

Fuzzing: Challenges and Reflections

Marcel Böhme

Monash University, Australia

Cristian Cadar

Imperial College London, UK

Abhik Roychoudhury

National University of Singapore, Singapore

Abstract—Fuzzing is a method to discover software bugs and vulnerabilities by automatic test input generation which has found tremendous recent interest in both academia and industry. Fuzzing comes in the form of several techniques. On one hand, we have symbolic execution, which enables a particularly effective approach to fuzzing by systematically enumerating the paths of a program. On the other hand, we have random input generation, which generates large amounts of inputs per second with none or minimal program analysis overhead. In this article, we summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting, and were validated in a subsequent survey. We take a forward-looking view of the software vulnerability discovery technologies and provide concrete directions for future research.

Introduction

The Internet and the world's Digital Economy runs on a shared, critical open-source software (OSS) infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, fuzzing is one of the most promising techniques in this regard. *Fuzzing* is an automatic bug and vulnerability discovery technique which continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: blackbox,

greybox and whitebox fuzzing.

Blackbox fuzzing generates inputs without any knowledge of the program. There are two main variants of blackbox fuzzing: mutational and generational. In mutational blackbox fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational blackbox fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (<http://community.peachfuzzer.com>) is one popular blackbox fuzzer.

Greybox fuzzing leverages program instrumentation to get lightweight feedback which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented

at compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations, and thus increase code coverage, are added to the seed corpus. The coverage-feedback allows a greybox fuzzer to gradually reach deeper into the code. In order to identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing greybox fuzzing tools include AFL (<https://lcamtuf.coredump.cx/afl/>), LibFuzzer (<https://lvm.org/docs/LibFuzzer.html>), and Honggfuzz (<https://github.com/google/honggfuzz>).

Whitebox fuzzing is based on a technique called *symbolic execution* [6], which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back-end in whitebox fuzzing are Satisfiability Modulo Theory (SMT) solvers, which allow for reasoning about (quantifier free) first-order logic formulas with equality and function/predicate symbols drawn from different background theories. Whitebox fuzzers calculate the *path condition* of an input i —the set of inputs which traverse the same path as i . The path condition is represented as an SMT formula, e.g. $i[0] = 42 \wedge i[0] - i[1] > 7$.

Given a seed input s , the path condition is calculated and mutated (as opposed to mutating the program input). The mutated path condition is then sent to a constraint solver to generate new inputs. The main benefit of this technique is that by carefully keeping track of path conditions of all inputs seen so far, it always generates an input traversing a new path (new control flow). Existing whitebox fuzzing tools include KLEE [5] and SAGE [10].

In this article, we provide reflections on recent advances in the field as well as concrete directions for future research. We discuss recent impact and enumerate open research challenges from the perspective of both practitioners and researchers. For a detailed, technical review, we refer the reader to Godefroid [9].

Recent Impact

Fuzzing for automatic bug and vulnerability discovery has taken by storm both the software industry and the research community. The research problem of finding bugs in a program by auto-

matic input generation has a long-standing history which began well before Miller’s inception of the term “fuzzing” in 1990 (<http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html>). Yet, only now do we see mainstream deployment of fuzzing technology in industry.

Using greybox fuzzing, Google has discovered more than 16,000 bugs in the Chrome browser over the past eight years and more than 11,000 bugs in over 160 open-source software projects over the past three years (<https://google.github.io/clusterfuzz/#trophies>). Microsoft credits its whitebox fuzzing tool SAGE with saving millions of dollars during the development of Windows 7 [10]. Trail of Bits has been developing various fuzzing tools, including DeepState, a unit testing framework that allows developers to fuzz the various units of their system (<https://github.com/trailofbits/deepstate>). The 2016 DARPA Cyber Grand Challenge had machines attack and defend against other machines by exploiting and hardening against software vulnerabilities. The Mayhem system [4], which was awarded two million dollars for winning the competition, made extensive use of whitebox fuzzing (<https://www.darpa.mil/news-events/2016-08-04>).

