The Art, Science, and Engineering of Fuzzing: A Survey

Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo

Fuzzing: Hack, Art, and Science*

Patrice Godefroid

Qiang Liu

Motivation

Vibrant Fuzzing Community

- Over a thousand public repos related to fuzzing on GitHub
- 112 fuzzers (big-4 security cons and three SE cons, from 08 to 19)
- 78 fuzzing related papers (from 20 to 21)

However

- Less description than source code and manual page -> Lose tracks
- Terminology fragmentation -> Hinder knowledge progress
  - AFL: test case minimization, funfuzz: test case reduction

We need to consolidate and distill the large amount of progress in fuzzing!

Outline

Three Parts

- Fuzzing Terminology
- A unified Fuzzing Model
- Stages in Fuzz Testing
  - Design Choices
  - Trade-offs

Fuzzing Terminology

PUT: Program Under Test

Definitions

- Fuzzing: Fuzzing is the execution of the PUT using input(s) sampled from an input space that **protrudes** the expected input space of the PUT.
- Fuzz Testing: The use of fuzzing to test if a PUT violates a correctness policy.
- Fuzzer: A program that performs fuzz testing on a PUT.
- Fuzz Campaign: A specific execution of a fuzzer with a specific correctness policy.
- Bug Oracle: A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific **correctness policy**.
- Fuzz Configuration: A fuzz configuration of a **fuzz algorithm** comprises **the parameter value(s)** that control(s) the fuzz algorithm.

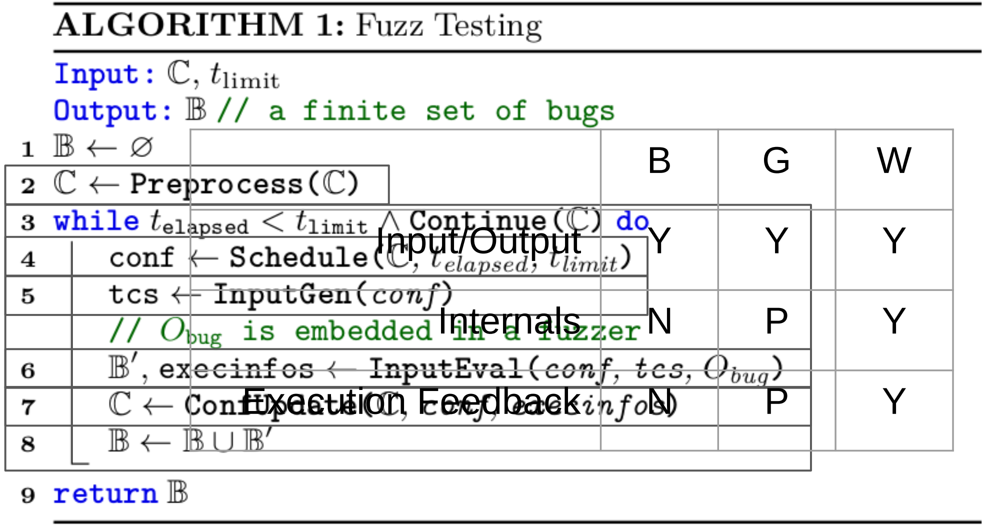
Model Fuzzer

Two parts

- Preprocess
- Fuzz Loop

Fuzzers

- Black-box Fuzzer (B)
- Grey-box Fuzzer (G)
- White-box Fuzzer (W)



ALGORITHM 1: Fuzz Testing

```
Input: $\mathbb{C}$, $t_{\text{limit}}$
Output: $\mathbb{B}$ // a finite set of bugs
$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) do
    conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, $t_{limit}$)
    tcs $\leftarrow$ InputGen($conf$)
    // $O_{\text{bug}}$ is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ InputEval($conf$, $tcs$, $O_{bug}$)
    $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, $conf$, $execinfos$)
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$
```

