(State of) The Art of War: Offensive Techniques in Binary Analysis

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, et al.
UC Santa Barbara

Leland Liu, Nolan Pofi
Importance of Binary Analysis

- Wide spread use of binaries
- Usually written in C or C++
- Compilers aren’t necessarily secure
- Need to be able to verify a binary
Automated Analysis is Hard

- Each analysis must make trade-offs
- Two main areas for trade-offs
  - Replayability
  - Semantic Insight
- Some analyses attempt both and run into scalability issues
- Environment model also causes issues for high semantic understanding analyses
Example Problems

- Three memcpy calls with two buffer overflows at lines 10 and 30
- Fuzzing might not be able to trigger execution of the first memcpy
- Depending on information stored a static analysis might report line 16
Problems with Offensive Techniques

● Two main issues with implementations
● Firstly, most implementations live and die as research prototypes
● Secondly, replication is impractical due to re-implementation
● Combine to distort comparisons
Introducing Angr

- Binary Analysis framework
- Open source and built in python
- [https://github.com/angr/angr](https://github.com/angr/angr)
- `pip install angr`
- Attempts to solve first issue
- Hopes to be groundwork to build new analyses
Darpa Cyber Grand Challenge

- Teams had to find, exploit, and patch vulnerabilities autonomously
- Solve environment model by creating new OS DECREE
- All data provided is open-source
- This data is used to test angr
Key Contributions

- Open source binary analysis framework
- Presents a base for comparing analyses and their results
- Provides solutions to common issues with large scale analysis
- Improved implementations of state of the art analysis techniques
Analysis Engine Design

● Main theme: **Modularity and Abstraction**

● Made up of submodules and uses many existing tools
  ○ Operates on Valgrind’s Vex IR
  ○ CLE to load binaries
  ○ SimuVEX to model the program state
  ○ Claripy and Z3 to represent data and data types

● Full Program Analysis representation
Full Program Analysis representation

- **Project** represents a binary and libraries
- **Path Group** tracks execution paths
- **Analysis** manages the lifecycle of an analysis
CFG Recovery

- Empowers the rest of the analysis
- Main roadblock is recovering all indirect jump targets
- Uses two recovery passes (**CFGAccurate** and **CFGFast**)
- Generates a lot of extra data in the process

CFGAccurate

- Starts at program entry
- Goes through Forced Execution, Symbolic Execution, and Backward Slicing
- Maintains a list of unresolved indirect jumps (1_j) it has encountered
Forced Execution, Symbolic Execution, Backward Slicing

- Main goal: Fast pass CFG recovery
  - Executes both branches of every conditional branch

```
if x == 1 jmp B
B
if y == 5 jmp D
if z == 10 jmp F
C
```

```
A
if x == 1 jmp B

B
if y == 5 jmp D
D

C
if z == 10 jmp F
F

E
G
```
Forced Execution, Symbolic Execution, Backward Slicing

- Main goal: Add indirect jumps to CFG
  - Uses constraint solver to find all possible targets of an indirect jump

```plaintext
add rax, 1
...
jmp $rax*0x4+0x8
```
Forced Execution, Symbolic Execution, Backward Slicing

- Main goal: Resolve missing targets
  - Similar to symbolic execution, but with context

```
func abc(int x):
    add rax, x
    ...
    jmp $rax*4+8
```
CFGFast

- Outputs a high coverage graph
- At minimum, identifies the location and content of functions
- Does not have much control flow info
- Does not understand inter-functional reachability
- 3 steps: function identification, recursive disassembly, indirect jump resolution
**CFGFast - Implementation**

- **Function Identification**
  - Identifies functions using hard-coded signatures
- **Recursive Disassembly**
  - Finds direct jumps in identified functions
- **Indirect Jump Resolution**
  - Resolves intra-function control flow
Value Set Analysis

- Main goal: Approximates values in each register/abstract location at each program point
- An advanced static analysis that can be run on a CFG
- Uses numeric and pointer analysis
- Used for memory corruption detection
Symbolic Execution

- One of the most core functionalities of angr
- Uses Claripy’s Z3 interface to populate SimuVEX
- Uses angr’s Path and PathGroup abstractions
  - Combats path explosion by selectively merging paths
Under-Constrained Symbolic Execution

- Based off of UC-KLEE
- Executes each function individually
- Lack of context = under constrained
  - Allows for analyzing large complex data structures
  - Helps filter out security violation false positive
- Improvements over UC-KLEE
  - Treats global memory as under constrained
  - Additional path limiter
  - Additional security false positive filters
Symbolic Assisted Fuzzing

- Introduces as symbolic assisted fuzzer: Driller
- Uses angr as its symbolic tracer and AFL as its fuzzer
  - Avoids symbolic execution’s problem: path explosion
  - Avoids fuzzing’s problem: low semantic insight
- Starts fuzzing with AFL, but starts up angr if AFL is stuck
Automatic Exploit Generation

- angr can exploit vulnerabilities, which allows for hijacking of the instruction pointer (IP)
- Uses concolic execution with crashing inputs
  - Categorizes the crash & measures exploitability
- Simplest exploitable bug: IP overwrite
  - Detect symbolic bits and constrain IP
Exploit Hardening

- Created a ROP chain compiler
- Can generate ROP chains to fulfil arbitrary end goals
- Goal is to upgrade exploits to make them harder to mitigate
Exploit Hardening - Implementation

- Gadget Discovery
  - Scan and categorize all executable code

- Gadget Arrangement
  - Produces impactful ROP gadget sequences

- Payload Generation
  - Uses an SMT solver to combine arrangements into a chain
Results - CFG Recovery

- Compared CFG Fast and CFG Accurate to IDA Pro 6.9
- Information on IDA is limited - believed to be more like CFG Fast
- CFG Fast has higher code coverage and edges found
- CFG Accurate lower code coverage
## Results - Vulnerability Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Replayable</th>
<th>Semantic Insight</th>
<th>Scalability</th>
<th>Crashes</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Symbolic Execution</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Veritesting</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Dynamic Symbolic Execution + Veritesting</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>Fuzzing (AFL)</td>
<td>Yes</td>
<td>Low</td>
<td>High</td>
<td>68</td>
<td>0</td>
</tr>
<tr>
<td>Symbolic-Assisted Fuzzing</td>
<td>Yes</td>
<td>High</td>
<td>High</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>VSA</td>
<td>No</td>
<td>Medium</td>
<td>High</td>
<td>27</td>
<td>130</td>
</tr>
<tr>
<td>Under-constrained Symbolic Execution</td>
<td>No</td>
<td>High</td>
<td>High</td>
<td>25</td>
<td>346</td>
</tr>
</tbody>
</table>

### TABLE IV

**Evaluation Results Across All Vulnerability Discovery Techniques**

<table>
<thead>
<tr>
<th>CGC Qualifying Position</th>
<th>Binaries Crashed</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>77</td>
</tr>
<tr>
<td>Second</td>
<td>12</td>
</tr>
<tr>
<td>Third</td>
<td>57</td>
</tr>
<tr>
<td>Fourth</td>
<td>9</td>
</tr>
<tr>
<td>Fifth</td>
<td>23</td>
</tr>
<tr>
<td>Sixth</td>
<td>57</td>
</tr>
<tr>
<td>Seventh</td>
<td>44</td>
</tr>
<tr>
<td>Eighth (did not qualify)</td>
<td>39</td>
</tr>
<tr>
<td>Ninth (did not qualify)</td>
<td>65</td>
</tr>
</tbody>
</table>

### TABLE III

**Number of Crashed Binaries for the Top 9 Competitors in the CGC Qualification Event**
DSE vs Fuzzing

- DSE struggles at larger paths
- Shines at shallow paths
- Difference in code coverage
  - DSE 330 blocks avg
  - Fuzzing 689 avg
- Combined the CFG is better then CFG accurate
Non-Replayable Analyses

- UC-angr found 25 true vulnerabilities
- With 346 false positives - 93% false positive rate
- VSA found 27 true vulnerabilities
- With 130 false positives - 82.8% false positive rate
- Comparatively low result to the replayable techniques
- Used aggressive false positive filtering
Results - AEG and Exploit Hardening

- Only for four crashes were exploits generated
- Binaries were designed for crashes not exploits
- Current techniques are fairly basic and can’t handle complex vulnerabilities

- 2 out of 4 exploits were able to be hardened
- Can’t reason about stack pivots
Conclusion

- Implemented State of the art techniques
- Techniques share up to 90% of code
- Built in Open Source
- Helps shift focus to improving techniques