Fuzzing, Symbolic Execution, and Expert Guidance for Better Testing

Ismet Burak Kadron
University of California, Santa Barbara

Yannic Noller
National University of Singapore

Rohan Padhye
Carnegie Mellon University

Tevfik Bultan
University of California, Santa Barbara

Corina S. Păsăreanu
Carnegie Mellon University

Koushik Sen
University of California, Berkeley

***Abstract*—Hybrid program analysis approaches, that combine static and dynamic analysis, have resulted in powerful tools for automated software testing. However, they are still limited in practice, where the identification and removal of software errors remains a costly manual process. In this paper we argue for hybrid techniques that allow minimal but critical intervention from experts, to better guide software testing. We review several of our works that realize this vision.**

1. Introduction

As all aspects of human society increasingly rely on software systems, there is an increased need for scalable techniques and tools that can detect and eliminate software bugs effectively. In the last decade, hybrid approaches that combine static and dynamic analysis have resulted in powerful tools for automated software testing. However, due to the limited scalability of automated techniques, in practice, identification and removal of software errors remains a predominantly manual process that requires tremendous amounts of effort by developers. In this article we argue for software analysis techniques that allow minimal but critical intervention from expert users. The insight is that humans and computers have complementary strengths: automated techniques excel at repetitive search at a scale that is hard for humans, whereas humans excel at contextual and semantic reasoning, which are hard for computers. By combining these strengths in a principled way, software testing tools can achieve error detection at scale. While the idea of combining automated testing and manual input is not new, we find that researchers still strive for push-button technologies, while the human guidance is seldom acknowledged and often implemented in ad-hoc ways. We put forward a

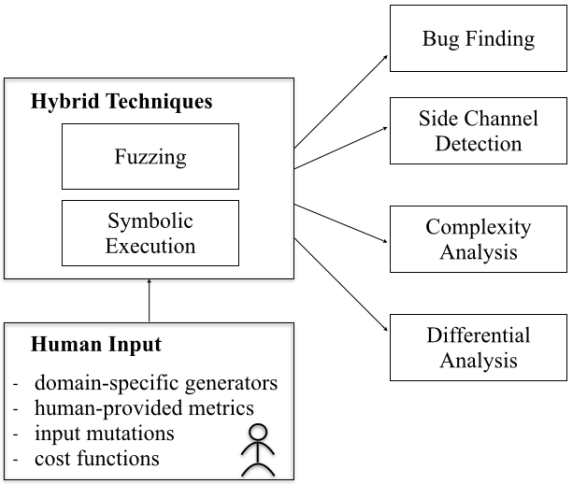

Figure 1. Software Testing with Expert Human Input

vision, illustrated in Figure 1, where the central effort of software testing relies on automated tools, but where human intervention is envisioned to be minimal, while still being indispensable. In this article, we highlight some of our recent relevant work, that leverages small expert input to address challenging analysis problems that can not be solved fully automatically. Specifically we describe fuzzing with human intervention in the form of domain-specific generators and feedback via human-provided metrics; information leakage analysis based on fuzzing, where the inputs are modified based on human-defined mutations; a combination of symbolic execution with fuzzing for estimating computational complexity of programs, where the human intervention is provided through user-defined cost functions to be maximized by the program analysis, and various differential analysis types where the user defined cost functions are leveraged across multiple program executions.

2. Domain-Specific Fuzzing with Generators and Waypoints

Fuzz testing or fuzzing is an automated software testing method that injects invalid, malformed, or unexpected inputs into a system to reveal software defects. Conventional fuzzing tools such as Google's AFL are designed to work in a *push-button* fashion. At the most, they offer knobs to control simple fuzzing heuristics such as timeouts and choice of mutation operators. These control points do not scale to more complex use cases when more fine-grained control is desired in order to customize testing objectives. We argue that the users of fuzzing tools are often domain experts who know something about the code they are testing, the input formats that a program expects, and/or what constitutes a good or bad input. Based on this insight, we have been investigating two extension points for enabling domain experts to incorporate their knowledge.

2.1. Domain-Specific Input Generation

We enable users to control the type of inputs generated by fuzzers using arbitrary generator functions. This requirement is common when testing programs that process complex input formats, such as compilers that operate on highly structured input program—conventional binary fuzzing tools struggle to generate diverse valid inputs of this form. We piggy-back on the well known abstraction of property-based testing, popularized by tools such as Quickcheck [3]. A property test validates an assertion of the form $\forall x \in T : p(x) \Rightarrow q(x)$, where T is some data type, e.g., abstract syntax-tree (AST) of a programming language when fuzzing a compiler, x is an instance of that type (e.g., a program), $p(x)$ is a precondition (e.g., the program is well-typed) and $q(x)$ is a postcondition (e.g., a compiler produces valid assembly for this program). In tools like Quickcheck, this test is validated by randomly sampling inputs $x \in T$ using a domain-specific sampling function, say `Generator<T>`. For example, an AST generator constructs random syntax trees using a recursive procedure and making pseudo-random choices for the types of nodes to construct and values to populate. Our key insight is that we can simply perform mutations on the pseudo-random choices made by the input generators; by recording these choice sequences and replaying input generators with slightly mutated sequences, we can effectively produce new structured inputs (e.g., ASTs) that differ slightly from the original [15]. This trick allows us to use feedback-directed fuzzing algorithms that are originally designed for mutating inputs represented as bitstreams on user-specified input formats. Our JQF [14] framework implements this idea by building on top of a Java port of Quickcheck. This framework further allows domain experts to choose from a number

of search strategies such as a coverage-guided evolutionary search, reinforcement learning, or a hybrid approach of integrating a concolic execution engine.

The JQF framework has enabled several domain-specific implementations including targeting Apache Spark applications, analyzing model-driven engineering tools, and for autograding classroom programming assignments.

2.2. Domain-Specific Input Feedback

Coverage-guided fuzzing is used for many different applications such as differential testing, discovering vulnerabilities, crashes, directed testing and so on. Researchers have also modified fuzzing to leverage domain-specific information to improve their target application. Researchers implement many of these modifications in an ad-hoc manner and the implementation of these modifications is non-trivial.

To unify these approaches with a single framework and help users develop domain-specific fuzzing applications, we proposed FuzzFactory [16], a framework for developing domain-specific fuzzing applications without requiring changes to mutation and search heuristics. To help with targeting different domains, users can define domain-specific feedback mechanisms to specify the metric for optimization. These feedback mechanisms map program locations to metrics based on the executed input. For example, if we want to generate inputs that maximize memory consumption, we can define the feedback mechanism as mapping program locations where `malloc` is called to a set of natural numbers, representing the amount of memory allocated in bytes at that program location (computed at runtime). FuzzFactory uses this information to selectively save intermediate inputs, called *waypoints*, to augment coverage-guided fuzzing. These intermediate inputs are selected if they improve on the domain-specific feedback metric. Saving these intermediate inputs enables the fuzzer to make progress for challenging input constraints and to maximize the domain-specific objectives. Finally, FuzzFactory allows the user to compose multiple domains. This is done in a manner such that inputs that improve any one of the domain-specific objectives are saved to help fuzzing and generate new inputs. This in turn enables the fuzzer to be flexible and to target multiple goals simultaneously.

We used FuzzFactory to implement six domain-specific fuzzing applications, three re-implementations of prior work and three novel mechanisms: maximizing dynamic memory allocations for finding resource consumption issues, surpassing checksums via minimizing hamming distance between integer comparisons, and targeting coverage beyond a fixed code location for validating incremental changes. We evaluated the effectiveness of these domain-specific fuzzing applications on benchmarks from Google's `fuzzer-test-suite`. In these benchmarks, FuzzFactory can be used to fuzz different types of applications without modifying the underlying search algorithm. Multiple domains can be composed to perform better than the sum of their parts, such as combining domain-specific feedback about strict equality comparisons and dynamic memory allocations and to enable the automatic generation of LZ4 bombs and PNG bombs [9]: these are tiny inputs that lead to dynamic allocations of 4GB in `libarchive` and 2GB in `libpng` respectively (more than the theoretically maximum decompression ratio, due to an implementation error).

The FuzzFactory has also been used by others for directed fuzzing applications, as in KC-Fuzz [18].

3. Human-Guided Input Generation for Side-Channel Detection

We discuss here an application of fuzzing to the challenging problem of finding side channels in complex software applications. Information leaks in cloud-based software services are an urgent threat, and *side-channel* leaks, where private information can be extracted by analyzing observable side effects of computation, are becoming increasingly important in applications that communicate over the internet. Well-known side-channel attacks include those based on physical side effects such as power, timing, cache usage or electromagnetic radiation, and CPU-level branch prediction and race conditions, such as the Spectre [7] and Meltdown [8] attacks. In applications that communicate over the network, the side effects of communication can be observed via network packet sizes and timings even

when the communication is encrypted.

To perform side-channel analysis for networked applications, we developed AutoFeed [6], [5]. The tool performs profiling of the system with valid inputs in order to extract observables (such as network packet timings and payload sizes) and to quantify the correlation between the observables and secret program information. Such profiling is typically done with manually generated input sets, requiring significant effort. To increase automation, AutoFeed uses a feedback-driven approach similar to fuzzing. However, the goal is not to reach error states or crash the system, but rather to generate different program behaviors using *valid inputs* in order to discover side-channels that leak information.

The user helps the analysis by providing *seed inputs* for the target system and a set of *mutators* which transform valid input into new valid inputs. As an example, consider `Airplan`, an air traffic control system from the DARPA STAC [6] benchmark where users can upload, edit, and analyze flight routes by cost, flight time, passenger and crew capacities. A seed input for this problem is a (weighted) graph of airports and flight routes. For this example we wrote two mutators: a RemoveFlight mutator takes a file containing a map of flights and removes one of the direct flights between two airports; similarly, an AddFlight mutator adds a direct flight to the map. Such mutators are written by hand, but can be reused for the analysis of many secrets, for the same application or for similar applications.

The user also designates the *secret of interest*, i.e., some aspect of the program data that they consider sensitive and whose leakage they want to detect and quantify. For instance, for `Airplan`, the *secret* is the number of airports in a route map uploaded by a user. AutoFeed then repeatedly executes the target system, generates new inputs, captures the network traffic, and adjusts input generation and system execution strategies based on the feedback it obtains by analyzing the captured traffic.

Since it is difficult for users to know which mutators will be most important for a secret, AutoFeed measures the effect of different mutators on leakage estimation, and weighs them accordingly in a feedback-driven way. By focusing the computational effort on those mutators that have greater effect on the leakage estimation of the most promising features, the input space is explored efficiently.

Specifically, AutoFeed first executes the application of interest (App) with the initial seed inputs to generate an initial set of traces. Based on these traces, it calculates the number of inputs to generate per iteration that corresponds to a given input time budget per iteration. Then, it applies the mutators on the seed inputs to get new inputs, executes App on these inputs, and uses the generated traces to obtain an initial estimation of the information leakage. Using these initial leakage results, AutoFeed uses heuristics to compute weights for mutators, corresponding to the likelihood of applying that mutator during input generation. In computing weights, each mutator is applied to a variety of the seed inputs to generate new inputs. After the input generation, AutoFeed checks if the mutator has changed a field of the input affecting the secret or the top k features that leak the most information.

After these initialization steps, AutoFeed starts executing its main loop for feedback-driven exploration of the input state space for obtaining an accurate estimation of information leakage. In each loop iteration, AutoFeed uses the mutators to generate new inputs, executes the App on new inputs to generate traces, and updates the leakage estimation using all the traces captured so far. When the change in the leakage estimate falls below a small value, AutoFeed terminates execution and reports the computed leakage.

AutoFeed is able to successfully generate inputs and correctly quantify the information leakage for challenging benchmarks [6]. Furthermore, experiments conducted with AutoFeed demonstrate that, with the feedback-driven input generation, the leakage estimates converge faster than the version of AutoFeed without mutator weighing. By combining automated analysis with manually written mutators, AutoFeed can achieve better estimates for information leakage, and improve the state of the art in network side-channel detection and quantification.

4. Hybrid Techniques Combining Symbolic Execution with Fuzzing

Fuzzers are good at finding so called *shallow bugs* but they may fail to execute deep program

paths, i.e., paths that are guarded by specific conditions in the code. This is because fuzzers have little knowledge about the inputs that affect specific conditions. (Dynamic) symbolic execution is a program analysis technique that works by collecting and solving constraints based on the conditions from the code, and it is thus particularly well-suited for such cases. However, it is usually much more costly in computational resources than plain fuzzing.

Hybrid analysis techniques, that combine fuzzing with symbolic execution, aim to leverage the strengths of each technique while mitigating their limitations. Fuzzing can produce a large amount of inputs in a short time. Symbolic execution complements the fuzzer by using its constraint solving capabilities. At the same time, it uses the concrete inputs from the fuzzer to mitigate its own scalability issues.

4.1. Complexity Analysis with User-defined Costs

We report here on an application of a hybrid technique to (algorithmic) complexity analysis. Characterizing the complexity of a program has many practical applications, ranging from understanding and fixing performance bottlenecks, performing compiler optimizations and finding security vulnerabilities related to denial-of-service attacks.

In our work on the algorithmic complexity analysis with BADGER [11], we investigate the worst-case algorithmic behavior of software with regard to a *cost function*. For many program types it is interesting to maximize the *execution time* or the *memory consumption*. For these cases, the cost function can be implemented automatically by measuring the clock time, counting the executed instructions, or regularly polling the current memory consumption.

There are also program types, e.g., smart contracts for cryptocurrency, for which the focus needs to be on some other, domain-specific resource, such as *gas* for the Ethereum blockchain. Exceeding the allocated budget of this resource might lead to loss of cryptocurrency and complexity analysis can help avoid such situations. To include domain-specific resources in our analysis, we use guidance from the user.

Specifically, in BADGER, we allow user-defined cost values which are added as an annotation, i.e., an additional function call `addCost(int)`. Our fuzzing component picks up these values and uses them as additional factors for the fitness evaluation of mutated inputs. Our symbolic execution component also uses the values, to prioritize paths for the next exploration. Additionally, when the user-defined costs include symbolic values, a constraint solver is instructed to maximize the cost values for the generated inputs. While in the described scenario the annotations are used to focus the analysis on the values of single variables, they can be further used to *combine* values from several variables, enabling the developer to combine existing cost models without any change to the underlying search engine.

4.2. Differential Hybrid Analysis

The user-defined annotations can also enable more challenging, differential analyses, that reason over multiple program executions.

For example, in our work on DIFFUZZ [10] we focus on a side-channel analysis using a form of differential fuzzing, where we measure the *cost difference* (δ) for two program executions, obtained by running a program on the same public inputs, but with different secret values. If for any public and secret value combination, we can identify a cost difference $\delta > 0$, then we have found a secret-dependent path, which indicates a potential side-channel vulnerability. For our fuzzing campaign we define the goal to maximize this cost difference, which also helps to further assess the severity of the discovered side-channel vulnerability.

The work leverages the user-defined cost annotations originally developed for single-program analysis, without requiring modifications to the cost-guided fuzzer.

We transfer this concept to a general hybrid differential analysis framework with HYDIFF [12]. To showcase the flexibility of the approach, we demonstrate three analysis types: regression analysis, side-channel detection, and robustness checking for neural networks. For all three cases, the developer uses annotations to guide the search.

5. Related Work and Conclusion

Apart from technical approaches, the success of human-guided software testing depends on its deployment and interaction with its environment. For example, in related work, Böhme et al. [2] propose an interactive human-in-the-loop repair approach. To label auto-generated inputs, they leverage a learning-based mechanism. Whenever the learner would label an input as failing, it will *query* the developer, to manually investigate the corresponding program behavior. The number of queries is bounded to limit the manual effort. While this approach mitigates the general oracle problem, a recent study [13] indicates that developers in practice would prefer rather low interaction with such repair tools.

