# Fuzzing — A Survey

Valentin Huber

November 8, 2023

## Contents



# 1 Related Work

- A systematic review of fuzzing techniques[1]

# 2 Random notes

- “Today, testing is the primary way to check the correctness of software.

Billions of dollars are spent on testing in the software industry, as testing usually accounts for about 50% of the cost of software development. It was recently estimated that software failures currently cost the US economy alone about $60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost."[2]

# 3 Fuzzing types

## 3.1 Random input generation

### 3.1.1 An Empirical Study of the Reliability of UNIX Utilities (1990)

- [3]
- OG Fuzzing paper
- Started because in a stormy night, electrical interference on a dial-up connection
- Authors were surprised by amount of crashes, and artificially produced those.
- Generates random data (all chars/only printable chars, with or without NULL), throws them against a program
- Were able to crash or hang between 24 and 33% of programs on different UNIX systems
- Different error categories: pointer and array errors, unchecked return codes, input functions, sub-processes, interaction effects, bad error handling, signed characters, race conditions and undetermined.

## 3.2 Symbolic Execution

### 3.2.1 DART (2005)

- [2]
- Automated extraction of interface and env based on static source-code parsing
- Starts with random input, then uses symbex (without calling it symbex) to choose a different path

- Introduces a lot of concepts that I understand to be base level for symbex
- Has a unclear distinction to symbex, argues that symbex is stuck at expressions that aren't an issue with the symbex I know
- Concolic execution, fallback on concrete value whenever stuck
- Works on C code
- Positioned against static code analysis, which produces a lot of false positives while errors reported by DART are "trivially sound"[2]
- Run on a Pentium III 800MHz
- "As illustrated by the examples in Section 2, DART is able to alleviate some of the limitations of symbolic execution by exploiting dynamic information obtained from a concrete execution matching the symbolic constraints, by using dynamic test generation, and by instrumenting the program to check whether the input values generated next have the expected effect on the program."[2]

#### 3.2.2 SAGE (2008)

- [4]
- First Whitebox Fuzzing paper so far.
- Developed at Microsoft.
- Does minor optimization to be able to perform partial symbex
- New invention: "Generational Search" — flips every branching condition after a symbex run to test in the next run, thus requiring fewer symbex runs overall.
- Uses concolic symbex whenever it gets too complex (i.e. interaction with the environment). It then checks whether the expected execution path is actually chosen and if not recovers (so-called "divergence").
- Runs on x86, Windows, file-reading applications.
- Found some vulnerabilities in media parsing engines and Office 2007.
- Further findings: symbex is slow (duh), at least two orders of magnitude compared to concrete execution.

- Divergences are common (60% of runs). This is because a lot of instructions were concretized to help with performance.
- No clear correlation between coverage and crashes, only weak effect when using a block coverage based heuristic to choose next execution.

#### 3.2.3 KLEE (2008)

- [5]
- Wide array of tests including GNU COREUTILS, BUSYBOX, MINIX, and HISTAR (430K LOC, 452 programs)
- Tests programs and OS Kernel (HISTAR)
- Found multiple high-profile errors (ten fatals in COREUTILS, three older than 15 years)
- Compares functionality of different implementations of the same specs
- Checks each error on the real binary, so no false positives theoretically (but because non-determinism and bugs in KLEE there are some in practice)
- Works on LLVM basis (so not binary, doesn't work for projects where source code is unavailable)
- Extensive env modelling, including command line args, files, file metadata, env variables, failing system calls
- Path explosion combated with copy-on-write in state
- Performs query optimization (expression rewriting like mathematical simplifications, and using more efficient operations), constraint set simplification, constraint independence and a counter-example cache
- Alternates between random and coverage-optimized choice of next branch to execute
- New development: Better env modelling (not just dropping back on concrete values)

# 4 TODOs

## 4.1 Related

- Grammar-based Whitebox Fuzzing[6] (follow-up to SAGE[4])
- AFLGo (Directed Greybox Fuzzing)[7] (follow-up to Grammar-based Whitebox Fuzzing I think)

## 4.2 New

- AFL++[8]
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution[9]
- Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach[10]
- Magma: A Ground-Truth Fuzzing Benchmark[11]
- T-Fuzz: fuzzing by program transformation[12]
- Learn&Fuzz: Machine Learning for Input Fuzzing[13]

# References


[1] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[2] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), p. 213–223, Association for Computing Machinery, 2005.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, p. 32–44, dec 1990.

[4] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network and Distributed System Security Symposium*, 2008.

[5] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (USA), p. 209–224, USENIX Association, 2008.

[6] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), p. 206–215, Association for Computing Machinery, 2008.

[7] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 2329–2344, Association for Computing Machinery, 2017.

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[9] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Network and Distributed System Security Symposium*, 2016.

[10] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, (New York, NY, USA), p. 1475–1482, Association for Computing Machinery, 2018.

[11] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, jun 2021.

[12] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, 2018.

[13] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 50–59, 2017.