Miryung Kim: Research and Scholarship

My research focuses on software engineering (SE). I design automated software analysis and development tools to improve developer productivity and software correctness. I conduct user studies with professional software engineers and carry out statistical analysis to allow data-driven decisions for designing novel SE tools.

Full Professor (UCLA, 2019-current)

I have taken a leadership role in creating and defining the new area of Software Engineering for Data Analytics and Machine Learning (SE4DA and SE4ML). Based on my group's research, I was invited to give a keynote talk at ASE 2019, a technical briefing at ICSE 2020, and Distinguished Lectures at UIUC and UMN.

Automated Test Input Generation for Big Data and Machine Learning Modern big data applications pose new challenges in exhaustive, automatic testing due to the complexity of custom application logic and long execution latency. In **BigTest** (FSE 2019 / ICSE 2020 Demo), we designed a new symbolic execution testing approach to reduce test data size and testing time. Such test size reduction can enable local testing as opposed to running on the entire data using cloud computing resources. BigTest achieved size reduction by 10^5 to 10^8 orders of magnitude, while revealing 2X more manually-injected faults than the previous approach. In **BigFuzz** (ASE 2020, ICSE 2021 Demo), we re-invented fuzz testing to big data applications to reduce latency and to test application logic as opposed to framework code. BigFuzz performs automated source to source transformation to construct an equivalent DISC application suitable for fast test generation, speeding up testing time by 78X to 1477X compared to random fuzzing. On the front of **Deep Learning Testing** (FSE 2020, PerCom 2020), we investigated trade-offs between defect detection, naturalness, and output diversity and found that increasing neuron coverage leads to fewer defects detected, less natural inputs, and more biased class preferences. We also investigated applicability of adversarial attack algorithms for testing self-driving vehicles.

Software Debloating for Size Reduction and Security Modern software is bloated. Demand for new functionality has led developers to include more and more features, many of which become unneeded or unused. This phenomenon, known as software bloat, results in software consuming more resources and an unnecessary increase in attack surfaces. To meet US Navy's needs to debloat software systems for improved security, we produced an end-to-end Java bytecode debloating framework called JShrink (FSE 2020): it statically removes the body of uninvoked methods through call graph analysis and type dependency analysis, while accounting for dynamic language features to ensure safety.

Redesign Big Data Systems for Debuggability Performance and correctness debugging is difficult for big data applications. In **PerfDebug** (SoCC 2019), we designed a post-mortem performance debugging tool that automatically finds input records responsible for *latency abnormalities*. We demonstrated that remediation such as removing the single most expensive record or simple code rewrite suggested by PerfDebug can achieve up to 16X performance improvement. In **FlowDebug** (SoCc 2020), we re-designed fine-grained taint analysis to identify a precise set of input records by combining it with a statistical technique called *influence function* analysis. FlowDebug significantly improves debugging precision by up to 99.9 percentage points and avoids repetitive re-runs required for post-mortem analysis by a factor of 33.

Automated SE Tools for Heterogeneous Computing In HeteroRefactor (ICSE 2020), we designed a new dynamic invariant analysis and automated refactoring for a heterogeneous application development with FPGA. It combines FPGA-specific dynamic invariant detection, automated source transformation, and selective offloading. HeteroRefactor can significantly reduce manual coding effort and save FPGA BRAM and DSP resources by up to 83%. In HeteroFuzz (FSE 2021), we designed a new fuzz testing technique to identify behavior divergence between CPU and FPGA.

Associate Professor (UCLA, 2014-2019)

I assembled a cross-disciplinary research team (software engineering and database systems) to pioneer a new sub-field on SE tools for big data analytics. I conducted scientific, comprehensive studies of professional data scientists.

