LAVA: Large-scale Automated Vulnerability Addition

Engin Kirda
Andrea Mambretti
Wil Robertson
Northeastern University

Brendan Dolan-Gavitt
NYU Tandon

Patrick Hulin, Tim Leek, Fredrich Ulrich, Ryan Whelan
MIT Lincoln Laboratory
This Talk

• In this talk, we explore how to **automatically add large numbers of bugs to programs**

• Why would we want to do this?
  • Computer programs don't have enough bugs
  • We want to put backdoors in other people's programs
This Talk

• In this talk, we explore how to *automatically add large numbers of bugs to programs*

• Why would we want to do this?
  • Computer programs don't have enough bugs
  • We want to put backdoors in other people's programs
In this talk, we explore how to **automatically add large numbers of bugs to programs**

Why would we want to do this?

- Computer programs don't have enough bugs
- We want to put backdoors in other people's programs
Finding vulnerabilities in software automatically has been a major research and industry goal for the last 25 years

**Academic**

- An Empirical Study of the Reliability of UNIX Utilities
  - Burton P. Miller
  - Lars Frederiksen
  - Bryan So

- KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
  - Cristian Cadar, Daniel Dunbar, Dawson Engler
  - Stanford University

- KLEE (2005)

**Commercial**

- Driller: Augmenting Fuzzing Through Selective Symbolic Execution
  - Nick Systermans, John Gosan, Christopher Saleh, Andrew Dyczia, Ruiya Wang, Erico Erbetta, Tae Manhakimchi, Cristopher Kraemer, Giovanni Vigna
  - U.C. Santa Barbara

- Driller (2015)

**LAVA: Large-Scale Automated Vulnerability Addition**
Vulnerability Discovery

- Finding vulnerabilities in software automatically has been a major research and industry goal for the last 25 years.

**Academic**

- An Empirical Study of the Reliability of UNIX Utilities
  - Barton P. Miller
  - Lars Frederiksen
  - Bryan S. So

**Commercial**

- KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
  - Cristian Cadar, Daniel Dunbar, Dawson Engler
  - Stanford University

- Driller: Augmenting Fuzzing Through Selective Symbolic Execution
  - Nick Strategos, John Groos, Christopher Salls, Andrew Driehuis, Ravee Wang, Accopi Ershova, Tar Oikarinen, Christopher Kruegel, Giovanni Vigna
  - UC Santa Barbara

**Does this work??**

LAVA: Large-Scale Automated Vulnerability Addition
Debugging the Bug Finders

• Lots of work that claims to find bugs in programs

• **Lack of ground truth** makes it very difficult to evaluate these claims

• If Coverity finds 22 bugs in my program, is that good or bad?

• What are the false positive and **false negative** rates?
Some existing bug corpora exist, but have many problems:

- Synthetic (small) programs
- Don't always have triggering inputs
- Fixed size – tools can “overfit” to the corpus
What About Real Vulnerabilities?

- Real vulnerabilities with proof-of-concept exploits are essentially what we want.
- But there just aren't that many of them. And finding new ones is expensive!

<table>
<thead>
<tr>
<th>Software</th>
<th>Price Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Reader</td>
<td>$5,000–$30,000</td>
</tr>
<tr>
<td>MAC OSX</td>
<td>$20,000–$50,000</td>
</tr>
<tr>
<td>Android</td>
<td>$30,000–$60,000</td>
</tr>
<tr>
<td>Flash or Java Browser Plug-ins</td>
<td>$40,000–$100,000</td>
</tr>
<tr>
<td>Microsoft Word</td>
<td>$50,000–$100,000</td>
</tr>
<tr>
<td>Windows</td>
<td>$60,000–$120,000</td>
</tr>
<tr>
<td>Firefox or Safari</td>
<td>$60,000–$150,000</td>
</tr>
<tr>
<td>Chrome or Internet Explorer</td>
<td>$80,000–$200,000</td>
</tr>
<tr>
<td>iOS</td>
<td>$100,000–$250,000</td>
</tr>
</tbody>
</table>

Forbes, 2012
Debugging the Bug Finders

• Existing corpora are fixed size and static – it's easy to optimize to the benchmark

• Instead we would like to automatically create bug corpora

• Take an existing program and automatically add new bugs into it

• Now we can measure how many of our bugs they find to estimate effectiveness of bug-finders
Goals

• We want to produce bugs that are:
  • **Plentiful** (can put 1000s into a program easily)
  • **Distributed** throughout the program
  • Come with a **triggering input**
  • Only manifest for a **tiny fraction of inputs**
  • Are likely to be **security-critical**
Sounds Simple... But Not

- Why not just change all the `strncpys` to `strcpys`?
  - Turns out this breaks most programs for every input – trivial to find the bugs
  - We won't know how to trigger the bugs – hard to prove they're "real" and security-relevant
  - This applies to most local, random mutations
Our Approach: DUAs

- We want to find parts of the program's input data that are:
  - **Dead**: not currently used much in the program (i.e., we can set to arbitrary values)
  - **Uncomplicated**: not altered very much (i.e., we can predict their value throughout the program's lifetime)
  - **Available** in some program variables

These properties try to capture the notion of *attacker-controlled data*

- If we can find these **DUAs**, we will be able to add code to the program that uses such data to trigger a bug
New Taint-Based Measures

• How do we find out what data is dead and uncomplicated?

