

Mentoring and Supervising Students

Miryung Kim

Professor and Vice Chair of Graduate Studies, UCLA

Amazon Scholar, Amazon Web Services

Quick Intro: Miryung Kim

- SE for data intensive computing and heterogenous computing
- Emphasis on quality
- Industry studies on large scale re-architecting / data scientists
 - Microsoft Research
 - Amazon Web Services
- Keynotes at ASE / ISSTA & Distinguished lectures: CMU, UIUC, Max Planck Inst, UMN, UC Irvine, UC Riverside



Code Mining, Debugging
and Refactoring for Java



Data-driven insights
industry collaboration



What are your products (or deliverables)?

Philosophy



“Focus on the students, since graduating great students means you’ll produce great research, while focusing on the research may or may not produce great students.”

I am inspired by my adviser, [David Notkin](#)'s philosophy about working with students, which he inherited from his academic father, Nico Habermann.

Nico Habermann

David Notkin

Bill Griswold (UC San Diego)

Kevin Sullivan (U Virginia)

Gail Murphy (UBC)

Michael Ernst (U
Washington)

Jonathan Aldrich (CMU)

Vibha Sazawal

Tao Xie (Peking)

Miryung Kim (UCLA) and
many more

Baishakhi Ray (PhD 2013, Associate Prof @ **Columbia**)

Na Meng (PhD 2014, Associate Prof @ **Virginia Tech**)

Myungkyu Song (Postdoc 2016, Associate Prof @ U
Nebraska Omaha)

Tianyi Zhang (PhD 2019, Asst Prof @ **Purdue**)

Muhammad Ali Gulzar (PhD 2020, Asst Prof @ **Virginia
Tech**)

Jason Teoh (PhD 2021, Twitter => Databricks)

Qian Zhang (Postdoc 2022, Asst Prof @ **UC Riverside**)

Hong Jin Kang (Postdoc 2024, Asst Prof @ **U Sydney**
to start in Fall 2024)

Nurturing Next Generation of Leaders



**ACM SIGSOFT Influential Educator
Award 2022**

- Multiply impact through people
- Gain trust and respect
- Seek out candid feedback

What are Dos and Don'ts?

Hiring, Admissions & Departing

Do

- Help out other students / serve on communities
- Mix co-advise and sole-advise
- Screen early and often
- Get involved in admissions committee
- Be cool with rejections & switching advisors, etc.
- Create a comfortable environment for students to **disagree with you**
- Let advisees make their own career decisions

Don't

- Be anxious
- Hire too many at once
- Spend start up too fast or too slow
- Influence and persuade too much

Research Advising

Do

- Schedule regular meetings
- Provide feedback repetitively (in the order of X times)
- Get senior students get involved in grant writing
- Pair up senior / juniors and create sub teams

Don't

- Cancel regular meetings for paper deadlines, grant deadlines, etc.
- Write introduction, conclusion, motivating examples and table skeletons and request students to fill in results
- Rewrite papers (if you must, do it sparingly)

Build Team and its Culture

- Reading group
- Stop by and work in the lab
- Scrum
- Lunch & team building
- Conference trips
- Reward & recognize awards, papers, etc.
- Demos
- In-person vs. zoom
- Respect boundaries: *No slack, text, and email on weekends, nights, etc. (If you do, write no need to respond at the end)*

Frequent, persistent written feedback

②

found that the optimization library they compared against [7] produced incorrect results. Unlike classical programs, quantum programs have many quantum-specific pragmas and framework APIs. For example, in Pyquil, quantum application developers can change the compilation style to `Pragma Native/Greedy/Partial`. Many bugs and unexpected behaviours may arise from these pragmas.

An appealing solution is to apply rigorous formal verification to check that a compiler does what it intends to. However, the applicability of formal verification on compilers is limited. Firstly, unlike the classical compiler that a bit only has two states 0 and 1, a single qubit has infinite states, which leads to state explosion. Most of the work restricts their targets or optimization like rotation merging [27], which limits their scope. Secondly, lots of errors are reported besides compilation and optimizations. For example, there are issues that report errors of Pyquil due to the wrong initialization of the quantum virtual machine [4], and another issue reports an error of Cirq when user sets synthesis options to a certain value [5]. Only checking compilation is not enough to detect errors in the quantum hardware.

We present QDrr, a general and comprehensive testing framework for quantum frameworks. QDrr takes a quantum program and a quantum framework as input, and reports quantum errors or unexpected behaviours and corresponding quantum programs that trigger the errors. Our key insight is three-fold:

1. First, to fix the state explosion and expand the scope, we apply a fuzzing log combined with differential executions to test the quantum frameworks. This is based on the insights that in classical computing, differential testing and fuzzing are used as an alternative way to the perfect testing.
2. Second, to reduce the state explosion, we use a quantum program variation generation based on quantum pragmas, QDrr generates and executes equivalent programs. Then QDrr compares the results of the execution to check if there are any unexpected behaviours.
3. Third, since the results of multiple executions are distributions, we use the Kolmogorov-Smirnov test (K-S test) to check the equivalence of program results. In statistics, the K-S test is a nonparametric test of the equality of continuous [13].

