Research Statement

Wen Li (li.wen@wsu.edu)

Motivation. The software development landscape now widely embraces the use of multiple programming languages to harness their respective strengths, such as C's efficiency and Python's programmability. This trend is evident in domains like Android OS and machine learning frameworks like PyTorch. However, this approach introduces challenges due to the complexity of multi-language systems. Developers must grapple with vulnerabilities arising from this complexity, lacking effective tools for identifying security threats.

Conventional techniques like program analysis and fuzzing primarily focus on single-language software, limiting their effectiveness for vulnerability discovery. Program analysis struggles with understanding control and data flows across different languages, leading to incomplete vulnerability detection. Grey-box fuzzing faces challenges adapting to various languages, incomplete coverage, and ensuring reproducibility of vulnerabilities. Additionally, runtime systems hosting multilingual software compound the issue, as they are often built using multiple languages. This makes security clearance within such runtimes a significant challenge in contemporary software development.

Summary. During my Ph.D. journey, I tackled the challenges of multi-language software vulnerabilities with a multifaceted approach. I conducted empirical investigations into vulnerability susceptibility [2, 4, 3], leading to the creation of PolyCruise [5], a *dynamic cross-language information flow analysis* technique. PolyCruise effectively detected real-world vulnerabilities in open-source multi-language systems. Recognizing the importance of test input coverage, I developed PolyFuzz [6], a holistic gray-box fuzzing methodology. By incorporating *sensitivity analysis and whole-system coverage measurements*, PolyFuzz generated powerful test inputs, strengthening vulnerability identification. Shifting the focus to runtime security, I crafted PyRTFuzz [1], a *two-level collaborative fuzzing framework* for Python interpreter runtimes. This contribution enhanced fundamental software system safeguarding.

Looking forward, my plan involves strengthening cross-language security testing by integrating holistic fuzzing and comprehensive cross-language data flow facts. Furthermore, I will delve into compiler and language runtime testing. First, I intend to enhance collaborative fuzzing by merging program analysis and semantic mining techniques. And then generalize these methodologies across diverse runtimes, encompassing platforms like JVM and JavaScript engines. Additionally, I am enthusiastic about exploring the intersection of machine learning frameworks and runtimes. This convergence between machine learning and security holds immense potential for pioneering research endeavors.

1 Current Research

In my Ph.D. research, I have undertaken significant projects across various domains. These projects encompass a security analysis of application-level systems, including endeavors like PolyCruise and PolyFuzz. Additionally, I've ventured into the realm of language runtimes with PyRTFuzz.

Cross-language Information Flow Analysis. Within the realm of multilingual systems, the interaction between different languages relies on diverse interfacing mechanisms [5, 7]. These complexities create two main challenges for achieving comprehensive cross-language program analysis: (1) The first challenge stems from varying semantics among the heterogeneous languages. Conventional single-language methods can't be directly applied due to these differences. It's also impractical to conduct separate analyses on individual language components and merge the outcomes. (2) The second challenge emphasizes scalability, especially for large-scale real-world systems. Multilingual systems often encompass more extensive codebases compared to single-language systems.

To tackle the first challenge, I embarked on *dynamic information flow analysis (DIFA)*, a method that effectively overcomes the constraints of static semantic analysis. DIFA involves observing program behaviors during execution, offering a means to navigate the intricacies arising from disparate language semantics. Specifically, this dynamic analysis operates in a real-time, online manner directly within

the program's memory. As the program runs, it dynamically computes information flow between predetermined sources and sinks, a process known as *run-time information flow computation (IFC)*. The outcomes of this computation materialize as flow facts, seamlessly integrated into an evolving dynamic information flow graph (DIFG). Using the insights from DIFG as a foundation, it's possible to create applications specialized in detecting specific vulnerabilities.

To overcome the second challenge, I introduced static symbolic analysis using a custom intermediate representation called *Language-Independent Symbolic Representation (LISR)*. The procedure initiates with lightweight analyses optimized for the particular programming language, converting code segments into the language-agnostic LISR format. Following this, *symbolic dependence analysis is* employed on these LISRs in conjunction with source and sink lists for each code unit. LISRs' inherent language-independent quality facilitates the applicability of this analysis technique across diverse programming languages. The insights gleaned from the symbolic dependence analysis guide a targeted approach to instrumentation, minimizing the extent of required modifications and consequent impact on runtime performance.

