Week 12A: Hybrid Fuzzing I

Stefan Nagy
University of Utah
How are semester projects going?

Problems with your tools?  Successfully up and running?
Questions?
Input Generation Recap
Recap: Model-agnostic Mutation

- Random mutation operators
  - Bit and byte flips
    - Single, two, or four bits in a row
  - Arithmetic operators
    - Additions/subtractions of both endians
  - Inject “fun” values (-1, 256, 1024, etc.)
    - Values that often cause weird behavior

| 11 | 11 | 00 | 11 | 11 | 11 | 11 | 11 | 11 |
| 11 | 11 | 11 | 12 | 11 | 11 | 11 | 11 | 11 |
| 11 | 11 | 11 | 11 | FF | 11 | 11 | 11 | 11 |
Recap: Model-guided Generation

- **Follow a pre-defined input specification**
  - Pre-defined input grammars
  - Dynamically-learned grammars
  - Domain-specific generators

- **Produces many more valid inputs**
  - Model-agnostic inputs are often discarded because they fail basic input sanity checks
Recap: Symbolic Execution

```python
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.   if (x - y > 0):
6.     assert false
7.   return (x, y)
```

Possible path constraints:
- \((A > B)\) and \((B-A > 0)\) = unsatisfiable
- \((A > B)\) and \((B-A \leq 0)\) = satisfiable
- \((A \leq B)\) = satisfiable

Diagram:
- **A > B**
  - \(x: A+B\)
  - \(y: B\)
- **A <= B**
  - \(x: A\)
  - \(y: B\)

**Possible Paths**
- L2 → L3 → L4 → unsatisfiable
- L2 → L3 → L4 → L6 → unsatisfiable
- L2 → L3 → L4 → L7 → satisfiable
- L2 → L3 → L4 → L6 → satisfiable
Recap: Taint Tracking

- **Track input bytes’ flow throughout the program**
  - Identify input “chunks” that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls
  - Mutate these chunks via:
    - Random mutation
    - Insertion of fun or useful tokens
Summary of Input Generation

- **Model-agnostic**: brute-force your way to valid inputs
  - Random insertions, deletions, and splicing

- **Model-guided**: follow a pre-defined input specification
  - Follow “rules” to create highly-structured inputs

- **White-box approaches**:
  - **Symbolic execution**: solve branches as *symbolic* expressions
  - **Concolic execution**: solve branches as *concrete* values
  - **Taint tracking**: infer critical input “*parts*” and mutate those

Source: The Art, Science, and Engineering of Fuzzing: A Survey
Trade-offs

- **Model-agnostic:** great on simple, easy-to-solve branches
  - Need a lot of luck to solve multi-byte conditionals, checksums

- **Model-guided:** more valid inputs leads to higher coverage
  - Out of luck if specification is not defined or hard-to-define

- **White-box approaches:**
  - **Symbolic / concolic exec:** precise solving of multi-byte conditionals
  - **Taint tracking:** easily identifies key data objects, branch constraints
  - Far too **heavyweight** to deploy on all generated inputs

Source: The Art, Science, and Engineering of Fuzzing: A Survey
Recap: What does your code coverage tell you?

- **Edge coverage:**
  - Strictly increases with time
    - Ideally increases the whole time
  - Always look at **multiple trials**
    - Studies show at least **5 trials**
  - All fuzzers eventually **plateau**
    - Early plateaus indicate you are stuck
    - Revisit your approach and try again
      - Combine multiple techniques
Recap: What does your code coverage tell you?

- **Edge coverage:**
  - Strictly **increases** with time
    - Ideally increases the whole time
  - Always look at **multiple trials**
    - Studies show at least **5 trials**
  - All fuzzers eventually **plateau**
    - **Early plateaus** indicate you are stuck
    - Revisit your approach and try again
      - **Combine multiple techniques**

Hybrid Fuzzing
Hybrid Fuzzing
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

Input Generation

Conventional ( ~N inputs )

Trace code coverage, Monitor execution

New coverage

Triggers bugs

No new coverage
How most hybrid fuzzers work

Input Generation

Conventional (∼N inputs)

Alternative

Trace code coverage, Monitor execution

New coverage

Triggers bugs

No new coverage
How most hybrid fuzzers work

Input Generation

Conventional (~N inputs)

Trace code coverage, Monitor execution

New coverage

Triggers bugs

No new coverage

Alternative (<< N inputs)
How most hybrid fuzzers work

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

Conventional (e.g., AFL)
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)
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)

```
local queue
```

Alternative (e.g., symex)

```
Solve!
```

```
local queue
```
How most hybrid fuzzers work

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

Conventional (e.g., AFL)

- Local queue

Alternative (e.g., symex)

- Solve!
- Local queue
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) vs. Alternative (e.g., symex)
How most hybrid fuzzers work

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

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

- Local queue

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

- Local queue
How most hybrid fuzzers work

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

Conventional (e.g., AFL)
- Local queue
- Sync!

Alternative (e.g., symex)
- Local queue
- New code coverage?
How most hybrid fuzzers work

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

**Question:** What could go wrong?

5–7 minute group discussion
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
What could go wrong?

- Ineffective seed scheduling
What could go wrong?

- Ineffective seed scheduling
What could go wrong?

- Ineffective seed scheduling
What could go wrong?

- Ineffective seed scheduling
Solution: path prioritization

- **Idea:** invoke heavier-weight generation only *strategically*
  - **Demand launch** (e.g., Driller): when fuzzer gets “stuck”
    - Perform concolic exec when progress stalls
    - Not stuck? Continue random fuzzing
  - **Cost-based launch** (e.g., DigFuzz): on “costly” paths
    - Prioritize solving rare or unseen branches
    - Infer via lightweight program analysis
Trade-offs

- **Demand launch:** need an accurate way to determine stalling
  - Time-based: no new coverage in some time interval
  - Coverage-based: rate of change drops below some threshold
  - These heuristics are fundamentally ad-hoc

- **Cost-based launch:** subject to imprecision
  - Observed coverage provides an incomplete picture
    - Rare branches may guard ultimately *fruitless paths*
  - More precise approach is analyzing the entire program
    - Really difficult for large or *closed-source* programs
What (else) could go wrong?

- Discrepancies in program structure
  - Missing branches or paths
    - E.g., from Instrumentation differences
    - Obstructs from incomplete information
    - Not a very common problem
  - Disagreeing coverage metrics
    - E.g., basic blocks versus edges
    - Will affect test case syncing phase
    - Many test cases won’t be seen as novel
What could go wrong?

- Discrepancies in program structure
Questions?