For what purpose? Also follow American not British singular "check your grammar." also follow American not British spelling behavior not behaviour

ICSE 2024, April 2024, Lisbon, Portugal

Algorithm 1 A pattern is mined to separate positives \mathcal{P} from negatives N . Patterns containing the user's suggested code lines are favored.

Require:

- \mathcal{P} ← positive instances
- N ← negative instances
- \mathcal{R} ← all instances
- C ← code lines suggested by the user
- S ← maximum pattern size to be considered

```
1: function SURF_PATTERN
2:    $D \leftarrow ()$ 
3:   for  $s \in \text{enumerateSubgraphs}(\mathcal{R}, S)$  do
4:     if  $\text{match}(s, \mathcal{P}) \geq \text{match}(s, N)$  then
5:        $D \leftarrow D \cup s$ 
6:   end if
7: end for
8:  $G_P \leftarrow \text{sort}(D, \text{compareBy}(\text{containsCodeLine}(C)))$ 
9:  $G_N \leftarrow \text{sort}(\text{compareBy}(\text{discriminative}(\mathcal{R}))$ 
10:  $\text{then}(\text{compareBy}(\text{match}(\text{Population}(\mathcal{A}))))$ 
11: return filterSubgraphsThatSeparatePosAndNeg( $D, S$ )
12: end function
```

3.2 Importance metrics

Inspired by active learning techniques [18], we guide users toward informally and representatively wide lines.

Support For each code line, SURF counts the support of the pattern if the code line were included, e.g., a code line with a reported support of 10 in SURF means that the code line appears in 10 lines in the population. Support is computed over the entire population, ignoring their labels. While not all frequent patterns are useful [15, 21], infrequent code lines are not useful.

Information Gain We use information gain to measure how well a pattern separates the positive and negative instances after including the code line. Including a code line is analogous to splitting the data at a decision node in a decision tree. The instances matched by the original pattern are partitioned into two sets, one set of instances that match the new pattern, and one set of instances that do not. First, we compute the entropy of the three sets:

- G_P : positive and negative instances matched by the pattern.
- G_N : positive and negative instances matched after the pattern is excluded.
- G_C : positive and negative instances excluded after the pattern is updated.

Then, for each group G , entropy is computed using the proportion of positive instances (p_p) and negative instances (p_n):

$$\text{Entropy}(G) = -p_p \log_2(p_p) - p_n \log_2(p_n)$$

The information gain of including a code line is as follows:

$$\text{Entropy}(G_P) - \left(\frac{|G_N|}{|G_P|} \times \text{Entropy}(G_N) + \frac{|G_C|}{|G_P|} \times \text{Entropy}(G_C) \right)$$

If a pattern initially matches one positive and one negative instances, then $\text{Entropy}(G_P)$ is 1. Say the pattern is modified to include a code line, to not match the negative instance, then G_C contains one positive instance and zero negative instances, and G_N contains zero positive instances and one negative instance. Both $\text{Entropy}(G_N)$ and $\text{Entropy}(G_C)$ are 0. The information gain associated with the code line is $1 - (0 + 0)$, as the code line has successfully separated every pair of positive and negative instances.

EVALUATION DESIGN

We conducted a within-subject user study to evaluate SURF. We aim to answer the following research questions:

- (?) RQ1. Does SURF improve the participant's ability to comprehend the API usage distribution?
- (?) RQ2. How much effort reduction does SURF provide in inferring code patterns?
- (?) RQ3. What features in SURF do the participants perceive to be useful?

To answer the RQs, we created two case studies of programs using cryptographic APIs [49]. We analyze cryptographic APIs as many prior studies have demonstrated that fully automatic API usage mining methods do not succeed in capturing the desired patterns [11]. Most usages of the APIs are incorrect, limiting the effectiveness of frequency-based pattern mining techniques. The users must examine and refine the inferred patterns directly to use a concrete set of notation → so that can easily membership.

Or use a concrete set notation → so that can easily membership.

lep = 5 (10 = 2 + 0) N = 5 (1) see
com = 4 (10 = 2 + 0) N = 5 (1)

Also strong hint. Is there a citation? clean what term like

③

These 23 instances are... how long you run first, etc. Experimental setting. A concrete example. How you selected these 23 instances or manually. How you selected these 23 instances or manually. How you selected these 23 instances or manually.

within func. new would respect the required number of arguments for each operation. Three arguments for `add`.

