ICST2024 Most Influential Paper Award

Ask the mutants: Mutating Faulty Programs for Fault Localization

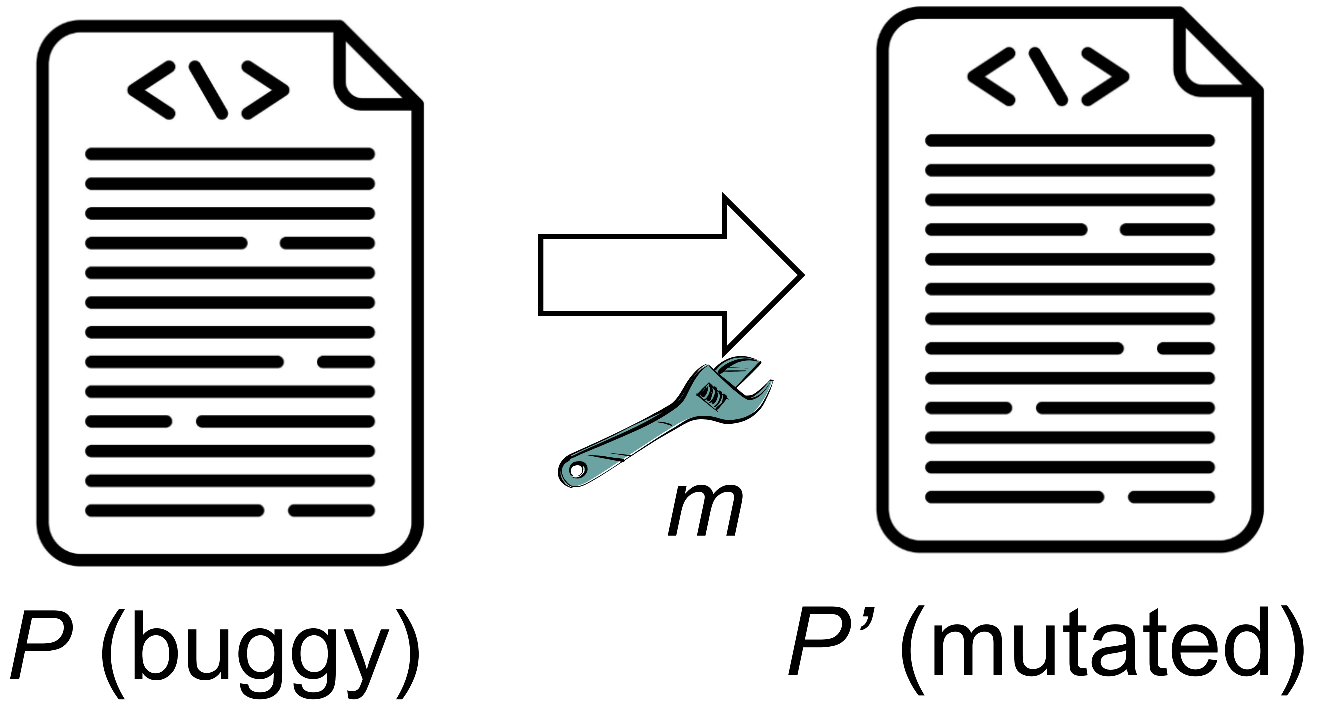
Seokhyeon Moon
SAMSUNG SDS

Yunho Kim
HANYANG UNIVERSITY 한양 1939

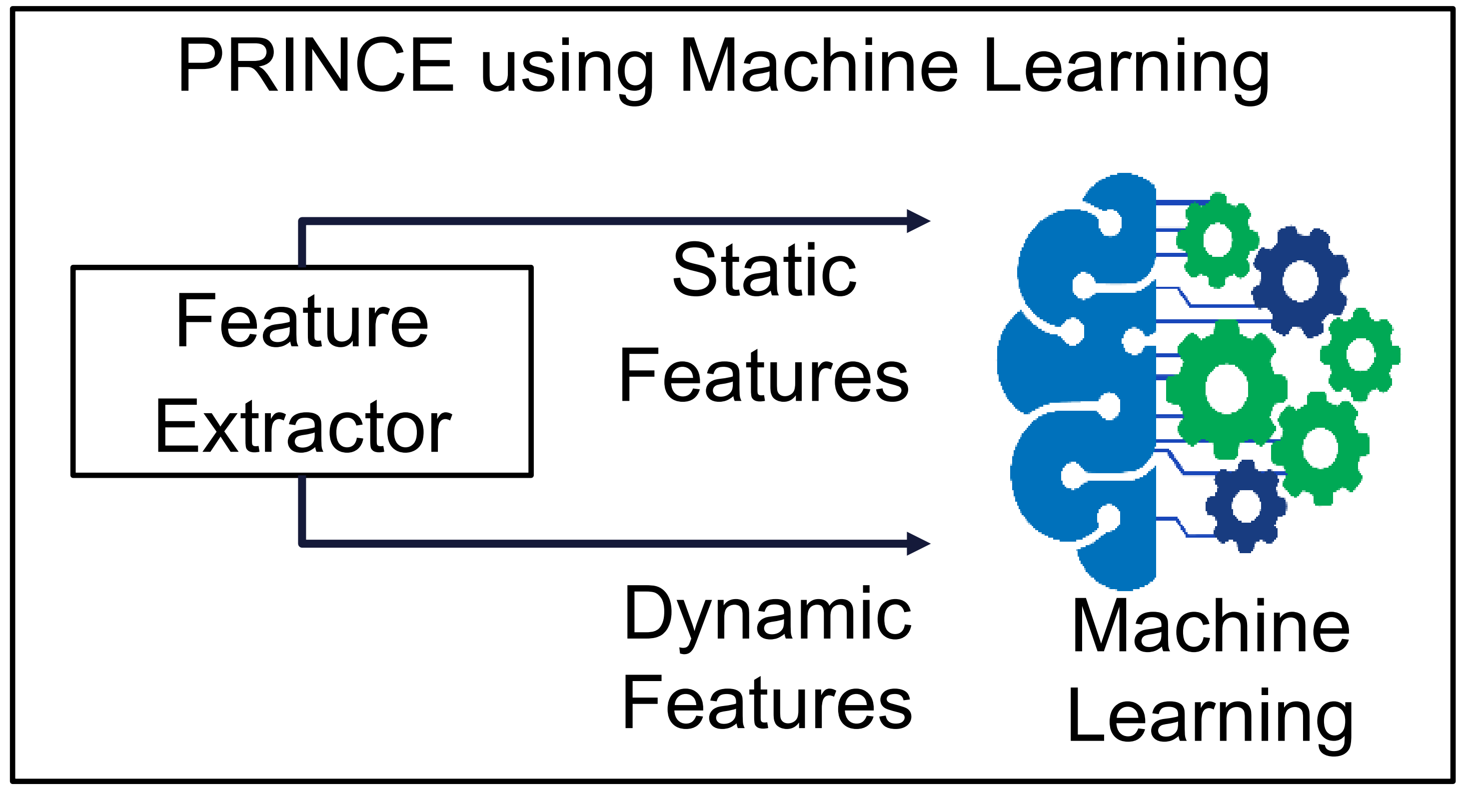
Moonzoo Kim and Shin Yoo
KAIST

Our Journey Over the Past 10 Years

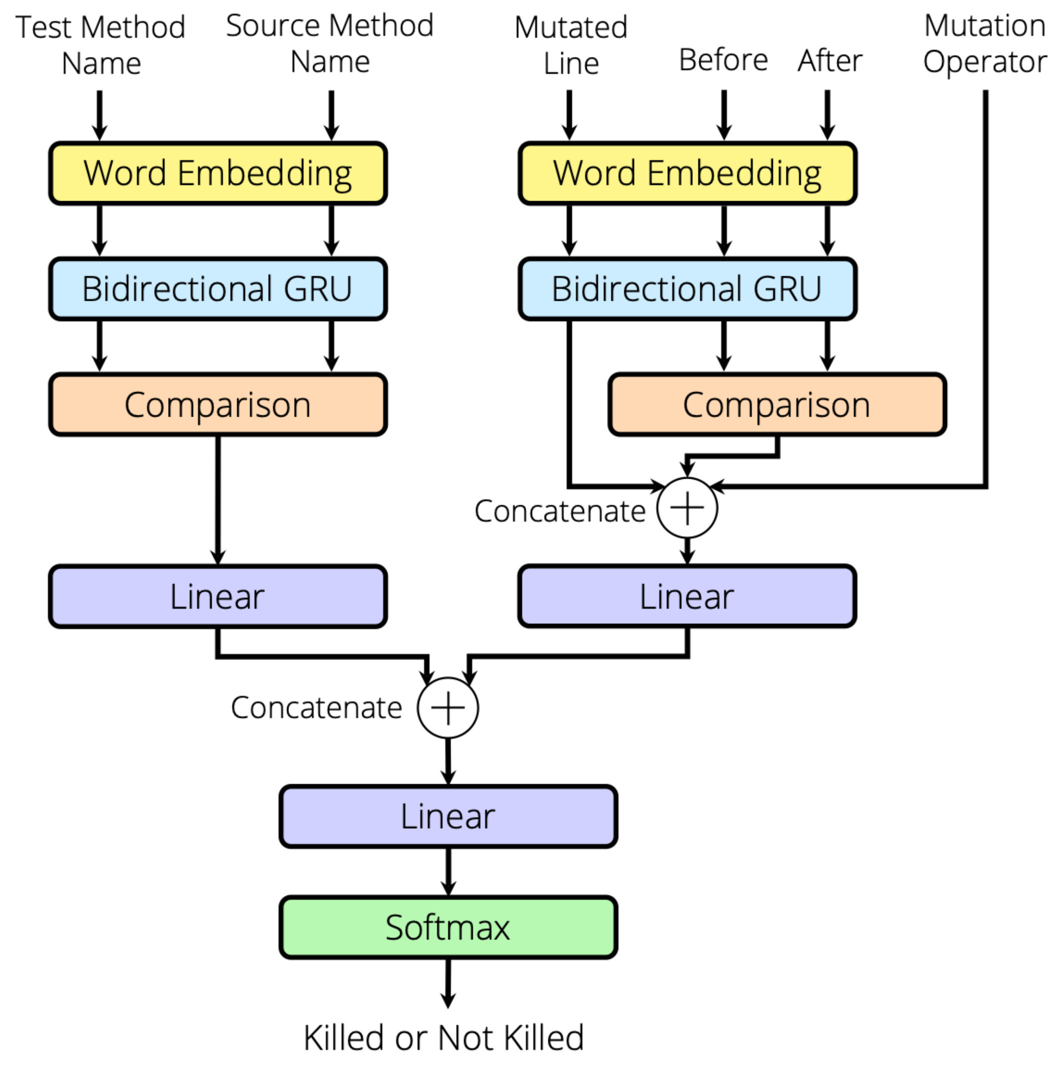
Birth of MUSE



Application of MUSE



What came after



Back in 2012…

Fault localization (FL) emerged as an active research topic after I joined SWTV group

FL in 2012: The Era of SBFL

SBFL and its improvements (and we did that too)

Visualization of Test Information to Assist Fault Localization

James A. Jones, Mary Jean Harrold, John Stasko
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{jjones,harrold,stasko}@cc.gatech.edu

ABSTRACT

One of the most expensive and time-consuming components of the debugging process is locating the errors or faults. To locate faults, developers must identify statements involved in failures and select suspicious statements that might contain faults. This paper presents a new technique that uses visualization to assist with these tasks. The technique uses color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases. Based on this visual mapping, a user can inspect the statements in the program, identify statements involved in failures, and locate potentially faulty statements. The paper also describes a prototype tool that implements our technique along with a set of empirical studies that use the tool for evaluation of the technique. The empirical studies show that, for the subject we studied, the technique can be effective in helping a user locate faults in a program.

Keywords

Software visualization, fault localization, debugging, testing

1. INTRODUCTION

Attempts to reduce the number of delivered faults[1] in software are estimated to consume 50% to 80% of the development and maintenance effort [4]. Among the tasks required to reduce the number of delivered faults, debugging is one of the most time-consuming [3, 15], and locating the errors

Pan and Spafford analyzed the debugging process and observed that developers consistently perform four tasks when attempting to locate the errors in a program: (1) identify statements involved in failures; (2) select suspicious statements that might contain faults; (3) hypothesize about suspicious faults; and (4) restore program variables to a specific state [10, page 2]. Our work addresses the second task—selecting suspicious statements that may contain the fault. To identify suspicious statements, programmers typically use debugging tools to manually trace the program, with a particular input, encounter a point of failure, and then backtrack to find related entities and potential causes.

There are a number of ways, however, that this approach can be improved. First, the manual process of identifying the locations of the faults can be very time consuming. A technique that can automate, or partially automate, the process can provide significant savings. Second, tools based on this approach lead developers to concentrate their attention locally instead of providing a global view of the software. An approach that provides a developer with a global view of the software, while still giving access to the local view, can provide more useful information. Third, the tools use results of only one execution of the program instead of using information provided by many executions of the program. A tool that provides information about many executions of the program can help the developer understand more complex relationships in the system. However, with large programs and large test suites, the huge ... such an approach, if reported ... difficult to interpret.

2002

FIESTA: Effective Fault Localization to Mitigate the Negative Effect of Coincidentally Correct Tests

Seokhyeon Mun, Yunho Kim, Moonzoo Kim*
CS Dept. KAIST, South Korea

Abstract

One of the obstacles for precise coverage-based fault localization (CFL) is th existence of Coincidentally Correct Test cases (CCTs), which are the test case that pass despite executing a faulty statement. To mitigate the negative effect of CCTs, we have proposed Fault-weIght and atomizEd condition baSed local izaTion Algorithm (FIESTA) that transforms a target program with compoun conditional statements to a semantically equivalent one with atomic condition to reduce the number of CCTs. In addition, FIESTA utilizes a fault weight o a test case t that indicates "likelihood" of t to execute a faulty statement. W have evaluated the effectiveness of the transformation technique and the faul weight metric through a series of experiments on 12 programs including five non trivial real-world programs. Through the experiments, we have demonstrate that FIESTA is more precise than Tarantula, Ochiai, and Op2 for the targe programs on average. Furthermore, the transformation techniq ...

