Notes

Valentin Huber

Institute of Applied Information Technology
Zürich University of Applied Sciences ZHAW
contact@valentinhuber.me

December 12, 2023

Contents

1 Related Work

Article Title	Paper	Cit#	Date
All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) [1]	IEEE Symposium on Security and Privacy	1055	05/2010
Symbolic execution for software testing in practice: preliminary assessment [2]	Conference on Software Engineering	492	05/2011
Fuzzing: The State of the Art [3]	DSTO Defence Science and Technology Organisation Australia	98	02/2012
Symbolic execution for software testing: three decades later [4]	ACM Computing Surveys	1035	02/2013
An orchestrated survey of methodologies for automated software test case generation [5]	Journal of Systems and Software	879	08/2013
Network protocol fuzz testing for information systems and applications: a survey and taxonomy [6]	Multimedia Tools and Applications	21	11/2016
A Survey of Dynamic Analysis and Test Generation for JavaScript	ACM Computing Surveys	93	09/2017
A Survey of Symbolic Execution Techniques [7]	ACM Computing Surveys	726	05/2018
A systematic review of fuzzing techniques [8]	Science Computers & Security	112	06/2018
Fuzzing: A Survey [9]	Open Access	242	06/2018
Fuzzing: State of the Art [10]	IEEE Transactions on Reliability	196	06/2018
Evaluating Fuzz Testing [11]	CCS SIGSAC	561	10/2018
Fuzzing: hack, art, and science [12]	Comms of the ACM	113	01/2020
A Systematic Review of Search Strategies in Dynamic Symbolic Execution [13]	Computer Standards & Interfaces	7	10/2020
A Survey of Hybrid Fuzzing based on Symbolic Execution [14]	CIAT 2020	3	01/2021

Article Title	Paper	Cit#	Date
Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems [15]	IEEE IoT Journal	23	02/2021
Fuzzing: Challenges and Reflections [16]	IEEE Software	96	05/2021
Firmware Fuzzing: The State of the Art [17]	ACM Internetware	5	07/2021
Exploratory Review of Hybrid Fuzzing for Automated Vulnerability Detection	IEEE Access	0	09/2021
Research on Fuzzing Technology for JavaScript Engines [18]	ACM CSAE	3	10/2021
The Art, Science, and Engineering of Fuzzing: A Survey [19]	IEEE Transactions on Software Engineering	395	11/2021
Ethereum Smart Contract Analysis Tools: A Systematic Review [20]	IEEE Access	41	04/2022
Fuzzing: A Survey for Roadmap [21]	ACM Computing Surveys	81	09/2022
Fuzzing vulnerability discovery techniques: Survey, challenges and future directions [22]	Science Computers & Security	16	09/2022
Embedded fuzzing: a review of challenges, tools, and solutions [23]	Springer Cybersecurity	7	09/2022
Fuzzing of Embedded Systems: A Survey [24]	ACM Computing Surveys	11	07/2023
A Survey on the Development of Network Protocol Fuzzing Techniques [25]	MDPI Electronics	0	07/2023
Demystify the Fuzzing Methods: A Comprehensive Survey [26]	ACM Computing Surveys	0	10/2023
A systematic review of fuzzing [27]	Application of soft computing	0	10/2023

2 Authors

Author	Works	Reviews
Cristian Cadar	KLEE [28]/KLEE-FP [29], EGT [30]/EXE [31], RWset [32], Covrig [33], KATCH [34], Automatic testing of symbolic execution engines [35], JFS [36], ZESTI [37]	[2], [4]
Koushik Sen	DART [38], CUTE [39], ZEST [40], FuzzFactory [41], FairFuzz [42], JQF [43], PerfFuzz [44], RFUZZ [45], RLCheck [46], QuickSampler [47], PARTEMU [48]	[4]
George Klees		[11]
Andrew Ruef	Build It, Break It, Fix It [49]	[11]
Benji Cooper		[11]
Shiyi Wei	FIXREVERTER [50]	[11]
Michael Hicks	FIXREVERTER [50], Build It, Break It, Fix It [49], ZAFL [51], UnTracer [52], Hardware Fuzzing Pipeline [53], CGPT [54]	[11]
Patrice Godefroid	DART [38], SMART [55], SAGE [56], Learn&fuzz [57], GWF [58]	[2]

3 Improvements in Papers

- Initial seed selection: up-front analysis [59]–[61], grammar [62], [63]
- Mutation: symbex to choose how many bits to flip [64], taint analysis [65]–[68], dynamic slicing [69], seed properties [70], grammars [71], [72], language constructs knowledge [73]
- Eval: symbex when stuck [65], [74], general symbex [75], speedup through OS optimizations [76] or other low-level primitives [72], [77], [78], removing checks [79], fine-grained runtime analysis [80]
- Observation: longer running time [81], different behavior [82], additional instrumentation [66], [67], static analysis-guided searching [68], [83]
- Seed selection: areas of interest reached [42], [68], [84], [85], different algorithm [86], [87]