Motivated by this observation that fast evolving languages and IRs would need flexible and adaptable custom input mutations, we propose a new approach, called SYNTRIFuzz, that synthesizes custom mutations. SYNTRIFuzz constructs parameterized mutation templates from existing test cases and instantiates context-dependent concrete mutations on the fly. The key novelty of SYNTRIFuzz is that its mutation-synthesis capability is different from simply recombining existing seed inputs. SYNTRIFuzz decomposes existing test cases into into parameterized mutation templates, where mutation context and mutation operations are enumerated on the fly. Then during the fuzzing loop, it matches the target context against the mutation context and instantiates concrete mutations in terms of parse tree edits. Its edits are parameterized edits, where the content of mutations (`arg0` and `arg1`) are parameterized with respect to argument names or operation names, making them reusable.

Our evaluation on 4 MLIR-based compiler projects, `MLIR`, `X`, `Y`, `Z` parameterized mutation templates from 212 X, Y, Z test cases for each project respectively. We find that on average 26% of mutants generated by SYNTRIFuzz show significant changes over baseline grammar-based fuzzing. 39% of those produced by SYNTRIFuzz successfully compile compared to 37% of mutants produced by our baseline, demonstrating SYNTRIFuzz's capability to synthesize complex edits. On average, each mutation template encodes 317 parameterized variations, showing SYNTRIFuzz's potential to obviate the need for defining different custom mutations individually by hand, demonstrating the versatility of parameterized mutation templates. In terms of overall cumulative coverage, SYNTRIFuzz achieves an average of 0.3% higher cumulative branch coverage over grammar-based fuzzing. In terms of fault detection potential, SYNTRIFuzz can detect 554 out of 778 injected faults by synthesizing custom mutations. Without custom mutations, only 542 out of 778 faults could be found. Our qualitative investigation shows SYNTRIFuzz is capable of synthesizing test cases that cannot be found by other fuzzers without specialized mutations.

In summary, our contributions are:

- (1) We design a novel compiler fuzzing technique that obviates the needs of defining custom mutations a priori, which is impractical when the target language is highly extensible and quickly evolving.
- (2) Our method automatically synthesizes and applies multi-edit, dependence aware, custom mutations on the fly. The key enabler is constructing parameterized mutation templates from code examples and adapting edits to its context.
- (3) We show that our method has similar overall performance to grammar-based fuzzing, but is also capable of detecting faults that cannot be efficiently found by other means.

2.1 BACKGROUND

MLIR (Multi-Level Intermediate Representation) is a modular compiler framework that differs from traditional approaches by enabling developers to extend the intermediate representation. Rather than defining a single monolithic IR with a fixed set of types and operations, MLIR is extensible by design. The core abstraction in MLIR is a dialect. Each MLIR dialect consists of a set of operations, types, and attributes which collectively define a domain specific representation. However, this also presents a challenge for program generation, since dialect operations vary greatly in terms of their semantics. As shown in Listing 77, both the enclosing "func. func" structure and the "tf. Add" operation are considered MLIR operations; and thus follow the same syntactic rules. Without further specialization of an MLIR grammar, existing fuzzer would not distinguish between

Why do you call this mutant? mutant like mutant fault. do you mean generated in part 5.

What is the same syntactic rule. explain.

Adjustment according to PhD Career Stage

Year 1-2	Year 3-4	Year 5-6
Confidence Pair-up Submission experience	Exploration Ownership Idea Formation Independence Industry internship	More Independence Mentoring Help with grant writing

What about other aspects of mentoring?

Vice Chair of Graduate Studies at UCLA

graduate curriculum

phd progress tracking

graduate student orientation

aspect of administration

computer science

award selection process

graduate education level

enrollment management

graduate admission

graduate orientation

- Establish clear process
- Consistent and fair
- Empower staff teams
- Navigate HR challenges

Faculty in Residence

- 8 years on-campus residential life mentor

- Noticed students' desire for CS from other depts
- Sensed students' anxiety about finance

- Learn to work with influence
- After 5pm student experiences
- Cooperate with campus units





Academic Support

Professor Miryung Kim, Computer Science, UCLA

“Reflecting Faculty in Residence”

Less Serious Version

- Miryung has degrees in Computer Science. She did well in school. She saw her adviser enjoying mentoring students. So she became a Professor, and she has been going to school for quite some time now. She likes learning something new. In particular, she gets inspiration from what people do in industry. She enjoys helping students, guiding students to graduation and working with others. Seeing students grow makes her proud and her job rewarding. Though she tried to be helpful, some students did not work out for her. Everyone is different and unique. Also each and everyone also changes over time. She is learning how to work with different people.