2012

FL in 2012: Criticism on FL

Inaccurate FL, unrealistic scenario

- Not accurate enough
  - Top 1%: 1,000 / 100,000 lines

- Unrealistic FL assumption
  - Developers follow the ranked list (They will only if the list is very short)

Are Automated Debugging Techniques Actually Helping Programmers?

Chris Parnin and Alessandro Orso
Georgia Institute of Technology
College of Computing
{chris.parnin|orso}@gatech.edu

ABSTRACT

Debugging is notoriously difficult and extremely time consuming. Researchers have therefore invested a considerable amount of effort in developing automated techniques and tools for supporting various debugging tasks. Although potentially useful, most of these techniques have yet to demonstrate their practical effectiveness. One common limitation of existing approaches, for instance, is their reliance on a set of strong assumptions on how developers behave when debugging (*e.g.*, the fact that examining a faulty statement in isolation is enough for a developer to understand and fix the corresponding bug). In more general terms, most existing techniques just focus on selecting subsets of potentially faulty statements and ranking them according to some criterion. By doing so, they ignore the fact that understanding the root cause of a failure typically involves complex activities, such as navigating program dependencies and rerunning the program with different inputs. The overall goal of this research is to investigate how developers use and benefit from automated debugging tools through a set of human studies. As a first step in this direction, we perform a preliminary study on a set of developers by providing them with an automated debugging tool and two tasks to be performed with and without the tool. Our results provide initial evidence that several assumptions made by automated debugging techniques do not hold in practice. Through an analysis of the results, we also provide insights on potential directions for future work in the area of automated debugging.

second activity, *fault understanding*, involves understanding the root cause of the failure. Finally, *fault correction* is determining how to modify the code to remove such root cause. Fault localization, understanding, and correction are referred to collectively with the term *debugging*.

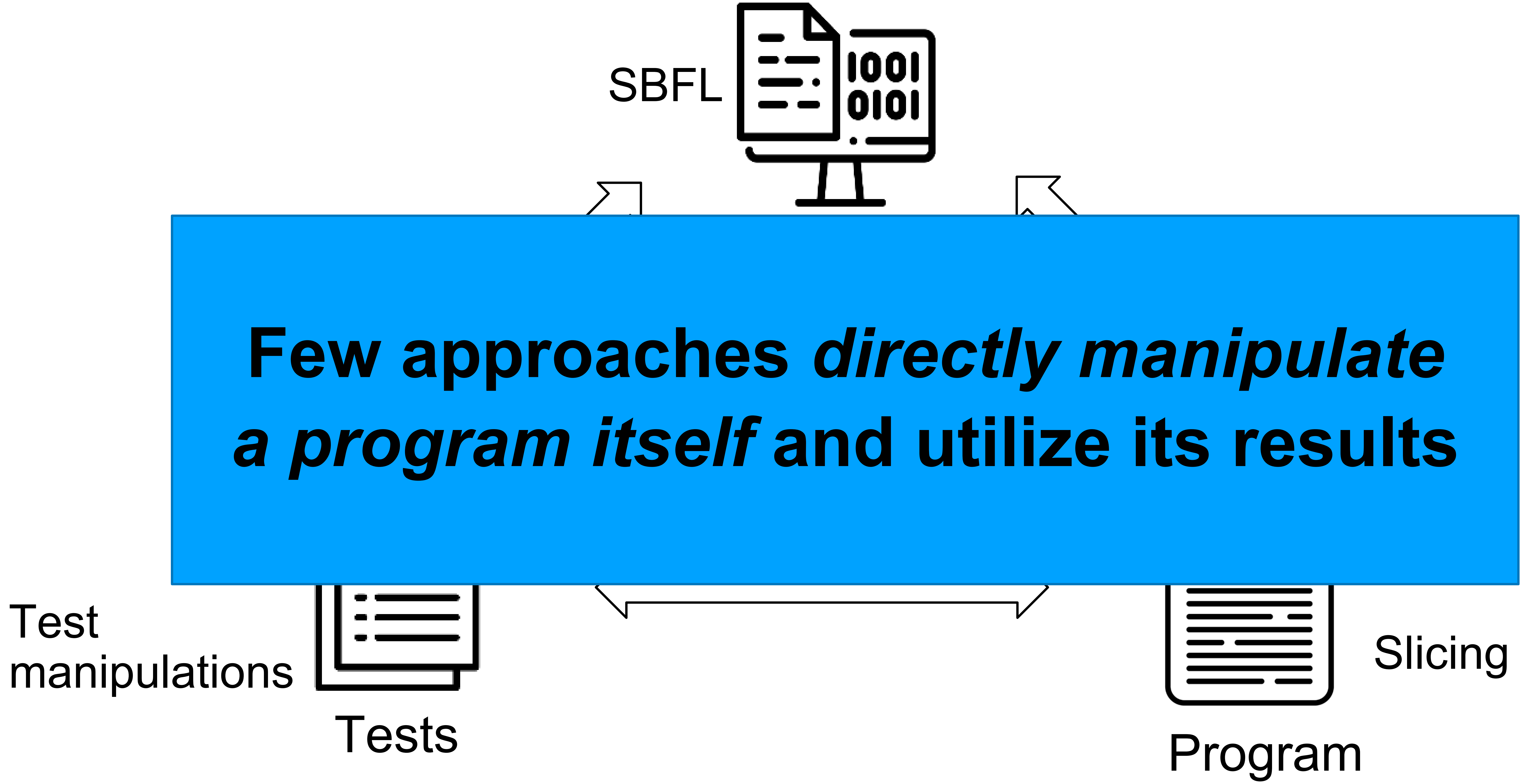
Debugging is often a frustrating and time-consuming experience that can be responsible for a significant part of the cost of software maintenance [25]. This is especially true for today's software, whose complexity, configurability, portability, and dynamism exacerbate debugging challenges. For this reason, the idea of reducing the costs of debugging tasks through techniques that can improve efficiency and effectiveness of such tasks is ever compelling. In fact, in the last few years, there has been a great number of research techniques that support automating or semi-automating several debugging activities (*e.g.*, [1, 3, 8, 11, 21, 29–31]). Collectively, these techniques have pushed forward the state of the art in debugging. However, there are several challenges in scaling and transitioning these techniques that must be addressed before the techniques are placed in the hands of developers.

In particular, one common issue with most existing approaches is that they tend to assume *perfect bug understanding*, that is, they assume that simply examining a faulty statement in isolation is always enough for a developer to detect, understand, and correct the corresponding bug. This simplistic view of the debugging process can be compelling, as it allows for collecting some obje
effectiveness of a debugging techniq
mon ground for comparing alternati

2011

FL in 2012: Focused on Passively Observable Results

Execute a program and utilize the execution results



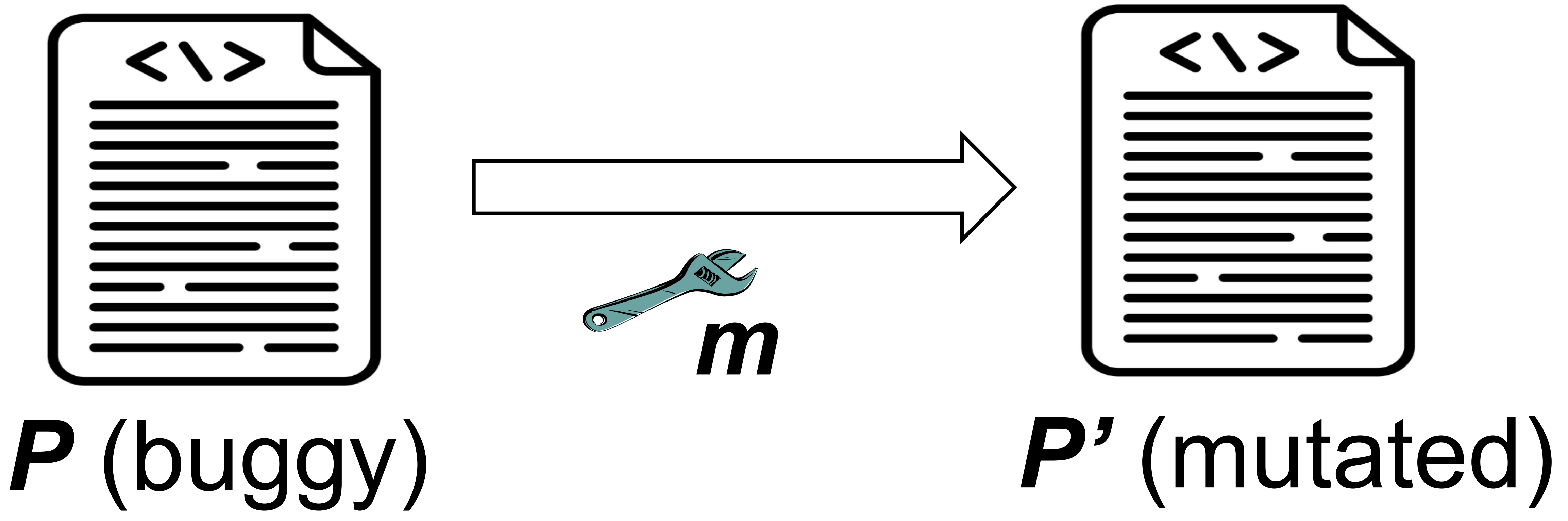
Few approaches *directly manipulate a program itself* and utilize its results

MUSE Main Idea

Mutating programs can give us hints

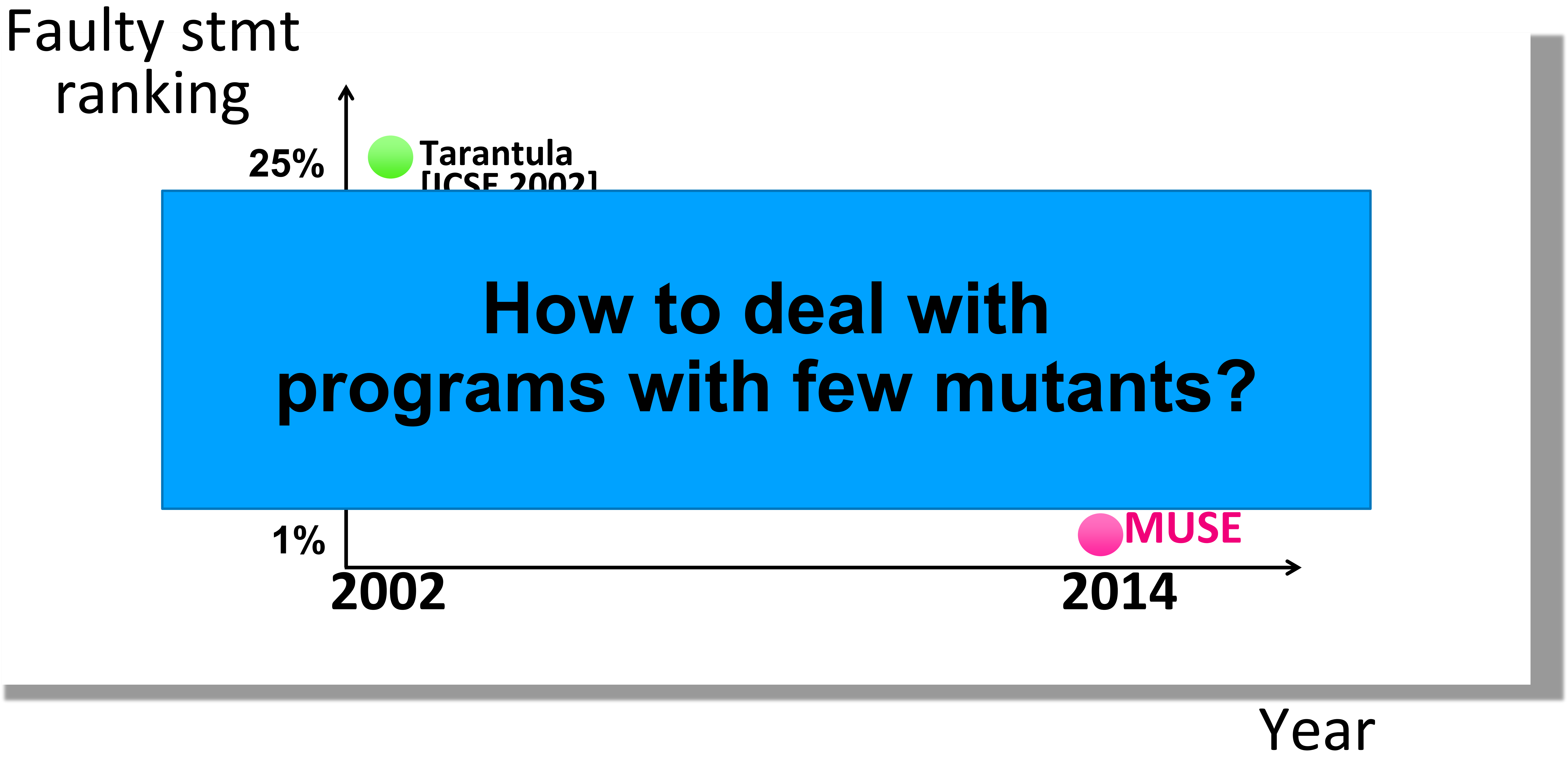
- (1) if ***m*** destroys ***P***
  - passed tests → failed tests↑
  - ‘destroy’ likely at correct statements

