The software that is being written is getting bigger and increasingly complex. This growing complexity can be attributed to multiple factors ranging from high consumer expectations for new features, diversification of employed software stacks to meet those expectations, and increasing proliferation of AI-based tooling allowing developers to write code at break-neck speed [1].

As prior studies have shown, rapid software development often leads to the introduction of bugs at a similar pace [2, 3]. Bugs can lead software into unintended states, some of which adversaries may exploit to mount attacks, ranging from making the software perform unintended functions to leaking sensitive data. Consequently, testing code—especially newly-added code—to ensure it is bug-free is crucial.

Dynamic software testing enables scalable analysis by reasoning about program behavior based on the paths exercised by test inputs. Fuzzing is a dynamic testing technique highly effective at uncovering vulnerabilities at scale [4] and is widely endorsed by government agencies [5] and corporations [6, 7]. However, existing fuzzing approaches struggle to uncover bugs that are deeply hidden in the software or lack effective oracles to flag their discovery. Furthermore, current tooling is not designed with the user experience in mind creating friction for widespread adoption.

My research vision is to *enhance software testing techniques to uncover hard-to-detect bugs and to democratize fuzzing by streamlining its integration into the software development pipeline*. To expand the frontier of bugs fuzzing can uncover, I have contributed towards improving the core components of dynamic testing. My work has uncovered 25 previously unknown vulnerabilities, with 5 CVEs assigned. Part of my research has been upstreamed into popular state of-the-art fuzzers like AFL++[1] and LibAFL[2]. Furthermore, some of my works were developed in close collaboration with major industry players, including Oracle. My key contributions have been in:

- Augmenting the input generation process using the target's domain knowledge to make input space exploration more efficient [8, 9].
- Developing specifications to find bugs that were previously hard to find automatically [10].
- Crafting program representations that allow fuzzers to rigorously test software patches while also expanding their coverage to hard-to-fuzz code regions [11, 12].

Going forward, my research will focus on three main directions to realize my research vision. First, I will develop highly automated testing solutions that help developers detect semantic bugs, which are challenging to identify automatically due to their reliance on specialized domain knowledge. Second, as the software ecosystem becomes increasingly heterogeneous with diverse software components interacting, it is crucial to test the interfaces where these interactions occur. To address this, I will create methodologies and metrics for assessing the correctness of cross-component interactions at scale. Finally, to encourage the widespread adoption of fuzzing, it is essential to minimize the friction developers face when integrating it into their development pipelines. I will achieve this by building agentic solutions to streamline end-to-end testing integration and by providing enriched feedback that helps developers better understand the quality of their testing efforts.

Prior and Current Research

My research till now has focused on making fuzzing more effective at discovering bugs hidden deep within software, especially those requiring a complex set of triggering preconditions. I achieved this by investigating strategies to enhance various components of dynamic testing, including input generation, specification creation, and program representation refinement for testing by incorporating domain knowledge from the software under test. Additionally, in the spirit of open science and to promote research reproducibility, all my published research is accompanied by reproducible software artifacts.

Optimizing Input Generation [8, 9]. Uncovering deep bugs in targets that accept structured input, such as language interpreters, demands the generation of syntactically valid inputs. To achieve this, existing fuzzers use

[1] https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators/gramatron

[2] https://github.com/AFLplusplus/LibAFL/tree/main/fuzzers/structure_aware/baby_fuzzer_gramatron

context-free grammars (CFG) to produce test inputs and apply grammar-aware mutations to maintain syntactic validity. However, when fuzzers use CFG production rules to generate inputs, they end up biased in their sampling of the input space due to the structure of the rules. Additionally, existing mutation operators in fuzzers perform small-scale mutations, which can be wasteful if the fuzzer is exploring grammar regions irrelevant to triggering deep bugs. With Gramatron [8], I introduce *grammar automatons*, a structural transformation of the input grammar which eliminates the sampling bias, and aggressive mutation operators to ensure the fuzzer does not get stuck in local minimas of coverage, enabling deeper testing of the software target. This framework uncovered 10 previously unknown vulnerabilities across three popular language interpreters including bugs that interpreter developers described as revealing "significant misunderstandings" of core mechanisms[3]. Owing to its evident effectiveness, Gramatron was upstreamed as a dedicated fuzzing mode in popular state-of-the-art fuzzers, AFL++, and LibAFL.