What has enabled this recent surge of interest in fuzzing? First, there is a tremendous *need*. Life and business are increasingly permeated by software systems, and a security vulnerability in even the smallest system can have dire consequences. Second, we now have the *incentives* and the required mindset. Some software companies have established lucrative bug bounty programs that pay top dollar for critical bugs. Anyone, including the reader, can offer vulnerability rewards on bug bounty platforms, such as HackerOne (<https://www.hackerone.com/>), which provides ethical coordination and responsible disclosure. Independent security researchers can report the discovered vulnerabilities and collect the bounties. Some stakeholders take matters into their own hands, with several companies continuously fuzzing their own software.

Third, we now have the *tools*. Many fuzzers are open-source, freely available, easy to use, and very successful in finding bugs. For instance, the KLEE symbolic execution engine (<https://klee.github.io/>) has been freely available, maintained and widely-used for more than ten years. As

a result, several companies, such as Baidu, Fujitsu and Samsung, have used and extended it to test their software products. Similarly, the AFL greybox fuzzer (<http://lcamtuf.coredump.cx/afl/>) is highly effective and easy to use. Its trophy case includes bugs and security vulnerabilities found in a large number of open-source systems.

Lastly, this open-science approach and meaningful engagement between industry and academia has facilitated *rapid advances* in fuzzing. For instance, fuzzers are getting faster, find more types of bugs and work for more application domains.

Challenges

In September 2019, we organized a Shonan meeting on Fuzzing and Symbolic Execution in Shonan Village Center, Japan (<https://shonan.nii.ac.jp/seminars/160/>). The meeting brought together thought leaders, distinguished researchers, tool builders, founders, and promising young scientists from the greybox and whitebox fuzzing (symbolic execution) communities. Below, we discuss the main challenges identified during the meeting. We phrase the challenges as research questions and hope that they provide guidance and direction going forward.

Automation

Automated vulnerability discovery is a game between adversaries. Given the same resources, the adversary with the fuzzer that finds more vulnerabilities has the advantage.

[C.1] More Software. *How can we fuzz efficiently more types of software systems?* We already know how to fuzz command-line tools (AFL, KLEE) and APIs (LibFuzzer). The fuzzer generates inputs and observes the program's output. The community is actively working on how to fuzz programs that take highly-structured inputs, such as file parsers or object-oriented programs. However, fuzzing *cyber-physical systems*, which interact with the environment as part of their execution, or *machine learning systems*, whose behavior is determined by their training data, is an under-explored area.

How do we fuzz *stateful software*, such as protocol implementations, which can produce different outputs for the same input? Most greybox and whitebox fuzzers are written with a single

programming language in mind. How do we fuzz *polyglot software* which is written in several languages? How do we fuzz *GUI-based programs* that take as inputs a sequence of events executed on a user interface? For whitebox fuzzing, we already know how symbolic execution can formulate constraints on numeric or string-based input domains. However, given a program whose input domain is defined by a grammar and/or protocol, how can a symbolic execution tool effectively formulate constraints on such „structured” input domains?

[C.2] More Bug Types. *How can the fuzzer identify more types of vulnerabilities?* A significant portion of current works on fuzzing focus on simple oracles, such as finding crashes. We need studies of security-critical classes of bugs that do not manifest as crashes and develop oracles that can efficiently detect them. Vulnerabilities are often encoded as assertions on the program state. Using such assertions, we already know how we can discover memory- or concurrency related errors. The discovery of side-channel vulnerabilities, such as information leaks or timing, cache, or energy-related side-channels is currently an active research topic [15]. Going forward, we should invent techniques to automatically detect and invoke privilege escalation, remote code execution, and other types of critical security flaws, not only in C/C++ but also in other programming languages.