	B	G	W
Input/Output	Y	Y	Y
Internals	N	P	Y
Execution Feedback	N	P	Y

Preprocess

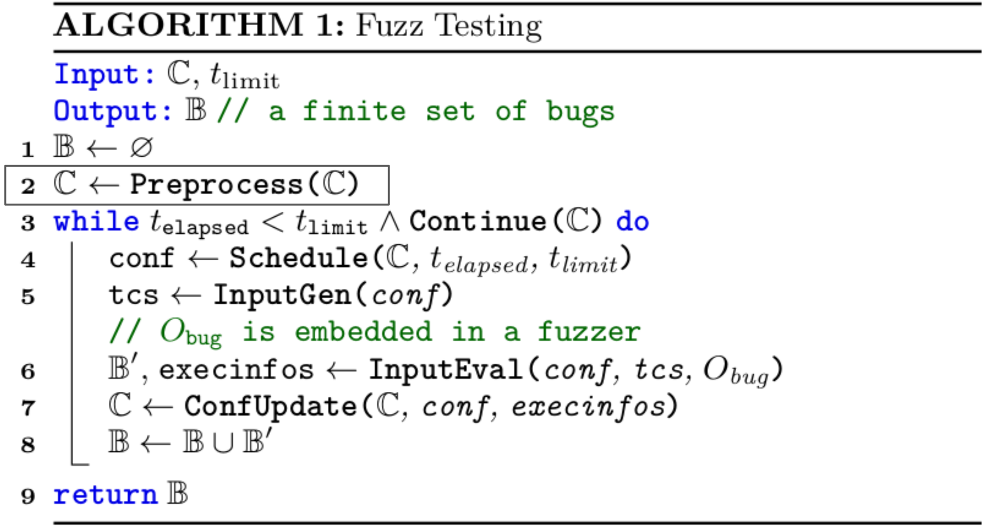
Instrumentation

Seed Selection

Seed Trimming

Driver Application

- Hard to directly fuzz
- Manual and one-time effort
  - FuzzGen (Sec’20)
  - WINNE (NDSS’21)
  - APICraft (Sec’21)
- Diverse in implementation



ALGORITHM 1: Fuzz Testing

```
Input: $\mathbb{C}$, $t_{\text{limit}}$
Output: $\mathbb{B}$ // a finite set of bugs
$\mathbb{B} \leftarrow \varnothing$
$\boxed{\mathbb{C} \leftarrow \texttt{Preprocess}(\mathbb{C})}$
while $t_{\text{elapsed}} < t_{\text{limit}} \land \texttt{Continue}(\mathbb{C})$ do
    conf $\leftarrow$ $\texttt{Schedule}(\mathbb{C}, t_{elapsed}, t_{limit})$
    tcs $\leftarrow$ $\texttt{InputGen}(conf)$
    // $O_{\text{bug}}$ is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ $\texttt{InputEval}(conf, tcs, O_{bug})$
    $\mathbb{C} \leftarrow \texttt{ConfUpdate}(\mathbb{C}, conf, execinfos)$
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$
```

Preprocess Cont’d

Instrumentation

Design Choices	Semantics Level	Overhead	Library
Static Instrumentation	Source Code, IR, Binary	Low	Separate
Dynamic Instrumentation	Binary	High	Unified

Usage Scenarios

- Execution Feedback
  - BitMap: Direct -> Colliding
- Thread Scheduling
  - Romdon Scheduling: Effective

- In-Memory Fuzzing
  - Snapshots/Fork server
  - In-Memory API Fuzzing
    - AFL Persistent Mode: Not Reproducible
    - **Delta Debugging (ViDeZZo)**

Preprocess Cont’d

Seed Selection

- Minset: find a minimal set of seeds that maximizes a coverage metric
- Coverage metric

	Coverage Metric	Preference
AFL	branch cov with **logarithmic** counter	differ in orders of magnitude
Honggfuzz	# instr, # BB, # branch	longer executions