- (2) if ***m*** (partially) fixes ***P***
  - failed tests → passed tests↑
  - ‘fix’ likely at faulty statements



MUSE Result

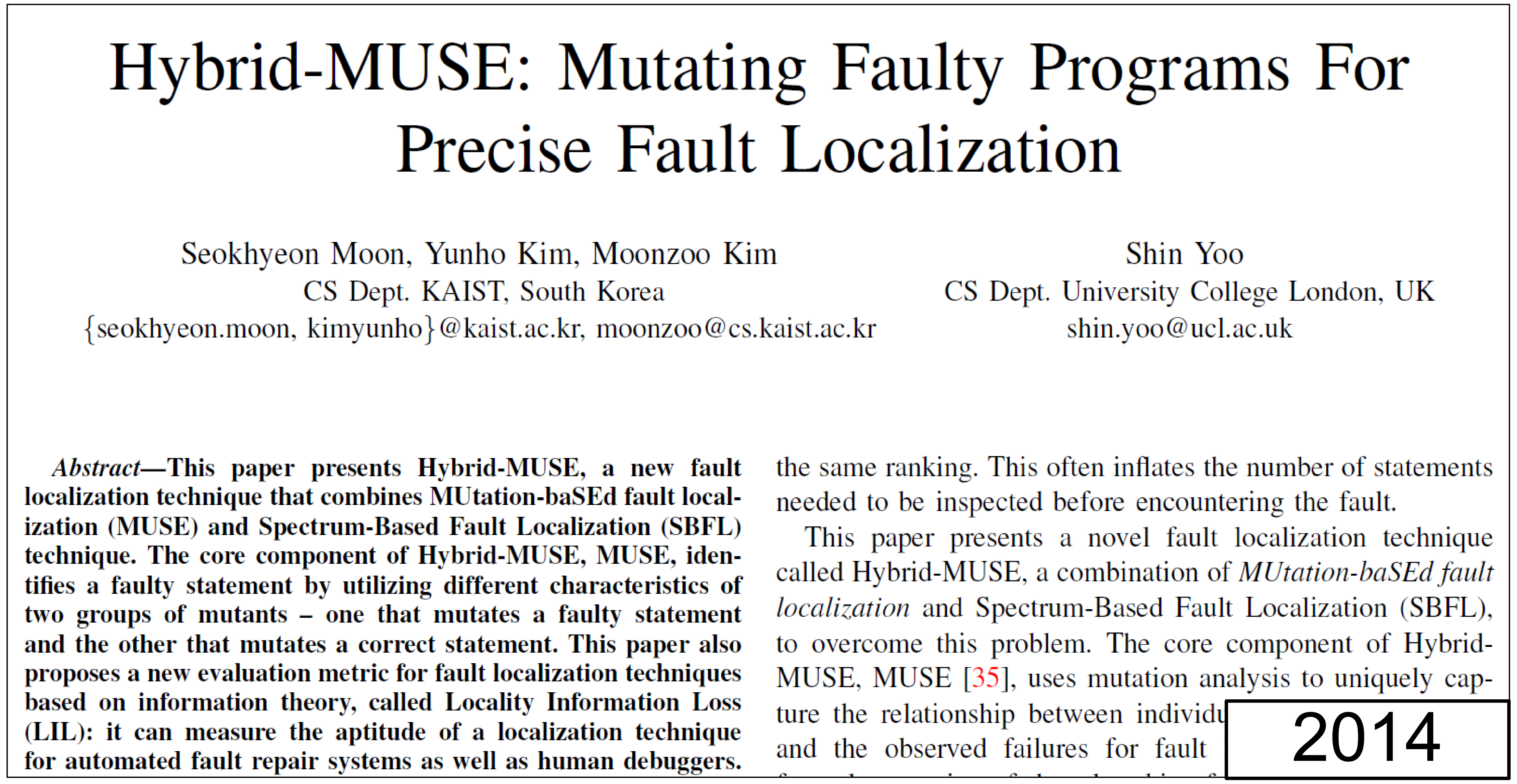
Ouperforms Op2 which is theoretically proven to be optimal



How to deal with programs with few mutants?

HybridMUSE: Combine MUSE with SBFL

SBFL assists MUSE for programs with few mutants



Hybrid-MUSE: Mutating Faulty Programs For Precise Fault Localization

Seokhyeon Moon, Yunho Kim, Moonzoo Kim
CS Dept. KAIST, South Korea
{seokhyeon.moon, kimyunho}@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

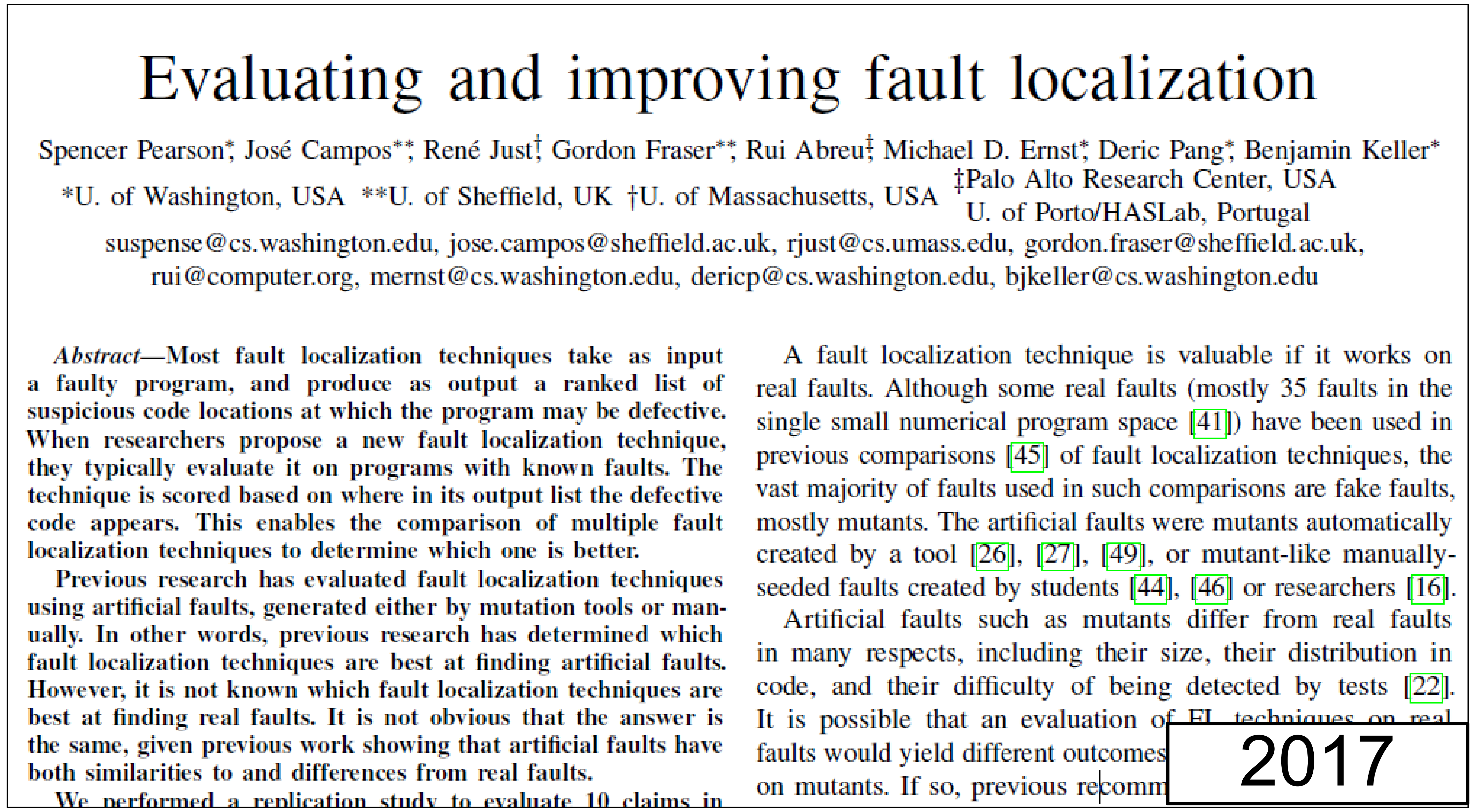
Shin Yoo
CS Dept. University College London, UK
shin.yoo@ucl.ac.uk

Abstract—**This paper presents Hybrid-MUSE, a new fault localization technique that combines MUtation-baSEd fault localization (MUSE) and Spectrum-Based Fault Localization (SBFL) technique. The core component of Hybrid-MUSE, MUSE, identifies a faulty statement by utilizing different characteristics of two groups of mutants – one that mutates a faulty statement and the other that mutates a correct statement. This paper also proposes a new evaluation metric for fault localization techniques based on information theory, called Locality Information Loss (LIL): it can measure the aptitude of a localization technique for automated fault repair systems as well as human debuggers.**

the same ranking. This often inflates the number of statements needed to be inspected before encountering the fault.

This paper presents a novel fault localization technique called Hybrid-MUSE, a combination of *MUtation-baSEd fault localization* and Spectrum-Based Fault Localization (SBFL), to overcome this problem. The core component of Hybrid-MUSE, MUSE [35], uses mutation analysis to uniquely capture the relationship between individu... and the observed failures for fault ...

2014



Evaluating and improving fault localization

Spencer Pearson*, José Campos**, René Just†, Gordon Fraser**, Rui Abreu‡, Michael D. Ernst*, Deric Pang*, Benjamin Keller*
*U. of Washington, USA **U. of Sheffield, UK †U. of Massachusetts, USA ‡Palo Alto Research Center, USA U. of Porto/HASLab, Portugal
suspense@cs.washington.edu, jose.campos@sheffield.ac.uk, rjust@cs.umass.edu, gordon.fraser@sheffield.ac.uk, rui@computer.org, mernst@cs.washington.edu, dericp@cs.washington.edu, bjkeller@cs.washington.edu

Abstract—**Most fault localization techniques take as input a faulty program, and produce as output a ranked list of suspicious code locations at which the program may be defective. When researchers propose a new fault localization technique, they typically evaluate it on programs with known faults. The technique is scored based on where in its output list the defective code appears. This enables the comparison of multiple fault localization techniques to determine which one is better.**

Previous research has evaluated fault localization techniques using artificial faults, generated either by mutation tools or manually. In other words, previous research has determined which fault localization techniques are best at finding artificial faults. However, it is not known which fault localization techniques are best at finding real faults. It is not obvious that the answer is the same, given previous work showing that artificial faults have both similarities to and differences from real faults.

A fault localization technique is valuable if it works on real faults. Although some real faults (mostly 35 faults in the single small numerical program space [41]) have been used in previous comparisons [45] of fault localization techniques, the vast majority of faults used in such comparisons are fake faults, mostly mutants. The artificial faults were mutants automatically created by a tool [26], [27], [49], or mutant-like manually-seeded faults created by students [44], [46] or researchers [16].

Artificial faults such as mutants differ from real faults in many respects, including their size, their distribution in code, and their difficulty of being detected by tests [22]. It is possible that an evaluation of FL techniques on real faults would yield different outcomes ... on mutants. If so, previous recomm...

2017

- Uses both MUSE and SBFL
  - $Susp_{Hybrid\_MUSE}(s) = norm\_susp(MUSE, s) + norm\_susp(sbfl, s)$

where $norm\_susp(flt, s) = \frac{Susp_{flt}(s) - min(flt)}{max(flt) - min(flt)}$

HybridMUSE Result

HybridMUSE outperforms MUSE and SBFL

Subject Program	% of Executed Stmts Examined	
	MUSE	Hybrid-MUSE
flex	17.72	**4.38**
grep	1.62	**0.91**
gzip	7.58	**0.84**
sed	1.16	**0.45**
space	5.63	**1.67**
Average	6.74	**1.65**

