Research Statement

Qingkai Shi

Programming language and software engineering technologies set the foundation for securing both conventional software systems and modern software born in the new age of IoT and 5G. Our general research interest focuses on the most fundamental part of programming language and software engineering technologies and theories, and aims for hunting software vulnerabilities in a more precise and efficient manner. In the past several years, we have made a few achievements in both academia and industry.

- In academia, our research frequently appears in prestigious venues in the area of programming language (PLDI, OOPSLA), software engineering (ICSE, ESEC/FSE), and cybersecurity (S&P). Our research also has won me many awards including ACM SIGSOFT Distinguished Paper Award and Hong Kong Ph.D. Fellowship.
- In the open-source community, our research has discovered over one hundred vulnerabilities in popular open-source software, including Apache, MySQL, and Firefox. Many of these discovered software vulnerabilities have been assigned CVE identifiers due to their security impacts.
- In industry, our research has been deployed in many Global 500 companies to secure their products. In 2020, the startup, Sourcebrella Inc, that commercializes our research was acquired by Ant Group to help ensure the quality of its product, Alipay, a popular digital payment app with over a billion monthly active users.

In the next sections, we will describe the most impactful research projects we have worked on in recent years (§1 and §2). We conclude the research statement with a brief discussion on future work (§3).

1 Static Code Analysis for Industrial-Sized Code

In 2014, the infamous software vulnerability, Heartbleed, was disclosed and shocked the world since the host code library, OpenSSL, is widely used in almost every corner of our everyday life. As a researcher of programming language, we were also shocked because static code analysis had been extensively studied for decades and should have been able to discover such software vulnerabilities easily. However, the reality is brutal. The discovery of Heartbleed makes it clear that there are still many problems that have not been addressed yet. From 2015, we started developing our own static analysis framework, known as Pinpoint, and aimed to push forward a revolution in this area. Despite still many challenges, it is fortunate that our research has addressed several key problems in static code analysis and allows successful industrial deployment in many companies. Proudly, we have made a few achievements as discussed before and frequently published papers in top-tier venues. Here are five representative papers where Paper 3, Paper 4, and Paper 5 are my dissertation works.

1. **Qingkai Shi**, Yongchao Wang, and Charles Zhang. Indexing Context-Sensitive Reachability. In *Proceedings of the 36th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'21)*. ACM, 2021.
2. **Qingkai Shi**, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-Sensitive Sparse Analysis without Path Conditions. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*. pp 930–943. ACM, 2021.
3. **Qingkai Shi** and Charles Zhang. Pipelining Bottom-up Data Flow Analysis. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*. pp 835–847. ACM, 2020.
4. **Qingkai Shi**, Rongxin Wu, Gang Fan, and Charles Zhang. Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*. pp 812–823. ACM, 2020.
5. **Qingkai Shi**, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. pp 693–706. ACM, 2018.

Fast and Precise Static Code Analysis [PLDI'18, ICSE'20a, ICSE'20b, ISSTA'20a, PLDI'21, OOPSLA'21a]. Our static analysis follows a sparse design to achieve better performance. Different from a conventional dataflow analysis that propagates dataflow facts via control flows, a sparse analysis propagates dataflow facts via data-dependency relations to skip unnecessary control flows, thus being more scalable. Generally, our static analysis consists of two steps: resolving the data-dependency relations in a given program and then looking for one or more data-dependency paths that may lead to software vulnerabilities.

(1) Context-Sensitivity. The detection of many software vulnerabilities, such as null exceptions and taint issues, can be formulated as a CFL-reachability problem on a given data-dependency graph. CFL-reachability is different from conventional graph reachability as it requires that each return operation should correctly match the call operation in a given data-dependency path. Unfortunately, it is well-known that the CFL-reachability problem is of sub-cubic time complexity and quadratic space complexity, which is not affordable for analyzing large-scale software. We developed Flare, an approach to reducing the CFL-reachability problem to the problem of conventional graph reachability [OOPSLA'21a]. This reduction allows us to address the long-standing "sub-cubic bottleneck" by benefiting from existing indexing schemes of conventional graph reachability. These indexing schemes enable us to respond to each CFL-reachability query in almost constant time at the cost of only a moderate space overhead.