• Two new taint-based measures:
  
  • *Liveness*: a count of how many times some input byte is used to decide a branch
  
  • *Taint compute number*: a measure of how much computation been done on some data
Dynamic Taint Analysis

• We use *dynamic taint analysis* to understand the effect of input data on the program

• Our taint analysis requires some specific features:
  
  • Large number of labels available
  
  • Taint tracks *label sets*

  • Whole-system & fast (enough)

• Our open-source dynamic analysis platform, **PANDA**, provides all of these features

\[
c = a + b ; \ a: \{w,x\} ; \ b: \{y,z\} \\
c \leftarrow \{w,x,y,z\}
\]

[https://github.com/moyix/panda](https://github.com/moyix/panda)
Taint Compute Number (TCN)

// a, b, n are inputs
1: int c = a + b;
2: if (a != 0xdeadbeef)
3:    return;
4: for (int i = 0; i < n; i++)
5:    c += s[i];

TCN measures how much computation has been done on a variable at a given point in the program.
Liveness

```
// a, b, n are inputs
1: int c = a+b;
2: if (a != 0xdeadbeef) return;
3: for (int i=0; i<n; i++)
4: c+=s[i];
```

b: bytes {0..3}
n: bytes {4..7}
a: bytes {8..11}

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0..3}</td>
<td>0</td>
</tr>
<tr>
<td>{4..7}</td>
<td>n</td>
</tr>
<tr>
<td>{8..11}</td>
<td>1</td>
</tr>
</tbody>
</table>

Liveness measures how many branches use each input byte
An Attack Point (ATP) is any place where we may want to use attacker-controlled data to cause a bug.

Examples: pointer dereference, data copying, memory allocation, ...

In current LAVA implementation we just modify pointer dereferences to cause buffer overflow.
LAVA: Large-Scale Automated Vulnerability Addition

**Approach: Overview**

1. **Clang**
   - Instrument source with taint queries
   - Run instrumented program on inputs
   - Find attacker-controlled data and attack points
   - Inject bug into program source, compile and test with modified input

2. **PANDA record**
   - PANDA record + taint analysis

3. **Injectable bugs**

4. **Input corpus**

5. **Effects**
LAVA Bugs

• Any (DUA, ATP) pair where the DUA occurs before the attack point is a potential bug we can inject

• By modifying the source to add new data flow the from DUA to the attack point we can create a bug

DUA + ATP = 🐜
LAVA Bug Example

• PANDA taint analysis shows that bytes 0-3 of `buf` on line 115 of `src/encoding.c` is attacker-controlled (dead & uncomplicated)

• From PANDA we also see that in `readcdf.c` line 365 there is a read from a pointer – if we modify the pointer value we will likely cause a bug in the program

```c
encoding.c 115: } else if (looks_extended(buf, nbytes, *ubuf, ulen)) {
```

```c
readcdf.c 365: if (cdf_read_header(&info, &h) == -1)
```
LAVA Bug Example

• PANDA taint analysis shows that bytes 0-3 of `buf` on line 115 of `src/encoding.c` is attacker-controlled (dead & uncomplicated)

• From PANDA we also see that in `readcdf.c` line 365 there is a read from a pointer – if we modify the pointer value we will likely cause a bug in the program
When the input file data that ends up in buf is set to 0x6c6176c1, we will add 0x6c6176c1 to the pointer info, causing an out of bounds access
### Evaluation: How Many Bugs?

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Num Src Files</th>
<th>Lines C code</th>
<th>N(DUA)</th>
<th>N(ATP)</th>
<th>Potential Bugs</th>
<th>Validated Bugs</th>
<th>Yield</th>
<th>Inj Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>5.22</td>
<td>19</td>
<td>10809</td>
<td>631</td>
<td>114</td>
<td>17518</td>
<td>774</td>
<td>38.7%</td>
<td>16</td>
</tr>
<tr>
<td>readelf</td>
<td>2.25</td>
<td>12</td>
<td>21052</td>
<td>3849</td>
<td>266</td>
<td>276367</td>
<td>1064</td>
<td>53.2%</td>
<td>354</td>
</tr>
<tr>
<td>bash</td>
<td>4.3</td>
<td>143</td>
<td>98871</td>
<td>3832</td>
<td>604</td>
<td>447645</td>
<td>192</td>
<td>9.6%</td>
<td>153</td>
</tr>
<tr>
<td>tshark</td>
<td>1.8.2</td>
<td>1272</td>
<td>2186252</td>
<td>9853</td>
<td>1037</td>
<td>1240777</td>
<td>354</td>
<td>17.7%</td>
<td>542</td>
</tr>
</tbody>
</table>

- We ran four open-source programs each on a single input and generated candidate bugs.