While Gramatron addresses the challenges of testing software with a known input structure, in certain domains, such as closed-source firmware on embedded devices, the accepted input structure is often unknown. Vendor-developed applications in embedded firmware are particularly under-vetted, making them more prone to vulnerabilities. Analyzing these applications is challenging because inferring their required input structure is non-trivial. To solve this issue, in FirmFuzz [9], I leverage the web application interfaces of these embedded firmware to generate syntactically valid inputs in conjunction with full-system emulation to trigger deep paths within the target applications. Using this domain-informed strategy, FirmFuzz uncovered seven previously undisclosed vulnerabilities with four CVE assigned. With EPOXY [13], we improved the state of security for embedded firmware even further by leveraging compiler-based instrumentation to enforce the principle of least privilege.

Enhancing Specifications [10]. Existing fuzzers are primarily designed to detect memory safety bugs, leveraging well-defined specifications that enable checking when an input triggers memory safety violations. However, identifying logical bugs—errors that violate an application's intended functionality—poses a different challenge, as there are no readily available specifications to detect when these bugs are triggered. One common scenario where such logic-based vulnerabilities arise for which no readily available specifications exist is during the deserialization of untrusted input [14]. Automatically uncovering these vulnerabilities requires navigating a large input space and flagging inputs that trigger these deserialization bugs. In Crystallizer [10], I address this challenge by formulating *gadget graphs* which Crystallizer uses to systematically navigate the large input space and also detect deserialization bugs when they are triggered. Crystallizer helped uncover previously unknown deserialization vulnerabilities in popular enterprise applications such as Apache Pulsar and Kafka. In addition, it also outperformed existing state-of-the-art automated solutions in uncovering such vulnerabilities.

Refining Program Representations [11, 12] The program representation employed by a fuzzer shapes the type of feedback it receives from the software under test during input evaluation, thereby guiding how the fuzzer explores and navigates the input space of the software under test. Consequently, it is crucial that the fuzzer's program representation aligns with its testing objectives. Building on this insight, we developed optimized program representations tailored for two distinct use cases: (i) incorporating directedness into the fuzzing process, and (ii) maximizing the code coverage achieved by the fuzzer.

In certain software quality control tasks, such as patch or regression testing, it is necessary to test specific code locations rather than the entire application. Existing directed fuzzers use distance minimization on a control-flow graph representation of the program to generate inputs that bring the fuzzer closer to the target location. However, this approach incurs significant overhead for each test case evaluation, slowing progress toward testing the desired location. In SieveFuzz [11], I introduce *tripwiring*, a technique that refines the control-flow graph representation to bias the fuzzer toward exploring target-reachable paths, eliminating the high overhead of distance calculations. By augmenting the program representation with tripwiring, SieveFuzz accelerates testing of target locations by 1.4x compared to state-of-the-art directed fuzzers.

In contrast to directed fuzzing, general-purpose fuzzing aims to maximize the code a fuzzer can reach to increase its bug discovery rate. This necessitates program representations specifically designed to maximize a fuzzer's code

[3] https://github.com/mruby/mruby/issues/5042#issuecomment-661586028

coverage. As part of FOX [12], we introduce *frontier branches*, a program representation allowing a fuzzer to leverage finer-grained data flow information to maximize coverage without significantly expanding the state space to explore. Our frontier-branch based representation coupled with control theory-guided optimization routines enabled uncovering up to 26% more coverage than current state-of-the-art fuzzers expanding the reach of fuzzers to previously untested code regions in heavily analyzed software. As part of my ongoing work, I am developing a program representation designed to maximize bug discovery by making the bug-triggering input space visible to fuzzers through sanitizer instrumentation. We have put together this work on bug-oriented fuzzing exploration into a proposal for a Google Research Award.

Future Research Directions

Semantic Bug Discovery. Semantic bugs cause an application to enter invalid states that violate its intended functionality. Uncovering such bugs is challenging because synthesizing oracles to detect them requires a specification against which inputs can be checked. Additionally, there are currently no generalized feedback metrics to quantify how much of the semantic space has been explored. To tackle this issue, I will adopt data-driven approaches to synthesize semantic models of the software under test. These models will not only serve as oracles to flag semantic violations but also act as coverage feedback mechanisms for test input generators. Finally, in addition to the semantic models, I will use formal language theory based approaches to create input space models that testing frameworks can use to track previously tested regions of the input space, ensuring more efficient use of compute resources for testing.

Short-term Plan. I will extend the work done as part of Crystallizer [10] by developing techniques to uncover semantic bugs rooted in the deserialization of trusted input, with a language-agnostic approach. I will first investigate known exploitation primitives to identify common semantics they use and then develop tooling to perform lightweight, semantic-guided searches over codebases to discover previously unknown primitives automatically.

Agentic solutions to Democratize Fuzzing. One key to improving the security of deployed software is the widespread adoption of fuzzing across all application domains. Employing fuzzing in and of itself is not hard, owing to the methodology's simplistic design. However, with current tooling, ensuring that the testing being performed is effective is non-trivial, requiring in-depth knowledge about the application under test and the fuzzer being used. My key insight is to enhance the developer experience when deploying fuzzing by enriching the feedback that fuzzers provide while testing. I will create debugging frameworks that help a developer model inputs the fuzzer is generating, identify preconditions that a fuzzer is having a hard time solving (fuzz blockers) and ways for a developer to easily test different strategies to overcome the testing roadblocks. Furthermore, I will propose methodologies to help developers identify which fuzzer-found crashes to prioritize for triaging and patching. Finally, to democratize the usage of fuzzing and make it easier for developers to integrate it into their development pipeline, I will build LLM-powered agents that can both recommend and deploy effective fuzzing setup based on developers' testing needs.

Short-term Plan. I will develop visualization techniques that can handle large codebases and allow developers to pinpoint where fuzz blockers are and how close they are to being solved. The visualization tooling will be interactive, enabling developers to direct fuzzing compute resources toward more promising blockers.

Evaluate Cross-Interface Interactions. Modern software systems consist of heterogeneous applications with varying safety guarantees and memory models that communicate across multiple layers of abstraction. Ensuring the security of these systems requires verifying the correctness of the interfaces through which communication occurs between diverse software components. This involves validating that an adversary cannot exploit an interface in a way that compromises the safety guarantees of the interacting components. I will develop testing frameworks that are designed to perform directed testing of interfaces between components. I will create feedback metrics that will allow a testing framework to quantify how robustly interfaces have been tested.

Short-term Plan. Rust is becoming an increasingly popular systems language due to its memory safety guarantees. However, a subset of features that enable interoperability with memory-unsafe languages can undermine Rust's safety guarantees [15]. Existing static solutions for evaluating Rust's interoperability with unsafe languages are incomplete and dynamic solutions require significant manual effort. I will develop hybrid testing solutions that combine dynamic, feedback-driven fuzzing approaches to explore the program state space, augmented by statically gathered target information. We are preparing a grant proposal for this research direction, which we plan to submit next year.

References

[1] Stack Overflow Developer Survey 2024. https://survey.stackoverflow.co/2024/ai/, July 2024.

[2] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[3] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *2010 IEEE 21st international symposium on software reliability engineering*. IEEE, 2010.

[4] Google. Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz, .

[5] Paul Black, Barbara Guttmann, and Vadim Okun. Guidelines on minimum standards for developer verification of software. https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf, 2021.

[6] Microsoft. A brief introduction to fuzzing and why it's an important tool for developers. https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/, 2020.

[7] Google. Use-cases of fuzzing. https://github.com/google/fuzzing/blob/master/docs/why-fuzz.md, .

[8] **Prashast Srivastava** and Mathias Payer. Gramatron: Effective Grammar-aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[9] **Prashast Srivastava**, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated IoT Firmware Introspection and Analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things (IOTSP)*, 2019.

[10] **Prashast Srivastava**, Flavio Toffalini, Kostyantyn Vorobyov, François Gauthier, Antonio Bianchi, and Mathias Payer. Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2023.

[11] **Prashast Srivastava**, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*, 2022.

[12] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, **Prashast Srivastava**, and Suman Jana. FOX: Coverage-guided Fuzzing as Online Stochastic Control. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[13] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, **Prashast Srivastava**, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (IEEE SP)*, 2017.

[14] OWASP. Owasp a08: Software and data integrity failures. https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/.

[15] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *NDSS*, 2022.