[C.3] More Difficult Bugs. *How can we find “deep bugs” for which efficient oracles exist, but which nevertheless evade detection?* There are bugs that evade discovery despite long fuzzing campaigns, e.g. because they are guarded by complex conditions, or because existing techniques require impractical amounts of resources to find them. Are there certain kinds of deep bugs that can be found efficiently with specialized approaches? Structure-aware and grammar-based fuzzing as well as the integration of static analysis and symbolic execution with greybox fuzzing are promising directions [11], [19]. Second, software also changes all the time—techniques that can target software patches will prove essential for finding bugs as they are introduced [2], [14]. Third, we should investigate strategies to boost fault finding, such as AFLFast which enables

faster crash detection in greybox fuzzers [1], and study the utility of GPUs and other means of efficient parallelization to maximize the number of executions per unit time [16]. Finally, ranking bugs in terms of their importance can also improve the effectiveness of fuzzing in practice.

[C.4] More Empirical Studies. *What is the nature of vulnerabilities that have evaded discovery despite long fuzzing campaigns? Why have they evaded discovery? We need empirical studies to understand the nature and distribution of security vulnerabilities in source code.*

The Human Component

[C.5] Human-In-The-Loop. *How can fuzzers leverage the ingenuity of the auditor? Many researchers think of fuzzing as a fully automated process that involves the human only at the beginning when the software system is prepared for the fuzzer and at the end when the fuzzer-discovered vulnerabilities need to be reported. In reality, security auditors use fuzzers in an iterative manner. During our meeting, Ned Williamson, a prolific security researcher at Google, demonstrated his semi-automated approach to vulnerability discovery. Ned would first audit the code to identify units that may contain a security flaw. He would prepare the unit for fuzzing, run the fuzzer for a while, and identify roadblocks for the fuzzer. Ned would manually patch out the roadblock to help the fuzzer make better progress. If the fuzzer spends more time fuzzing less relevant portions of the code, Ned would adjust the test driver and re-focus the fuzzer. Once a potential vulnerability is found, he would backtrack, add each roadblock back, and adjust the vulnerability-exposing input accordingly.*

This semi-automated process raises several research questions. How can we facilitate a more effective communication between fuzzer and security auditor? How can the security auditor dynamically direct the fuzzer? How can the fuzzer explain what prevents it from progressing, and how can the auditor instruct the fuzzer to overcome the roadblock?

[C.6] Usability. *How can we improve the usability of fuzzing tools? Ethical hacking requires a very special set of skills. Fuzzing already simplifies the process by automating at least the*

test input generation. How can we make fuzzing more accessible to developers and software engineers? How can we make it easier to develop test drivers for fuzzers? How can we integrate fuzzing into the day-to-day development process, e.g., as component of the CI-pipeline or as a fuzz-driven unit testing tool in the IDE? Particularly our industry participants and respondents identified usability as most important.

How can we prepare the output of a fuzzer for human consumption? A fuzzer produces an input that crashes the program and the developer must find out why it crashes. How can we extend the fuzzer such that it generates a detailed bug report or even a bug fix for each identified vulnerability? Automated repair techniques which have emerged recently can help in this regard [13]. Recent work on Linux kernel fuzzing [18] discusses techniques to address usability challenges while deploying the kernel fuzzer syzkaller on enterprise Linux distributions. Generalizing such enhancements to a fuzzer for general-purpose software remains a challenge.

Fuzzing Theory

It is important for any discipline to stand on a firm scientific foundation. We have seen many technical advances in the engineering of fuzzing tools. But why do some fuzzers work so much better than others? What are their limitations? We want to be able to explain interesting phenomena that we have observed empirically, make predictions and extrapolate from these observations. To do this, we need a sound theoretical model of the fuzzing process.