Seed Trimming

- Smaller seeds: less memory and higher throughput
- Different intuitions

	Intuition
AFL	code coverage
USec’14	smaller size
MoonShine	dependency

Both of them can be in ConfUpdate

Schedule

Fuzz Configuration Scheduling (FCS) Problem

- Optimization: # bugs, coverage
- Confliction: exploration v.s. exploitation

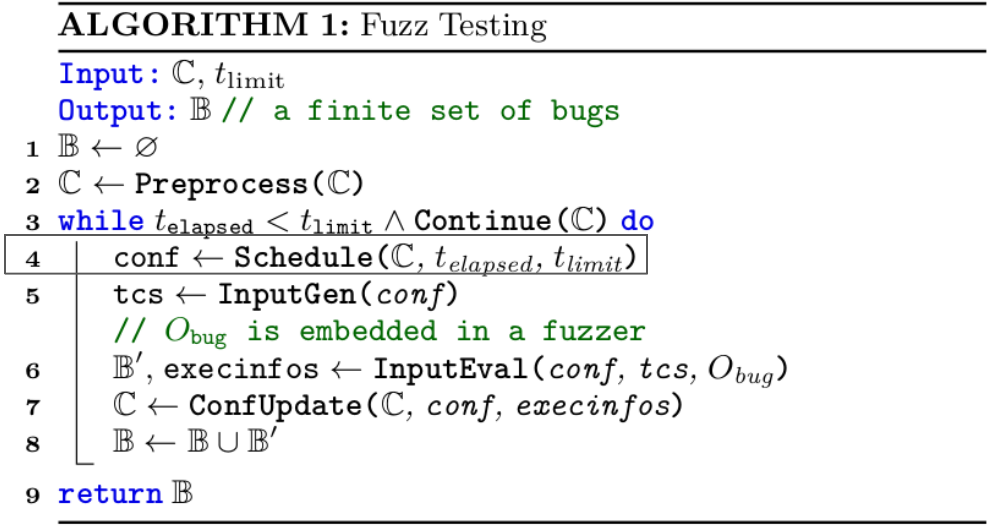
Line 4: C with new information

Black-box FCS Algorithm

- # of crashes and # of runs
- Probability Theory

Grey-box FCS Algorithm

- Richer coverage
- Evolutionary Algorithm: Fitness



ALGORITHM 1: Fuzz Testing

```
Input: $\mathbb{C}$, $t_{\text{limit}}$
Output: $\mathbb{B}$ // a finite set of bugs
1 $\mathbb{B} \leftarrow \varnothing$
2 $\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
3 while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) do
4     conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, $t_{limit}$)
5     tcs $\leftarrow$ InputGen($conf$)
      // $O_{\text{bug}}$ is embedded in a fuzzer
6     $\mathbb{B}'$, execinfos $\leftarrow$ InputEval($conf$, $tcs$, $O_{bug}$)
7     $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, $conf$, $execinfos$)
8     $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
9 return $\mathbb{B}$
```

Schedule Cont'd

AFL

- Fastest and Smallest

AFLFast: FAST power schedule

- Start with a small "energy" value to **ensure initial exploration** among configurations and increase exponentially up to a limit to **quickly ensure sufficient exploitation**
- Normalize the energy by the number of generated inputs that exercise the same path, thus promoting explorations of **less-frequently** fuzzed configurations

AFLGo

- Target specific program locations

Input Generation

Model-based (Generation)	Predefined Model	specs: tool-specific, grammar, protocol, syscall, file format	
	Inferred Model*	preprocessing	binary, seeds, API logs, …
		configuration updating	kinds of dynamic behaviors
	Encoder Model	MutaGen: Mutate the encoder program.	
Model-less (Mutation)	Bit-Flipping	# bits to flip; each PUT has a specific mutation ratio.	
	Arithmetic Mutation	integer i: i +/- r	
	Block-based Mutation	block: add, append, delete, replace, shuffle, crossover	
	Dictionary-based Mutation	“ELF\x00”	

*How to automate the generation of input grammars for complex formats, perhaps using machine learning, is another challenge.

Input Generation Cont’d

White-box Fuzzers

Dynamic Symbolic Execution	Expensive*	specify uninterested parts of a PUT
		alternate between concolic testing and grey-box fuzzing
Guided Fuzzing: costly program analysis + test case generation		hot bytes, control/data flow features
PUT Mutation: change PUT and recover		checksum, branches

* How to engineer exhaustive symbolic testing (that is, a form of verification) in a cost-effective manner is still an open problem for large applications.

