I specialize in **software engineering**, with a particular focus on **program analysis** and **software testing**. Ensuring software correctness is critical, as software increasingly governs many aspects of modern life. However, traditional formal-semantics-based program analysis often struggles with scalability when addressing the complexity of modern software systems. At the same time, empirical methods like software testing, while practical, inevitably miss certain behaviors, leaving critical gaps in verification. The overarching objective of my research is to develop **scalable and reliable methodologies** that bridge these gaps. To achieve this, I employ **interdisciplinary statistical techniques to analyze dynamic information from program execution**, advancing the precision and robustness of software systems.

Over the past few decades, the widespread adoption of *program analysis* and software testing has become integral to ensuring the reliability and security of software applications. Conventional program analysis relies on formal semantics, which assigns rigorous mathematical meaning to the syntax of a programming language, to deduce a program's semantic features. However, formal semantics are limited in their ability to handle the heterogeneous features prevalent in modern software, such as network communication, system-level behavior, and third-party libraries, which are often beyond what formal semantics cover.

Software testing, which finds defects by actively executing the software, has gained notable attention since formally proving correctness is often unfeasible due to the vast program state. However, its reliability is challenged by its inherent incompleteness: there is always an unseen behavior in the software, and whether there is an undetected defect and if the testing process will find it, given the limited number of test cases, is unknown.

My research addresses the inherent limitations of program analysis and software testing fundamentally by reframing these tasks as *statistical problems* and solve them by employing the statistical methods to the dynamic information from the program execution. The primary advantage of *inferring* the program behavior statistically for program analysis is that they *operate irrespective of the system's complexity*, even in cases where the entire system is unknown, inaccessible, and/or undecidable. Statistical inference for software testing, focusing on the distribution of program executions in operational environments, provides *predictions* and *guarantees* for the software testing process in practice. In addressing the aforementioned limitations, I leverage a diverse range of statistical methods, including *causal inference*, *biostatistics*, and *machine learning*, drawn from fields such as *ecology*, *linguistics*, and *social sciences*. These methods are not only adapted but also customized to address the intricacies of software engineering.

I have pursued two problem-driven research directions addressing challenges in program analysis and software testing, along with a theoretical direction integrating interdisciplinary statistical methods into software engineering:

- **Counterfactual program analysis [1, 2, 3, 9, Section 1]**: I have developed statistical methodologies to infer program dependencies, determining which program elements affect others, by considering counterfactual events in sample program executions. Our work demonstrates that these statistical methods can identify dependencies not recognized by conventional program analysis.
- **Reasoning the unseen in software testing [4, 6, 7, 8, Section 2]**: I have developed statistical methodologies to estimate the likelihood of unseen events in software testing, providing reliable interpretations of testing results and predicting the future performance of the testing process. These methodologies outperform existing models based on formal semantics and statistical approaches that mishandle the unseen in practical scenarios.
- **Refining interdisciplinary statistical methods for SE [2, 4, 5, 6, Section 3]**: Statistical models from fields like social science and ecology offer valuable insights, but software's distinct traits—discreteness, determinism, and structural dependencies—require tailored approaches. To address these challenges, we develop specialized methodologies to enhance the applicability of statistical methods in software engineering.

Long-term Vision

Building on a foundation of scalable and reliable program analysis and software testing, my research integrates statistical methods with dynamic program information to address the challenges of modern software systems. **This is just the beginning: advancing statistical methodologies in software engineering promises to tackle enduring challenges and open transformative research avenues**. Modern software development generates vast amounts of data, and leveraging this data—particularly through machine learning—has become standard practice. Yet, other statistical methods, which offer complementary benefits, remain underexplored.

For example, my work on *unseen event estimation* [4, 5, 6, 7, 8] addresses data scarcity by estimating missing behaviors and improving test coverage. Similarly, *counterfactual causal analysis*[1, 2, 3, 9] provides explanation-based insights, bridging gaps where traditional machine learning methods fall short. Looking ahead, I aim to integrate advanced statistical approaches—such as *Bayesian modeling*, *extreme value theory*, and *robust inference*—into software engineering. These methods offer untapped potential for addressing scalability, uncertainty, and data sparsity, paving the way for a paradigm shift in how software is analyzed and tested to ensure reliability and efficiency.

