ISSN 2186-7437

NII Shonan Meeting Report

No.160

Fuzzing and Symbolic Execution

Marcel Boehme
Cristian Cadar
Abhik Roychoudhury

September 24–27, 2019

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Fuzzing and Symbolic Execution: Reflections, Challenges, and Opportunities

Organizers:
Marcel Boehme (Monash University)
Cristian Cadar (Imperial College London)
Abhik Roychoudhury (National University of Singapore)

September 24–27, 2019

1 List of Participants

- Antonia Bertolino, ISTI-CNR
- Marcel Boehme, Monash University
- Tevfik Bultan, University of California at Santa Barbara
- Cristian Cadar, Imperial College London
- Maria Christakis, Max Planck Institute for Software Systems
- Brendan Dolan-Gavitt, New York University
- Xiang Gao, National University of Singapore
- Gregory Gay, Chalmers University of Technology and University of Gothenburg
- Milos Gligoric, University of Texas at Austin
- Patrice Godefroid, Microsoft Research
- Peter Goodman, Trail of Bits
- Alessandra Gorla, IMDEA Software Institute
- Yue Jia, Facebook and University College London
- Sarfraz Khurshid, University of Texas at Austin
- Caroline Lemieux, University of California Berkeley
- Paul Marinescu, Facebook
- Darko Marinov, University of Illinois at Urbana Champaign
- Sergey Mechtaev, University College London

- Filip Niksic, University of Pennsylvania
- Martin Nowack, Imperial College London
- Mauro Pezze, Universita della Svizzera Italiana
- Van-Thuan Pham, Monash University
- Michael Pradel, University of Stuttgart
- Abhik Roychoudhury, National University of Singapore
- Kostya Serebrany, Google
- David Trabish, Tel Aviv University
- Willem Visser, Stellenbosch University
- Ned Williamson, Google
- Norihiro Yoshida, Nagoya University
- Andreas Zeller, CISPA Helmholz Center for Information Security and Saarland University
- Xiangyu Zhang, Purdue University

2 Meeting Schedule

Check-in Day: September 23 (Mon)

- Welcome banquet

Day 1, September 24 (Tue)

- 7:00 - 9:00 Breakfast
- 9:00 - 9:25 Introduction by Shonan staff and seminar organizers
- 9:25 - 10:30 Self-introduction by participants
- 10:30 - 11:00 Coffee break
- 11:00 - 12:00 Keynote: Kostya Serebryany on Fuzzing in Industry
- 12:00 - 13:30 Lunch
- 13:30 - 14:00 Group photo
- 14:00 - 15:00 Industry Discussion Panel (Moderators: Andreas Zeller and Paul Marinescu)
- 15:00 - 15:30 Demo: Semi-automated Vulnerability Discovery (Ned Williamson)
- 15:30 - 16:00 Afternoon Tea

- 16:00 - 18:00
  - Talk 1: Caroline Lemieux
  - Talk 2: Xiangyu Zhang
  - Talk 3: Maria Christakis
  - Talk 4: Van-Thuan Pham
  - Talk 5: Gregory Gay
  - Talk 6: Peter Goodman
- 18:00-19:30 Dinner

Day 2, September 25 (Wed)

- 7:00-9:00 Breakfast
- 9:00-10:00 Keynote: Patrice Godefroid on Symex in Industry
- 10:00-10:30 Coffee break
- 10:30-12:10
  - Talk 7: Mauro Pezze
  - Talk 8: Martin Nowack
  - Talk 9: Willem Visser
  - Talk 10: Sarfraz Khurshid
  - Talk 11: Darko Marinov
- 12:10-14:00 Lunch
- 13:30-15:00 Panel (Reflections & Challenges)
- 15:00-15:15 Afternoon tea
- 15:15-16:15
  - Talk 12: Tevfik Bultan
  - Talk 13: Sergey Mechtaev
  - Talk 14: Gao Xiang
- 16:15-16:30 Break
- 16:30-17:30
  - Talk 15: Andreas Zeller
  - Talk 16: Milos Gligoric
  - Talk 17: Paul Marinescu
- 18:00-18:10 Discussions of Thursday's Program
- 18:00-19:30 Dinner