Input Evaluation

Bug Oracles

- Fatal Signals -> Sanitizers
- Memory and Type Safety (ASAN)
  - spatial and temporal memory safety, control flow integrity
- Undefined Behaviors (MSAN, UBSAN, TSAN)
  - uninitialized memory, misaligned pointers, division by zero, dereferencing null pointers, and integer overflow, data races
- Others
  - Input Validation: manually specific patterns: XSS, SQL injection
  - Semantic Difference: differential testing: semantic bugs

Execution Optimization

- Fork-server, In-memory Fuzzing, In-memory API Fuzzing

Input Evaluation Cont’d

Triage

- Deduplication
  - Stack Backtrace Hashing
    - widely used
    - but “some crashes do not occur near the code that caused the crash“
  - Coverage-based Deduplication
    - “the crash covered a previously unseen edge”
  - Semantic-aware Deduplication
    - “root cause analysis”
- Prioritization
  - Exploitability
- Minimization
  - Delta-debugging, C-Reduced

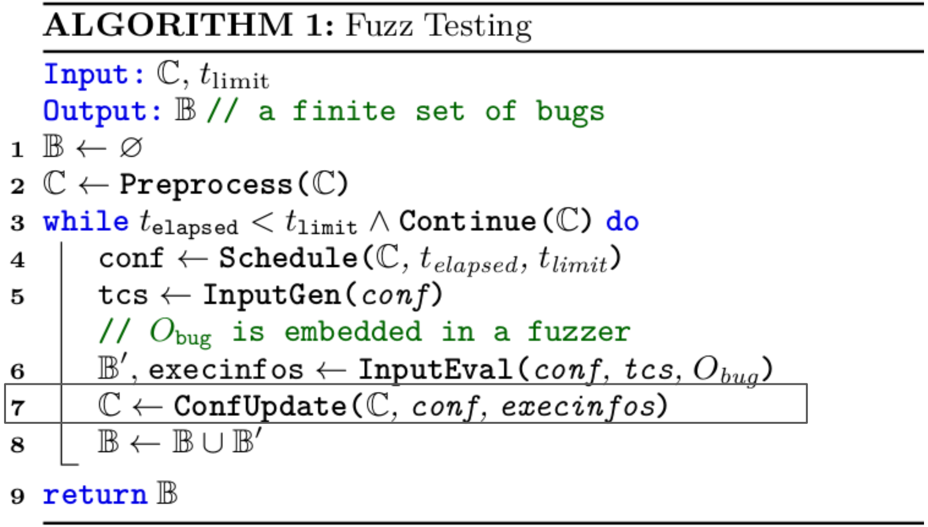
Configuration Updating

Most black-box fuzzers don’t update configurations

White-box fuzzers generate conf for each test case.

Evolutionary Seed Pool Update

- Fitness function: node or branch cov
- Refined fitness function
  - AFL: # taken branches
  - VUzzer: weights of BB



```
ALGORITHM 1: Fuzz Testing
Input: $\mathbb{C}$, $t_{\text{limit}}$
Output: $\mathbb{B}$ // a finite set of bugs
$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) do
    conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, $t_{limit}$)
    tcs $\leftarrow$ InputGen($conf$)
    // $O_{\text{bug}}$ is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ InputEval($conf$, $tcs$, $O_{bug}$)
    $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, $conf$, $execinfos$)
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$
```

Maintaining a Minset

- Avoid creating too many confs
- Variants:
  - Completely remove useless configurations
  - A culling procedure to mark minset configurations as being favorable

Conclusion

Summary

- Rich taxonomy
- A general purpose model fuzzer
- Design decisions in each stage

Hopefully help bring some more uniformity to future works, particularly in the terminology and in the presentation of fuzzing algorithms

Discussion

Questions in presentation?

When can/should we do a survey paper?

What are the generic steps to do a survey paper?

Some new directions?

Discussion

New papers since 2020: https://wcventure.github.io/FuzzingPaper/

- Speed
  - Towards Systematic and Dynamic Task Allocation for Collaborative Parallel Fuzzing (ASE’21)
  - Hardware Support to Improve Fuzzing Performance and Precision (CCS’21)
- Benchmark
  - FuzzBench: An Open Fuzzer Benchmarking Platform and Service (FSE’21)
- New feedback
  - The Use of Likely Invariants as Feedback for Fuzzers (USec’21)
  - IJON: Exploring Deep State Spaces via Fuzzing (SP’20)
  - SAVIOR: Towards Bug-Driven Hybrid Testing (SP’20)
- Emerging PUTs
  - compiler, interpreter, database, library, kernel, Hypervisor, firmware, RTL design
  - distributed systems
  - commits