The rest of the research statement will elaborate on the **research topics** I have been working on and the **short-term research plans for the next five years**.

1 Counterfactual Program Analysis

When undesired behaviors, such as bugs, arise in software, it is crucial to understand *how* and *why* these behaviors occur. Software operation is a sequence of interdependent instructions, and the software itself is a complex system composed of diverse elements for instructions, such as functions, statements, and variables. Identifying how these elements interact—i.e., *the dependency relations between program elements*—is fundamental to program analysis.

In my research, I have introduced a novel paradigm for dependency analysis, *observation-based dependency analysis* [3, 1, 2], which leverages statistical methods to infer dependencies between program elements. This approach employs *counterfactual reasoning*: if altering the value of program element B causes a change in the value of program element A, it is inferred that program element A depends on program element B.

Observation-based dependency analysis is purely data-driven and, thus, overcomes key limitations of traditional formal semantics-based methods. It avoids reliance on formal semantics, making it effective for features they cannot cover, reducing false positives from over-approximation, and addressing scalability challenges posed by undecidability. By leveraging statistical methods, this approach works even in heterogeneous systems or when parts of the system are unknown or inaccessible. These methods are *data-driven*, *empirically validated*, and provide a *quantifiable* approximation of dependencies, offering practical solutions to an inherently undecidable problem.

Following are the research works I have conducted in this direction.

Identifying the dependency. In our prior work, MOAD [1] successfully approximated program element dependencies using statistical methods by reframing dependency as the likelihood that one program element affects another. We developed an efficient sampling strategy to capture changes in program elements during intervened executions and applied statistical techniques to estimate these probabilities. Our evaluation showed that, for programs analyzable by conventional methods, MOAD achieved high accuracy in identifying elements influencing a target, outperforming traditional approaches. For programs beyond conventional analysis, MOAD uncovered hidden dependencies, such as those mediated through file I/O or network communication, which traditional methods fail to detect.

Quantifying the strength of the dependency. Not all dependencies are equal; A variable's value may exhibit sensitivity to changes in some variables (strong dependency), be rarely affected by others (weak dependency), or remain unresponsive to changes in the rest (no dependency). Reframing the program dependency as a sensitivity to changes can overcome the limitations of the undecidable nature of the program semantics and provide a nuanced understanding of the program operation. In our work, CPDA [2], we utilized causal inference to quantify the dependency strength solely from dynamic information. Our empirical findings revealed that fine-grained dependency information can group program elements into functional clusters, enhancing debugging productivity. Our Subsequent research [9] demonstrated the effectiveness of this information in software testing. Mutating pairs of program elements with strong dependencies are more likely to generate higher-order mutants that are challenging to detect with existing test cases.

2 Reasoning the Unseen in Software Testing

Software testing is fundamentally a *sampling process*, where test cases from the operational distribution—i.e., the distribution of program executions in practice—are executed to uncover defects. Given the vast space of possible test cases, exhaustively covering all program behaviors is infeasible. Consequently, software testing is inherently vulnerable to unseen behaviors, leading to two concrete challenges: 1) the interpretation of test results becomes unreliable, and 2) there is a never-ending concern about the sufficiency of the testing process.

Another research direction I have pursued focuses on developing a statistical inference model to address unseen behaviors in software testing. By quantifying the likelihood of unseen behaviors and incorporating it into test results, the model provides a *reliable interpretation of testing outcomes* [4, 7]. Additionally, it predicts the future performance of the testing process with statistical guarantees, enabling a rational assessment of testing effectiveness and serving as a decision-maker for resource allocation [8, 6]. These advancements enhance the *practicality of software testing.*

The following are the research works I have conducted in this direction.

Extrapolating the greybox fuzzing. Fuzzing, an automated software testing technique generating numerous test cases, is one of the industry's most widely adopted methods. Yet, little is known about how to determine the effectiveness of fuzzing, whether it will uncover new defects, or what its future performance will be. While blackbox fuzzing, with its consistent sampling distribution, is relatively predictable, greybox fuzzing introduces complexities due to adaptive bias, where the sampling distribution evolves based on test input execution coverage. In our work [8], we introduced a novel statistical model for predicting greybox fuzzing performance. Leveraging ecological statistics, our model forecasts future coverage increases in the stochastic process. To address adaptive bias, we partition the coverage record into sub-records, apply ecological statistics to each, and regress predicted coverage increases for extrapolating future performance. Our evaluation shows that this adaptive bias-aware model outperforms existing approaches, offering improved predictions for greybox fuzzing performance.

Reliable information leakage analysis. Information leakage analysis quantifies the information leaked from a secret source to a public sink during program execution. Existing statistical methods rely on mutual information estimation, which is sensitive to missing observations, often leading to either a significant bias or a false sense of security. In our work [7], we developed a novel mutual information estimator that addresses missing observations, enabling accurate and secure leakage estimation even with limited data. Our evaluation shows that our estimator outperforms existing methods in both accuracy and security, enhancing the reliability of information leakage analysis.

Estimating the reaching probability. Quantitative reachability analysis measures the probability of reaching a specific program state, such as an errornous state or a defect-inducing state, during execution. While conventional methods compute this probability through symbolic execution and model counting, we developed the statistical reachability analysis [4] that estimates the reaching probability from the sample program executions. In small-scale programs with known operational distribution, our approach outperformed conventional methods in both accuracy, due to the limited coverage of the formal semantics, and time cost. Our method also demonstrated its effectiveness in large-scale real-world software with an unknown operational distribution, as encountered in software fuzzing.

3 Refining Interdisciplinary Statistical Methods for SE

The software domain has unique characteristics that set it apart from fields where statistical methods are commonly applied, such as ecology, linguistics, and social sciences. Unlike these nature-based environments, the software domain features a discrete and deterministic nature, structural dependencies, and unnatural distributions. Recognizing these distinct traits, we refine statistical methods to improve their applicability in software engineering.

Program-specific characteristics. Programs exhibit a highly structured nature, with dependencies between program elements at the core of their operation. When applying statistical methods to software engineering tasks, we account for this structural aspect, refining the methods for more accurate estimations. For instance, in our reachability analysis [4], we proposed a structure-aware reaching probability estimator. Unlike existing statistical estimators (e.g., Laplace smoothing), which treat all unreached program elements equally, our estimator distinguishes them based on structural dependencies, assigning probabilities accordingly. Similarly, in residual risk analysis [6], we recognized the importance of structural dependency. Residual risk—the risk of a defect remaining after testing—is typically upper-bounded by the discovery probability of uncovered program elements. While existing methods assume independent coverage events, our structure-aware analysis incorporates structural dependencies, i.e., control-flow, providing a significantly tighter upper bound on residual risk.

Recognizing the deterministic nature of software enables more accurate identification of dependencies between program elements. In causal analysis, structure discovery identifies causal relationships from observational data. While existing methods are designed for probabilistic systems, we introduced a novel structure discovery method [2] tailored to deterministic dependencies in software. Specifically, if the value of program element A changes when B is manipulated, A is unequivocally dependent on B. Our method significantly improves causal structure accuracy for program dependency analysis compared to existing approaches.

Distribution-specific estimation. Many statistical methods propose a single general estimator for the unknown quantity of the underlying distribution, irrespective of its shape. While some well-known distributions, like Zipf's law, are common in natural environments, the software domain often exhibits heterogeneous and unnatural distributions. In such cases, a distribution-specific estimator may outperform the general estimator. Our recent work [5] contributes to a purely statistical domain, focusing on estimating the missing probability mass of a distribution. Combining statistical theory with optimization methods, specifically genetic algorithms, we introduce a 'distribution-free methodology' to discover a 'distribution-specific estimator' that surpasses the general estimator in performance.

4 Future Research Directions – Next Five Years Plan

Despite its success, statistical program analysis faces distinct challenges that are fundamentally different from the conventional program analysis. Those challenges includes, but not limited to,

C1. *Making sufficiently precise statements about properties of rarely executed components*: Most of the hard-to-find bugs lie in the rarely executed program components, and missing the behavior of those components can lead to a misprediction of the analysis result, which can be critical in various safety-critical scenarios.

C2. *Efficiently adjusting statistical program analysis in the presence of program evolution*: Re-executing the updated program is required in order to generate the new analysis result, which is time-consuming.

C3. *Adapting the statistical reasoning to the different domain/distribution*: For every empirical research, the domain shift is a critical issue. It occurs when the distribution of the observational data is different from the distribution of the data that the model is trained on, which can lead to a significant bias in the analysis result.

In the short term, I aim to address these challenges through specific research plans, some already in progress.

Modularized statistical program analysis (C1,2,3). I propose a new statistical program analysis that is *modularized* and *compositional* by design. The methodology involves generating observational data for each software component, inferring its statistical reachability model, and composing these models to analyze the entire system. By focusing on individual components, ample data is collected even for rarely observed ones, contributing to the final analysis result. The modularized approach simplifies adaptation to software changes; only updated components require re-analysis, enabling reuse of prior analysis. Additionally, observation and inference for each component can be parallelized, significantly reducing analysis time.

The main challenge is composing statistical models for each component to derive the final analysis for the entire software system. This involves adapting analysis from the independent domain of each component to the context-aware domain of whole program execution—a topic explored in the next research direction. This ongoing research recently secured funding from CASA–Cyber Security in the Age of Large-Scale Adversaries, with me as the sole PI.

Adapting to the different domain (C3). The domain shift is a prevalent issue in the software testing due to the varied execution environments and differences between in-house and production settings. Previously the machine-learning community has developed the domain adaptation method to address the domain shift, which I aim to adapt to the software domain. Two potential approaches are to be explored: 1) composing non-parametric models from each component using a covariate shifting method (e.g., importance sampling) and 2) designing parametric models for each component for the Markov chain model of the reachability.

I propose collaboration with industry partners, highlighting the paramount importance of this issue in practical applications. Many bugs occur in the software, even though it is tested during the development, due to the discrepancy between the development and production environment, and it could cause a significant loss in the industry. The industry's vested interest in overcoming domain shift challenges makes this research particularly appealing, as it aligns with their pressing concerns. Through this collaboration, we aim to validate the effectiveness of proposed solutions in real-world settings, offering practical insights to mitigate domain shift challenges within the industry.

Integration of static and statistical program analysis (C1). Static and dynamic analyses complement each other in program analysis. Static analysis scales effectively in systems with full semantic coverage and provides formal guarantees, while dynamic analysis addresses unknown or inaccessible systems through execution observation with statistical guarantees. Static analysis excels at capturing rare behaviors triggered by specific conditions (e.g., `if (val == 42)`), which are difficult to observe in arbitrary executions. I will explore integrating static and statistical analyses to combine their strengths: using statistical methods to infer the behavior of components not analyzable by static methods, expanding the analysis domain while maintaining both formal and statistical guarantees. The main challenge is identifying suitable components for each method and integrating their guarantees.

References

[1] Lee, S., Binkley, D., Feldt, R., Gold, N., and Yoo, S. Observation-based approximate dependency modeling and its use for program slicing. *Journal of Systems and Software 179* (Sept. 2021), 110988.

[2] Lee, S., Binkley, D., Feldt, R., Gold, N., and Yoo, S. Causal program dependence analysis. *Science of Computer Programming 240* (Feb. 2025), 103208.

[3] Lee, S., Binkley, D., Gold, N., Islam, S., Krinke, J., and Yoo, S. Evaluating lexical approximation of program dependence. *Journal of Systems and Software 160* (Feb. 2020), 110459.

[4] Lee, S., and Böhme, M. Statistical Reachability Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, Nov. 2023), ESEC/FSE 2023, Association for Computing Machinery, pp. 326–337.

[5] Lee, S., and Böhme, M. How Much is Unseen Depends Chiefly on Information About the Seen, Feb. 2024.

[6] Lee, S., and Böhme, M. Structure-aware Residual Risk Analysis, 2025.

[7] Lee, S., Minocha, S., and Böhme, M. Accounting for Missing Events in Statistical Information Leakage Analysis. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (New York, NY, USA, 2025), ICSE '25, Association for Computing Machinery, pp. 1–12.

[8] Liyanage, D., Lee, S., Tantithamthavorn, C., and Böhme, M. Extrapolating Coverage Rate in Greybox Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, Apr. 2024), ICSE '24, Association for Computing Machinery, pp. 1–12.

[9] Oh, S., Lee, S., and Yoo, S. Effectively Sampling Higher Order Mutants Using Causal Effect. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (Apr. 2021), pp. 19–24.