[C.7] How can we assess residual security risk if the fuzzing campaign was unsuccessful? Blackbox and whitebox fuzzing sit on two ends of a spectrum. A whitebox fuzzer might provide a formal guarantee about the absence of detectable vulnerabilities. If we assume that a symbolic execution engine can enumerate all paths in piece of code and the oracle is encoded as assertions, then whitebox fuzzing can formally verify the absence of bugs. If it can enumerate only some paths in reasonable time, we can still provide partial guarantees [8]. To make symbolic execution applicable in practice, correctness or completeness are traded for scalability. How does this trade-off affect the guarantees?

In contrast, a *blackbox fuzzer* can never guarantee the absence of vulnerabilities for all inputs. What is the residual risk that at the end of a fuzzing campaign a bug still exists in the program that has not been found? If we model blackbox fuzzing as a random sampling from the program’s input space, we can leverage methods from applied statistics to *estimate* the residual risk.

A *greybox fuzzer* uses program feedback to boost the efficiency of finding errors. However, this program feedback introduces an adaptive bias. How do we account for this adaptive bias when assessing residual risk? To answer such questions, we should develop statistical and probabilistic frameworks, and methodologies for sound estimation with quantifiable accuracy.

[C.8] *What are the theoretical limitations of blackbox, greybox, and whitebox fuzzing?* Black- and greybox fuzzers are highly efficient—but at the cost of effectiveness. Unlike whitebox fuzzers, they struggle to generate inputs that exercise paths frequented by few inputs. This tension raises several research questions. Given a program and a time budget, how can we select that fuzzing technique, or combination of techniques, which finds the most vulnerabilities within the time budget? How do program size and complexity affect the scalability and performance of each technique? How much more efficient is an attacker that has an order of magnitude more computational resources? With an understanding of the limitations of existing approaches, we can develop more advanced techniques.

Evaluation and Benchmarks

In order to validate a claim of superiority for novel fuzzing tools and techniques, we need sound methods for evaluation. Generally speaking, the better fuzzer finds a larger number of important bugs in software that we care about within a reasonable time. But what is a “reasonable time”, “software that we care about”, or “important bugs”? If no important bugs are found, how do we measure effectiveness? How do we prevent over-fitting? What is a fair baseline for comparison?

To measure progress, we need to develop reasonable standards for comparison against previous work. We encourage the community to be open about releasing tools, benchmarks, and ex-

perimental setup publicly for anyone to reproduce the results and to build upon.

Benchmarks

[C.9] *How can we evaluate specialized fuzzers?* There are programs that take structured and those that take unstructured inputs. There are stateful and stateless programs. There are programs where the source code is available and programs where only the compiled binary is available. There are programs that take inputs via a file, a GUI, or an API. Extending fuzzing to different types of software systems is a key technical challenge (C.1).

Similarly, some fuzzers are specialized for a specific purpose. For instance, there are fuzzers that seek to reach a program location [2], [14] or that focus on exposing specific types of bugs, such as performance bugs [3].

However, existing benchmarks are often *not* designed for these specialized tasks. If there is no previous work, we need standards for researchers to choose suitable subject programs and baselines for comparison.

[C.10] *How can we prevent overfitting to a specific benchmark?* For any benchmark suite, there is always the danger of overfitting. Despite a demonstration of superiority on the benchmark subjects, a fuzzer might still be inferior in general. What are reasonable strategies to mitigate overfitting? Can we propose a fair and sound policy to collect benchmarks? How can we avoid “single-source” types of benchmarks that are contributed by just one group and might give undue control to a single set of people?

Fuzzing tool competitions could be part of the solution for challenges C9 and C10. One model, inspired by constraint solving and verification competitions, is to have different competition categories, such as coverage-based fuzzing, directed fuzzing and so on. Within each category, there can be a further division based on the type of bugs and applications the fuzzer is suited for. Tool builders can submit their own benchmarks and fuzzers, which would allow independent scrutiny of the entire process. TEST-COMP (<https://test-comp.sosy-lab.org/>) is an existing competition that illustrates this model.

A second model is to come up with challenge

problems in the form of buggy programs, and have tool developers directly apply the fuzzers to find the hidden bugs. This has the advantage of tool developers configuring their tools in the best possible way for each task, but makes independent reproduction of the results more challenging. RODE0DAY (<https://rode0day.mit.edu/>) is an existing competition that illustrates this model.

Another approach is a continuous evaluation, where fuzzers are repeatedly used to fuzz real programs. For instance, as a concrete outcome of our Shonan meeting, Google has developed FUZZBENCH (<https://github.com/google/fuzzbench>) and committed computational resources to evaluate submitted fuzzers on submitted benchmarks. In addition to scientific evaluation of technical advances, this approach allows direct application of these technical advances to a large set of actual open-source software, to make critical software systems safer and more secure.

Measures of Fuzzer Performance

During the evaluation of two fuzzing techniques, which quantities should we compare? What do we measure? Today, fuzzers are typically evaluated in terms of their effectiveness and efficiency. When we are interested in security vulnerabilities, a fuzzer's *effectiveness* for a software system is determined by the total number of vulnerabilities a fuzzer has the capability of finding. In contrast, a fuzzer's *efficiency* for a software system is determined by the rate at which vulnerabilities are discovered.

[C.11] *Are synthetic bugs representative?* For evaluation, buggy software systems can be generated efficiently simply by injecting artificial faults into an existing system [7]. We need to study empirically whether such synthetic bugs are indeed representative of real and important security vulnerabilities. If they are not representative, how are they different from actual vulnerabilities? What can we do to make synthetic bugs more like real bugs? Which types of vulnerabilities are not represented in synthetic bug benchmarks?

[C.12] *Are real bugs, which have previously been discovered with other fuzzers, representative?* Another approach is to collect actual vulnerabilities that have been found through other means into a benchmark. However, this process

is tedious, such that the sample size may be relatively low which would affect the generality of the results. Secondly, the evaluation only establishes that the newly proposed fuzzer finds at least the same vulnerabilities that have been found before. It does not evaluate how well the newly proposed fuzzer finds new vulnerabilities. How representative are the discovered vulnerabilities of all (undiscovered) vulnerabilities? We could build a large, shared database of vulnerabilities in many software systems that have been found by several fuzzers or auditors over a period of time.

[C.13] *Is coverage a good measure of fuzzer effectiveness?* When no suitable bug benchmark is available, we need other means of evaluating the effectiveness of a fuzzer. Code coverage is the classic substitute measure. The intuition is that vulnerabilities cannot be exposed if the code containing the vulnerability is never executed. How effective is coverage really at measuring the capability of a fuzzer to expose vulnerabilities? We need empirical studies that assess how strongly the increase in different coverage metrics correlates with an increase in the probability to find a vulnerability. In addition to code coverage, there are many other measures of coverage such as GUI, constraint, model, grammar, or state coverage. We should conduct empirical studies to determine correlation and agreement of various proxy measures of effectiveness.

[C.14] *What is a fair choice of time budget?* It is not possible to measure fuzzer effectiveness directly. If our measure is the number of bugs found, then effectiveness is the total number of bugs the fuzzer finds *in the limit*, i.e., when given infinite time. Instead, researchers can derive a trivial lower bound on the effectiveness, i.e., the total number of bugs a fuzzer finds, by fixing a time budget. Currently, this time budget is typically anywhere between one hour and one day. However, an extremely effective fuzzer may take some time to generate test cases during which time another fuzzer can generate several orders of magnitudes more test cases [12]. If the chosen time budget is too small, the faster, yet less effective fuzzer might *appear* more effective. Thus, we should develop standards that facilitate a fair choice of time budget when evaluating the effectiveness of a fuzzer.

Techniques versus Implementations

[C.15] *How do we evaluate techniques instead of implementations?* In order to demonstrate claims of superiority of a proposed technique, researchers compare an implementation of the proposed technique to that of an existing technique. In the implementation, the researcher can make engineering decisions that can substantially affect the effectiveness of the fuzzer [17]. For instance, a comparison between the AFL greybox fuzzer against the KLEE whitebox fuzzer to determine whether a whitebox fuzzing technique outperforms a greybox fuzzing technique should always be taken with a grain of salt. If possible, the proposed technique (e.g., an improvement to greybox fuzzing) is implemented directly into the baseline (e.g., AFL).

Survey

To request feedback from the larger community on the identified challenges, we surveyed further experts from industry and academia. Our objective was to identify points of contention, to add challenges or reflections that we might have overlooked, and to solicit concrete pathways or initiatives for some of the identified challenges. We sent an email invitation to software security experts who have previously published in fuzzing or have professional work on automatic vulnerability discovery. Out of 24 respondents, 14 work in academia and 10 work in industry; 3 attended the Shonan meeting.

The survey participants marked improving automation (71%), building a theory of fuzzing (63%), and finding valid measures of fuzzer performance (63%) as their Top-3 most important challenges. While practitioners and researchers were mostly in agreement, practitioners demonstrated a particularly greater interest in the development of human-in-the-loop approaches (+0.8 Likert points). On average, a respondent marked all identified challenges as important or very important on a 5-point Likert-scale. No major additional challenges were identified. Other survey results were directly added to the corresponding sections.

Awareness and Education

Fuzzing is used today in corporations in a significant manner, often on a daily basis, for de-

tecting bugs and security flaws. Despite advances in static analysis and formal verification, fuzzing remains the primary automatic mechanism for vulnerability discovery in most software products. However, the security of our software systems is in the hands of each and every software engineer, including future volunteers that contribute to critical open-source software. We believe awareness and education, in the small and in the large, are of paramount importance.

One mechanism is the organization of security-oriented hackathons and Capture-the-Flag (CTF) competitions. For instance, the *Build it Break it Fix it* contest from Maryland (<https://builditbreakit.org/>) represents an early successful attempt in this direction. The community could also move towards competitions between fuzzing tools (such as FuzzBench, Test-Comp, and Rode0Day) or organize regular fuzzing camps.

Another mechanism is to teach about fuzzing in software engineering and cyber-security courses. The second and third authors were actively involved in designing and delivering such courses at the university level. A key challenge in developing such educational content is that the students need to be exposed to several tools, which takes a significant amount of the students' time. The recent development of online books [20] can alleviate some of these issues by presenting an integrated resource and repository for getting familiarised with various variants of fuzzing.

Acknowledgments

We thank the participants at the Shonan Meeting on Fuzzing and Symbolic Execution, and the survey respondents. This work was partially funded by the Australian Research Council (ARC) through a Discovery Early Career Researcher Award (DE190100046). This project has received funding from European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819141) and from the UK EPSRC through grant EP/R011605/1. This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems and funded by National Research Foundation (NRF) Singapore under National Cybersecurity R&D

(NCR) programme.

About the Authors

Marcel Böhme is an ARC DECRA Fellow and a Senior Lecturer at Monash University in Australia. Marcel leads his research group with a reproducibility policy (<https://mboehme.github.io/manifesto>), which means that experiment data and tools are usually published with the peer-reviewed article to facilitate open science. For instance, his most recent fuzzer, Entropic, was integrated into Google's LibFuzzer and now runs on more than 25,000 machines every day to discover security vulnerabilities in more than 300 open source projects. He received his PhD from the National University of Singapore, is a member of the ACM, and can be reached at marcel.boehme@acm.org.

Cristian Cadar is a Professor in the Department of Computing at Imperial College London, where he leads the Software Reliability Group. His research interests span the areas of software engineering, computer systems and software security, with a focus on building practical techniques for improving the reliability and security of software systems. Cadar has a PhD in Computer Science from Stanford University, and undergraduate and Master's degrees from the Massachusetts Institute of Technology. He is a Member of the ACM and IEEE and can be reached at c.cadar@imperial.ac.uk.

Abhik Roychoudhury is Provost's Chair Professor of Computer Science at the National University of Singapore. His research interests are in program analysis, software security and trustworthy systems. He is the Director of the National Satellite of Excellence in Trustworthy Software Systems at Singapore. He helped set up the Singapore Cybersecurity Consortium, a grouping of 30 Singapore-based companies engaging in research and translation in cybersecurity. He received his PhD in Computer Science from Stony Brook University. He is a senior member of IEEE, and can be reached at abhik@comp.nus.edu.sg.

■ REFERENCES

1. Marcel Böhme, Van-Thuan Pham, Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. ACM Conference on Computer and Communications Security (CCS'16).
2. Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, Abhik Roychoudhury. 2016. Directed Greybox Fuzzing. ACM Conference on Computer and Communications Security (CCS'17).
3. Jacob Burnim, Sudeep Juvekar, Koushik Sen. WISE: Automated test generation for worst-case complexity. International Conference of Software Engineering (ICSE 2009).
4. Sang Kil Cha, Thanassis Avgerinos, Alexandre Robert, David Brumley. 2012. Unleashing Mayhem on Binary Code. IEEE Symposium on Security and Privacy (S&P'12).
5. Cristian Cadar, Daniel Dunbar, Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. USENIX Conference on Operating Systems Design and Implementation (OSDI'08).
6. Cristian Cadar, Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM vol. 56, no. 2 (Feb. 2013).
7. Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, Ryan Whelan. LAVA: Large-Scale Automated Vulnerability Addition. IEEE Symposium on Security and Privacy (IEEE S&P 2016).
8. Antonio Filieri, Corina S. Pasareanu, Willem Visser. Reliability Analysis in Symbolic Pathfinder. International Conference of Software Engineering (ICSE 2013).
9. Patrice Godefroid. Fuzzing: Hack, Art and Science. Communications of the ACM, vol. 63, no. 2 (Jan. 2020).
10. Patrice Godefroid, Michael Y. Levin, David Molnar. Automated White-box Fuzz Testing. Network and Distributed System Security Symposium (NDSS 2008).
11. Christian Holler, Kim Herzig, Andreas Zeller. Fuzzing with Code Fragments. USENIX Security Symposium (USENIX Security 2012).
12. George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, Michael Hicks. Evaluating Fuzz Testing. ACM Conference on Computer and Communications Security (CCS 2018).
13. Claire Le Goues, Michael Pradel, Abhik Roychoudhury. Automated Program Repair. Communications of the ACM vol. 62, no. 12 (Dec. 2019).
14. Paul Dan Marinescu, Cristian Cadar. KATCH: High-Coverage Testing of Software Patches. ACM Symposium on the Foundations of Software Engineering (ESEC-FSE 2013).
15. Shirin Nilizadeh, Yannic Noller, Corina S. Pasareanu. DiffFuzz: differential fuzzing for side-channel analysis.

- International Conference of Software Engineering (ICSE 2019).
16. Ajitha Rajan, Subodh Sharma, Peter Schrammel, Daniel Kroening. Accelerated test execution using GPUs. Automated Software Engineering Conference (ASE 2014).
 17. Eric F. Rizzi, Sebastian Elbaum, Matthew B. Dwyer. On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE. International Conference on Software Engineering (ICSE 2016).
 18. Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, Jianguang Sun. Industry practice of coverage-guided enterprise Linux kernel fuzzing. ACM Symposium on the Foundations of Software Engineering (ESEC-FSE 2019).
 19. Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. Network and Distributed System Security Symposium (NDSS 2016).
 20. Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, Christian Holler. 2019. The Fuzzing Book. <https://www.fuzzingbook.org/>