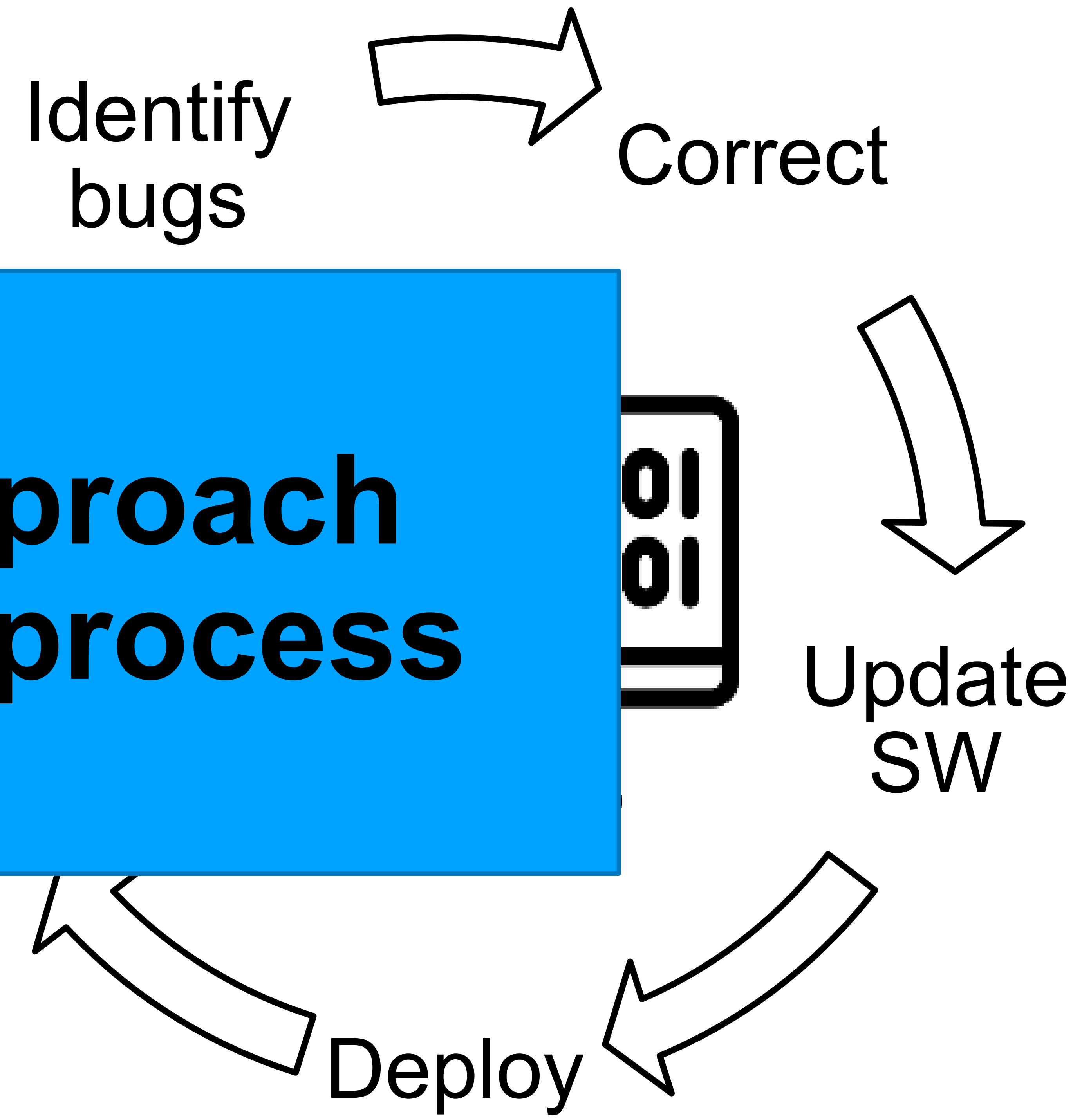
Mutant Sampling Rate	% of Executed Stmts Examined	Rank of a Faulty Stmt
1%	6.20	123.50
10%	4.60	95.53
40%	2.79	61.21
70%	1.83	45.59
100%	1.65	41.60

- Op2: 9.64%

Practical FL for the Industry

Auto FL techniques have been studied for decades, but few tools are employed in the industry

- FL is closely related to other processes
  - Test, correct, update, deploy, etc.



Needs an integrated approach for the entire debugging process

- Numerous de
  - Lack of tests, pr
  - Various usages (by developers)

An Approach for the Entire Debugging Process

Test Generation ↔ Fault Localization ↔ Fix candidates

Sapienz: Multi-objective Automated Testing for Android Applications

Ke Mao Mark Harman Yue Jia
CREST Centre, University College London, Malet Place, London, WC1E 6BT, UK
k.mao@cs.ucl.ac.uk, mark.harman@ucl.ac.uk, yue.jia@ucl.ac.uk

ABSTRACT

We introduce SAPIENZ, an approach to Android testing that uses multi-objective search-based testing to automatically explore and optimise test sequences, minimising length, while simultaneously maximising coverage and fault revelation. SAPIENZ combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation. SAPIENZ significantly outperforms (with large effect size) both the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-revealing sequence length. When applied to the top 1,000 Google Play apps, SAPIENZ found 558 unique, previously unknown crashes. So far we have managed to make contact with the developers of 27 crashing apps. Of these, 14 have confirmed that the crashes are caused by real faults. Of those 14, six already have developer-confirmed fixes.

CCS Concepts

•**Software and its engineering → Software testing and debugging;** *Search-based software engineering;*

Keywords

Android; Test generation; Search-based software testing

1. INTRODUCTION

There are over 1.8 million apps available from the Google

Where test automation does occur, it typically uses Google's Android Monkey tool [36], which is currently integrated with the Android system. Since this tool is so widely available and distributed, it is regarded as the current state-of-practice for automated software testing [53]. Although Monkey automates testing, it does so in a relatively unintelligent manner: generating sequences of events at random in the hope of exploring the app under test and revealing failures. It uses a standard, simple-but-effective, default test oracle [22] that regards any input that reveals a crash to be a fault-revealing test sequence.

Automated testing clearly needs to find such faults, but it is no good if it does so with exceptionally long test sequences. Developers may reject longer sequences as being impractical for debugging and also unlikely to occur in practice; the longer the generated test sequence, the less likely it is to occur in practice. Therefore, a critical goal for automated testing is to find faults with the *shortest possible* test sequences, thereby making fault revelation more actionable to developers.

Exploratory testing is "simultaneous learning, test design, and test execution" [11], that can be cost-effective and is widely used by industrial practitioners [21, 43, 46] for testing in general. However, it is particularly underdeveloped for mobile app testing [41, 42]. Although there exist several test automation frameworks such as Robotium [10] and Appium [3], they require human-imp… inhibiting full automation.

We introduce SAPIENZ, the first…

2016

SapFix: Automated End-to-End Repair at Scale

A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott
Facebook Inc.

***Abstract*—We report our experience with SAPFIX: the first deployment of automated end-to-end fault fixing, from test case design through to deployed repairs in production code[1]. We have used SAPFIX at Facebook to repair 6 production systems, each consisting of tens of millions of lines of code, and which are collectively used by hundreds of millions of people worldwide.**

INTRODUCTION

Automated program repair seeks to find small changes to software systems that patch known bugs [1], [2]. One widely studied approach uses software testing to guide the repair process, as typified by the GenProg approach to search-based program repair [3].

Recently, the automated test case design system, Sapienz [4], has been deployed at scale [5], [6]. The deployment of Sapienz allows us to find hundreds of crashes per month, before they even reach our internal human testers. Our software engineers have found fixes for approximately 75% of Sapienz-reported crashes [6], indicating a high signal-to-noise ratio [5] for Sapienz bug reports. Nevertheless, developers' time and expertise could undoubtedly be better spent on more creative programming tasks if we could automate some or all of the comparatively tedious and time-consuming repair process.

The deployment of Sapienz automated test design means that automated repair can now also take advantage of automated software test design to automatically re-test candidate patches. Therefore, we have started to deploy automated repair, in a tool called SAPFIX, to tackle some of these crashes. SAPFIX

In order to deploy such a fully automated end-to-end detect-and-fix process we naturally needed to combine a number of different techniques. Nevertheless the SAPFIX core algorithm is a simple one. Specifically, it combines straightforward approaches to mutation testing [8], [9], search-based software testing [6], [10], [11], and fault localisation [12] as well as existing developer-designed test cases. We also needed to deploy many practical engineering techniques and develop new engineering solutions in order to ensure scalability.

SAPFIX combines a mutation-based technique, augmented by patterns inferred from previous human fixes, with a reversion-as-last resort strategy for high-firing crashes (that would otherwise block further testing, if not fixed or removed). This core fixing technology is combined with Sapienz automated test design, Infer's static analysis and the localisation infrastructure built specifically for Sapienz [6]. SAPFIX is deployed on top of the Facebook FBLearner Machine Learning infrastructure [13] into the Phabricator code review system, which supports the interactions with developers.

Because of its focus on deployment in a continuous integration environment, SAPFIX makes deliberate choices to sidestep some of the difficulties pointed out in the existing literature on automated program repair (see Related Work section). Since SAPFIX focuses on null-dereference faults revealed by Sapienz test cases as code is submitted for review it can re-use the Sapienz fault localisati… on null-dereference errors also means th…

2019

EvoFuzz: Improving and extending EvoSuite

Moon and Jhi, SBFT'24 [Java Testing Tool Competition Winner, 1st place]

- Goal: supporting the entire debugging
  - •Currently in improving test generation

- Built on top of EvoSuite
  - •Readily available features for FL (test generation, execution, mutation, etc)
  - •**Won of 10 / 11 SBFT (=SBST) competitions (2013 ~ 2023, except 2015)**

- Improving and extending EvoSuite
  - •Support JDK8~17 and Spring6.0.x+
  - •**SBFT'24 competition 1st place** (2nd EvoSuite)
  - •NEXT: FL and repair

EvoFuzz at the SBFT 2024 Tool Competition

Seokhyeon Moon
shyeon.mun@samsung.com
Technology Research, Samsung SDS
Seoul, Republic of Korea

Yoon-Chan Jhi
yoonchan.jhi@samsung.com
Technology Research, Samsung SDS
Seoul, Republic of Korea

ABSTRACT

EvoFuzz is an automated fuzzing tool that integrates fuzzing techniques into EvoSuite to improve code coverage. It first uses EvoSuite to generate a test suite, which is then utilized for fuzzing. During this process, EvoFuzz ensures the generated test suite is executable by performing strict code validity checks. Additionally, EvoFuzz actively explores new variables/values to achieve more code coverage. Our experimental results on the SBFT2024 benchmark demonstrate that EvoFuzz outperforms EvoSuite in both code coverage and mutation kill ratio.

KEYWORDS

Software testing, Fuzzing, Test case generation, Code validity

ACM Reference Format:
Seokhyeon Moon and Yoon-Chan Jhi. 2024. EvoFuzz at the SBFT 2024 Tool Competition . In *2024 ACM/IEEE International Workshop on Search-Based and Fuzz Testing (SBFT '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3643659.3648556

1 INTRODUCTION

Despite EvoSuite's success in achieving high code coverage and bug detection over several years, our investigation revealed opportunities for improvement, particularly in the comprehensive exploration of the expansive search space inherent in program variables.

cases. Out of the 9007 test cases generated for the 210 target subjects (we conducted 3 repetitions of the experiment for all 70 classes within the benchmark), only 7994 were compiling. Given that the fuzzing procedures are initiated from the test drivers derived from the generated test suite, fuzzing with non-compiling test drivers can result in test cases that achieve no coverage at all.

Recognizing that the efficacy of fuzzing efforts is contingent upon the executable nature of the test suite, EvoFuzz integrates both fuzzing techniques [3] and a robust mechanism for ensuring test suite executability. This integration enables EvoFuzz to address the identified shortcomings of EvoSuite, improving the overall effectiveness of the fuzzing process and the code coverage.

2 EVOFUZZ

EvoFuzz is a prototype research tool that consists of two key steps. In the initial step, EvoFuzz employs EvoSuite's test suite generation algorithm to create a set of test cases. During this process, EvoFuzz identifies mutable variables/values within each test case, establishing a foundation for fuzzing. In the subsequent the step, EvoFuzz systematically iterates over the generated test suite, strategically mutating the values of the identified mutable variables.

2024

Remarks

MUSE and HybridMUSE have added new dimensions for FL

- MUSE and HybridMUSE
  - (1) **Mu**tation-ba**se**d FL*
  - (2) Combination of MUSE with SBFL

* (independent MBFL studies: Metallaxis-FL, FIFL, etc.)

- An integrated approach is essential for the practical FL
  - •FL usage scenarios are not perfectly working frequently
  - •Contribute to this for both industry and academia

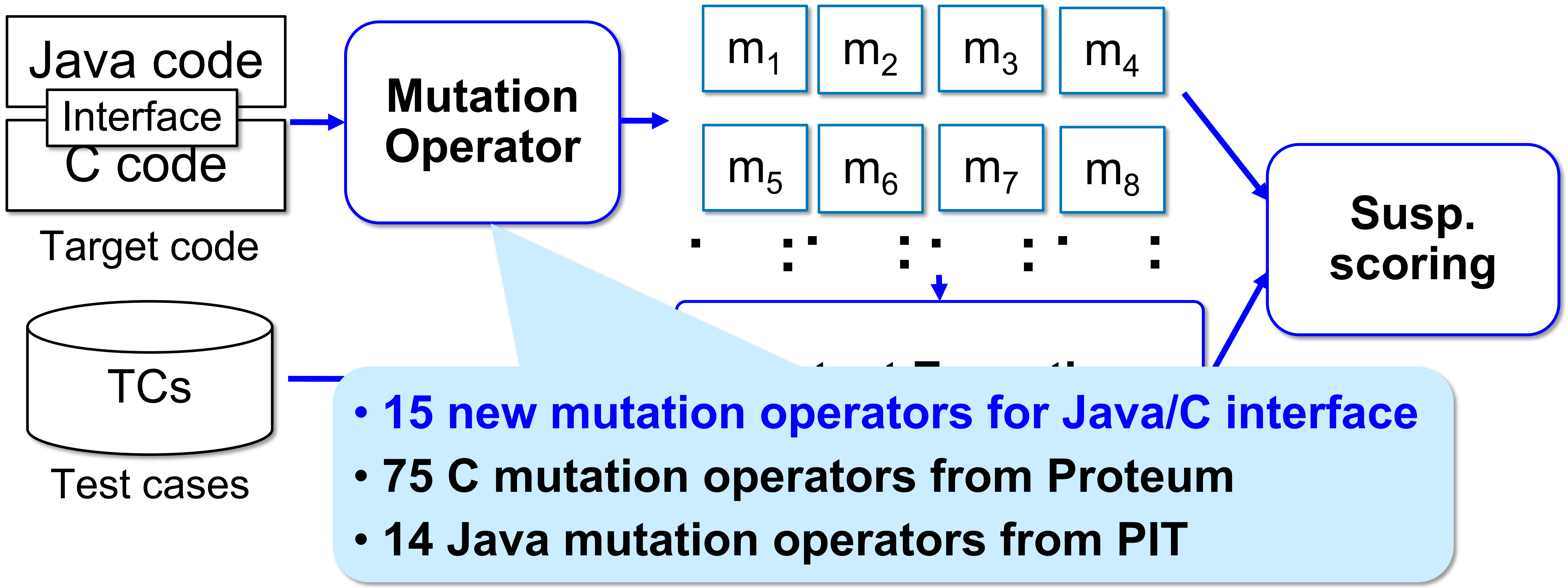
Applications of MUSE

Yunho Kim | Assistant Professor @ Hanyang University, Korea

Fault Localization for Multi-lingual Programs

Hong et al., ASE 15 and IST 17

- MUSEUM extends MUSE to localize faults caused by language interface specification violation



- MUSEUM can find a root cause of a complex real-world bug
  - Eclipse Bug 322222 had survived **more than 1 year** with 14 duplicate bug reports and more than 100 comments

What Do We Need for Effective FL?