- Because validating all possible bugs would take too long, we instead validated a random sample of 2000 per program.

- **Result:** extrapolating from the yield numbers, a single run gives us up to \(~200,000\) real bugs.
### Evaluation: What Influences Yield?

<table>
<thead>
<tr>
<th>$mTCN$</th>
<th>$[0, 10)$</th>
<th>$[10, 100)$</th>
<th>$[100, 1000)$</th>
<th>$[1000, +\text{inf}]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0, 10)$</td>
<td>51.9%</td>
<td>22.9%</td>
<td>17.4%</td>
<td>11.9%</td>
</tr>
<tr>
<td>$[10, 100)$</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$[100, +\text{inf}]$</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

- TCN strongly affects yield
- No bugs that involved TCN greater than 10 were useable
- Liveness has a weaker correlation with yield – even fairly live data can be sometimes be used if TCN is low
Evaluation: Can Tools Find Them?

• We took two open-source bug-finding tools and tried to measure their success at finding LAVA bugs

  • A coverage-guided fuzzer (FUZZER)

  • A symbolic execution and constraint solving tool (SES)

• (Actual names withheld since this is just a preliminary study)
Results: Specific Value

<table>
<thead>
<tr>
<th>Program</th>
<th>Total Bugs</th>
<th>Unique Bugs FUZZER</th>
<th>Unique Bugs SES</th>
<th>Found Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>28</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>base64</td>
<td>44</td>
<td>7</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>md5sum</td>
<td>57</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>who</td>
<td>2136</td>
<td>0</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Total</td>
<td>2265</td>
<td>16</td>
<td>27</td>
<td>41</td>
</tr>
</tbody>
</table>

Less than 2% of injected bugs found
**Results: Range-Triggered Bugs**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Bug Type</th>
<th>Range</th>
<th>FUZZER</th>
<th>SES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2⁰</td>
<td>2⁷</td>
<td>2¹⁴</td>
</tr>
<tr>
<td>Fuzzer</td>
<td></td>
<td>0</td>
<td>0</td>
<td>9%</td>
</tr>
<tr>
<td>SES</td>
<td></td>
<td>8%</td>
<td>0</td>
<td>9%</td>
</tr>
</tbody>
</table>

Note that the names of tools under evaluation are being used by LAVA-1, used the repository with a fuzzed version of the input verified. The first corpus we created, none of which trigger our injected bugs. Additionally, we constructed tiny example buggy programs and used them to verify that we were able to use each tool at care, high-profile tools. For each tool, we expended significant effort, either in terms of careful setup and use of tools, or other. FUZZER and SES are both state-of-the-art, high-profile tools. For each tool, we expended significant

**TABLE III: Percentage of bugs found in**

<table>
<thead>
<tr>
<th>Range</th>
<th>Bug Type</th>
<th>Tool</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2⁰</td>
<td>Fuzzer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2⁷</td>
<td>Fuzzer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2¹⁴</td>
<td>SES</td>
<td>8%</td>
<td>9%</td>
</tr>
<tr>
<td>2²¹</td>
<td>SES</td>
<td>8%</td>
<td>9%</td>
</tr>
<tr>
<td>2²⁸</td>
<td>SES</td>
<td>8%</td>
<td>9%</td>
</tr>
</tbody>
</table>

**Frequency**

**Fig. 11: Normalized ATP trace location**

**Fig. 12: Fraction of trace with perfectly normal or realistic**

**Frequency**

**Histogram of rdfs$V3**

**Histogram of rdfs$V2**
• The burning question in everyone's mind now: are these bugs realistic?

• This is hard to measure, in part because realism is not a well-defined property!

• Our evaluation looks at:
  • How injected bugs are distributed in the program
  • What proportion of the trace has normal data flow

• Ultimately, the best test of realism will be whether it helps bug-finding software get better
Results: Realism

Fig. 10: Normalized DUA trace location
Fig. 11: Normalized ATP trace location

LAVA: Large-Scale Automated Vulnerability Addition
Limitations and Caveats

• General limitations:
  • Some types of vulnerabilities probably can't be injected using this method – e.g., weak crypto bugs
  • More work is needed to see if these bugs can improve bug-finding software

• Implementation limits:
  • Currently only works on C/C++ programs in Linux
  • Only injects buffer overflow bugs
  • Works only on source code
Future Work

• Continuous on-line competition to encourage self-evaluation

• Use in security competitions like Capture the Flag to re-use and construct challenges on-the-fly

• Improve and assess realism of LAVA bugs

• More types of vulnerabilities (use after free, command injection, ...)

• More interesting effects (prove exploitability!)
Conclusions

• Presented a new technique that is capable of quickly injecting massive numbers of bugs

• Demonstrated that current tools are not very good at finding these bugs

• If these bugs prove to be good stand-ins for real-world vulnerabilities, we can get huge, on-demand bug corpora
Questions?