Being the pioneer of the cross-language dynamic information flow analysis approach, PolyCruise integrates a language-independent real-time data flow analysis guided by these dependencies, effectively surmounting the challenges posed by language diversity. Additionally, PolyCruise has succeeded by uncovering *14 previously unknown security vulnerabilities* in real-world multilingual systems, including well-known instances like NumPy. This impressive feat underscores the importance and practical applicability of the approach.

Holistic Grey-box Fuzzing. In multi-language software analysis and testing, the coverage of test inputs critically limits dynamic information flow analysis [5], which can be reduced through techniques like grey-box fuzzing. However, existing fuzzing techniques are tailored exclusively for single-language software, mostly focusing on C/C++. Even seemingly multilingual fuzzers target only one language unit. Applying these fuzzers to multilingual code could treat other language units as black boxes, disregarding cross-language interactions and significantly reducing the effectiveness of fuzzing.

Hence the endeavor to conduct comprehensive fuzzing on real-world multi-language systems encounters two significant challenges. The first challenge involves achieving extensive coverage while accommodating the extensibility of different programming languages. The second challenge centers on generating inputs capable of efficiently testing information flow across varied language units using greybox fuzzing. To tackle the initial challenge, a *specialized intermediate representation named SAIR* is introduced for measuring fuzzing metrics. A comprehensive static program analysis on this representation reduces the need for language-specific analyses. This approach achieves broad and adaptable coverage. To overcome the second challenge, PolyFuzz integrates *sensitivity analysis-driven seed generation*. This involves training regression models to discern the connections between inputs and branch variables. The trained models predict values for seed blocks with constant branch values as inputs; then, these blocks are assembled into seeds to alter branch conditions effectively, thus enhancing fuzzing efficiency.

PolyFuzz is a leading holistic greybox fuzzer known for comprehensively testing multi-language systems. It's flexible, accommodating various language combinations such as C, Python, and Java. Furthermore, PolyFuzz consistently outperforms single-language fuzzers, achieving significant *code coverage enhancements (10%-52.3%)* and uncovering *more bugs (1-10 vulnerabilities)* across multi-language and single-language programs. Impressively, PolyFuzz has discovered 14 previously unknown vulnerabilities, underscoring its effectiveness.

Collaborative Fuzzing for Language Runtime. The security of the language runtime is crucial for multi-language software hosting, as it provides essential services like memory management, code execution, and resource access. Current approaches fail to test Python runtimes due to three primary challenges. Firstly, comprehensive Python runtime fuzzing requires close collaboration between two fuzzing levels, generating programs and concrete inputs, to thoroughly exercise interpreter-runtime library interactions. Secondly, fuzzing the interpreter demands diverse yet valid Python applications. However, achieving both requirements simultaneously, especially for Python, remains largely unknown. Lastly, generating quality input values is challenging for application fuzzers in general, and even more so for Python due to its dynamic typing, which hinders format-aware input generation essential for

effective Python application fuzzing.

To overcome these challenges, a *two-level collaborative fuzzing* approach integrates generation-based and mutation-based techniques at Level-1 and Level-2, respectively. Operating within a holistic fuzzing loop, this approach addresses Challenge 1 by utilizing shared coverage feedback.

At Level-1 of *generation-based fuzzing*, PyRTFuzz employs a unique approach using SLang, a scripting language incorporating elements from Python's syntax, semantics, and features. This involves combining selected primitives onto a Python runtime API in a random manner to form an application specification. This specification is then translated by the SLang compiler into a Python application with diverse control flow complexities. Additionally, this strategy spans various runtime domains, which improves the strength and thoroughness of the generated test inputs or applications. This effectively addresses Challenge 2. At Level 2 of *mutation-based fuzzing*, a novel mutation strategy is presented. This strategy considers the application input's specific data types, utilizing insights from runtime API descriptions to understand data types, format, and structure. Guided by the variable data types, this specialized mutator generates suitable values customized for each runtime API, which significantly improves the accuracy and efficacy of the mutation strategy, effectively addressing Challenge 3.

PyRTFuzz combines generation-based fuzzing at the compiler level and mutation-based fuzzing at the application-testing level for the primary Python implementation (CPython). Through its usage, PyRTFuzz identified *61 new, demonstrably exploitable bugs*. These bugs span issues within the interpreter, most occurring in the runtime libraries. The findings underscored PyRTFuzz's scalability, cost-effectiveness, and potential for continued bug discovery. The collaborative two-level fuzzing approach in PyRTFuzz holds promise for extending its application to other language runtimes, particularly those using interpreted languages.