4 Review Papers

4.1 Symbolic execution for software testing: three decades later

- [4]
- "Note that we do not aim to provide here a comprehensive survey of existing work in the area, but instead choose to illustrate some of the main challenges and proposed solutions by using examples from the authors' own work." [4]
- "A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints)." [4]
- Two techniques that alleviate this problem:
  - Concolic Testing (like DART [38]): Run concrete and symbolic execution at the same time, keep mapping between values, solve path constraint with one sub-constraint flipped to get input for an other path.
  - Execution-Generated Testing (EGT) (like EXE [31] and KLEE [28]): Only execute symbolically if any operands are symbolic
- These handle imprecision in symbex (like interaction with outside code (that is not instrumented for symbex), constraint solving timeouts, unhandled instructions (floating point), or system calls) by just using concrete values.
  - If none of the operands are symbolically, just use them
  - If any are, use the concrete values (direct in concolic, solution from path constraint in EGT)
- Downside: missing some feasible paths, and therefore sacrificing completeness.
- Challenges:
  - Path Explosion: program path count usually exponential in the number of static branches in the code.
    - Symbex helps by only looking at possible branches. Example: EXE [31] on `tcpdump`: only 42% of instructions contained symbolic operands, less than 20% of of symbolic branches have both sides feasible [31]
    - Prioritization of which path to explore next using heuristics (like statement or branch coverage (and using static analysis to guide), favouring statements that were run the fewest number of times, or random)
    - Interleave random and symbolic execution
    - Pruning redundant paths ("if a program path reaches the same program point with the same symbolic constraints as a previously explored path, then this path will continue to execute exactly the same from that point on and thus can be discarded." [32])
    - Lazy test generation
    - Static path merging
  - Constraint Solving: Dominates runtime

    - Irrelevant constraint elimination: Generally, we go from a solvable constraint set (namely the current execution with the solution being the current concrete values) to one where only one constraint changes (the one we flipped). Typically, major major parts of the constraint set are not influenced by the change and can be excluded from what is passed to the solver. We can then just use the values from the previous iteration.
    - Incremental solving: Reuse the results of previous similar queries, because subsets of the constraints are still solved by the same results and supersets often do not invalidate existing solutions.
  - Memory Modeling: Things like modelling `int`s as mathematical integers being imprecise since it ignores over-/underflows, and pointers being hard to deal with. "On the one end of the spectrum is a system like DART [38] that only reasons about concrete pointers, or systems like CUTE [39] and CREST [88] that support only equality and inequality constraints for pointers, which can be efficiently solved.35 At the other end are systems like EXE [31], and more recently KLEE [28] and SAGE [56] that model pointers using the theory of arrays with selections and updates implemented by solvers like STP or Z3." [4]
  - Handling Concurrency: Testing usually difficult because of the inherent non-determinism. "Concolic testing was successfully combined with a variant of partial order reduction to test concurrent programs effectively." [4]

4.2 Evaluating Fuzz Testing

- [11]
- "We examined 32 recently published papers on fuzz testing located by perusing top-conference proceedings and other quality venues, and studied their experimental evaluations." [11]
- "14 out of 32 papers we examined used AFL as a baseline in their evaluation." [11]
- On `nm`, `objdump`, `cxxfilt`, `gif2png`, and `FFmpeg`
- To do proper evaluation on algorithm `A`, do this:
  - Choose baseline fuzzer `B`
  - Choose benchmark suite
  - Choose performance metric (ideal: number of bugs)
  - Choose set of config parameters like seed and duration of run
- If not done right:
  - "Fuzzing performance under the same configuration can vary substantially from run to run." [11] This happens if only one run is looked at, because they are nondeterministic and based on randomness. Perform many runs and check for statistically significant differences.
  - "Fuzzing performance can vary over the course of a run." [11]
  - Different seeds can lead to very different results.

  - "14 out of 32 papers we examined used code coverage to assess fuzzing effectiveness." [11] This might seem decent, but isn't necessarily, and looking at number of bugs is more precise. Deduplicating inputs that trigger the same bug is ineffective ("We found that all 57,142 crashing inputs deemed "unique" by coverage profiles were addressed by 9 distinct patches." [11]). Stack hashes aren't great either and are subject to false negatives. Solution: Assess against different versions of a program with/without applied bugfixes, or by using a synthetic suite.
  - Performance varies based on the chosen program under test. Choosing a diverse collection of programs is therefore critical.

- "There are many different dynamic analyses that can be described as "fuzzing." A unifying feature of fuzzers is that they operate on, and produce, concrete inputs. Otherwise, fuzzers might be instantiated with many different design choices and many different parameter settings." [11]
- Papers between 2012 and 2018, 25/32 between 2016 and 2018
- Started with 10 high-impact papers published in top security venues, chased citations, keyword search.
- Advances discussed in papers:
  - Initial seed selection: up-front analysis [59]–[61], grammar [62], [63]
  - Mutation: symbex to choose how many bits to flip [64], taint analysis [65]–[68], dynamic slicing [69], seed properties [70], grammars [71], [72], language constructs knowledge [73]
  - Eval: symbex when stuck [65], [74], general symbex [75], speedup through OS optimizations [76] or other low-level primitives [72], [77], [78], removing checks [79], fine-grained runtime analysis [80]
  - Observation: longer running time [81], different behavior [82], additional instrumentation [66], [67], static analysis-guided searching [68], [83]
  - Seed selection: areas of interest reached [42], [68], [84], [85], different algorithm [86], [87]

4.3 Symbolic Execution for Software Testing in Practice – Preliminary Assessment

5 Random notes

5.1 Random 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." [38]
- "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." [58]
- "KLEE is a redesign of EXE" [4]
- I think the EXE paper also introduced STP
- "The process begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout." [11]
- "Different fuzzers record different observations when running the program under test. In a "black box" fuzzer, a single observation is made: whether the program crashed. In "gray box" fuzzing, observations also consist of intermediate information about the execution, for example, the branches taken during execution as determined by pairs of basic block identifiers executed directly in sequence. "White box" fuzzers can make observations and modifications by exploiting the semantics of application source (or binary) code, possibly involving sophisticated reasoning. Gathering additional observations adds overhead. Different fuzzers make different choices, hoping to trade higher overhead for better bug-finding effectiveness." [11]
- "In any of these cases, the output from the fuzzer is some concrete input(s) and configurations that can be used from outside of the fuzzer to reproduce the observation. This allows software developers to confirm, reproduce, and debug issues." [11]