Studies of Professional Data Scientists I initiated academia and industry coalition to investigate the emerging role of data scientists. In collaboration with Microsoft Research, I conducted the first in-depth interview study of data scientists (ICSE 2016). We then conducted the largest scale, comprehensive study of almost 800 professional data scientists (TSE 2018). This quantification and sub-categorization of data scientists is important—although many companies are hiring data scientists and universities are creating new graduate programs, we lack scientific understandings of who data scientists are. My studies should inform how organizations should train this new workforce effectively and improve their productivity.

Automated Debugging for Big Data Analytics The current big data computing model lacks the kinds of debugging features found in traditional desktop computing, forcing data scientists to debug their applications by trial and error. To address these challenges, we designed a set of novel interactive debugging primitives for big data computing, called BigDebug (ICSE 2016). It emulates a breakpoint through incremental replay from the latest materialization point and streams selected program states on demand by leveraging dynamic guard compilation and deployment. BigDebug also enables a user to identify the origin of faulty outputs by re-architecting the underlying big data runtime with *data provenance* (VLDB 2015, VLDBJ 2017). Replay debugging can be optimized through *incremental computation* (SoCC 2016). To reduce the burden of manual debugging, we designed BigSift (SoCC 2017), a new automated debugging technique that combines delta debugging (DD) and data provenance (DP) with several key optimizations. BigSift improves fault localizability by $\sim 10^3$ to 10^7 and reduces the debugging time as much as $66\times$.

Mining, Visualizing, and Assessing Code Examples at Scale There is a growing interest in leveraging large collections of open-source repositories such as GitHub. The hypothesis here is that common application programming interface (API) usage inferred from a large code corpus may represent a desirable pattern that a programmer must follow. In ExampleCheck (ICSE 2018), we designed an automated API usage pattern inference technique that can generalize common API usage patterns in terms of data flow and control predicates from GitHub repositories and such inferred API usage patterns can prevent API misuse in StackOverflow. In ExampleStack (ICSE 2019), we designed another API usage adaptation mining technique that takes a code example from StackOverflow as input and automatically suggests how to modify the example by mining variations among similar clones found in GitHub repositories.

Despite this growing interest of mining source repositories, there is no easy way for a user to understand the commonalities and variances among a massive number of related code examples. In collaboration with UC Berkeley, we designed Examplore (CHI 2018) that summarizes hundreds of code examples in one synthetic code skeleton. This work lies between software engineering and human computer interaction to tackle the new frontier of mining software repositories research—*usability* and *information delivery.*

Coping with Code Duplication in Software Systems Code duplication created by copy and paste is common in large software. To help developers ensure that they applied similar changes to all relevant locations, we designed an interactive code search, called Critics (ICSE 2015). It allows a developer to interactively create an *abstract diff template* by parameterizing the edit content. By matching the template against the entire program, it finds clones and detects anomalies. We designed an automated code transplantation and differential testing technique to compare behavioral differences (ICSE 2017), called Grafter. We investigated a new automated clone removal refactoring approach that can handle *type variations*, *method variations*, *variable, expression variations*, *multiple variables to return*, and *non-local jump statements* (ICSE 2015), pushing the limit of duplication removal.

Assistant Professor (UT Austin, 2009-2014)

At UT Austin, a majority of my research focused on explicitly supporting *systematic changes*. This theme is based on the insight that high-level software modifications are often systematic, requiring similar but not identical changes to similar contexts.

Automating Systematic Changes Making similar changes to multiple locations is error prone and time consuming. Existing approaches are inadequate—refactoring engines support only pre-defined, semantics-preserving transformations. Source transformation tools require developers to prescribe edits in a formal syntax in advance. We developed a set of novel example-based program transformation techniques. Sydit (PLDI 2010) learns an abstract context-aware program transformation script from a change example and applies it to a user-selected target. We showed that Sydit produces code that is 96% similar to the manual editing done by programmers.

This Sydit project led to the development of LASE (ICSE 2013), which finds change locations automatically and applies systematic program edits. Our insight was to avoid over-generalization and over-specialization through the use of multiple examples. LASE was able to apply bug fixes to the locations missed by human developers, correcting errors of omissions. LASE showed the potential to automate transplantation of program edits to expedite complex software repair.