2 Future Research

Looking ahead, the widespread adoption of multi-language software is anticipated across various domains. Ensuring the security of both the software and the hosting language runtimes becomes paramount within these diverse domains. Expanding upon my current endeavors, my forthcoming objectives are outlined as follows: Firstly, my unwavering commitment lies in advancing security testing for cross-language applications. Subsequently, my focus shifts toward the refinement of collaborative fuzzing techniques. This strategic enhancement aims to achieve a thorough testing framework for the Python runtime and subsequently, to extrapolate this refined approach to encompass a broader spectrum of language runtimes. Furthermore, I am dedicated to addressing the forefront of current interests. I am actively immersed in the security analysis of AI compilers and runtimes, encompassing prominent platforms such as PyTorch and MindSpore.

Cross-language Security Testing. PolyCruise and PolyFuzz have made significant advancements in cross-language information flow analysis and comprehensive grey-box fuzzing. However, there is still room for improving cross-language vulnerability discovery. While PolyFuzz has enhanced fuzzing efficiency through semantic relationships and system coverage, its current limited semantics don't accurately model complex data flow information, which is crucial for vulnerability exploration, as prior research indicates. To address this, my next focus is on seamlessly integrating cross-language information flow analysis into holistic fuzzing. The plan involves developing methods like taint-guided holistic fuzzing using insights from cross-language data flow analysis and exploring directed fuzzing based on potentials identified by cross-language data flow analysis. These strategies aim to concentrate fuzzing efforts on areas highlighted by data flow analysis as having higher vulnerability likelihood, thus optimizing resources and intensifying security weakness identification.

Language Runtime Fuzzing. The SLang-based approach has successfully generated applications with varying control flow complexities tailored to specific runtime APIs. Nevertheless, it falls short in representing realistic software scenarios, mainly due to the lack of potential dependencies among runtime APIs. This deficiency hampers the ability to simulate real-world application interactions, potentially limiting its effectiveness in capturing intricate software behaviors. Additionally, comprehensive testing of interpreters necessitates consideration of broader language features or characteristics

beyond control flow structures. Focusing solely on control flow could result in incomplete testing, overlooking critical language behaviors.

Furthermore, the security of mainstream language compilers and runtimes hosting various languages has emerged as a critical concern within multi-language software. This challenge is particularly pronounced in languages like JVM, JavaScript, Python, and their combined scenarios. Despite the theoretical promise of the proposed approach to span different language runtimes, practical implementation encounters obstacles stemming from the nuanced differences among languages. Bridging the gap between theoretical potential and practical application demands a focused endeavor to develop an adaptable framework. This framework should be flexible to accommodate the unique attributes of diverse language ecosystems, thereby enhancing runtime security across the spectrum of multi-language software and ensuring robust protection across varied language runtimes.

AI Compiler and Runtime Security Assurance. Much like traditional language compilers and runtimes catering to applications across various domains, the security of AI compilers and runtimes is gaining increasing importance within artificial intelligence and machine learning. As AI technologies advance and integrate into diverse applications, security assurance within AI compiler and runtime environments becomes a central concern, encompassing several crucial dimensions. Firstly, these compilers and runtimes often encounter sensitive information due to the nature of the data they handle. Consequently, establishing secure processing environments is paramount to avert unauthorized data exposure. Moreover, akin to conventional software, AI runtimes can exhibit vulnerabilities susceptible to exploitation by malicious actors. Furthermore, the gamut of security threats extends to issues such as insecure execution or the injection of malicious models. These intricacies demand meticulous consideration within the architecture and operation of AI runtimes. As the landscape of AI technology evolves, secure compilers and runtimes will persist as a linchpin in constructing dependable and credible AI systems across various domains. As a significant stride within the scope of my long-term research, I am fully committed to investigating robust and feasible techniques that ensure the security of AI compilers and runtimes.

References

[1] Wen Li. Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. `https://github.com/awen-li/PyRTFuzz`, 2023.

[2] Wen Li, Li Li, and Haipeng Cai. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 847–859, 2022.

[3] Wen Li, Li Li, and Haipeng Cai. Polyfax: a toolkit for characterizing multi-language software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1662–1666, 2022.

[4] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2023.

[5] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. {PolyCruise}: A {Cross-Language} dynamic information flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2513–2530, 2022.

[6] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. Polyfuzz: Holistic greybox fuzzing of multi-language systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[7] Haoran Yang, Weile Lian, Shaowei Wang, and Haipeng Cai. Demystifying issues, challenges, and solutions for multilingual software development. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1840–1852. IEEE, 2023.