5.2 Introduction

- "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." [38]
- "The attack (WannaCry) startled the global economy by hitting its impact on around 230K–300K computers in about 150 countries, leading to an estimated substantial financial impact of US $4–$8 billion worldwide" [26]

- "Software testing is the most commonly used technique for validating the quality of software, but it is typically a mostly manual process that accounts for a large fraction of software development and maintenance." [2]
- Fuzzing is one of several software vulnerability techniques. [89]
- "Compared with other techniques, fuzzing requires few knowledge of targets and could be easily scaled up to large applications, and thus has become the most popular vulnerability discovery solution, especially in the industry." [9]
- "The term "fuzz" was originally coined by Miller et al. in 1990 to refer to a program that "generates a stream of random characters to be consumed by a target program" [90]" [19]
- "There are many different dynamic analyses that can be described as "fuzzing." A unifying feature of fuzzers is that they operate on, and produce, concrete inputs. Otherwise, fuzzers might be instantiated with many different design choices and many different parameter settings." [11]
- "Google could find 20K vulnerabilities in Chrome using fuzz testing" [26]
- Fuzzing is used by lots of big players: Google, Microsoft, DoD, Cisco, Adobe all employ fuzzing as part of their secure development practices, and many of those have contributed to or written their own open-source or commercial fuzzers [26].

5.3 Theoretical Principles

- Two approaches: Random mutation as described in *An Empirical Study of the Reliability of UNIX Utilities* by Miller et al. [90] and pure symbolic execution, as introduced in [91].
- The latter is infeasible for large programs and for any program that interacts with the environment, the former in its purest form is not very effective.
  - "A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints)." [4]
  - "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 grammarless strategies do not improve coverage much above the initial set of seed inputs." [58]
- Generally:
  - "The process begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout." [11]
  - "Different fuzzers record different observations when running the program under test. In a "black box" fuzzer, a single observation is made: whether the program crashed. In "gray box" fuzzing, observations also consist of intermediate information about the

execution, for example, the branches taken during execution as determined by pairs of basic block identifiers executed directly in sequence. "White box" fuzzers can make observations and modifications by exploiting the semantics of application source (or binary) code, possibly involving sophisticated reasoning. Gathering additional observations adds overhead. Different fuzzers make different choices, hoping to trade higher overhead for better bug-finding effectiveness." [11]

  - "In any of these cases, the output from the fuzzer is some concrete input(s) and configurations that can be used from outside of the fuzzer to reproduce the observation. This allows software developers to confirm, reproduce, and debug issues." [11]

- "Many of these tools also automatically find well-defined bugs, such as assertion errors, divisions by zero, NULL pointer dereferences, etc." [1]
- Symbex-based fuzzing is powerful: SAGE [56] "reportedly found a third of the Windows 7 bugs between 2007-2009" [3]

5.4 Methods

- Large scientific body of work
- Review papers
  - Well cited
  - New
  - Specific topics to see if challenges and solutions differ
- Gathered by
  - Search engines
  - Cited in review papers
  - Lists in review papers (as in [26])
  - Cited in important primary papers
- Excluded survey papers that specifically focus on a different approach such as machine learning such as [92], [93]
- This section summarizes their contribution and lists relevant primary works (as opposed to survey papers) discussed. Primary works mentioned without discussion are omitted.

5.5 Results

5.5.1 Impossible Constraints

- "A key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints)." [4]
- Two techniques that alleviate this problem:

  - Dynamic Symbolic Execution (DSE), or Concolic Testing (like DART [38] and its successor CUTE [39], and CREST [88]): Run concrete and symbolic execution at the same time, keep mapping between values, solve path constraint with one sub-constraint flipped to get input for an other path. "A key observation in DART is that imprecision in symbolic execution can be alleviated using concrete values and randomization" [2]
  - Execution-Generated Testing (EGT) [30] (like EXE [31] and KLEE [28]): Only execute symbolically if any operands are symbolic
- These handle imprecision in symbex (like interaction with outside code (that is not instrumented for symbex), constraint solving timeouts, unhandled instructions (floating point), or system calls (`read`, interrupts, etc.)) by just using concrete values.
  - If none of the operands are symbolically, just use them
  - If any are, use the concrete values (direct in concolic, solution from path constraint in EGT)
- Downside: missing some feasible paths, and therefore sacrificing completeness.
- Further Ideas: Special Constraint Solvers that improve floating point based constraint handling (like FloPSy [94]) and complex mathematical constraints (like CORAL [95] and its extension [96])

5.5.2 System Calls

- "Additionally, symbolic execution creates conflicts while handling system calls, since it does not support modeling all possible system calls and inter-process communication, such as pipes or sockets. Likewise, the non-deterministic behavior of system calls complicates the generation of inputs that consistently trigger specific paths." [26]
- HFL [97] is a kernel fuzzer that heavily relies on symbolic execution. It lists three main issues the authors had to overcome: "(1) indirect control transfers determined by system call arguments (2) controlling and matching internal system state via system calls, and (3) nested argument type inference for invoking system calls" [97]. To solve those issues, HFL "(1) converts implicit control transfers to explicit transfers, (2) infers system call sequence to build a consistent system state, and (3) identifies nested arguments types of system calls" [97].

5.5.3 Environment Interaction

- System calls (such as calls to `read` or interrupts) pose an obvious obstacle to pure symbolic execution, since they may introduce new symbolic variables or, more importantly, have side effects. This can be mitigated by manually creating summaries of these side effects (as done in EXE [31] and KLEE [28]), or, again, employing concolic execution with all the upsides and drawbacks discussed before.
- "[...KLEE's [28]] ability to handle interactions with the outside environment — e.g., with data read from the file system or over the network — by providing models designed to explore all possible legal interactions with the outside world." [2]
- The path constraints per instruction can also be generated automatically, like in ASSIE [98].
- Further automation allows Cinger [99] to analyze a PUT and prompt the user to present models only for the program parts that actually introduce imprecision.

