Week 12B: Hybrid Fuzzing II

Stefan Nagy

University of Utah
How are semester projects going?

Smoothly?

Obstacles?
Questions?
Hybrid Fuzzing Recap
What is hybrid fuzzing?

- **Combining crude fuzzing with smarter fuzzing**
  - E.g., random + concolic execution (Driller, QSYM, Savior)
  - E.g., random + taint tracking (VUzzer, RedQueen, Angora)

- **Goal is to balance strengths of both techniques**
  - Use generic fuzzing for **most test cases**
    - Use **speed** to brute-force easy branches
  - Deploy more elegant approach **selectively**
    - Focus its **precision** on harder branches
How most hybrid fuzzers work

- Leverage AFL-style parallel fuzzing mode with conventional fuzzer as parent

Conventional (e.g., AFL)

Alternative (e.g., symex)
How most hybrid fuzzers work

- Leverage AFL-style **parallel fuzzing** mode with conventional fuzzer as parent

**Conventional** (e.g., AFL)

**Alternative** (e.g., symex)

- Local queue

**Sync!**

**New code coverage?**
What could go wrong?

- **Ineffective seed scheduling**
  - There are fundamental differences in speed
    - AFL can solve basic branch conditionals fast
    - Fancier approaches generally are much slower
  - Heavyweight approaches are best applied to a **subset** of paths
    - Invoking on all paths will lead to **path explosion**
    - E.g., by the time it’s solved, fuzzer is already way past
Questions?
Adventures in Hybrid Fuzzing:
Driller (NDSS’17)
Fuzzing

0. def f(x) {
1. if x > 10 {
2.   if x < 100:
3.     print "You win!"
4.   else:
5.     print "You lose!"
6. }else:
7.   print "You lose!"

1 ⇒ "You lose!"
593 ⇒ "You lose!"
183 ⇒ "You lose!"
4 ⇒ "You lose!"
498 ⇒ "You lose!"
48 ⇒ "You win!"

Where fuzzing falls short

0. \texttt{def f (x) {}
1. \texttt{    if x > 10 {}
2. \texttt{        if x^2 == 152399025:}
3. \texttt{            print "You win!"
4. \texttt{        else:}
5. \texttt{            print "You lose!"
6. \texttt{    }else:
7. \texttt{        print "You lose!"
8. \texttt{}}}else:
9. \texttt{    print "You lose!"

Symbolic Execution to the rescue!

0. `def f (x) {
1.   if x > 10 {
2.     if \( x^2 \) == 152399025:
3.       print "You win!"
4.     else:
5.       print "You lose!"
6.   }else:
7.   print "You lose!"

Driller

- **Idea:** invoke concolic execution via **demand launch**
  - Heuristic 1: a pre-determined # of mutations based on test case length
  - Heuristic 2: after a pre-determined time interval without new coverage

- **Concolic executor based on** **angr**
  - Binary-level instrumentation and analysis framework
  - Heavily maintained and used in many research projects
  - Translates, analyzes binary in **intermediate form** (VEXIR)
Driller in action

AFL-found coverage

AFL-found test cases
Driller in action

Execute

AFL-found test cases
Driller in action

If `strcmp(input, "MAGIC")` is not equal to "MAGIC", the test cases are solved.
Driller in action

if strcmp(input, "MAGIC")

Execute

Fork

Unsolved branch

Solve

!= "MAGIC"

AFL-found test cases

== "MAGIC"

Concrete test case
Driller in action

AFL-found coverage
Driller in action

Continue execution

Unsolved branch

if \( x^2 = 152399025 \)

Fork

!= 12345

AFL-found test cases

== 12345

Concrete test case

Solve
AFL-found coverage

Driller in action
When to turn solving elsewhere?

- When the path is already **fully solved**
  - Track all branches and which have been solved
  - A fundamental piece of info that is tracked

![Diagram showing decision process](image)
When to turn solving elsewhere?

- **When the path is already fully solved**
  - Track all branches and which have been solved
  - A fundamental piece of info that is tracked

- **When symbolic executor cannot solve**
  - Biggest culprit: hashes

```plaintext
if MD5(input) == "......."
```

A very large search space!
Questions?
Adventures in Hybrid Fuzzing: QSYM (USENIX’18)
Problem: relying on an IR is costly

<table>
<thead>
<tr>
<th>Executor</th>
<th>checksum</th>
<th>md5sum</th>
<th>sha1sum</th>
<th>md5sum (mosml)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>0.008</td>
<td>0.014</td>
<td>0.014</td>
<td>0.001</td>
</tr>
<tr>
<td>KLEE</td>
<td>26.243</td>
<td>32.212</td>
<td>73.675</td>
<td>0.285</td>
</tr>
<tr>
<td>angr</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>462.418</td>
</tr>
</tbody>
</table>

**Table 1:** The emulation overhead of KLEE and angr compared to native execution, which are underlying symbolic executors of S2E and Driller, respectively. We used checksum, md5sum, and sha1sum in coreutils to test KLEE, and md5sum (mosml) [12] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.

# Problem: relying on an IR is costly

<table>
<thead>
<tr>
<th>Executor</th>
<th>checksum</th>
<th>md5sum</th>
<th>sha1sum</th>
<th>md5sum (mosml)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>0.008</td>
<td>0.014</td>
<td>0.014</td>
<td>0.001</td>
</tr>
<tr>
<td>KLEE</td>
<td>26.243</td>
<td>32.212</td>
<td>73.675</td>
<td>0.285</td>
</tr>
<tr>
<td>angr</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>462.418</td>
</tr>
</tbody>
</table>

**Table 1:** The emulation overhead of KLEE and angr compared to native execution, which are underlying symbolic executors of S2E and Driller, respectively. We used `checksum`, `md5sum`, and `sha1sum` in coreutils to test KLEE, and `md5sum (mosml)` [12] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.

QSYM: operate on native instructions

- Omit lifting to intermediate representation
  - Use Intel PIN dynamic binary instrumentation

- Trade-offs:
  - A much higher implementation complexity
  - Significant decrease in symbolic instructions
    - 4X fewer than Driller

Problem: incomplete environment modeling

\[ x : A + B \]
\[ y : B \]

\[ x : A + B \]
\[ y : (A + B) - B = A \]

\[ x : (A + B) - A = B \]
\[ y : A \]

\[ x : \text{syscall} (...) \]
Problem: incomplete environment modeling

- $x : A + B$
- $y : B$
- $x : A + B$
- $y : (A + B) - B = A$
- $x : (A + B) - A = B$
- $y : A$
- $x : \text{syscall (…)}$

Non-trivial to model symbolically
Expensive to emulate and fork
QSYM: leave the environment as-is

- **Omit translating the environment**
  - Use **concrete execution** to model it
    - Model only relevant system calls
    - E.g., standard input, reads, etc.
  - **What about kernel state forking?**
    - Avoid—just **re-execute** from the start

- **Trade-offs:**
  - Re-execution adds **more overhead**
    - Cannot “go back in time” like Driller
Problem: overconstrained paths

```python
0. def f(x):
1.   if x > 10:
2.     if (x > 1000):
3.       if x^2 == 152399025:
4.         print "You win!"
5.       else:
6.         print "You lose!"
7.     else:
8.       print "You lose!"
9.   else:
10.   print "You lose!"
```

Problem: overconstrained paths

```python
0. def f (x) {
1.   if x > 10 {
2.     if (x > 1000){
3.       if x^2 == 152399025:
4.         print "You win!"
5.       else:
6.         print "You lose!"
7.     }else:
8.       print "You lose!"
9.   }else:
10.   print "You lose!"
```
QSYM: optimistically solve last constraint

```
0. def f(x) {
1.   if x > 10 {
2.     if (x > 1000){
3.       if x^2 == 152399025:
4.         print "You win!"
5.       else:
6.         print "You lose!"
7.       }else:
8.      print "You lose!"
9.   }else:
10.  print "You lose!"
```

**Trade-offs:**
- Does not always work
- Can just let the fuzzer quickly rule these out

Questions?
Adventures in Hybrid Fuzzing:
RedQueen (NDSS’19)
Problem: symbolic and concolic execution is slow

1. if( u64(input) == hash(input[8..len]) )
2. if( u64(input+8) == hash(input[16..len]) )
3. if( input[16] == 'R' && input[17] == 'Q')
4. print "You win!"
Problem: symbolic and concolic execution is slow

1. if( u64(input) == hash(input[8..len]) )
2. if( u64(input+8) == hash(input[16..len]) )
3. if( input[16] == 'R' && input[17] == 'Q')
4. print "You win!"

RedQueen’s solution: input-to-state tracking

- **Idea:** hook comparison instructions and identify their input bytes
  - Replace with *compared-to value* (lifted directly from the operand)

```c
if (x[0:3] == "ABCD")
```

```assembly
CMP (eax, 0x44434241)
```
RedQueen’s solution: input-to-state tracking

- **Idea:** hook comparison instructions and identify their input bytes
  - Replace with **compared-to value** (lifted directly from the operand)

```markdown
source

```if (x[0:3] == "ABCD")``` binary

CMP (eax, 0x44434241) ```

```markdown

A B C D X
```

```markdown
W W J D X
```

```markdown
W W J D X
```
Supporting other comparisons

- **Idea:** hook comparison instructions and identify their input bytes
  - Replace with *compared-to value* (lifted directly from the operand)

```
source
if (x[0:3] < 1234)
```

Replace (x[0:3], 1234 - 1)

Replace (x[0:3], 1234 + 1)
What about checksums?

- Finding these at the binary-level is difficult
  - **Assumption:** can identify input bytes that affect the checksum hash
  - **Colorize the input:** inject random bytes and see if they influence the outcome

```rust
if u64(input) == hash(input[8..len])
```
What about checksums?

- Then, **patch-out the checksum** with an always-true operation
  - **Assumption:** checksum is only passed if the input is well-formed
What about checksums?

- Then, **patch-out the checksum** with an always-true operation
  - **Assumption:** checksum is only passed if the input is well-formed
    - Thus, skipping over checksum *won’t matter if well-formed*
    - New input found afterwards? Great—restore the checksum

```
malformed  [ Field 1 ] [ Field 2 ] [ Field 3 ] [ Checksum ] [ Field 4 ]

well-formed [ Field 1 ] [ Field 2 ] [ Field 3 ] [ Checksum ] [ Field 4 ]

well-formed [ Field 1 ] [ Field 2 ] [ Field 3 ] patched-out [ Field 4 ]
```
Questions?