Day4: September 26 (Thu)

- 7:00-9:00 Breakfast
- 9:00-10:20
  - Talk 18: Antonia Bertolino
  - Talk 19: Yue Jia
  - Talk 20: Brendan Dolan-Gavitt
  - Talk 21: Filip Niksic
- 10:20-10:50 Coffee Break
- 10:50-12:10
  - Talk 22: Michael Pradel
  - Talk 23: Norihiro Yoshida
  - Talk 24: Alessandra Gorla
  - Talk 25: Marcel Boehme
- 12:10-13:30 Lunch
- 13:30-21:00 Excursion & banquet

Day4: September 27 (Fri)

- 7:00-9:00 Breakfast & Check out
- 9:00-9:40 Talks (Session Chair: Gregory Gay)
  - Talk 26: Abhik Roychoudhury
  - Talk 27: Cristian Cadar
- 9:40-10:20 Discussion: How to evaluate fuzzing/symex tools?
- 10:20-10:40 Coffee Break
- 10:40-11:00 Summary
- 11:00-12:00 Discussion
- 12:00-13:30 Lunch

3 Motivation for our Meeting

The problem of finding software errors by automated input generation—i.e., automated software testing—is currently investigated within several mostly isolated academic research communities:

- The symbolic execution (SE) research community casts testing as a constraint satisfaction problem and studies formal approaches to automated software testing.
- The search-based software testing (SBST) research community casts testing as an optimization problem and studies metaheuristics, such as genetic algorithms, to tackle automated software testing.

- The fuzzing research community is interested particularly in exposing security flaws in critical software systems, and leverages both techniques, with substantial recent success in an area called coverage-based greybox fuzzing.

This problem has also received significant attention from the industry. The industry is interested in finding errors at the very large scale, with several notable examples discussed next. Google is running the OSS-Fuzz project which has found thousands of errors in security-critical open source projects over the last three years. Microsoft is running Project Springfield, a cloud-based testing-as-a-service framework. Facebook has successfully deployed an automated Android testing tool, Sapienz, which is tightly integrated in their Continuous Deployment process.

Despite the shared high-level objective, there has been little interaction across these communities. As a concrete example, the fuzzing community leverages a compact terminology that is very different from that of the SE and SBST communities. The fuzzing community publishes in security venues (CCS, NDSS, S&P, TIFS) while the SE and SBST communities publish in software engineering venues (ICSE, FSE, ISSTA, ASE, TSE, TOSEM). Many ideas that have long been discussed in the SE and SBST communities are now being re-discovered in the fuzzing community.

The research communities have much to learn from industry, too. Driven by the need for scale and enabled by the availability of abundant computational resources, members of the industry have developed some of the most advanced and highly efficient automated testing tools. Much can be learned by studying these tools from a research perspective. On the one hand, industry has found practical solutions to open research problems. On the other hand, the industry faces interesting practical challenges that are yet unbeknownst to the research community. It is our position that the proposed meeting can help establish a conversation.

3.1 Objective

The purpose of this meeting was to bring together the thought leaders, distinguished researchers, tool builders, founders, and promising young researchers from each of these communities. We discussed recent advances and open challenges in automated software testing (with specific focus on security vulnerability detection) through the lens of both research and industry. We reviewed the state of the art, established common ground, and discussed a roadmap of important problems to address in the near- and long-term. We plan to document the results of this meeting for the larger research community and publish in a follow-up publication. We hope that this meeting sparks new collaborations across research communities, between research and industry, and among individuals with vastly different backgrounds and expertise. We also hope the meeting invigorates and accelerates research on software vulnerability detection by building and extending the theoretical foundation as well as by providing practical and immediate benefit to the industry.