5.5.4 Modelling

Things like modelling `int`s as mathematical integers being imprecise since it ignores over-/underflows, and pointers being hard to deal with.

- Issue: Dereferencing symbolic pointer, as in pointer which can be influenced from the input.
  - Separate pointer and integer constraints to still be able to argue about parts of the PUT, even when pointer constraints might be undecidable, as is done in CUTE [39]. CUTE only considers (in-)equality predicates with symbolic pointers.
  - EXE [31] and KLEE [28] regards symbolic pointers as array accesses. An object accessed with a symbolic pointer is copied as often as necessary to model all possible results. Or, in other words, "a sound strategy is to consider it a load from any possible satisfying assignment for the expression" [1].
  - "Symbolic memory addresses can lead to aliasing issues even along a single execution path. A potential address alias occurs when two memory operations refer to the same address." [1]
  - (Potentially) unsound assumptions: optionally rewrite all memory addresses as scalars based on name, like Vine [100]
  - Pass the dealiazing step to the SMT solver like CVC Lite [101] or STP [102].
  - Perform alias analysis. However, like in DART [38], "part of the allure of forward symbolic execution is that it can be done at run-time" [1].
  - EXE [31] and KLEE [28] "perform a mix of alias analyses and letting the SMT solver worry about aliasing" [1]
  - Other systems like DART [38] and CUTE [39] cannot handle non-linear constraints and therefore cannot deal with symbolic references.
- "On the one end of the spectrum is a system like DART [38] that only reasons about concrete pointers, or systems like CUTE [39] and CREST [88] that support only equality and inequality constraints for pointers, which can be efficiently solved. [39] At the other end are systems like EXE [31], and more recently KLEE [28] and SAGE [56] that model pointers using the theory of arrays with selections and updates implemented by solvers like STP or Z3." [4]

Symbolic target addresses of jump instructions are an obvious issue for symbolic execution based systems. Standard ways of handling these include:

- Concolic execution: Perform and trace the execution of a program under test, let it jump to the concrete address observed during this run, and finally perform symbolic execution on the trace. This leaves some potentially possible program states unexplored. Examples include CUTE [39].
- Pass the reasoning issue to the SMT solver. This however makes the SMT queries more complicated and since constraint solving is already an issue in many cases (see Section 5.5.5), this may not solve the issue after all.
- Use static analysis to locate possible jump targets.

5.5.5 Path Explosion

Search Space Reduction

- Pruning redundant paths ("if a program path reaches the same program point with the same symbolic constraints as a previously explored path, then this path will continue to execute exactly the same from that point on and thus can be discarded." [32]) Eliminating redundant paths by analyzing the values read and written by the program.
- Similarly, MoWF [103] uses knowledge gained by its built-in blackbox fuzzer to prune invalid inputs and thus prevents its symbolic execution engine to get stuck in input checking and error handling code. CESE [104] uses context-free grammars to limit its symbolic execution engine to interesting paths, as opposed to error handling during parsing. TCR [105] intelligently reduces existing test cases and prioritizes the remaining according to heuristics to maximize exploration efficiency.
- State-merging to reduce the number of states in memory, as in KLEE [28], Mayhem [65], S2E [106], BORG [107], and Cloud9 [108]
- Sharing among states/copy on write: KLEE [28]
- Transfer state from memory to disk, as in Mayhem [65], BORG [107], and SAGE [56]
- Caching function summaries for later use by higher-level functions. These can be generated automatically, like in SMART [55], HOTG [109], or DDCSE [110], or manually, like in PFA [111].
- Similarly, common complex structures like strings and regular expressions can be manually transformed into constraints. An example of this would be PFA [111].
- Lazy test generation (as in LATEST [112])
- Static path merging (as in KLEE-FP [29])
- partial order and symmetry reductions (as in GSE [113])
- Compact representation of path constraints (as in SAGE [56])

Using Advanced Data Structures

- Simple depth-first search, however this naïve approach gets stuck in non-terminating loops with symbolic conditions and is therefore rarely used. Both EXE [31] and KLEE [28] can however be configured to run in this mode.
- Symbex inherently helps by only looking at possible branches. Example: EXE [31] on `tcpdump`: only 42% of instructions contained symbolic operands, less than 20% of of symbolic branches have both sides feasible [31]
- Only explore symbolically until a user-defined timeout is reached, then use a black- or greybox fuzzer with random inputs that conform to the calculated constraints. This is called hybrid fuzzing [114].

Guiding the Execution Path Explosion: program path count usually exponential in the number of static branches in the code.