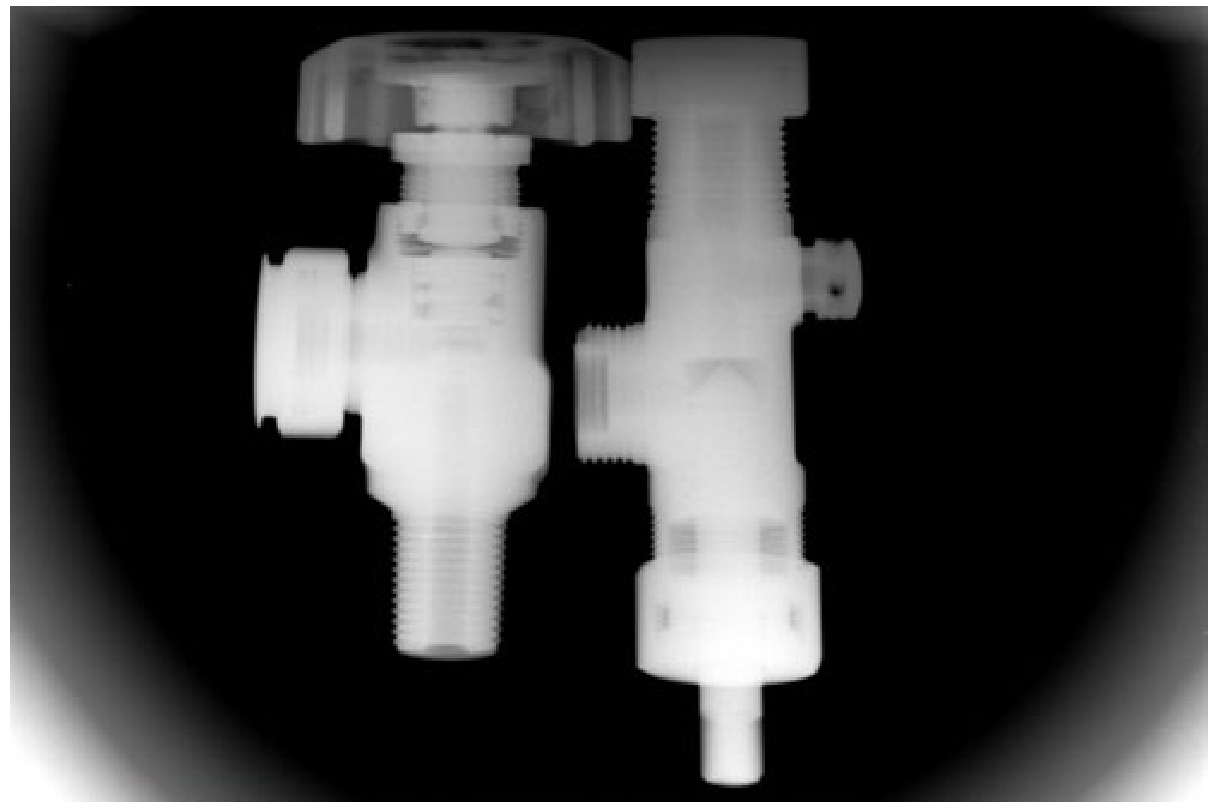
Pre-requisite: Generation of TCs w/ High Coverage

- The accuracy of FL highly depends on **the quality of passing/failing testcases.**
  - How to generate test inputs w/ high coverage is a critical pre-requite to successful FL.
- Thus, we developed a novel test input generation technique DeMiner through code mutation
  - For a target program *P, w*e utilize mutants of *P* as **syntactic shortcuts** to quickly reach deep execution states of *P.*

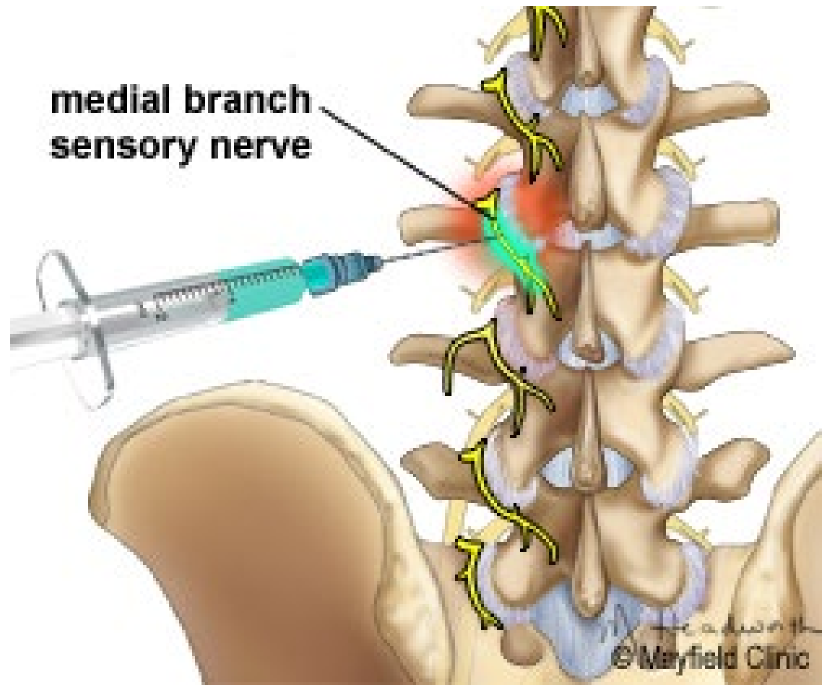
Invasive Software Testing

Kim et al., ICST 18 [Distinguished Paper Award], Kim and Hong., STVR 19

Non-invasive analysis



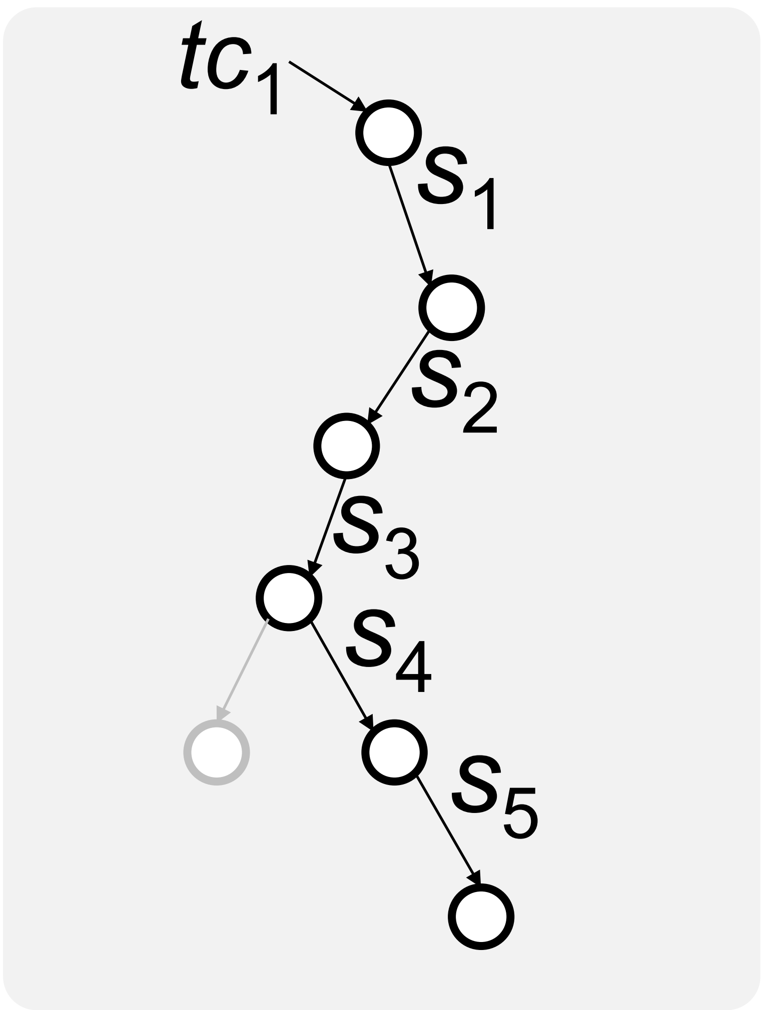
Invasive analysis



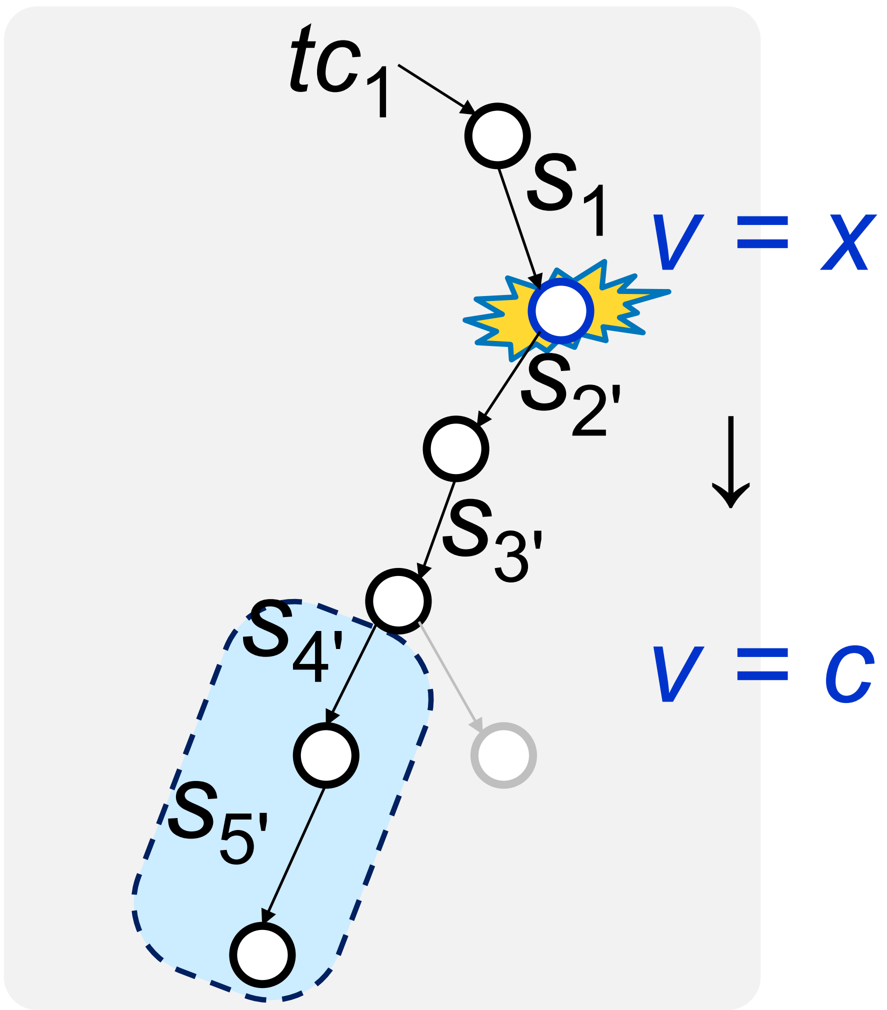
Invasive Software Testing

Kim et al., ICST 18 [Distinguished Paper Award], Kim and Hong., STVR 19

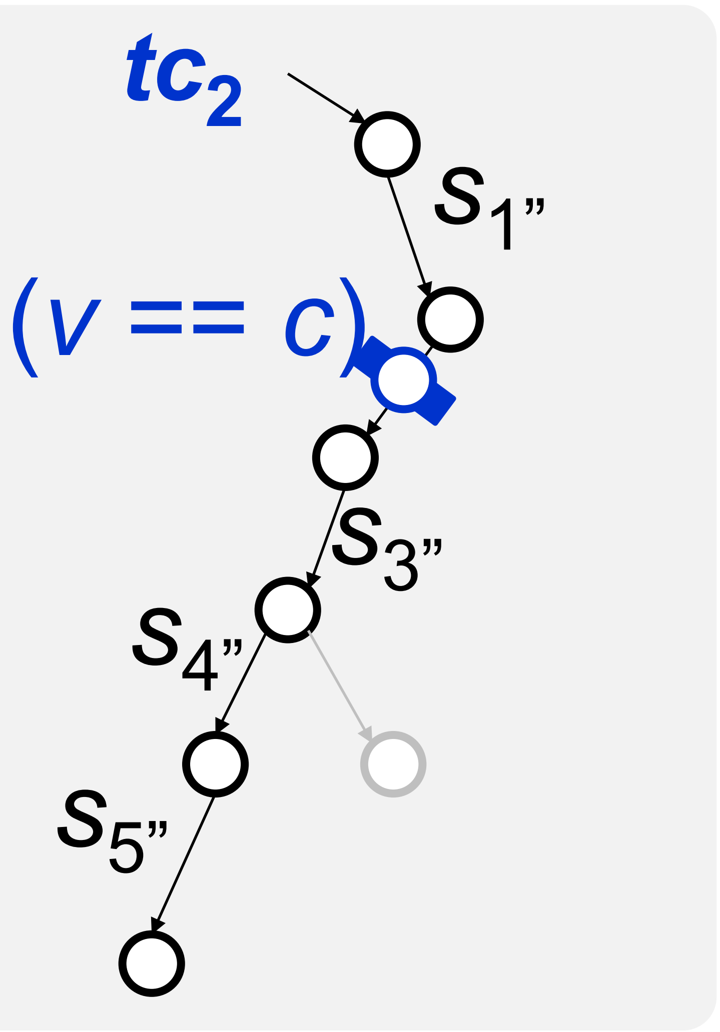
- Use *mutation* to quickly generate *diverse likely-test executions*
- Discover pre-conditions for reaching corner-cases
- Guide symbolic execution with the discovered pre-conditions



