Symbolic Execution in Fuzzing — A Survey

Valentin Huber

November 13, 2023

Contents

1 Related Work

Article Title	Paper	Cit#	Date
Symbolic execution for software testing: three decades later[1]	ACM Computing Surveys	1035	02/2013
A systematic review of fuzzing techniques[2]	Science Computers & Security	112	06/2018
Fuzzing: A Survey[3]	Open Access	242	06/2018
The Art, Science, and Engineering of Fuzzing: A Survey[4]	IEEE Transactions on Software Engineering	395	11/2021
Fuzzing: A Survey for Roadmap[5]	ACM Computing Surveys	81	09/2022
Fuzzing vulnerability discovery techniques: Survey, challenges and future directions[6]	Science Computers & Security	16	09/2022
Demystify the Fuzzing Methods: A Comprehensive Survey[7]	ACM Computing Surveys	0	10/2023

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."[8]
- "The blackbox and whitebox strategies achieved similar results in all categories. This shows that, when testing applications with highly-structured inputs in a limited amount of time (2 hours), whitebox fuzzing, with the power of symbolic execution, does not improve much over simple blackbox fuzzing. In fact, in the code generator, those grammar-less strategies do not improve coverage much above the initial set of seed inputs."[9]

3 Fuzzing types

3.1 Random input generation

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

- [10]
- 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)

- [8]
- 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"[8]
- 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."[8]

3.2.2 SAGE (2008)

- [11]
- 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.
- tl;dr: Runs concolic symbex, records run, flips every branch condition on its own, and solves the constraint formulas to generate inputs that choose a different path at each branch.
- Struggles with highly structured input like compilers and interpreters. Issue: "Due to the enormous number of control paths in early process- ing stages, whitebox fuzzing rarely reaches parts of the application beyond these first stages."[9].
- Also: Parsers sometimes use hash functions to match tokens, which make symbex impossible because they cannot be inverted.[9].

3.2.3 KLEE (2008)

- [12]
- 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)
- "KLEE uses search heuristics on symbolic execution to achieve high code coverage."[2]

3.2.4 Grammar-based Whitebox Fuzzing (2008)

- [9]
- Follow-up to SAGE[11]
- SAGE struggled with highly structured inputs. Which is where this paper comes in.
- "We present a dynamic test generation algorithm where symbolic execution directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver."[9]
- Two main parts:
  1. "Generation of higher-level symbolic constraints, expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional symbolic bytes read as input."[9]
  2. "A custom constraint solver that solves constraints on symbolic grammar tokens. The solver looks for solutions that satisfy the constraints and are accepted by a given (context-free) grammar."[9]
- Basically wrote their own custom token-based (as opposed to bit/byte-based) symbex engine.
- Does not mark input bytes as symbolic, but the tokens returned by the tokenization function in the parser, implemented based on SAGE[11]
  - Also tries to only do this without using a grammar, so symbex based on tokens without pruning invalid inputs.
- When negating constraints allows to generate input that will be parsed (does not use *any* byte, but one that will conform to the manually provided context-free grammar).
- Also allows to quickly prove that flipping certain conditions isn't possible (while still conforming to the grammar) without even running the code.
- If the parser has more constraints than the context-free grammar provided (like basic type checks or, e. g. in network protocols, the number k followed by k records, which cannot be represented as context-free grammar), this makes the system less efficient, but the outputs are still complete.
- Requires no source modifications
- "We use the official JavaScript grammar. The grammar is quite large: 189 productions, 82 terminals (tokens), and 102 nonterminals."[9]
- Downside: Requires some domain knowledge:
  - Formal grammar structure (available for many input formats)
  - Identifying the tokenization function in the parser that needs to be instrumented (apparently usually fairly straight-forward, by looking for functions with names that contain *token, nextToken, scan* or something similar)
  - Creating a de-tokenization function to generate input byte strings from input token strings generated by a context-free constraint solver.
- This system doesn't check the lexer and parser for bugs, but one can just use traditional whitebox fuzzing (they say that coverage is similar to other approaches, but will likely not cover the error handling as well)

- Tested on IE7s JS engine

4 TODOs

4.1 Related

- AFLGo (Directed Greybox Fuzzing)[13] (follow-up to Grammar-based Whitebox Fuzzing I think)
- SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft.[14]

4.2 New

- CUTE: a concolic unit testing engine for C (2005)[15] (discussed as early idea in [2])
- TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection[16] (Tainting, discussed in [2] as improving the efficiency of fuzzing by reducing search space)
- BuzzFuzz: Taint-based directed whitebox fuzzing[17] (discussed in [2] as improving the efficiency of fuzzing by reducing search space, specifically library and system calls)
- Dowser: A Guided Fuzzer for Finding Buffer Overflow Vulnerabilities[18, 19] (discussed in [2])
- The BORG: Nanoprobing Binaries for Buffer Overreads[20]
- MoWF — Model-based whitebox fuzzing for program binaries[21]
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution[22]
- ! S2E: a platform for in-vivo multi-path analysis of software systems[23]
- ! Mayhem: Unleashing MAYHEM on Binary Code[24]
- VUzzer: Application-aware Evolutionary Fuzzing[25]
- SYMFUZZ: Program-Adaptive Mutational Fuzzing[26]
- Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach[27]
- Magma: A Ground-Truth Fuzzing Benchmark[28]

4.3 Non-Symbex

- AFL++[29]
- Learn&Fuzz: Machine Learning for Input Fuzzing[30]
- T-Fuzz: fuzzing by program transformation[31]

References

[1] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, p. 82–90, feb 2013.

[2] 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.

[3] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, pp. 1–13, 2018.

[4] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[5] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, sep 2022.

[6] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Computers & Security*, vol. 120, p. 102813, 2022.

[7] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, oct 2023.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft.," *Queue*, vol. 10, p. 20–27, jan 2012.

[15] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, p. 263–272, sep 2005.

[16] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, pp. 497–512, 2010.

[17] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 474–484, 2009.

[18] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, (USA), p. 49–64, USENIX Association, 2013.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowser: A guided fuzzer for finding buffer overflow vulnerabilities," *login Usenix Mag.*, vol. 38, 2013.

[20] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, (New York, NY, USA), p. 87–97, Association for Computing Machinery, 2015.

[21] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, (New York, NY, USA), p. 543–553, Association for Computing Machinery, 2016.

[22] 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.

[23] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 46, p. 265–278, mar 2011.

[24] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, 2012.

[25] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Network and Distributed System Security Symposium*, 2017.

[26] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, pp. 725–741, 2015.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.