- Prioritization of which path to explore next using heuristics (like statement or branch coverage (and using static analysis to guide), favouring statements that were run the fewest number of times, or random). Examples: EXE [31], SAGE [56], CREST [88]
- "CREST [88] is an extensible platform for building and experimenting with heuristics for selecting which paths to explore" [4], allows implementing heuristics based on static analysis, namely control flow graphs.
- Prioritize inputs that increase path coverage the most, as implemented in many fuzzers, including SAGE [56]
- Guide towards changes in a patch: Directed Incremental Symbolic Execution [115], Directed Test Suite Augmentation [116], MATRIX RELOADED [117], and KATCH [34], which is based on KLEE [28].
- Guide towards interesting function calls, such as `malloc`, as in CRAXfuzz [118]
- Guide backwards from interesting parts of the PUT, as in DrillerGo [119]
- Reward inputs that lead to longer runtime, as in AGLT [120], or those that produce vastly different outputs based on very similar inputs, as in SRA [121].
- "we propose a novel approach called Fitnex, a search strategy that uses state-dependent fitness values (computed through a fitness function) to guide path exploration. The fitness function measures how close an already discovered feasible path is to a particular test target (e.g., covering a not-yet-covered branch)" [122]
- Chopped symbolic execution ignores (resp. only lazily executes) certain functions deemed uninteresting to focus on certain parts of the PUT and prevent path explosion. This can either be done manually or automatically based on some heuristics (like code unassociated with changes in a patch in Chopper [123]).
- Weigh the approximate cost of executing a certain path against its demand, as done in QuickFuzz [62]
- Assign probabilities to execution paths to reach deeper execution, as in DeepFuzz [124]
- Examine possible next seeds using machine learning and validate using symbex, as in MEUZZ [125]
- Probabilistic approach: Use Monte Carlo path optimization to quantify the difficulty of each path using grey-box fuzzing to then let the white-box fuzzer focus on the paths that are believed to be most challenging for grey-box fuzzing to make progress. [126]
- Guide execution towards code parts deemed to be interesting based on static analysis, such as pointer dereferences in loops as implemented in Dowser [83], potential bugs according to UndefinedBehaviorSanitizer [127] in SAVIOR [128], buffer over-reads in BORG [107] or more general prior static or dynamic program analysis such as in GRT [129] or VUzzer [68] to guide the symbolic execution engine.

6 Primary Papers

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

- [90]
- 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.

6.2 DART (2005)

- [38]
- 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" [38]
- 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." [38]

6.3 SAGE (2008)

- [56]
- 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 processing stages, whitebox fuzzing rarely reaches parts of the application beyond these first stages." [58].
- Also: Parsers sometimes use hash functions to match tokens, which make symbex impossible because they cannot be inverted. [58].

6.4 KLEE (2008)

- [28]
- 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 nondeterminism 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." [8]

6.5 Grammar-based Whitebox Fuzzing (2008)

- [58]
- Follow-up to SAGE [56]
- 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." [58]
- 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." [58]
  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." [58]

- 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 [56]
  - 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." [58]
- 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

7 TODOs

7.1 Related

- AFLGo (Directed Greybox Fuzzing) [84] (follow-up to Grammar-based Whitebox Fuzzing I think)
- SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. [130]

7.2 New

- CUTE: a concolic unit testing engine for C (2005) [39] (discussed as early idea in [8])
- CREST: Heuristics for Scalable Dynamic Test Generation [88] (discussed in [4])
- TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection [131] (Tainting, discussed in [8] as improving the efficiency of fuzzing by reducing search space)
- BuzzFuzz: Taint-based directed whitebox fuzzing [132] (discussed in [8] as improving the efficiency of fuzzing by reducing search space, specifically library and system calls)
- Dowser: A Guided Fuzzer for Finding Buffer Overflow Vulnerabilities [83] (discussed in [8])
- The BORG: Nanoprobing Binaries for Buffer Overreads [107]
- MoWF — Model-based whitebox fuzzing for program binaries [103]
- Driller: Augmenting Fuzzing Through Selective Symbolic Execution [74]
- ! S2E: a platform for in-vivo multi-path analysis of software systems [106]
- ! Mayhem: Unleashing MAYHEM on Binary Code [65]
- VUzzer: Application-aware Evolutionary Fuzzing [68]
- SYMFUZZ: Program-Adaptive Mutational Fuzzing [64]
- Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach [133]
- Magma: A Ground-Truth Fuzzing Benchmark [134]

7.3 Non-Symbex

- AFL++ [135]
- Learn&Fuzz: Machine Learning for Input Fuzzing [57]
- T-Fuzz: fuzzing by program transformation [79]

References

[1] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, Jul. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[2] C. Cadar, P. Godefroid, S. Khurshid, *et al.*, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, May 2011, pp. 1066–1071, ISBN: 9781450304450. DOI: 10.1145/1985793.1985995. [Online]. Available: https://doi.org/10.1145/1985793.1985995.

[3] R. McNally, K. K.-H. Yiu, D. A. Grove, and D. Gerhardy, "Fuzzing: The state of the art," *DSTO Defence Science and Technology Organisation*, Feb. 2012.

[4] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. [Online]. Available: https://doi.org/10.1145/2408776.2408795.

[5] S. Anand, E. K. Burke, T. Y. Chen, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.02.061. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213000563.

[6] T. L. Munea, H. Lim, and T. Shon, "Network protocol fuzz testing for information systems and applications: A survey and taxonomy," *Multimedia Tools and Applications*, vol. 75, no. 22, pp. 14 745–14 757, Nov. 2016, ISSN: 1573-7721. DOI: 10.1007/s11042-015-2763-6. [Online]. Available: https://doi.org/10.1007/s11042-015-2763-6.

[7] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[8] 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, Jun. 2018, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2018.02.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818300658.

[9] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, p. 6, Jun. 2018, ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y. [Online]. Available: https://doi.org/10.1186/s42400-018-0002-y.

[10] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018. DOI: 10.1109/TR.2018.2834476.

[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, Oct. 2018, pp. 2123–2138, ISBN: 9781450356930. DOI: 10.1145/3243734.3243804. [Online]. Available: https://doi.org/10.1145/3243734.3243804.

[12] P. Godefroid, "Fuzzing: Hack, art, and science," *Commun. ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020, ISSN: 0001-0782. DOI: 10.1145/3363824. [Online]. Available: https://doi.org/10.1145/3363824.

[13] A. Sabbaghi and M. R. Keyvanpour, "A systematic review of search strategies in dynamic symbolic execution," *Computer Standards & Interfaces*, vol. 72, p. 103 444, Oct. 2020, ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2020.103444. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548919300066.