Similarly, there are existing works for debugging [17] and static analysis [4] showing that assumptions made by automated techniques often do not hold in practice. Böhme et al. [1] discuss challenges for human-in-the-loop fuzzing and raise the question of how to include the usually highly automated fuzzing in the common software development workflow. Such considerations include the interaction with developers as well as the integration into existing CI pipelines.

We conclude that this demonstrates the need for more user studies to better understand (1) how much developers want to be involved in *human-guided* software testing techniques, (2) what kind of input and specifications they can provide, and (3) how they want to interact with the generated outputs. Such future research can be accomplished with, e.g., user surveys, interviews, and field studies.

In this article we advocated for the need of minimal, principled human intervention in making automated program analysis techniques practical. We have shown that through seemingly simple devices, such as user-defined generators, mutations, and cost functions, we can significantly improve testing and solve challenging problems, ranging from the traditional testing objectives of bug finding, to side channel analysis, algorithmic complexity estimation and differential analysis applied to complex software systems, which are difficult, if not impossible, to perform fully automatically. Both fuzzing and symbolic execution remain very active areas of research in the software engineering community, which we believe can further benefit from other ways of user intervention.

Acknowledgments

We would like to thank the reviewers for their detailed comments and suggestions, which helped us greatly in improving this article. This work was funded by NSF Award # 1901136 *SHF: Medium: Collaborative Research: HUGS: Human-Guided Software Testing and Analysis for Scalable Bug Detection and Repair*.

REFERENCES

1. M. Böhme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and opportunities. *IEEE Software*, pages 1–9, 2021.
2. M. Böhme, C. Geethal, and V. T. Pham. Human-in-the-loop automatic program repair. In *13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 274–285, 2020.
3. K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *5th International Conference on Functional Programming*, ICFP, 2000.
4. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.
5. I. B. Kadron and T. Bultan. TSA: A tool to detect and quantify network side-channels. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1760–1764. ACM, 2022.
6. İ. B. Kadron, N. Rosner, and T. Bultan. Feedback-driven side-channel analysis for networked applications. In *29th International Symposium on Software Testing and Analysis*, pages 260–271, 2020.
7. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.
8. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, pages 973–990, 2018.
9. MITRE Corporation. CWE-409: Improper handling of highly compressed data. https://cwe.mitre.org/data/definitions/409.html.

10. S. Nilizadeh, Y. Noller, and C. S. Păsăreanu. Diffuzz: Differential fuzzing for side-channel analysis. In *41st International Conference on Software Engineering*, ICSE'19, page 176–187. IEEE Press, 2019.
11. Y. Noller, R. Kersten, and C. S. Păsăreanu. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *27th International Symposium on Software Testing and Analysis*, ISSTA'18, pages 322–332, New York, NY, USA, 2018. ACM.
12. Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske. Hydiff: Hybrid differential software analysis. In *42nd International Conference on Software Engineering*, ICSE'20, page 1273–1285, New York, NY, USA, 2020. ACM.
13. Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury. Trust enhancement issues in program repair. In *44th International Conference on Software Engineering (ICSE'22)*, 2022.
14. R. Padhye, C. Lemieux, and K. Sen. JQF: Coverage-guided property-based testing in java. In *28th International Symposium on Software Testing and Analysis*, ISSTA'19, pages 398–401, 2019.
15. R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with Zest. In *28th International Symposium on Software Testing and Analysis*, ISSTA'19, pages 329–340, New York, NY, USA, 2019. ACM.
16. R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: domain-specific fuzzing with waypoints. *Object-oriented Programming, Systems, Languages, and Applications*, 3(OOPSLA):1–29, 2019.
17. C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, ISSTA '11, page 199–209, New York, NY, USA, 2011. ACM.
18. S. Wang, X. Jiang, X. Yu, and S. Sun. Kcfuzz: Directed fuzzing based on keypoint coverage. In *International Conference on Artificial Intelligence and Security*, pages 312–325. Springer, 2021.