3.2 Why Now?

We feel this was the right timing for such a meeting because the various individual technologies have gained maturity and hence a conversation across the subcommunities can be productive. The symbolic execution community has gained prominence with the development of mature tools like KLEE. The fuzz testing community has seen an explosion of works in the recent years, starting with the work of AFLFast which studies the science behind greybox fuzzing. Meanwhile the SBST community has also developed mature technologies like Evosuite. At the same time, all of these techniques could potentially benefit from learning approaches which glean information about the input domain or the application processing it. In this meeting, we discussed the common themes behind these techniques and identified the research opportunities in terms of cross-fertilization.

4 Summary of Discussions from Organizers

4.1 Overview

Fuzz testing is an automated vulnerability discovery method for programs, which is used routinely in corporations and organizations on a daily basis. Usually fuzz testing involves generating inputs via a biased random search with the goal of exposing crashes, as an attempt to remain one step ahead of the attacker. If a whitebox view of the program is available, the input generation search can be systematized via symbolic execution so that every input covers a different program path. Fuzzing and symbolic execution have reached a wide degree of maturity with significant well-used tools being developed in companies and academia. In this meeting we took a forward-looking view of the software vulnerability discovery technologies and studied the open problems and challenges in the field. We also summarized opportunities that exist to advance the field as discussed with researchers and practitioners in a recent Shonan Meeting. As a follow-up of the discussions from the meeting, Google has released the Fuzzbench service for the comparison and benchmarking of fuzzers `https://github.com/google/fuzzbench`

4.2 Keynotes

1. Kostya Serebryany, Google on **Fuzzing at Google Today & Tomorrow**

2. Patrice Godefroid, **Fuzzing at Microsoft**
   The Microsoft Security Development Lifecycle requires fuzzing (aka automatic security testing) at every untrusted interface of every product. To satisfy this requirement, a lot of expertise and tools have been developed inside Microsoft over the last 15 years. This short presentation will discuss fuzzing at Microsoft, the whitebox fuzzing technology pioneered at Microsoft Research (based on dynamic symbolic execution and constraint solving), and "Microsoft Security Risk Detection" (formerly named Project Springfield), the first commercial cloud fuzzing service. It will also discuss open challenges and future research directions.

4.3 Demo Session

- Ned Williamson of Google demonstrated his semi-automated approach to vulnerability discovery which involves cycles of two tasks: fuzzing and scaffolding. Using structure-aware API-based fuzzing to find bugs in Chrome and iOS kernel.

4.4 Talks

1. Caroline Lemieux
   Semantic Fuzzing with Input Generators
   Conventional fuzz testing excels at finding syntactic bugs for programs taking in binary input data. But how can we fuzz deeper, and find bugs deep in the semantic and logic processing stages of such programs? In this talk I will discuss Zest, a semantic fuzzing technique that combines input generators with coverage-guided fuzzing to reliably find semantic bugs in programs.

2. Peter Goodman
   DeepState
   This talk will be about how to bring fuzzing and symbolic execution to the fingertips of developers via unit testing. DeepState is a Google Test-like unit testing framework, which enables "parameterized" tests, where a programming can ask for a symbolic integer. DeepState makes it easy to connect to fuzzers like AFL, libFuzzer, Angora, and Eclipser, as well as symbolic executors like Manticore and Angr. Some cool things that have come out of this work is new programming idioms for trying to address scalability challenges of symbolic execution, as well as coming up with idioms for structuring tests in such a way that test case reduction becomes easier.

3. Alessandra Gorla
   Can NLP help fuzzing and symbolic execution?
   Recently I have been working on a technique based on NLP to analyze developers' comments in source code. We use it to generate valid inputs and generate test assertions for automatically generated inputs. Is there any possible use of NLP to improve fuzzing and symbolic execution?