[14] T. Zhang, Y. Jiang, R. Guo, X. Zheng, and H. Lu, "A survey of hybrid fuzzing based on symbolic execution," in *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies*, ser. CIAT 2020, Guangzhou, China: Association for Computing Machinery, Jan. 2021, pp. 192–196, ISBN: 9781450387828. DOI: 10.1145/3444370.3444570. [Online]. Available: https://doi.org/10.1145/3444370.3444570.

[15] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 390–10 411, Jul. 2021. DOI: 10.1109/JIOT.2021.3056179.

[16] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, May 2021. DOI: 10.1109/MS.2020.3016773.

[17] C. Zhang, Y. Wang, and L. Wang, "Firmware fuzzing: The state of the art," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, ser. Internetware '20, Singapore, Singapore: Association for Computing Machinery, Jul. 2021, pp. 110–115, ISBN: 9781450388191. DOI: 10.1145/3457913.3457934. [Online]. Available: https://doi.org/10.1145/3457913.3457934.

[18] Y. Tian, X. Qin, and S. Gan, "Research on fuzzing technology for javascript engines," in *Proceedings of the 5th International Conference on Computer Science and Application Engineering*, ser. CSAE '21, Sanya, China: Association for Computing Machinery, Dec. 2021, ISBN: 9781450389853. DOI: 10.1145/3487075.3487107. [Online]. Available: https://doi.org/10.1145/3487075.3487107.

[19] V. J. Manès, H. Han, C. Han, *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021. DOI: 10.1109/TSE.2019.2946563.

[20] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, Apr. 2022. DOI: 10.1109/ACCESS.2022.3169902.

[21] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, Sep. 2022, ISSN: 0360-0300. DOI: 10.1145/3512345. [Online]. Available: https://doi.org/10.1145/3512345.

[22] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Computers & Security*, vol. 120, p. 102 813, Sep. 2022, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102813. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822002073.

[23] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella, "Embedded fuzzing: A review of challenges, tools, and solutions," *Cybersecurity*, vol. 5, no. 1, p. 18, Sep. 2022, ISSN: 2523-3246. DOI: 10.1186/s42400-022-00123-y. [Online]. Available: https://doi.org/10.1186/s42400-022-00123-y.

[24] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022, ISSN: 0360-0300. DOI: 10.1145/3538644. [Online]. Available: https://doi.org/10.1145/3538644.

[25] Z. Zhang, H. Zhang, J. Zhao, and Y. Yin, "A survey on the development of network protocol fuzzing techniques," *Electronics*, vol. 12, no. 13, Jul. 2023, ISSN: 2079-9292. DOI: 10.3390/electronics12132904. [Online]. Available: https://www.mdpi.com/2079-9292/12/13/2904.

[26] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, Oct. 2023, ISSN: 0360-0300. DOI: 10.1145/3623375. [Online]. Available: https://doi.org/10.1145/3623375.

[27] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, "A systematic review of fuzzing," *Soft Computing*, pp. 1–30, Oct. 2023.

[28] 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*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[29] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and simd code," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, Salzburg, Austria: Association for Computing Machinery, 2011, pp. 315–328, ISBN: 9781450306348. DOI: 10.1145/1966445.1966475. [Online]. Available: https://doi.org/10.1145/1966445.1966475.

[30] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Model Checking Software*, P. Godefroid, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 2–23, ISBN: 978-3-540-31899-6.

[31] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008, ISSN: 1094-9224. DOI: 10.1145/1455518.1455522. [Online]. Available: https://doi.org/10.1145/1455518.1455522.

[32] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, Budapest, Hungary: Springer-Verlag, 2008, pp. 351–366, ISBN: 3540787992.

[33] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 93–104, ISBN: 9781450326452. DOI: 10.1145/2610384.2610419. [Online]. Available: https://doi.org/10.1145/2610384.2610419.

[34] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 235–245, ISBN: 9781450322379. DOI: 10.1145/2491411.2491438. [Online]. Available: https://doi.org/10.1145/2491411.2491438.

[35] T. Kapus and C. Cadar, "Automatic testing of symbolic execution engines via program generation and differential testing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 590–600. DOI: 10.1109/ASE.2017.8115669.

[36] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 521–532, ISBN: 9781450355728. DOI: 10.1145/3338906.3338921. [Online]. Available: https://doi.org/10.1145/3338906.3338921.

[37] P. Dan Marinescu and C. Cadar, "Make test-zesti: A symbolic execution solution for improving regression testing," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 716–726. DOI: 10.1109/ICSE.2012.6227146.

[38] 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*, ser. PLDI '05, Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223, ISBN: 1595930566. DOI: 10.1145/1065010.1065036. [Online]. Available: https://doi.org/10.1145/1065010.1065036.

[39] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005, ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. [Online]. Available: https://doi.org/10.1145/1095430.1081750.

[40] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 329–340, ISBN: 9781450362245. DOI: 10.1145/3293882.3330576. [Online]. Available: https://doi.org/10.1145/3293882.3330576.

[41] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: Domain-specific fuzzing with waypoints," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360600. [Online]. Available: https://doi.org/10.1145/3360600.

[42] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, Montpellier, France: Association for Computing Machinery, 2018, pp. 475–485, ISBN: 9781450359375. DOI: 10.1145/3238147.3238176. [Online]. Available: https://doi.org/10.1145/3238147.3238176.

[43] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 398–401, ISBN: 9781450362245. DOI: 10.1145/3293882.3339002. [Online]. Available: https://doi.org/10.1145/3293882.3339002.

[44] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 254–265, ISBN: 9781450356992. DOI: 10.1145/3213846.3213874. [Online]. Available: https://doi.org/10.1145/3213846.3213874.

[45] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8. DOI: 10.1145/3240765.3240842.

[46] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1410–1421, ISBN: 9781450371216. DOI: 10.1145/3377811.3380399. [Online]. Available: https://doi.org/10.1145/3377811.3380399.