Real-World Refactoring It is widely believed that refactoring improves software quality and productivity. To create a scientific foundation that can help engineers to make data-driven refactoring decisions, I conducted statistical analysis of version histories at Microsoft. We first investigated the relationships between API-level refactorings and bug fixes using open source projects (ICSE 2011) and found that the location and timing of API refactoring is strongly correlated with bugs. At Microsoft, I quantified the impact of a multi-year Windows 7 re-architecting effort (FSE '12). This project was particularly important within Microsoft because despite the multi-year refactoring investment, there had not been a systematic quantitative validation of the refactoring impact on productivity and quality measurements. We analyzed version history data, conducted a survey of over 300 developers, and interviewed the architects and development leads to assess the impact of refactoring. We found that refactoring contributed to the reduction of inter-module dependencies and post-release defects, showing the tangible benefit of refactoring. We continued investigation of refactoring impact on code size, churn, complexity, test coverage, failure, and organization metrics (TSE '14). This research provided a sound basis to justify continued investment in large-scale refactoring in multi-billion dollar software industry.

Code Change Abstraction Techniques We invented a suite of analysis tools that can help programmers investigate code modifications (Vdiff ASE 2010, LibSync OOPSLA 2010, RefFinder ICSME 2010, Repertoire FSE 2011, FaultTracer ICSM 2011, and SPA ASE 2013) We studied recurring maintenance effort related to software forking—creating a variant product by copying an existing project. Our recent study of NetBSD, OpenBSD, and FreeBSD finds that 10 to 15% of all changes in the BSD family are actually recurring patches ported from peer projects (FSE 2012). We then investigated the types of porting errors in practice, and designed a tool to automatically detect copy and paste bugs and porting inconsistencies (ASE 2013).

We also developed RefFinder, a logic-query approach to refactoring reconstruction. Our insight was that the skeleton of refactoring edits can be expressed as a logical constraint. RefFinder automatically discovers the types and locations of refactoring edits and is highly extensible, because supporting a new refactoring type requires simply adding a new rule (ICSM 2010). We also invented a new differencing algorithm for Verilog (ASE 2010, ACM SIGSOFT Distinguished Paper Award) to help hardware design experts during code reviews.

PhD Research (University of Washington, 2003-2008)

My doctoral research investigated why and how systematic changes occur through a study of code duplication and introduced rule-based program differencing to summarize systematic changes.

Code Clone Genealogies and Copy and Paste Programming Practices I investigated how and why duplicated code is actually created and maintained using two empirical analyses. I used an edit capture and replay approach to gather insights into copy and paste programming practices. To extend this type of change-centric analysis to programs without edit logs, I developed a clone genealogy analysis that tracks individual clones over multiple versions. By focusing on how code clones actually evolve, I found that clones are not inherently bad and that we need better support for managing clones. I developed a clone genealogy extractor that automatically reconstructs the history of similar code fragments from source code repositories. My study of clone genealogies indicated that merging code clones is not always beneficial or even applicable to many clones for several reasons (ESEC/FSE 2005). These findings have contributed to diverting research focus from automatic clone detection to clone management support.

Logical Program Differencing A high-level software change is often systematic, consisting of many homogeneous transformations at a code level. By identifying such homogeneity, my approach discovers structures that explain how individual code changes are related to one another. To explicitly capture such structures, I developed novel rule-based change representations (first order logic rules) and built program differencing tools using my rule inference algorithms. In my Ph.D thesis, I built two different rule-based representations. First, I invented a rule-based change representation that groups similar API-level refactorings. I built an inference algorithm that assesses each rule's likelihood to find method-level matches with high precision and recall (ICSE 2007). To help programmers recognize structural differences from line-level textual differences, I developed LSdiff that automatically documents latent patterns within structural differences as logic rules (ICSE 2009).

Software Tools and Industry Impact

As a software engineering researcher, I went extra miles in packaging and hardening research technologies so that our tools could be adopted by software professionals. The following evidence attests to the fact that our research has made impact to industry and our tools scale to industry projects.

- PepperData tech-transferred and commercialized our debugging and data provenance techniques for Apache Spark, BigDebug (ICSE 2016) and Titian (VLDB 2016).
- Huawei tech-transferred our interactive clone search, Critics (ICSE 2015) for their multi-version Android platform maintenance. Critics was also used and evaluated at Salesforce.com.
- BigDebug and BigSift (ICSE 2016 and SoCC 2017) were used by Broad Institute for Genomics Research to debug big data analytics written in Apache Spark.
- JShrink was selected by Office of Naval Research for tech transfer Java bytecode debloating to US Navy systems. PJR Corporation is currently partnering with UCLA for tech transfer our Java debloating technologies.
- Grafter, our automated patch transplantation technique (ICSE 2017) was demonstrated to US Air Force for cyber-physical system repair.
- Our study of Windows refactoring (FSE 2012 and TSE 2014) is the first to quantify refactoring benefits using multi-year version histories and to assess the real world impact of re-architecting on code size, defect reduction, dependency, and organization structure.
- Our study (TSE 2018) is the largest scale study of professional data scientists and the first to quantify their working styles and challenges.
- Our tools such as RefFinder (ICSME 2010) and LASE (ICSE 2013) were used and extended by multiple research groups.

Student Advising and Faculty Placement

Over the past several years, my research group produced several tenure-track faculty members.

- Baishakhi Ray (PhD 2013) joined University of Virginia as a tenure track faculty in 2015 and later moved to Columbia University. Her dissertation focused on analysis of cross-system porting and porting errors in software projects.
- Na Meng (PhD 2014, co-advised by Kathryn McKinley) joined Virginia Tech as a tenure track faculty in 2015 and recently was promoted to an Associate Professor with Tenure in 2021. Her dissertation focused on automating program transformations based on example systematic edits.
- Tianyi Zhang (PhD 2019) joined Purdue University as a tenure track faculty member. His dissertation focused on leveraging inherent repetitiveness and redundancies embedded in large software systems to help programmers inspect, test, and assess code fragments during code reuse.
- Muhammad Ali Gulzar (PhD 2020) joined Virginia Tech as a tenure track faculty member. His dissertation focused on designing automated testing and debugging techniques for big data analytics.

Honors and Awards

- **Keynote Speaker (ASE 2019)** ASE is a top conference in software engineering. My keynote focused on how the development of big data, machine learning, and AI affects software correctness and developer productivity.
- **Most Influential Paper Award, 10-Year Retrospective Impact Award from ICSME 2010** I and my former students received ICSME's 10-Year Retrospective Most Influential Paper Award. Our paper published in 2010, presented an automated logic-query based refactoring inference approach.
- **Distinguished Lectures** My lectures were about the current trends in AI/ML systems and my group's work on automated debugging and testing data-intensive applications. I was one of 3 distinguished speakers at UIUC and one of 5 distinguished speakers at UMN in 2020-2021.
- **Professional Services** I am a Program Co-Chair of ESEC/FSE 2022, the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. I was the co-organizer of a Dagstuhl Seminar on SE4ML—Software Engineering for AI-ML based Systems. I was a Program Co-Chair of ICSME 2019, the 35th IEEE International Conference on Software Evolution and Maintenance. I was an Associate Editor of IEEE Transactions on Software Engineering.
- **Awards** I received an NSF CAREER award, a Microsoft Software Engineering Innovation Foundation Award, an IBM Jazz Innovation Award, a Google Faculty Research Award, and an Okawa Foundation Research Award, an ACM SIGSOFT Distinguished Paper Award and Humboldt Fellowship from Alexander von Humboldt Foundation.