4. Gregory Gay
   A Brief Introduction to (Metaheuristic) Search-Based Test Generation
   Powerful optimization algorithms can be used to automate test generation, augmenting the ability of human developers to build, test, and deploy software. This talk briefly introduces the topic of metaheuristic search, and explores the relation of metaheuristic search to the broader world of fuzzing.

5. Michael Pradel
   Test Generation for Higher-Order Functions in Dynamic Languages
   Test generation has proven to provide an effective way of identifying programming errors. Unfortunately, current test generation techniques

are challenged by higher-order functions in dynamic languages, such as JavaScript functions that receive callbacks. In particular, existing test generators suffer from the unavailability of statically known type signatures, do not provide functions or provide only trivial functions as inputs, and ignore callbacks triggered by the code under test. This paper presents LambdaTester, a novel test generator that addresses the specific problems posed by higher-order functions in dynamic languages. The approach automatically infers at what argument position a method under test expects a callback, generates and iteratively improves callback functions given as input to this method, and uses novel test oracles that check whether and how callback functions are invoked. We apply LambdaTester to test 43 higher-order functions taken from 13 popular JavaScript libraries. The approach detects unexpected behavior in 12 of the 13 libraries, many of which are missed by a state-of-the-art test generator.

6. Xiang Gao
   Integrating automated program repair with detection.
   Existing program repair systems modify a buggy program so that the modified program passes given tests. The repaired program may not satisfy even the most basic notion of correctness, namely crash-freedom. In other words, repair tools might generate patches which over-fit the test data driving the repair, and the automatically repaired programs may even introduce crashes or vulnerabilities. We propose an integrated approach for detecting and discarding crashing patches. Our approach fuses test and patch generation into a single process, in which patches are generated with the objective of passing existing tests, and new tests are generated with the objective of filtering out over-fitted patches by distinguishing candidate patches in terms of behavior. We use crash-freedom as the oracle to discard patch candidates which crash on the new tests. In its core, our approach defines a grey-box fuzzing strategy that gives higher priority to new tests that separate patches behaving equivalently on existing tests. This test generation strategy identifies semantic differences between patch candidates, and reduces over-fitting in program repair.

7. Sergey Mechtaev
   Symbolic Execution with Existential Second-Order Constraints
   Symbolic execution often has to reason about pieces of code whose precise semantics is unknown. First, when executing a system that interacts with environment, symbolic execution might not have access to the code of external functions. Second, symbolic execution has been applied to automatically fix software bugs, and when analysing a buggy program, it has to abstrast away a fragment of code containing the bug. The insight if this talk is that the unknown code in both these cases can be efficiently abstracted away using symbolic variables over functions. To facilitate this, we propose an extension of symbolic execution to second-order logic that enables reasoning about symbolic functions whose domain of interpretations is restricted by a user-defined language. This extension helps to address the path explosion problem in the context of program repair, and enables automatic inferring of environment models from the usage context. This talks introduces symbolic execution with existential second-order constrains, discusses its possible applications, and formulates

technical challenges of integrating second-order reasoning into symbolic execution systems.

8. Antonia Bertolino
   Automated coverage testing for a scope
   Coverage measures provide an effective and practical adequacy criterion. However, aiming at *full* code coverage may not be a convenient choice in all cases. Depending on the usage context, some code entities -even if executable- might be irrelevant, thus in previous work we proposed a concept of scope-based coverage that can be applied to any coverage criterion and customized to different notions of a scope. This concept could usefully drive a more cost-effective technique of scope-guided dynamic execution which we propose as a possible future research challenge.

9. Maria Christakis
   Targeted Greybox Fuzzing with Static Lookahead Analysis
   Automatic test generation typically aims to generate inputs that explore new paths in the program under test in order to find bugs. Existing work has, therefore, focused on guiding the exploration toward program parts that are more likely to contain bugs by using an offline static analysis. In this talk, we introduce a novel technique for targeted greybox fuzzing using an online static analysis that guides the fuzzer toward a set of target locations, for instance, located in recently modified parts of the program. This is achieved by first semantically analyzing each program path that is explored by an input in the fuzzer's test suite. The results of this analysis are then used to control the fuzzer's specialized power schedule, which determines how often to fuzz inputs from the test suite. We implemented our technique by extending a state-of-the-art, industrial fuzzer for Ethereum smart contracts and evaluate its effectiveness on 27 real-world benchmarks. Using an online analysis is particularly suitable for the domain of smart contracts since it does not require any code instrumentation—adding instrumentation to contracts changes their semantics. Our experiments show that targeted fuzzing significantly outperforms standard greybox fuzzing for reaching 83% of the challenging target locations (up to 14x of median speed-up).

10. Filip Niksic
    Combinatorial Constructions for Effective Testing
    Many bugs in distributed systems can be characterized in terms of small combinatorial structures, and such characterizations can lead to combinatorial constructions of small sets of tests that can expose these bugs. In the talk I will present our work on detecting bugs that occur due to two or three key nodes in the system being separated in a network partition, and bugs that arise due to wrong relative ordering of a small number of events. The first kind of bugs can be detected by simulating a small number of network partitions, and the second kind can be detected by executing a small number of schedules, where by small we mean logarithmic, polylogarithmic, or polynomial in the size of the system.

11. Andreas Zeller
    Fast Fuzzing

12. Darko Marinov
    Fuzzing and Symbolic Execution for Regression Testing
    Fuzzing and symbolic execution can be great approaches for generating new tests that find bugs in the entire codebase. However, a developer who makes a change to the code may not be interested in bugs in the entire codebase but only in the bugs in the changed code (or code impacted by the changes). How can fuzzing and symbolic execution focus on the changed code? How can they synergistically work with the existing, manually written tests?

13. Norihiro Yoshida
    Symbolic Execution-based Approach to Extracting a Micro State Transition Table
    During embedded system development, developers frequently change and reuse the existing C source code for the development of a new but behaviorally similar product. Such frequent changes generally decrease the understandability of C source code although the developers have to understand how it behaves and how to reuse it. So far, much research has been done on symbolic execution techniques that statically analyze the behavioral aspect of given source code. In this paper, we propose a symbolic execution-based approach to extracting a Micro State Transition Table (MSTT) that helps developers understanding the behavioral aspect of embedded C source code based on a fine-grained state transition model.

14. Milos Gligoric
    Application of Fuzz Testing in Verification Projects
    Fuzz testing emerged as one of the most scalable techniques for generating test cases, which is able to increase code coverage and discover unexpected behavior. Although substantial work has been done in recent years, most of the focus has been on imperative languages (e.g., C and Java). In this talk, I will present our effort to apply fuzzing to a new domain with the goal to detect incomplete specification in verification projects.

15. Paul Marinescu
    Fuzzing the Long Tail
    Fuzzing tools are widely used by the security community to find bugs in critical pieces of software. This talk explores fuzzing from a different perspective, and looks at the challenges of building a fuzzing product, which scales to thousands of engineers and the codebase they produce and continuously modify.

16. Tevfik Bultan
    Side Channel Analysis Using a Model Counting Constraint Solver and Symbolic Execution
    A crucial problem in computer security is detecting information leakage via side channels. Information gained by observing non-functional properties of program executions (such as execution time or memory usage) can enable attackers to infer secret information accessed by the program. In this talk, I will discuss how symbolic execution, combined with a model counting constraint solver, can be used for detecting and quantifying side-channel leakage in programs, and also for identifying input sequences that

can be used to recover secrets. We implemented these results as an extension to the symbolic execution tool SPF using our model counting constraint solver ABC.

17. Mauro Pezzé
Efficient symbolic execution of programs with complex structured inputs to reveal 'deep bugs'
This talk will outline an approach to symbolically execute programs with complex structured inputs to generate concrete test cases for revealing deep bugs in heap data structures. We presented the work at ISSTA in 2017 and demonstrated a prototype tool at ICSE in 2018.

18. Xiangyu Zhang
ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery
Existing mutation based fuzzers tend to randomly mutate the input of a program without understanding its underlying syntax and semantics. In this talk, I will present a novel on-the-fly probing technique (called ProFuzzer) that automatically recovers and understands input fields of critical importance to vulnerability discovery during a fuzzing process and intelligently adapts the mutation strategy to enhance the chance of hitting zero-day targets. Since such probing is transparently piggybacked to the regular fuzzing, no prior knowledge of the input specification is needed. During fuzzing, individual bytes are first mutated and their fuzzing results are automatically analyzed to link those related together and identify the type for the field connecting them; these bytes are further mutated together following type-specific strategies, which substantially prunes the search space. We define the probe types generally across all applications, thereby making our technique application agnostic. Our experiments on standard benchmarks and real-world applications show that ProFuzzer outperforms AFL and its optimized version AFLFast, as well as other state-of-art fuzzers including VUzzer, Driller and QSYM. Within two months, it exposed 42 zero-days in 10 intensively tested programs, generating 30 CVEs.

19. Sarfraz Khurshid
Solution enumeration abstraction.
We introduce a new abstraction to direct back-end constraint solvers (such as those often employed for symbolic analysis) to enumerate solutions that are more useful in the context of the application domain where the solvers are employed.

20. Willem Visser, Stellenbosch
COASTAL: a Tool for Seamless Integration of Symbolic Execution and Fuzzing for Java
The past few years a number of research groups built tools where they combined fuzzing and symbolic execution, and in this talk we will discuss yet another case. The combination of these two technologies for bug finding is a no-brainer: fuzzing covers lots of cases with very little effort, but can get stuck generating inputs to highly constrained behaviours, for which symbolic execution is good. What makes our approach (COASTAL)

somewhat unique is that it uses concolic execution rather than classic symbolic execution and that the fuzzer and the concolic execution were built into the same framework, from scratch (in other words it is not two existing tools that are being combined). In this talk we will discuss the design decisions, the integrated architecture and show some examples.

21. Van-Thuan Pham
Coverage-guided Fuzzing for Automotive: Challenges and Opportunities
Fuzzing has been used to discover security bugs in cars, specfically in Electronic Control Units (ECUs) on the Control Area Network (CAN) bus. However, the current approach treats ECUs as black-boxes leading to limited results. This talk shares the challenges and opportunities to implement a coverage guided fuzzing (CGF) approach, which has been widely used in other domains (e.g., parser fuzzing), to improve the effectiveness of automotive fuzzing.

22. Martin Nowack
Fine-grain memory representation for Dynamic Symbolic Execution

One of the biggest challenges of dynamic symbolic execution (DSE) is the state space explosion problem: DSE tries to evaluate all possible execution paths of an application. For every path, it needs to represent the allocated memory and its accesses. Even though different approaches have been proposed to mitigate the state space explosion problem, DSE still needs to represent a multitude of states in parallel to analyse them. If too many states are present, they cannot fit into memory, and DSE needs to terminate them prematurely or store them on disc intermediately. With a more efficient representation of allocated memory, DSE can handle more states simultaneously, improving its performance. In this work, we introduce an enhanced, fine-grain and efficient representation of memory that mimics the allocations of tested applications. We tested Coreutils using three different search strategies with our implementation on top of the symbolic execution engine KLEE. We achieve a significant reduction of the memory consumption of states by up to 99.06% (mean DFS: 2%, BFS: 51%, Cov.: 49%), allowing to represent more states in memory more efficiently. The total execution time is reduced by up to 97.81% (mean DFS: 9%, BFS: 7%, Cov.:4%)—a speedup of 49x in comparison to baseline KLEE.

23. Brendan Dolan-Gavitt
Rode0day: Improving Software Security through Competition
Bug-finding tools such as fuzzers and symbolic execution engines are now a standard part of the software assurance toolkit, and there has been a surge of activity in academia and industry focused on improving these tools and using them to find security bugs in critical software. It remains difficult, however, to tell exactly how effective such tools are: they often find bugs, but lack of standard datasets and ground truth mean that the performance of a given bug-finder is hard to compare with any other. To solve this problem, we created Rode0day, a monthly bug-finding competition focused on evaluating and improving the state of the art in software security research, inspired by research competitions such as the ImageNet

ILSVRC and SMT-COMP. Every month, we release a set of real-world programs seeded with bugs; competitors find and submit crashing inputs, and we award points based on which bugs have been found. At the end of the month, we release the solutions and all submitted inputs, allowing competitors to diagnose and improve their tools and steadily improve performance. These results, combined with our detailed knowledge of how each bug works (since we inject them ourselves, at scale), allow empirical investigation into bug-finding. This talk will discuss the design, implementation, and results from the first year of Rode0day's competition.

24. Yue Jia
Automated Software Testing at Facebook and Some Challenges for Fuzzing GUI Applications

25. Abhik Roychoudhury
Symbolic Execution for Automated Repair

Automated program repair is an emerging technology where fix suggestions for errors and vulnerabilities are generated. In our work, we have investigated the role of symbolic execution in inferring specifications from tests, and then these specifications can be used to synthesize fix suggestions. Such an automated fixing engine can be potentially used as a back-end to a fuzzing engine, where vulnerabilities found by fuzzing are automatically fixed.

26. Cristian Cadar
Summaries of C String Loops for More Effective Symbolic Execution (and Refactoring)

Analysing and comprehending C programs that use strings is hard: using standard library functions for manipulating strings is not enforced and programs often use complex loops for the same purpose. We introduce the notion of memoryless loops that capture some of these string loops and present a counterexample-guided synthesis approach to summarise memoryless loops using C standard library functions, which has applications to testing, optimisation and refactoring.

Joint work with Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky and Noam Rinetzky.

27. Marcel Boehme
Assurances in Software Testing
As researchers, we already understand how to make testing more effective and efficient at finding bugs. However, as fuzzing (i.e., automated testing) becomes more widely adopted in practice, practitioners are asking: Which assurances does a fuzzing campaign provide that exposes no bugs? When is it safe to stop the fuzzer with a reasonable residual risk? How much longer should the fuzzer be run to achieve sufficient coverage? It is time for us to move beyond the innovation of increasingly sophisticated testing techniques, to build a body of knowledge around the explication and quantification of the testing process, and to develop sound methodologies to estimate and extrapolate these quantities with measurable accuracy.

In our vision of the future practitioners leverage a rich statistical toolset to assess residual risk, to obtain statistical guarantees, and to analyze the cost-benefit trade-off for ongoing fuzzing campaigns. We propose a general framework as a first starting point to tackle this fundamental challenge and discuss a large number of concrete opportunities for future research.

5 Topics covered in the seminar

The Internet and the world's Digital Economy runs on software no one is accountable for. Much of our critical software infrastructure is built upon (well-tested) open-source software libraries. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. However, there is no software without bugs. 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 that empower stakeholders, individual security researchers, and the open-source community to discover vulnerabilities automatically and at scale; and fuzzing is one of the most promising techniques.

Fuzzing is an automatic vulnerability discovery technique which continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques, which are conveniently named blackbox fuzzing, greybox fuzzing and whitebox fuzzing.

Blackbox fuzzing generates inputs without any knowledge of the program. However, it may have a view of the input domain from which the program inputs are drawn. There are two main variants of blackbox fuzzing: mutational and generational. In mutational blackbox fuzzing, the fuzzing campaign starts with one or more seed inputs, and a mutation ratio $0 \leq m \leq 1$. The process then selects a seed input and randomly flips $m{*}length_{seed}$ bits in it, where $length_{seed}$ is the length of the seed input. The process continues until a time budget is exhausted. In generational blackbox fuzzing, a grammar-like specification of the input format is provided. New inputs are generated that meet the grammar. Existing blackbox fuzzing tools include Spike and Peach fuzzers.

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. Given a seed input, mutations are applied on it to generate new inputs. As these new inputs are run, we can observe which of the instrumented control locations are visited, and accordingly make approximate inference of whether these inputs exposed new control flows. Such inputs are retained for further examination, via mutating them, which would in turn generate more inputs. In order to terminate the program when a vulnerability is discovered, *sanitizers* inject assertions into the program. Existing greybox fuzzing tools include American Fuzzy Lop (AFL) and LibFuzzer.

Whitebox fuzzing are based on a technique called *symbolic execution*, which uses program analysis and constraint solvers to systematically enumerate interesting program paths. Whitebox fuzzers calculates the *path condition* of an input i—the set of inputs which traverse the same path as i. The path condition is represented as a logical formula, e.g. $i[0] == 42 \wedge i[0] - i[1] > 7$. Thus, given a seed input s, we calculate its path condition, and mutate the path condition (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). Symbolic execution engines available include KLEE, SAGE and Symbolic Path Finder (SPF).

6 Summary of Identified Issues from Organizers

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. On the 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 enables a particularly efficient approach to fuzzing by generating millions of inputs per second with none or minimal program analysis overhead. In this seminar, we discussed the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a recent Shonan Meeting. We took a forward looking view of the software vulnerability discovery technologies and discussed concrete directions for future research.

7 Spreading the Message

Fuzzing is used today in corporations in a significant manner, often on a daily basis, for detecting software vulnerabilities. Despite advances in static analysis and formal verification, fuzz testing remains the primary mechanism of choice for finding security vulnerabilities in real-life. While fuzzing is routinely deployed in certain large corporations, more could be done to spread the "message" about the effectiveness of fuzzing, to enhance software security as a whole. We conclude the article by briefly mentioning certain measures that can be adopted in an actionable fashion, to enhance awareness about fuzzing.

One mechanism which is already adopted by various colleges and universities is the organization of security oriented hackathons and Capture-the-Flag (CTF) competitions. While there exist many such attempts all over the world, the Build-it Break it Fix it Contest from Maryland[1] represents an early attempt in this direction. As a next step, the community could move towards competitions between fuzzing tools (such as monthly competitions by Rode0Day, see `https://rode0day.mit.edu/`) or camps on fuzzing by enthusiasts.

We feel integration of fuzzing concepts into software engineering and cybersecurity education is of paramount importance. Two of us were actively involved in designing and delivering undergraduate courses on bug and vulnerability detection. A key challenge in developing such educational content is that the students need to be exposed to several tools and in the space of a semester significant amount of the students' time is spent on familiarizing with several tools. The recent development of online books[2] can alleviate some of these

[1] https://builditbreakit.org/

[2] https://www.fuzzingbook.org/

issues by presenting an integrated resource and repository for getting exposed to various variants of fuzzing.

Today, there is a large number of software systems connected to the internet that have been running for years without intervention. Thus even if there is greater awareness of fuzzing, systems may remain unpatched, even after vulnerabilities are detected via fuzzing. For instance, five years after the Heartbleed vulnerability in OpenSSL was patched, almost 100k systems were still vulnerable.[3] Because of this state of the practice, future practitioners in the field of software security will need to consider the possibility of automatic generation and/or application of patches.[4]

[3] `https://www.shodan.io/report/OWew7Zq7`

[4] Claire Le Goues, Michael Pradel and Abhik Roychoudhury, Automated Program Repair, Communications of the ACM, 62(12), 2019.