[47] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of sat solutions for testing," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 549–559, ISBN: 9781450356381. DOI: 10.1145/3180155.3180248. [Online]. Available: https://doi.org/10.1145/3180155.3180248.

[48] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of Real-World TrustZone software using emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 789–806, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/harrison.

[49] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build it, break it, fix it: Contesting secure development," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 690–703, ISBN: 9781450341394. DOI: 10.1145/2976749.2978382. [Online]. Available: https://doi.org/10.1145/2976749.2978382.

[50] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3699–3715, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong.

[51] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 1683–1700, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[52] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 787–802. DOI: 10.1109/SP.2019.00069.

[53] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel.

[54] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360607. [Online]. Available: https://doi.org/10.1145/3360607.

[55] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07, Nice, France: Association for Computing Machinery, 2007, pp. 47–54, ISBN: 1595935754. DOI: 10.1145/1190216.1190226. [Online]. Available: https://doi.org/10.1145/1190216.1190226.

[56] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," Nov. 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[57] 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)*, 2017, pp. 50–59. DOI: 10.1109/ASE.2017.8115618.

[58] 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*, ser. PLDI '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 206–215, ISBN: 9781595938602. DOI: 10.1145/1375581.1375607. [Online]. Available: https://doi.org/10.1145/1375581.1375607.

[59] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594. DOI: 10.1109/SP.2017.23.

[60] B. Shastry, M. Leutner, T. Fiebig, *et al.*, "Static program analysis as a fuzzing aid," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., Cham: Springer International Publishing, 2017, pp. 26–47, ISBN: 978-3-319-66332-6.

[61] J. Corina, A. Machiry, C. Salls, *et al.*, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2123–2138, ISBN: 9781450349468. DOI: 10.1145/3133956.3134069. [Online]. Available: https://doi.org/10.1145/3133956.3134069.

[62] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," *SIGPLAN Not.*, vol. 51, no. 12, pp. 13–20, Sep. 2016, ISSN: 0362-1340. DOI: 10.1145/3241625.2976017. [Online]. Available: https://doi.org/10.1145/3241625.2976017.

[63] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "Quickfuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, pp. 340–354, 2017, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.09.018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217302066.

[64] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 725–741. DOI: 10.1109/SP.2015.50.

[65] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.

[66] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[67] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 627–637, ISBN: 9781450351058. DOI: 10.1145/3106237.3106295. [Online]. Available: https://doi.org/10.1145/3106237.3106295.

[68] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, vol. 17, 2017, pp. 1–14.

[69] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 782–792, ISBN: 9781450336758. DOI: 10.1145/2786805.2786844. [Online]. Available: https://doi.org/10.1145/2786805.2786844.

[70] Y.-D. Lin, F.-Z. Liao, S.-K. Huang, and Y.-C. Lai, "Browser fuzzing by scheduled mutation and generation of document object models," in *2015 International Carnahan Conference on Security Technology (ICCST)*, 2015, pp. 1–6. DOI: 10.1109/CCST.2015.7389677.

[71] H. Yoo and T. Shon, "Grammar-based adaptive fuzzing: Evaluation on scada modbus protocol," in *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2016, pp. 557–563. DOI: 10.1109/SmartGridComm.2016.7778820.

[72] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2345–2358, ISBN: 9781450349468. DOI: 10.1145/3133956.3134103. [Online]. Available: https://doi.org/10.1145/3133956.3134103.

[73] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 1–11. DOI: 10.1109/ISSRE.2017.16.

[74] N. Stephens, J. Grosen, C. Salls, *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[75] B. Zhang, J. Ye, C. Feng, and C. Tang, "S2f: Discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, 2017, pp. 548–552. DOI: 10.1109/CIS.2017.00127.

[76] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2313–2328, ISBN: 9781450349468. DOI: 10.1145/3133956.3134046. [Online]. Available: https://doi.org/10.1145/3133956.3134046.

[77] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "Vdf: Targeted evolutionary fuzz testing of virtual devices," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., Cham: Springer International Publishing, 2017, pp. 3–25, ISBN: 978-3-319-66332-6.

[78] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[79] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[80] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," in *Network and Distributed System Security Symposium*, 2018.

[81] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2155–2168, ISBN: 9781450349468. DOI: 10.1145/3133956.3134073. [Online]. Available: https://doi.org/10.1145/3133956.3134073.

[82] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632. DOI: 10.1109/SP.2017.27.

[83] 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*, ser. SEC'13, Washington, D.C.: USENIX Association, 2013, pp. 49–64, ISBN: 9781931971034.

[84] 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*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344, ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. [Online]. Available: https://doi.org/10.1145/3133956.3134020.

[85] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 1032–1043, ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. [Online]. Available: https://doi.org/10.1145/2976749.2978428.

[86] A. Rebert, S. K. Cha, T. Avgerinos, *et al.*, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, San Diego, CA: USENIX Association, 2014, pp. 861–875, ISBN: 9781931971157.

[87] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 511–522, ISBN: 9781450324779. DOI: 10.1145/2508859.2516736. [Online]. Available: https://doi.org/10.1145/2508859.2516736.

[88] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[89] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 152–156. DOI: 10.1109/MINES.2012.202.

[90] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[91] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[92] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," *arXiv preprint arXiv:1906.11133*, 2019.

[93] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, vol. 15, no. 8, e0237749, 2020.

[94] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "Flopsy - search-based floating point constraint solving for symbolic execution," in *Testing Software and Systems*, A. Petrenko, A. Simão, and J. C. Maldonado, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–157, ISBN: 978-3-642-16573-3.

[95] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "Coral: Solving complex constraints for symbolic pathfinder," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 359–374, ISBN: 978-3-642-20398-5.

[96] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu, "Symbolic execution with interval solving and meta-heuristic search," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 111–120. DOI: 10.1109/ICST.2012.91.