1. Original program execution



2. Mutant execution



3. Generated execution w/ guide

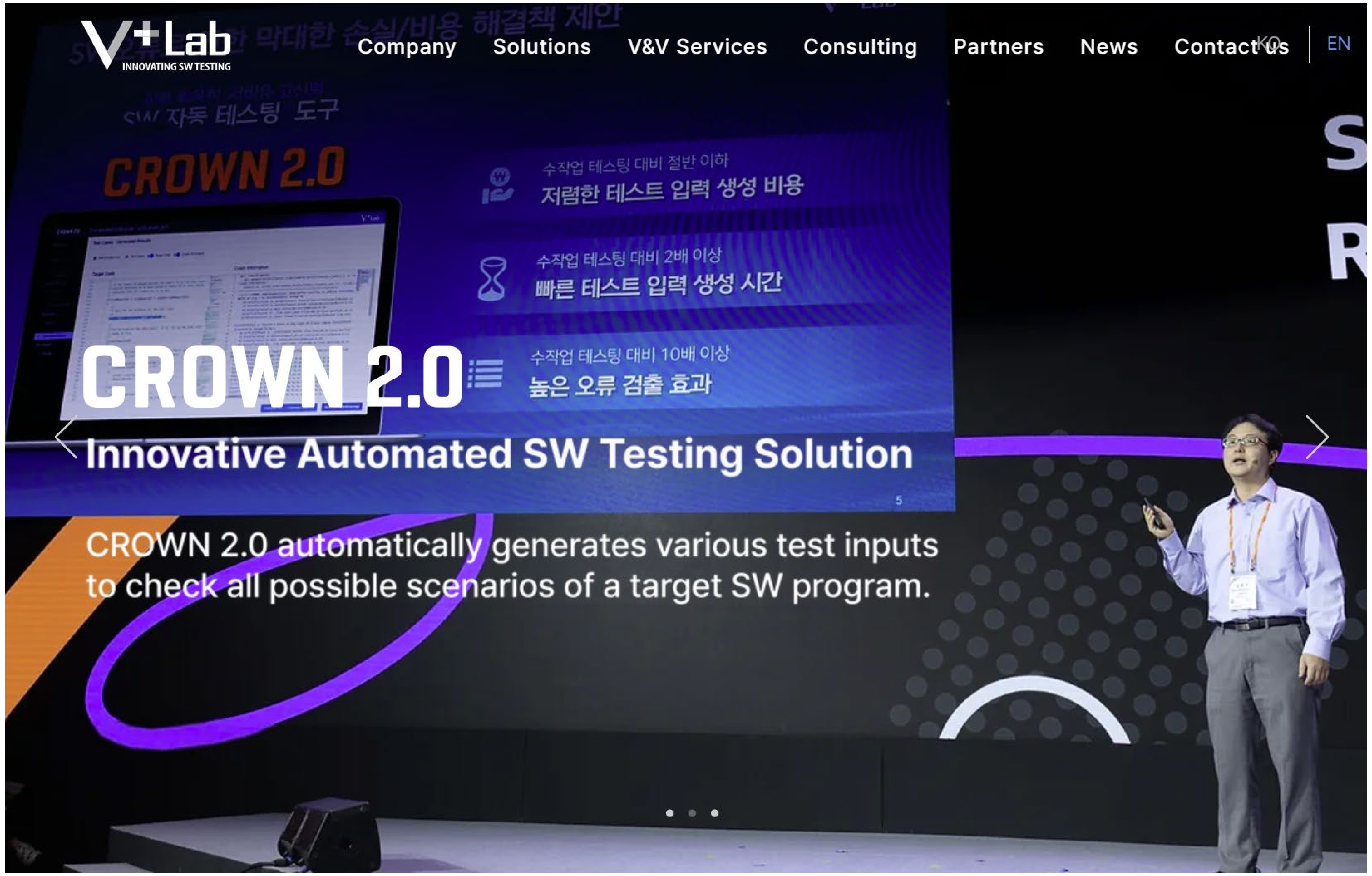
Truly Real-world Industrial Application

Yunho and Moonzoo founded VPlusLab Inc. in 2019 (https://vpluslab.kr)

- VPlusLab provides an automated test input generation solution CROWN 2.0 for safety critical sys.
  - For Hyundai Mobis Automotive S/W, CROWN achieved >80% MC/DC cov. [ICSE SEIP '19]
- Our customers:

HYUNDAI | HYUNDAI MOBIS | ETRI 한국전자통신연구원 Electronics and Telecommunications Research Institute

LIG Nex1 | POSTECH | NSR 국가보안기술연구소 National Security Research Institute



Can We Do Things Better?

How can we improve mutation with machine learning?

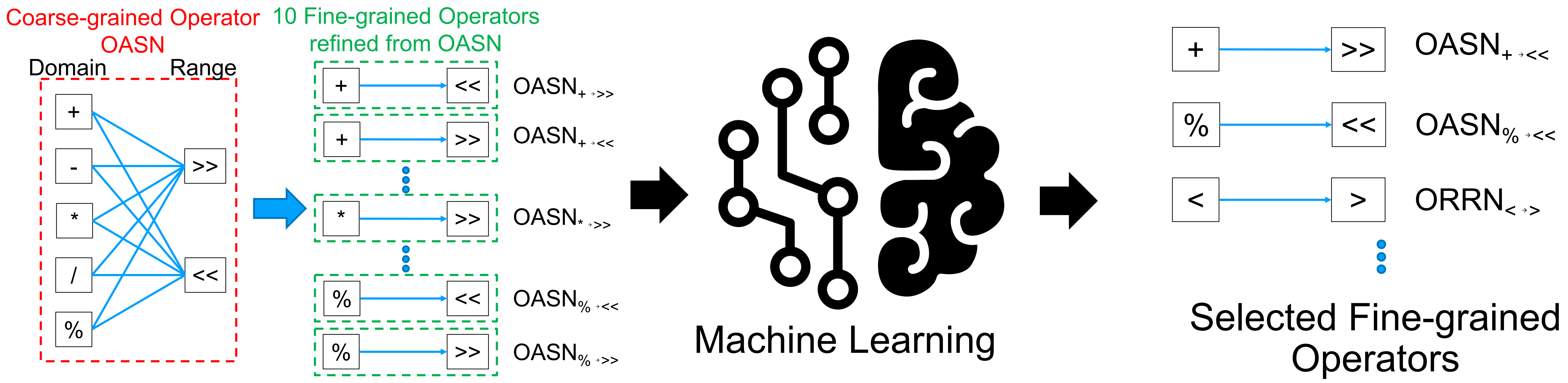
- So far, mutation can give great answers to testing and debugging problems from various mutant executions
  - Various mutant executions provide valuable data to learn the behaviors of software-under-analysis mutant analysis
- We can apply machine learning to learn a model from various mutant executions and apply it to testing and debugging

ML-based Mutant Selection

Phan et al., Mutation 18, Kim and Hong, STVR 21

- With great power comes great cost
- We need to select representative fine-grained mutation operators
  - We develop MUSIC to support fine-grained mutation operators for C/C++ (https://github.com/swtv-kaist/MUSIC)

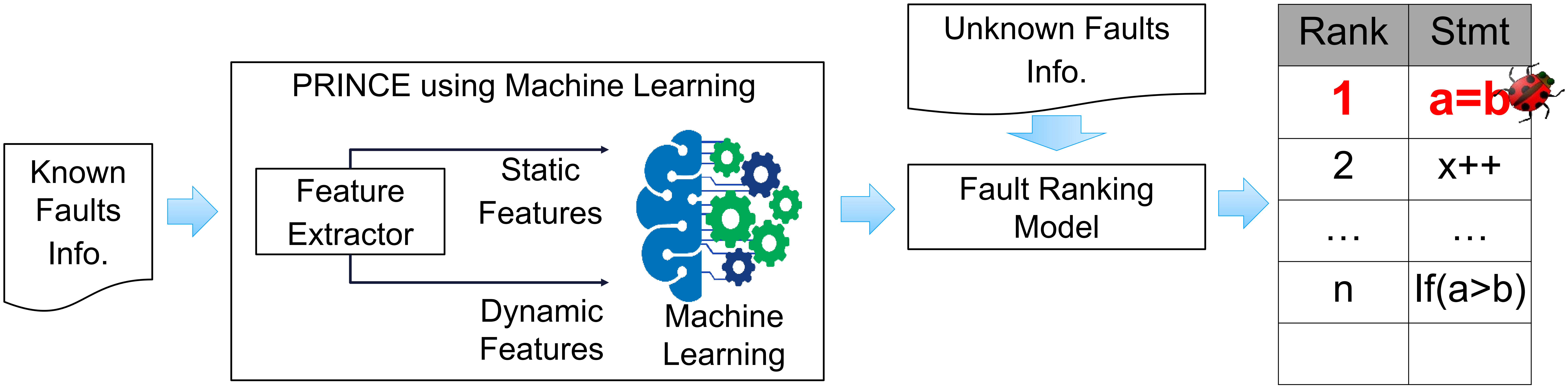
Rule: Arithmetic Operator → Shift Operator



ML-based Fault Localization

Kim et al., TOSEM 19

- A single fixed susp. formula cannot rule them all
- PRINCE learns a FL model from dynamic FL features and static code complexity.



What came after MUSE

Shin Yoo | Associate Professor @ KAIST

What happened?

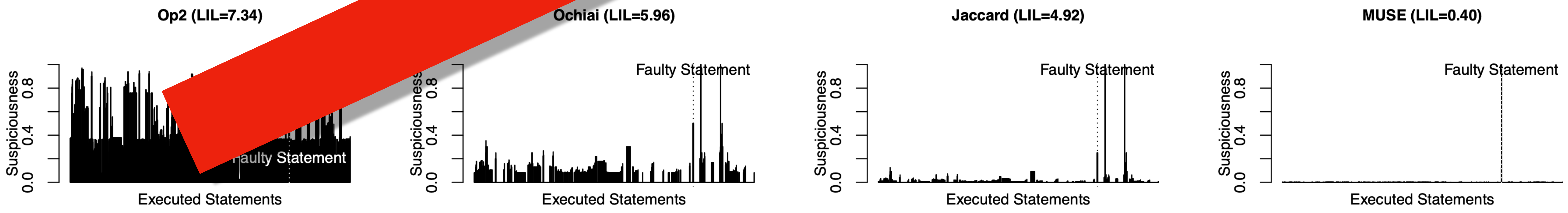
- “Wow, I’m old…”
- I could not attend ICST 2014 because of a certain toddler

Locality Information Loss (LIL)

A new way to evaluate FL results

- Ranking-based evaluation assumes linear consump[illegible] [illegible]sults by humans; what if machines (=APR) want the probability [illegible] [illegible]eing at location Y?
- FL performance should be measure[illegible] [illegible]s-entropy between the suspiciousness score distributi[illegible] [illegible]e ground truth!

Didn’t take off!



Cost of MUSE

MBFL may be accurate but is also very expensive!

- In fact, the more expensive it is, the more accurate it can be

Unified Debugging

Lou et al., ISSTA 2020

- Iteratively refine FL by running APR based on the intermediate FL results!
- APR = applying changes to code = mutation

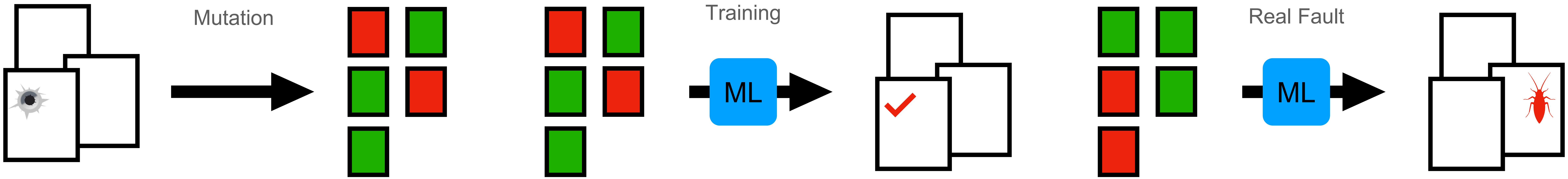
Ahead-of-Time MBFL

Kim et al., ISSRE 2021, IST 2023

Prof. Robert Feldt   Dr. Jinhan Kim   Gabin An (PhD Candidate)

- How can we do MBFL, without having to generate/compile/test all the mutants at the time of test failure?

- Can we do the expensive step in advance??



1. Do mutation analysis

2. Reverse relation and learn to predict mutation location from test results

3. Given a real failure, pretend if it is a mutant and ask the model where it is

Predictive Mutation Analysis

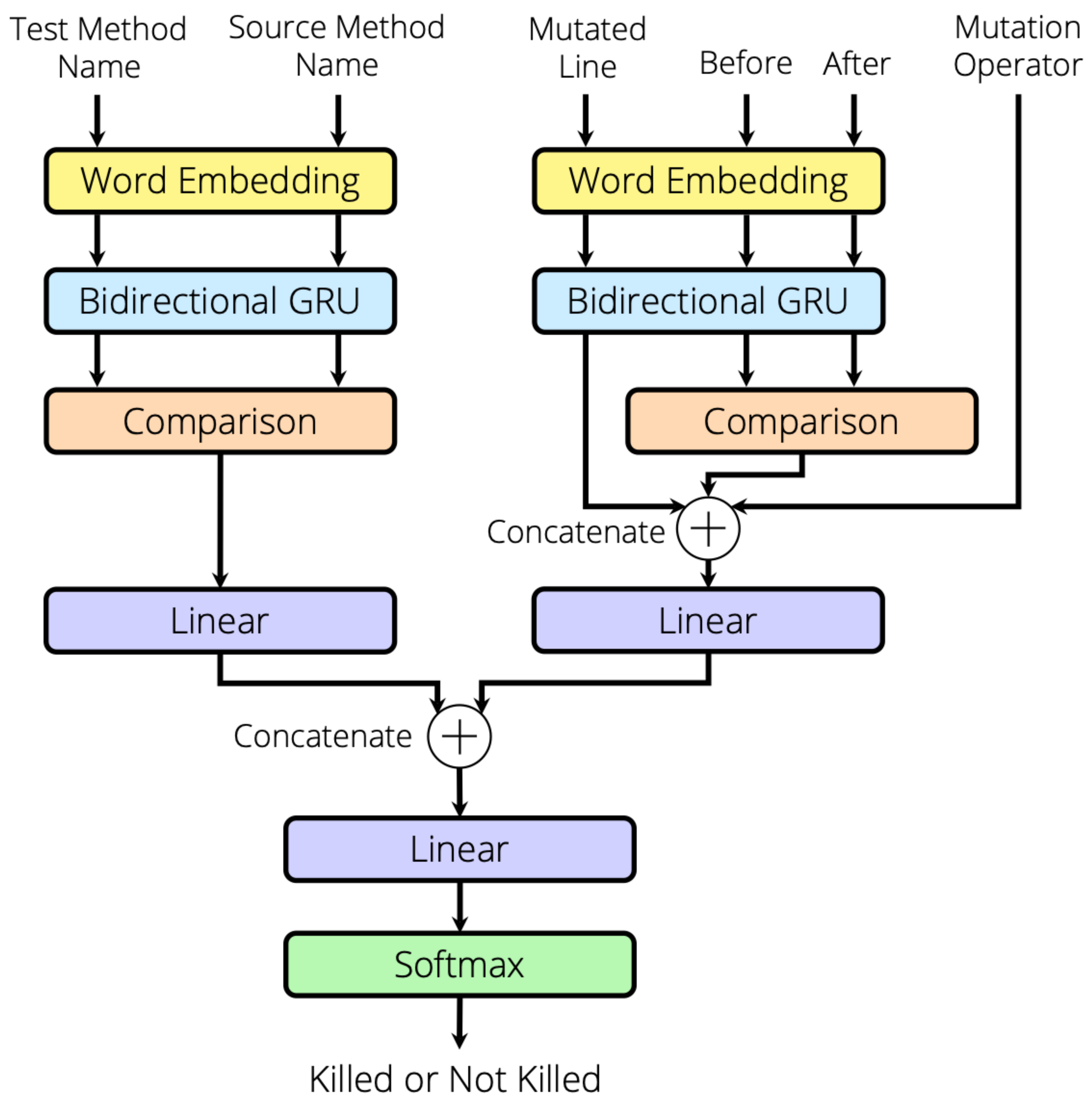
Kim et al., TOSEM 2021

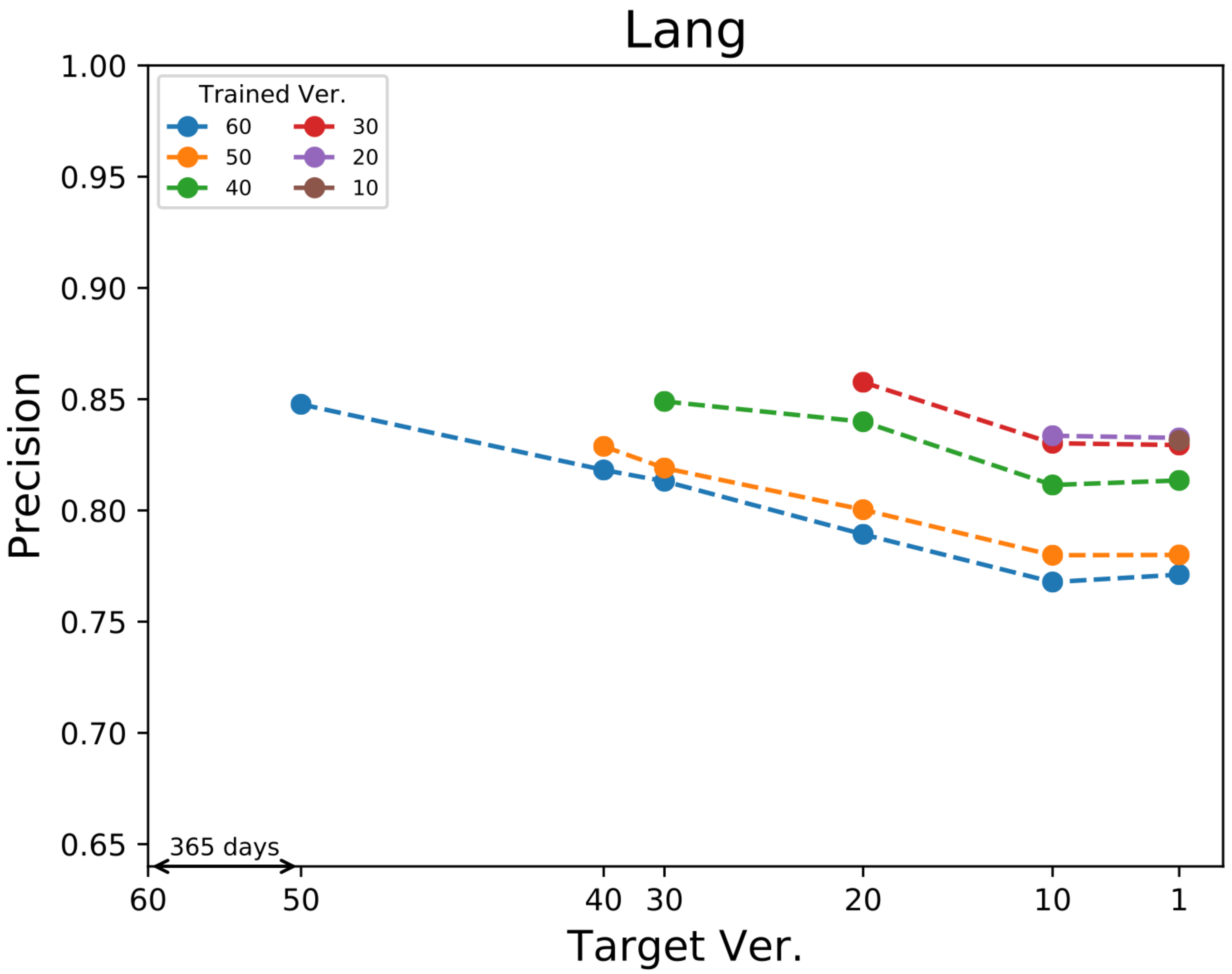
Prof. Robert Feldt

Dr. Jinhan Kim

Prof. Shin Hong

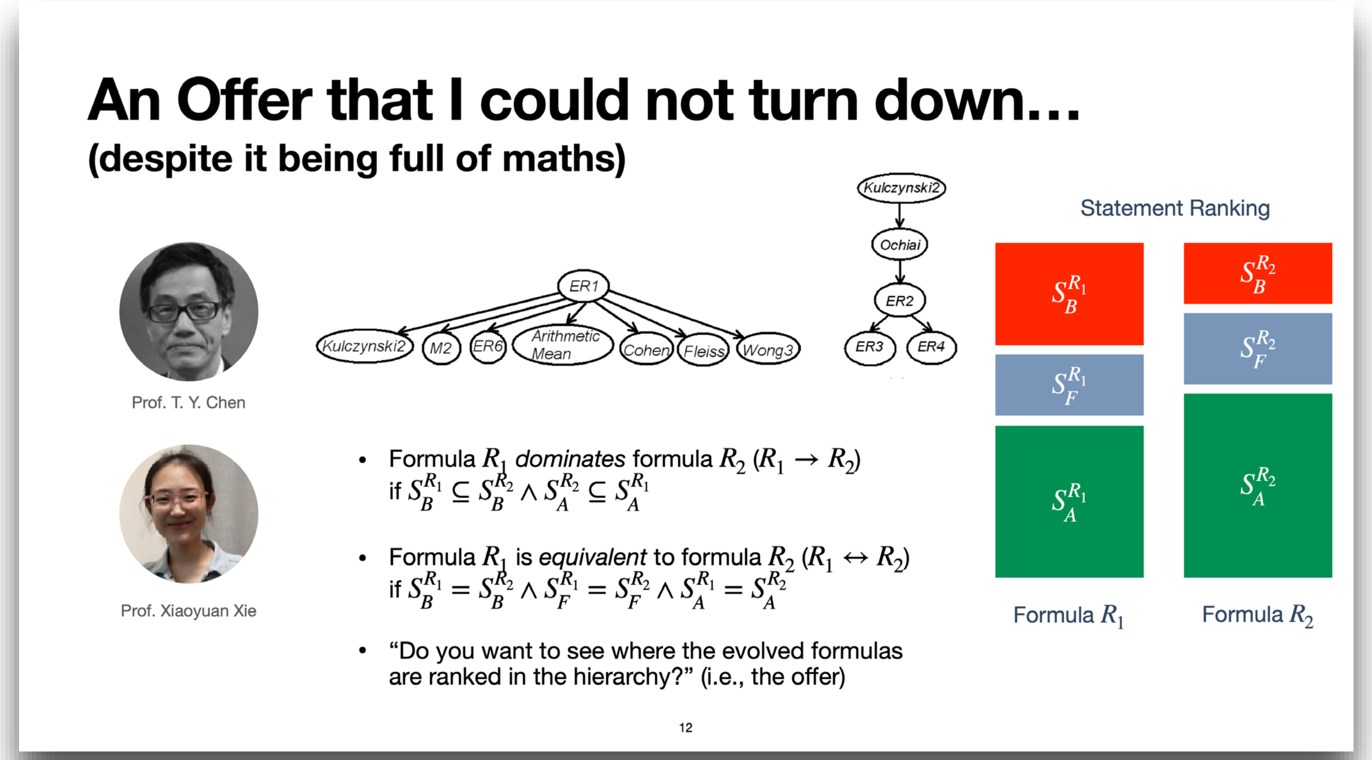
- What if a new test case fails, and you do not have any ahead-of-time mutation results?
- We try to predict the mutation analysis results by exploiting natural language channel.

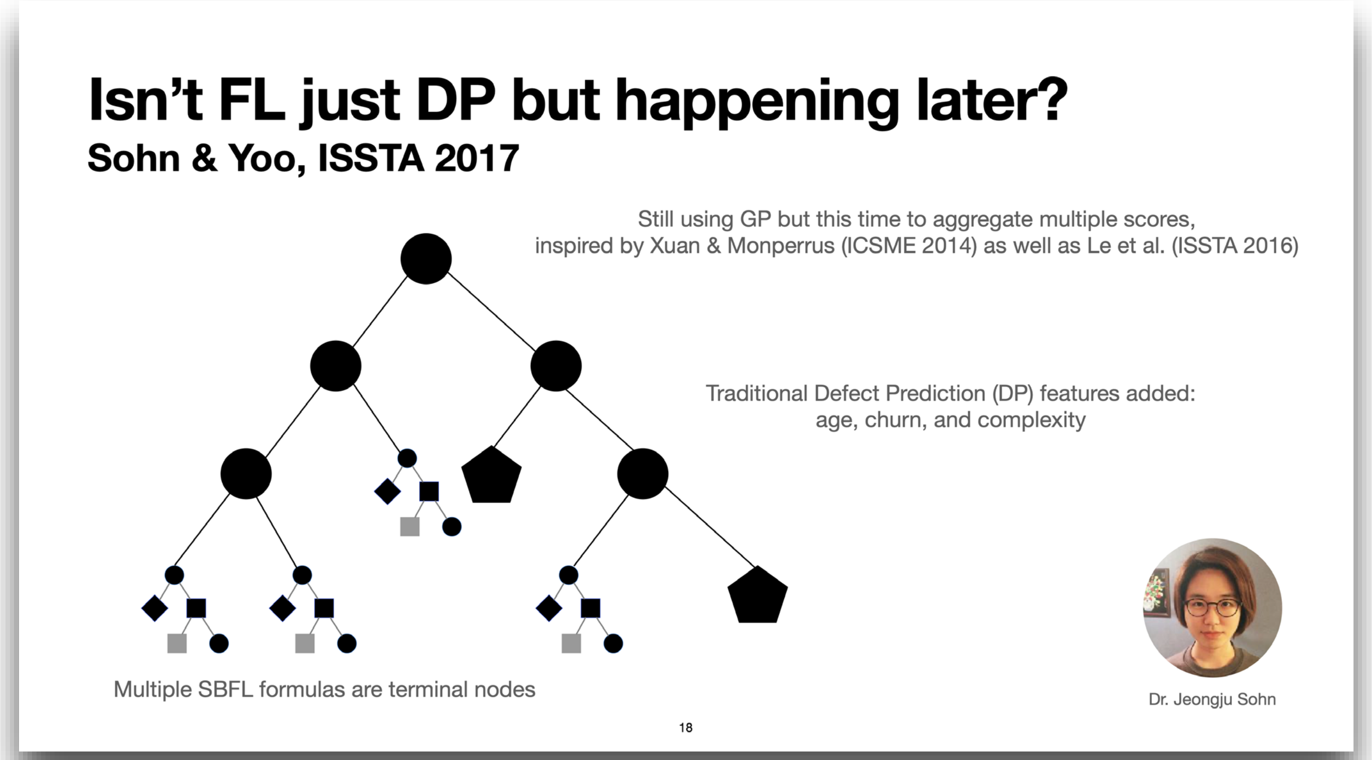


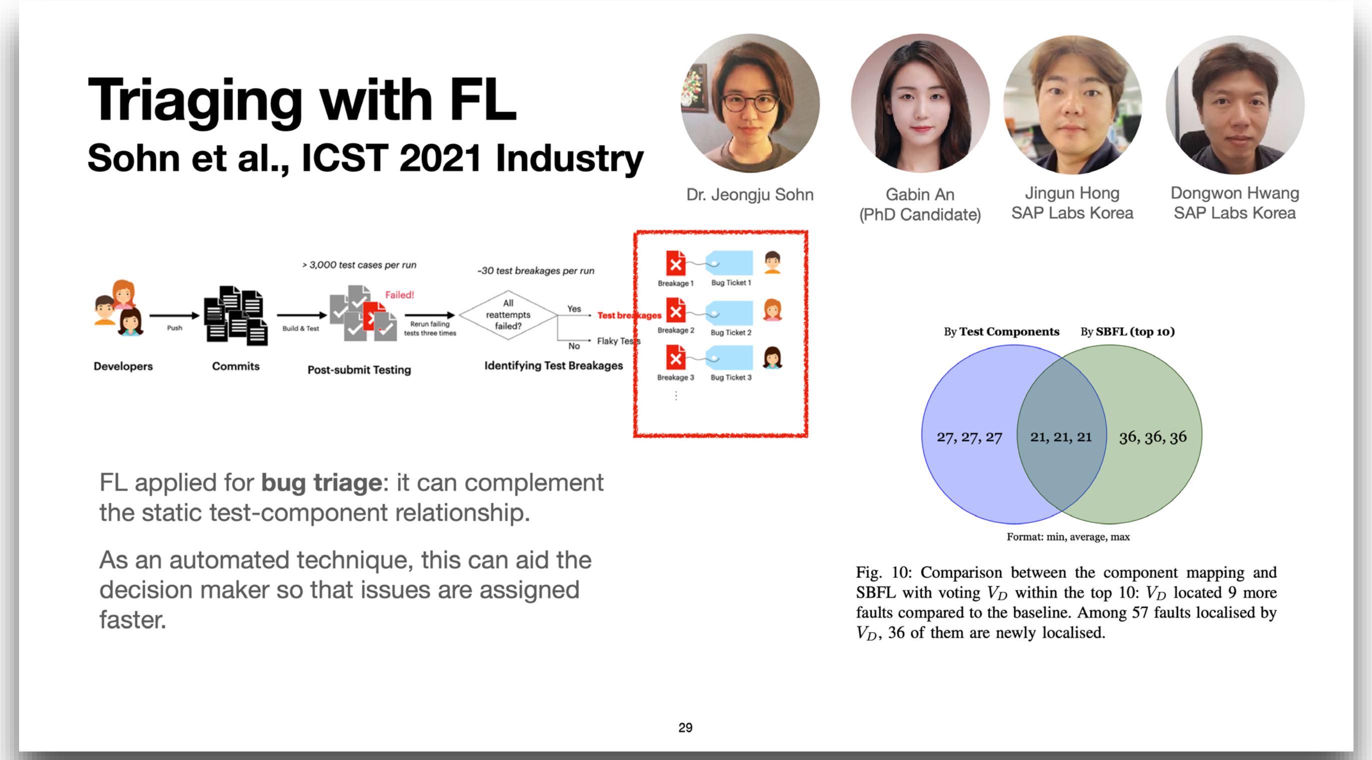


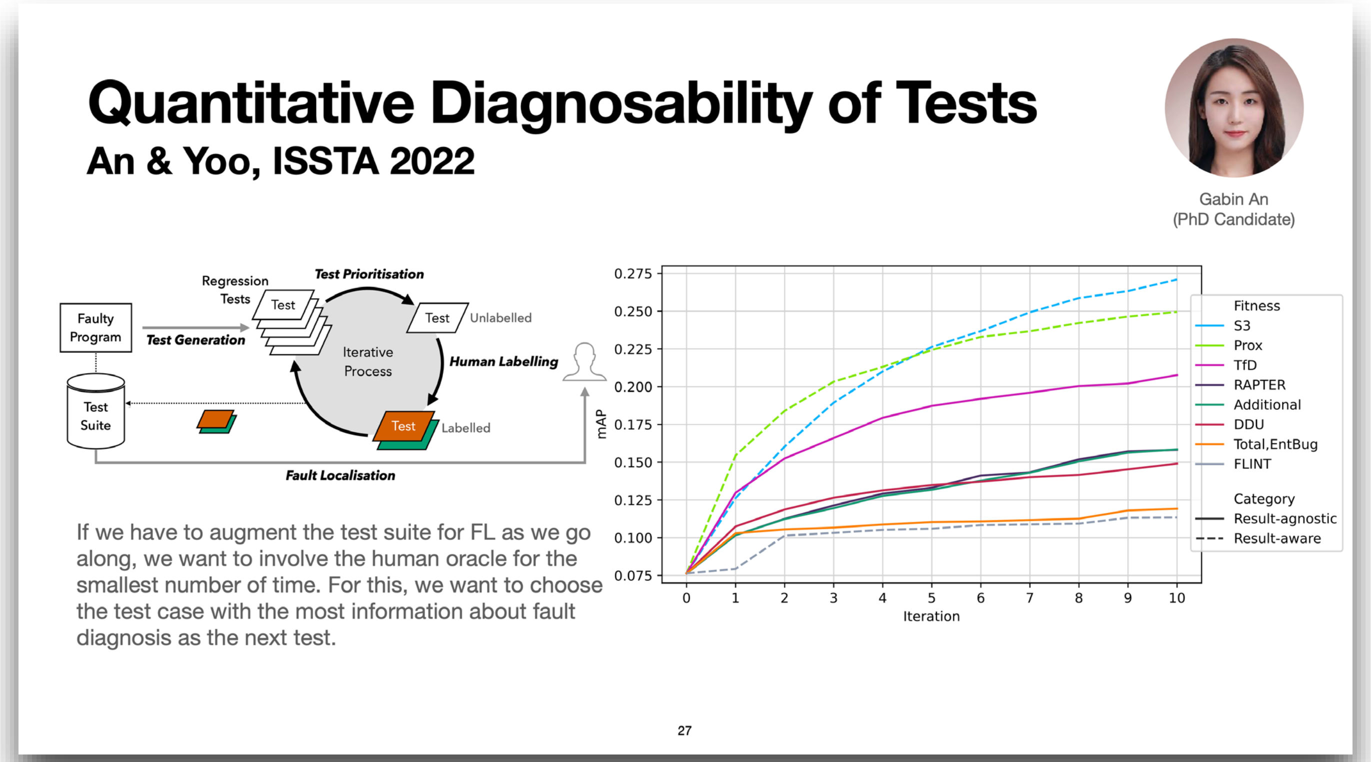
…And Many Other Mutation/FL Work

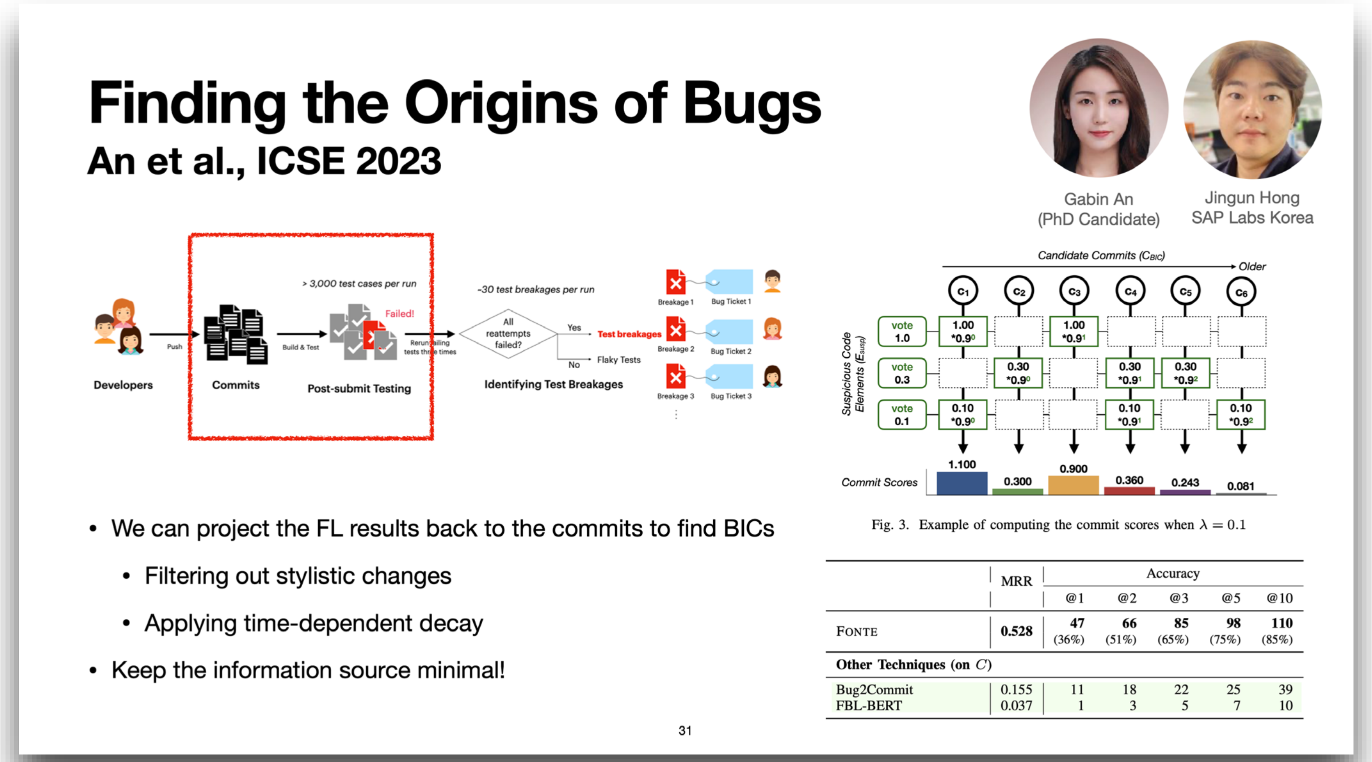
(that stemmed from my experience of ICST 2014 collaboration)

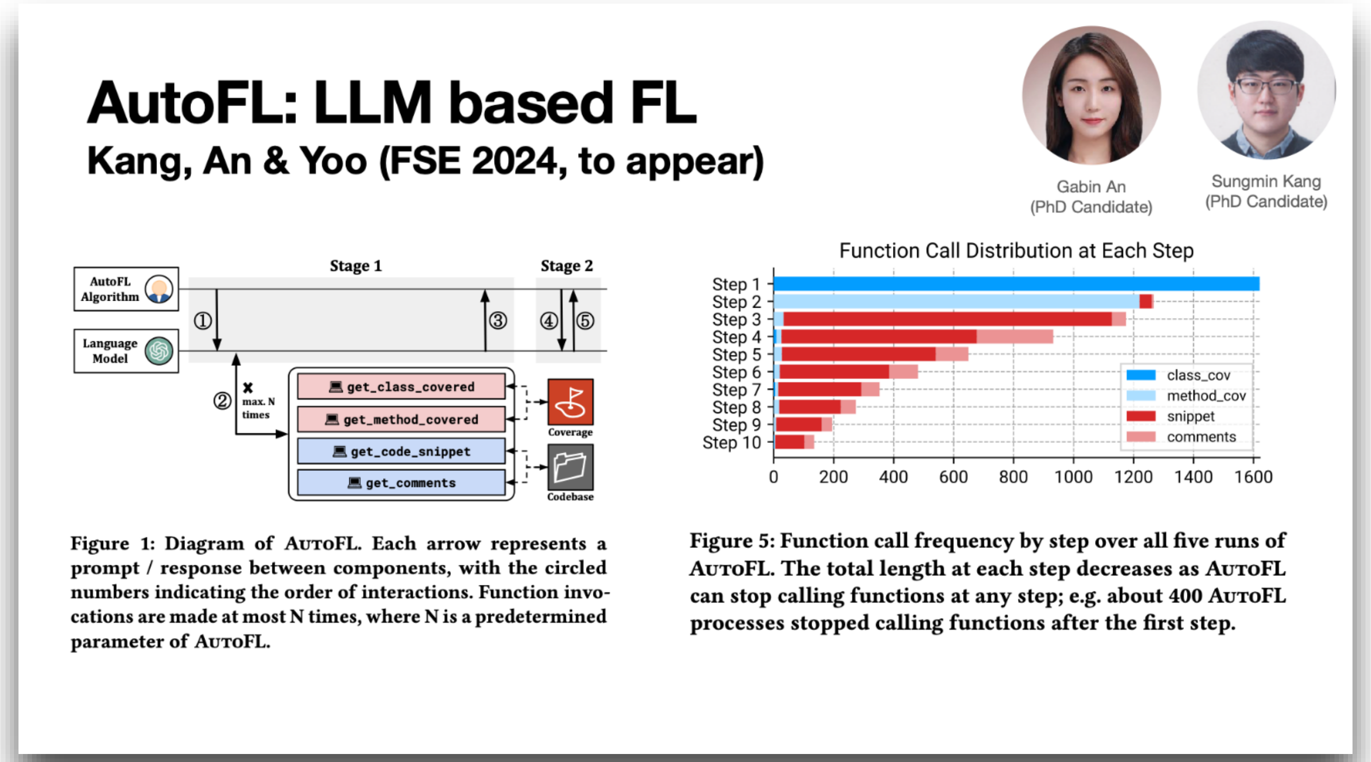












Summary

- Mutation is a fundamentally strong tool!
- Sticking to a single problem can be fun, if the problem is important


ICST 2026
Daejeon, Republic of Korea