(2) Path-Sensitivity. Reducing the false positive rate is a critical task to make a static code analyzer practical. Achieving path-sensitivity is an effective manner to reduce the false positives and, in recent years, we have made a lot of efforts in this research direction. Basically, to make our static analysis path sensitive, we need to (a) build path-sensitive data-dependency graph [PLDI'18] and (b) path-sensitively check the feasibility of a data-dependency path on the graph [ISSTA'20a, PLDI'21].

First, building path-sensitive data-dependency graph is hard because of the "pointer trap" — a precise pointer analysis limits the scalability of static code analysis and an imprecise one seriously undermines the analysis precision. To escape from the pointer trap, we present a holistic approach that decomposes the cost of high-precision pointer analysis by precisely discovering local pointer relations and delaying the expensive inter-procedural analysis through memorization [PLDI'18]. Such memorization enables the demand-driven computation of only the necessary inter-procedural pointer relations and path feasibility queries, which are then solved by a costly SMT solver. Experiments showed that our approach can build path-sensitive pointer relations for two million lines of code in just twenty minutes while conventional approaches cannot finish in twelve hours. The precise pointer relations further enable a fast and precise static code analysis that only reports around 20% false positives when used to check the use-after-free vulnerabilities or taint issues.

Second, checking the feasibility of a data-dependency path via SMT solving is expensive and a major source of the cost that creates non-negligible obstacles to deploying static analysis in practice. While SMT solving is expensive at a general setting, we observe that, in the environment of program analysis, SMT solvers can be significantly optimized by utilizing the program dependency relations [ISSTA'20a] and the program modular structure [PLDI'21]. The program dependency relations (data dependency and control dependency) allow us to design effective heuristics to optimize the DPLL algorithm, the core algorithm of an SMT solver. The program modular structure significantly reduces the size of path conditions, eliminating a large number of function clones. The experimental results demonstrated that, armed with the optimized SMT solver, our static analyzer achieves up to 10× speedup, significantly lowering the barrier to deployment in the industry.

(3) Parallelism. When checking industrial-sized software, we often parallelize the analyses of different program modules, e.g., functions, to benefit from multi-cores in a modern computer. However, such function-level parallelism is significantly limited by the calling dependence — functions with caller-callee relations have to be analyzed sequentially because the analysis of a function depends on the analysis results of its callees. To improve the parallelism, we propose a new parallel design, in which the analysis task of each function is elaborately partitioned into multiple sub-tasks [ICSE'20a]. These sub-tasks are pipelined and run in parallel, even though the calling dependence exists. Evaluation demonstrated that we can achieve 2×–3× speedup over the conventional parallel design. Such speedup is significant enough to make many overly lengthy analyses useful in practice.

(4) Extensibility. In the industry, we often need to check a few dozen or even hundreds of program properties at the same time. However, to the best of our knowledge, a very limited number of existing static analyses have studied this problem, despite its importance at an industrial setting. A major factor to this problem, as we observe, is that the core static analysis engine is oblivious of the mutual synergy among the properties being checked, thus inevitably losing many optimization opportunities. Our work leverages the inter-property awareness and captures the redundancies and inconsistencies when many properties are considered at once [ICSE'20b]. That is, before analyzing a program, we make optimization plans to reuse the analysis results of a property to speed up checking others. Our evaluation, where

we simultaneously checked twenty common properties, showed that our approach is more than 8× faster than existing ones but consumes only 1/7 of the memory.

Dealing with Specific Bug Categories [ICSE'19, ISSTA'20b, ISSTA'20c]. In addition to the general dataflow analysis framework discussed above, we also dived into the investigation of some specific bug categories and aim to employ their specific features to boost the static analysis.

(1) Hunting type-state bugs. We developed Smoke based on the insight that only a small proportion of program paths lead to software bugs. Specifically, Smoke is a two-staged analysis [ICSE'19] by first computing a succinct set of candidate buggy paths through a novel data structure known as the use-flow graph, followed by a precise and heavy-weight verification of these paths. Experimental results showed that we can finish checking industrial-sized projects, up to eight million lines of code, in forty minutes with an average false positive rate of 24.4%. This is significantly faster than recent industrial tools, with the speedup ranging from 5.2× to 22.8×. This work received an ACM SIGSOFT Distinguished Paper Award at the 41st ACM/IEEE International Conference on Software Engineering.

(2) Hunting code clones. Detecting code clones (Type 1 – Type 4) is a fundamental task in software engineering. Type-1, Type-2, and Type-3 code clones have been well addressed in recent years but Type-4, heterogeneous code snippets that share the same functionality but have different code structures or syntax, has not yet as it needs precise modeling of program semantics. On top of our path-sensitive data-dependency analysis, we understand the program semantics better than the state of the arts. Hence, our approach can detect Type-4 clones in a much more precise manner [ISSTA'20b].

(3) Hunting build script errors. Almost all software projects resort to build systems (e.g., CMake, Bazel and Ninja) to automatically transform the source code to executable software. Errors in build scripts are quite common and may lead to severe software vulnerabilities. To block the source of vulnerabilities in build scripts, we designed a dedicated static analyzer [ISSTA'20c], which reduces the error hunting problem into a graph reachability problem on a new data structure known as the unified dependence graph. This novel structure uniformly encodes static and dynamic dependencies and, thus, allows more effective error detection. In the experiments on forty-two mature open-source projects, we found over two thousand build script errors, all of which have been confirmed by software developers.

Validating the Correctness of SMT Solvers [ISSTA'21, ESEC/FSE'21]. SMT solvers, e.g., Z3 and CVC4, play a critical role in our static analyzers to check the bug-triggering condition. However, we observe that there are many bugs, either soundness bugs or crash bugs, in the solvers, which significantly affect the deployment of our static analyzer in practice. We proposed Falcon [ISSTA'21] and Sparrow [ESEC/FSE'21] to help validate the correctness of an SMT solver. From the technical aspect, Falcon was designed based on the observation that prior work only focused on generating various first-order formulas as the inputs but neglected the algorithmic configuration space of an SMT solver, which leads to under-reporting many deeply-hidden bugs. Differently, Falcon explores both the formula space and the configuration space, and reduces the search space according to the correlations between the two spaces. Sparrow is a skeletal approximation enumeration approach that specially designed for discovering soundness bugs. Its key idea is to enumerate the under- or over-approximations of a given test formula, and test the soundness of an SMT solver based on the relation between the original test input and the approximated test input. To date, Falcon and Sparrow have discovered over 1600 bugs in Z3 and CVC4, making our static code analyzers more stable.

2 Abstract- and Diversity-Guided Fuzz Testing

Dynamic program analysis complements the static one. Particularly, we are also working on fuzz testing, an automated software testing technique that is conducted to reveal coding errors and security loopholes in software, networks, or operating systems. Behind our recent works, the core idea is to guide the testing procedure via the abstraction of program behaviors [TSE'16, S&P'20, OOPSLA'21b, S&P'22] and the diversity of test inputs [TRel'16, ISSTA'20d, S&P'20]. Here are five representative papers on fuzz testing, where the label "*" means that I supervised the work and was marked as the corresponding author.

1. Heqing Huang, Yiyuan Guo, **Qingkai Shi***, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P'22)*. IEEE, 2022.
2. Heqing Huang, Peisen Yao, Rongxin Wu, **Qingkai Shi***, and Charles Zhang. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. pp 1613-1627. IEEE, 2020.

3. Yang Feng, **Qingkai Shi***, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. DeepGini: Prioritizing Massive Tests to Enhance the Robustness of Deep Neural Networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. pp 177–188. ACM, 2020.
4. **Qingkai Shi**, Zhenyu Chen, Chunrong Fang, Yang Feng, and Baowen Xu. Measuring the Diversity of a Test Set with Distance Entropy. In *IEEE Transactions on Reliability (TRel'16)*, Vol. 65, No. 1. pp 19-27. IEEE, 2016.
5. **Qingkai Shi**, Jeff Huang, Zhenyu Chen, and Baowen Xu. Verifying Synchronization for Atomicity Violation Fixing. In *IEEE Transactions on Software Engineering (TSE'16)*, Vol. 42, No. 3. pp 280-296. IEEE, 2016.

Abstract-Guided Fuzzing [TSE'16, S&P'20, OOPSLA'21b, S&P'22]. An abstract is a brief summary of program behaviors, which provide useful information to improve the performance of fuzzers. We developed Swan, a directed fuzzer that aims to validate if a concurrency bug is sufficiently fixed by synchronization [TSE'16]. By creating an execution trace of shared memory accesses (an abstract of the interleaving of multiple threads), we can discover a minimal set of thread interleavings to test without any knowledge of the bug to fix. We developed Pangolin [S&P'20] and Beacon [S&P'22], which compute the abstract of path conditions in the polyhedral and interval domain to improve the performance of fuzzing. Pangolin computes the polyhedral path abstract to preserve the exploration state in the concolic execution stage and allows more effective mutation and constraint solving over existing techniques. To the best of our knowledge, this is the first "incremental" hybrid fuzzer that can reuse previous computation results for optimization. Pangolin discovered 41 unseen bugs with 8 of them assigned with CVE identifiers. Beacon computes the interval path abstract to prune infeasible paths that cannot reach a given testing target, e.g., a possibly vulnerable point in the code. Beacon found 14 incomplete fixes of existing CVE-identified vulnerabilities and 8 new bugs while 10 of them are exploitable with new CVE identifies assigned. The methodology of computing high-quality polyhedral and interval abstract is summarized and published in OOPSLA, which is expected to be widely applicable other than fuzz testing [OOPSLA'21b].

Diversity-Guided Fuzzing [TRel'16, ISSTA'20d, S&P'20]. When the code of software is not available or it is expensive to compute the abstracts, we can consider the diversity of test inputs. The key insight of diversity-guided fuzzing is that similar test inputs exhibit similar program behaviors. Thus, to thoroughly test a program, we need to maximize the diversity of test inputs, which, intuitively, aims to evenly spread the test inputs in the input space. In 2016, we proposed a metric called distance entropy to measure the diversity of a given test set [TRel'16]. This idea then was applied to test intelligent software based on deep neural networks [ISSTA'20d]. While many works claimed that conventional code coverage metrics can be applied to guide the testing of deep neural networks, we demonstrated that this may be not true but the diversity-based method effectively works. We also applied the idea of diversity to guide our fuzzer, Pangolin [S&P'20] — given a path abstract that renders a bounded range of the input variables with respect to a path prefix, by sampling from such bounded search space, we are able to quickly generate a large number of new inputs that still satisfy this path condition and, meanwhile, to explore the subsequent paths sharing the same path prefix. Such a diversity-based sampling method saves a lot of computational resources that would have been used for solving path conditions.

3 Future Work

Software revolution is ongoing, especially in the new age of IoT and 5G. In the course of our research, we have noticed that a fundamental challenge for analyzing modern software is to deal with the *interoperability* among (1) multiple components in a single software application or (2) multiple software applications in an open environment. Our future research will center around the keyword, interoperability.

Modern software applications often rely on a large number of external libraries or frameworks, which have various versions and are written in different programming languages. The application code and the external code must cooperate to complete the application's functionality. However, we usually do not have any knowledge of the external code, which inevitably leads to a misunderstanding of the code semantics. Even with the knowledge of the external code, it is still challenging because the search space of the software application, including a great deal of external code, is extremely large. Meanwhile, different programming languages in an industrial-scale software application may involve different memory models and different type systems. These differences make it hard to perform a conventional program analysis since we have a lack of uniform approaches to managing different language semantics. Our short-term research will focus on addressing these interoperability problems in modern software by designing uniform program models and building big-code warehouses with the capability of cloud storage and fast query of program semantics.

In the long term, we are convinced that, in addition to focusing on the complexity of software itself, we also need to study the complex application environment of software as well as the interactions between software and the environment. This could be more challenging because, for example, in the age of IoT and 5G, it is more important to care about how a large number of systems cooperate through various communication channels. Given that a single software system has already been very complex, processing combinations of many software systems is apparently an extremely challenging task. However, it is meaningful to our everyday life as we never stop interacting with these software systems, such as the IoT software in smart home, smart city, agriculture, and medical treatment. All these applications, and many more, stand at the intersection of more research areas and will open the door to more interdisciplinary research. This is the point to which we see program analysis advancing in the next decades; this is the point which we will strive to explore in the future.

References

[S&P'22] Heqing Huang, Yiyuan Guo, **Qingkai Shi***, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P'22)*. IEEE, 2022.

[OOPSLA'21a] **Qingkai Shi**, Yongchao Wang, and Charles Zhang. Indexing Context-Sensitive Reachability. In *Proceedings of the 36th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'21)*. ACM, 2021.

[OOPSLA'21b] Peisen Yao, **Qingkai Shi***, Heqing Huang, and Charles Zhang. Program Analysis via Efficient Symbolic Abstraction. In *Proceedings of the 36th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'21)*. ACM, 2021.

[ESEC/FSE'21] Peisen Yao, Heqing Huang, Wensheng Tang, **Qingkai Shi**, Rongxin Wu, and Charles Zhang. Skeletal Approximation Enumeration for SMT Solver Testing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. pp 1141–1153. ACM, 2021.

[ISSTA'21] Peisen Yao, Heqing Huang, Wensheng Tang, **Qingkai Shi**, Rongxin Wu, and Charles Zhang. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. pp 322–335. ACM, 2021.

[PLDI'21] **Qingkai Shi**, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-Sensitive Sparse Analysis without Path Conditions. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*. pp 930–943. ACM, 2021.

[ISSTA'20a] Peisen Yao, **Qingkai Shi***, Heqing Huang, and Charles Zhang. Fast Bit-Vector Satisfiability. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. pp 38–50. ACM, 2020.

[ISSTA'20b] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and **Qingkai Shi**. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. pp 516–527. ACM, 2020.

[ISSTA'20c] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, **Qingkai Shi**, and Charles Zhang. Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. pp 463–474. ACM, 2020.

[ISSTA'20d] Yang Feng, **Qingkai Shi***, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. DeepGini: Prioritizing Massive Tests to Enhance the Robustness of Deep Neural Networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. pp 177–188. ACM, 2020.

[ICSE'20a] **Qingkai Shi** and Charles Zhang. Pipelining Bottom-up Data Flow Analysis. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*. pp 835–847. ACM, 2020.

[ICSE'20b] **Qingkai Shi**, Rongxin Wu, Gang Fan, and Charles Zhang. Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*. pp 812–823. ACM, 2020.

[S&P'20] Heqing Huang, Peisen Yao, Rongxin Wu, **Qingkai Shi***, and Charles Zhang. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. pp 1613-1627. IEEE, 2020.

[ICSE'19] Gang Fan, Rongxin Wu, **Qingkai Shi**, Xiao Xiao, Jinguo Zhou, and Charles Zhang. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19)*. pp 72–82. IEEE, 2019. **ACM SIGSOFT Distinguished Paper Award**.

[PLDI'18] **Qingkai Shi**, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. pp 693–706. ACM, 2018.

[TRel'16] **Qingkai Shi**, Zhenyu Chen, Chunrong Fang, Yang Feng, and Baowen Xu. Measuring the Diversity of a Test Set with Distance Entropy. In *IEEE Transactions on Reliability (TRel'16)*, Vol. 65, No. 1. pp 19-27. IEEE, 2016.

[TSE'16] **Qingkai Shi**, Jeff Huang, Zhenyu Chen, and Baowen Xu. Verifying Synchronization for Atomicity Violation Fixing. In *IEEE Transactions on Software Engineering (TSE'16)*, Vol. 42, No. 3. pp 280-296. IEEE, 2016.