[97] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *Network and Distributed System Security Symposium*, 2020.

[98] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 441–452, ISBN: 9781450312059. DOI: 10.1145/2254064.2254116. [Online]. Available: https://doi.org/10.1145/2254064.2254116.

[99] S. Anand and M. J. Harrold, "Heap cloning: Enabling dynamic symbolic execution of java programs," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 33–42. DOI: 10.1109/ASE.2011.6100071.

[100] D. Song, D. Brumley, H. Yin, *et al.*, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[101] C. Barrett and S. Berezin, "Cvc lite: A new implementation of the cooperating validity checker," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 515–518, ISBN: 978-3-540-27813-9.

[102] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 519–531, ISBN: 978-3-540-73368-3.

[103] 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*, ser. ASE '16, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 543–553, ISBN: 9781450338455. DOI: 10.1145/2970276.2970316. [Online]. Available: https://doi.org/10.1145/2970276.2970316.

[104] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 134–143, ISBN: 9781595938824. DOI: 10.1145/1321631.1321653. [Online]. Available: https://doi.org/10.1145/1321631.1321653.

[105] C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 160–170, ISBN: 9781450326452. DOI: 10.1145/2610384.2610392. [Online]. Available: https://doi.org/10.1145/2610384.2610392.

[106] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 46, no. 3, pp. 265–278, Mar. 2011, ISSN: 0362-1340. DOI: 10.1145/1961296.1950396. [Online]. Available: https://doi.org/10.1145/1961296.1950396.

[107] 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*, ser. CODASPY '15, San Antonio, Texas, USA: Association for Computing Machinery, 2015, pp. 87–97, ISBN: 9781450331913. DOI: 10.1145/2699026.2699098. [Online]. Available: https://doi.org/10.1145/2699026.2699098.

[108] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254088. [Online]. Available: https://doi.org/10.1145/2345156.2254088.

[109] P. Godefroid, "Higher-order test generation," *SIGPLAN Not.*, vol. 46, no. 6, pp. 258–269, Jun. 2011, ISSN: 0362-1340. DOI: 10.1145/1993316.1993529. [Online]. Available: https://doi.org/10.1145/1993316.1993529.

[110] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3.

[111] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 307–321, ISBN: 978-3-642-00768-2.

[112] R. Majumdar and K. Sen, "Latest: Lazy dynamic test input generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-36, Mar. 2007. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-36.html`.

[113] S. Khurshid, C. S. PĂsĂreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568, ISBN: 978-3-540-36577-8.

[114] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," M.S. thesis, School of Computer Science Carnegie Mellon University, May 2012.

[115] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, Oct. 2014, ISSN: 1049-331X. DOI: `10.1145/2629536`. [Online]. Available: `https://doi.org/10.1145/2629536`.

[116] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 257–266, ISBN: 9781605587912. DOI: `10.1145/1882291.1882330`. [Online]. Available: `https://doi.org/10.1145/1882291.1882330`.

[117] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 218–227. DOI: `10.1109/ASE.2008.32`.

[118] C.-C. Yeh, H. Chung, and S.-K. Huang, "Craxfuzz: Target-aware symbolic fuzz testing," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, 2015, pp. 460–471. DOI: `10.1109/COMPSAC.2015.99`.

[119] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 2637–2639, ISBN: 9781450367479. DOI: `10.1145/3319535.3363275`. [Online]. Available: `https://doi.org/10.1145/3319535.3363275`.

[120] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 43–52. DOI: `10.1109/ASE.2011.6100093`.

[121] R. Majumdar and I. Saha, "Symbolic robustness analysis," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 355–363. DOI: `10.1109/RTSS.2009.17`.

[122] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 359–368. DOI: `10.1109/DSN.2009.5270315`.

[123] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 350–360, ISBN: 9781450356381. DOI: `10.1145/3180155.3180251`. [Online]. Available: `https://doi.org/10.1145/3180155.3180251`.

[124] K. Böttinger and C. Eckert, "Deepfuzz: Triggering vulnerabilities deeply hidden in binaries," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., Cham: Springer International Publishing, 2016, pp. 25–34, ISBN: 978-3-319-40667-1.

[125] Y. Chen, M. Ahmadi, R. M. farkhani, B. Wang, and L. Lu, "MEUZZ: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, San Sebastian: USENIX Association, Oct. 2020, pp. 77–92, ISBN: 978-1-939133-18-2. [Online]. Available: `https://www.usenix.org/conference/raid2020/presentation/chen`.

[126] L. Zhao, P. Cao, Y. Duan, H. Yin, and J. Xuan, "Probabilistic path prioritization for hybrid fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1955–1973, 2022. DOI: 10.1109/TDSC.2020.3042259.

[127] "Undefinedbehaviorsanitizer." (2023), [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (visited on 11/25/2023).

[128] Y. Chen, P. Li, J. Xu, *et al.*, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1580–1596. DOI: 10.1109/SP40000.2020.00002.

[129] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "Grt: Program-analysis-guided random testing (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 212–223. DOI: 10.1109/ASE.2015.49.

[130] 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, no. 1, pp. 20–27, Jan. 2012, ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. [Online]. Available: https://doi.org/10.1145/2090147.2094081.

[131] 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*, 2010, pp. 497–512. DOI: 10.1109/SP.2010.37.

[132] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[133] 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*, ser. SAC '18, Pau, France: Association for Computing Machinery, 2018, pp. 1475–1482, ISBN: 9781450351911. DOI: 10.1145/3167132.3167289. [Online]. Available: https://doi.org/10.1145/3167132.3167289.

[134] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Jun. 2021. DOI: 10.1145/3428334. [Online]. Available: https://doi.org/10.1145/3428334.

[135] 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. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi.