return 0;
```
Coverage-based fuzzers may not be so efficient because they prioritize code that is most covered, which means it has less meaning and less coverage. Thus, fuzzer should prioritize the potentially vulnerable code.

```c
#define SIZE 1000
int main(int argc, char **argv){
    unsigned char source[SIZE];
    unsigned char dest[SIZE];
    char *input=ReadData(argv[1]);
    int i;
    i=argv[2];
    /\*magic byte check */
    if (input[0]==\'\'\')
        return ERROR;
    if (input[1]==0xA0)
        printf("Pass
    /\/*some nested if\*/
        if(strcmp(&input[1], some_if)
            printf("\n    /\* some code\*/
        ...
    } else{
        printf("Not pass the 1st check\n    
    } else{
        printf("//A buffer overflow vulnerability\*/
            func_v(input, source, dest);
        return 0;
    }

Thus, fuzzer should prioritize the potentially vulnerable code.
```
Detect which are vulnerable.

Prioritize to fuzz vulnerable code.

Fig. 2: The architecture of V-Fuzz.
Data preprocessing

Fig. 2: The architecture of V-Fuzz.

V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction
V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction

Vulnerability Prediction Model

Fig. 2: The architecture of V-Fuzz.

Output vulnerable probability
V-Fuzz: Evolutionary Fuzzing Assisted by Vulnerability Prediction

Data preprocessing

Generate inputs with tend to arrive at the vulnerable components
Experimental Results: Vulnerability Prediction

Top-K Accuracy $\geq 80$

Recall $\geq 66$

Fast Convergence

Vulnerability prediction model performs well.
Experimental Results: ACFG Extraction Time

ACFG extraction process is fast:
Released binaries: <100 s
Debugging binaries: <2.5 s
Vulnerability prediction model can distinguish between vulnerable functions and secure functions.
Experimental Results: The number of unique crashes.

V-Fuzz finds crashes quickly than AFL, AFLFast and VUzzer.
Experimental Results: The number of unique crashes.

TABLE 6: The number of unique crashes found for 24 hours.

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>Fuzzer</th>
<th>V-Fuzz</th>
<th>VUzzer</th>
<th>AFL</th>
<th>AFLFast</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>LAVA-M</td>
<td></td>
<td>659</td>
<td>321</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>base64</td>
<td>LAVA-M</td>
<td></td>
<td>128</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>who</td>
<td>LAVA-M</td>
<td></td>
<td>117</td>
<td>92</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pdftotext</td>
<td>xpdf-2.00</td>
<td></td>
<td>209</td>
<td>59</td>
<td>12</td>
<td>108</td>
</tr>
<tr>
<td>pdffonts</td>
<td>xpdf-2.00</td>
<td></td>
<td>581</td>
<td>367</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>pdftopbm</td>
<td>xpdf-2.00</td>
<td></td>
<td>50</td>
<td>25</td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td>pdf2svg + libpoppler</td>
<td>pdf2svg-0.2.3 libpoppler-0.24.5</td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MP3Gain</td>
<td>1.5.2</td>
<td></td>
<td>217</td>
<td>34</td>
<td>103</td>
<td>110</td>
</tr>
<tr>
<td>mpg321</td>
<td>0.3.2</td>
<td></td>
<td>321</td>
<td>184</td>
<td>40</td>
<td>17</td>
</tr>
<tr>
<td>xpstopng</td>
<td>libgxps-0.2.5</td>
<td></td>
<td>3,222</td>
<td>2,195</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>xpsops</td>
<td>libgxps-0.2.5</td>
<td></td>
<td>4,157</td>
<td>3,044</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>xpsotjpeg</td>
<td>libgxps-0.2.5</td>
<td></td>
<td>4,828</td>
<td>4,243</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>cflow</td>
<td>1.5</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>14,493</td>
<td>10,666</td>
<td>214</td>
<td>280</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td>1,114</td>
<td>820</td>
<td>16</td>
<td>21</td>
</tr>
</tbody>
</table>

V-Fuzz finds more unique crashes than AFL, AFLFast and VUzzer.
### TABLE 9: The CVEs found by V-Fuzz.

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>CVE</th>
<th>Vulnerability Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdftotext</td>
<td>xpdf&lt;=3.01</td>
<td>CVE-2007-0104</td>
<td>Buffer errors</td>
</tr>
<tr>
<td>pdffonts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pdftopbm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpg321</td>
<td>0.3.2</td>
<td>CVE-2017-11552</td>
<td>Buffer errors</td>
</tr>
<tr>
<td>MP3Gain</td>
<td>1.5.2</td>
<td>CVE-2017-14406</td>
<td>NULL pointer dereference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-14407</td>
<td>Stack-based buffer over-read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-14409</td>
<td>Buffer overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-14410</td>
<td>Buffer over-read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-12912</td>
<td>Buffer errors</td>
</tr>
<tr>
<td>libgxps</td>
<td>&lt;=0.3.0</td>
<td>CVE-2018-10767 (new)</td>
<td>Buffer errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2018-10733 (new)</td>
<td>Stack-based buffer over-read</td>
</tr>
<tr>
<td>libpoppler</td>
<td>0.24.5</td>
<td>CVE-2018-10768 (new)</td>
<td>NULL pointer dereference</td>
</tr>
</tbody>
</table>

V-Fuzz detects three new CVEs.
Conclusion

• In this paper, we design and implement V-Fuzz, an evolutionary fuzzer assisted by vulnerability prediction.

• We design and implement a vulnerability prediction model based on graph embedding network that can predict the vulnerable probabilities for binary functions.

• Compared with several state-of-the-art fuzzers, V-Fuzz can find more vulnerabilities quickly.

• V-Fuzz has discovered 10 CVEs, and 3 of them are 0-day vulnerabilities.