GAP TESTING: COMBINING DIVERSE TESTING STRATEGIES FOR FUN AND PROFIT

DR. BEN LIVSHITS

IMPERIAL COLLEGE LONDON

MY BACKGROUND

- Professor at Imperial College London
- Industrial researcher
- Stanford Ph.D.
- Here to talk about some of the technologies underlying testing
- Learn about industrial practice

- Work on a range of topics including
- Software reliability
- Program analysis
- Security and privacy
- Crowd-sourcing
- etc.

FOR FUNCTIONAL TESTING: MANY STRATEGIES

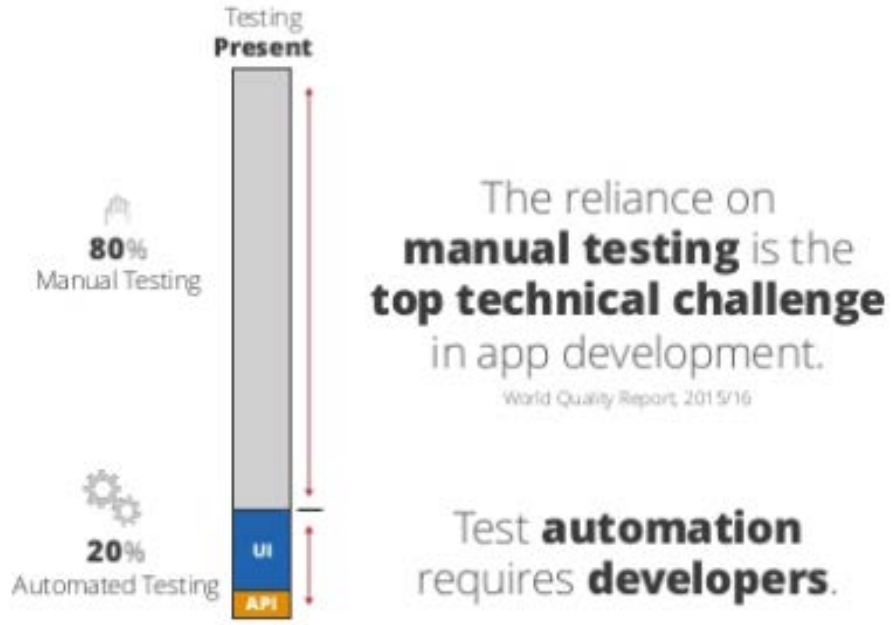
Human effort

- Test suites written by developers and/or testers
- Field testing
- Crowd-based testing
- Penetration testing

Automation

- (Black box) Fuzzing
- White box fuzzing or symbolic execution
- We might even throw in other automated strategies into this category such as static analysis

MANUAL VS. AUTOMATED



- My focus is on **automation**, generally
- However, ultimately, these two approaches should be **complimentary** to each other
- Case in point: consider the numerous companies that do **mobile app** testing, i.e. Applause
  - The general approach is to upload an app binary, have a crowd of people on call, they jump on the app, encounter bugs, report bugs, etc.
  - Generally, not many guarantees from this kind of approach
  - But it's quite useful as the first level of testing

https://www.slideshare.net/IosifItkin/extent2016-the-future-of-software-testing

MANUAL VS. AUTOMATED: HOW DO THEY COMPARE?

- Fundamentally, a difficult question to answer
- What is our goal
- Operational goals
  - Make sure the application doesn't crash at the start
  - Make sure the application isn't easy to hack into
- Development/design goals
  - Make sure the coverage is high or 100%, for some definition of what coverage is
  - Make sure the application doesn't crash, ever, or violate assertions, ever?

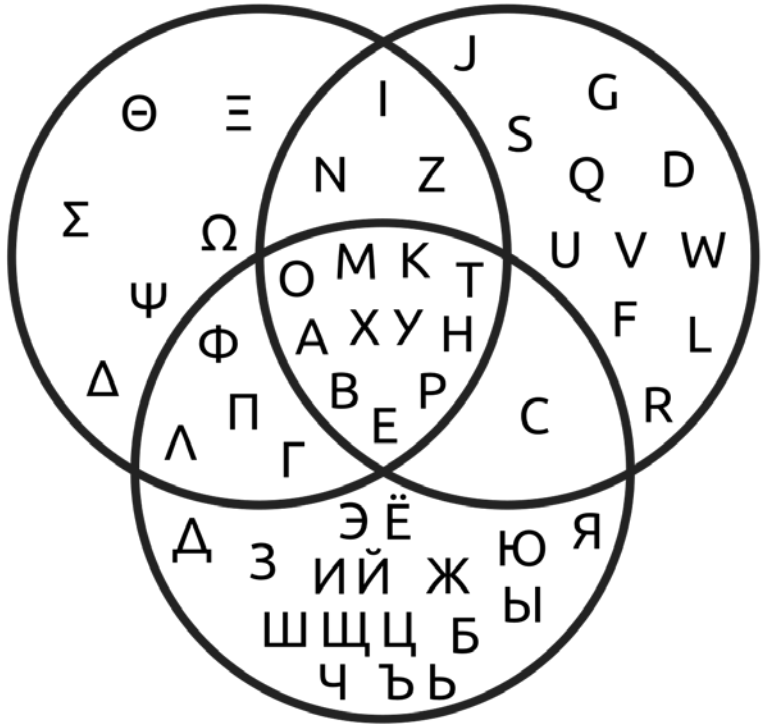
Do we have to choose?

MULTIPLE, COMPETING, UNCOORDINATED TECHNIQUES ARE NORMAL

- We would love to have a situation of when **one** solution delivers all the **value**
- Case in point: symbolic execution was advertised as a the best thing since sliced bread:
  - Precision of runtime execution
  - Coverage of static analysis
  - How can this go wrong?
- The practice of symbolic execution is unfortunately different
- Coverage numbers from KLEE and SAGE

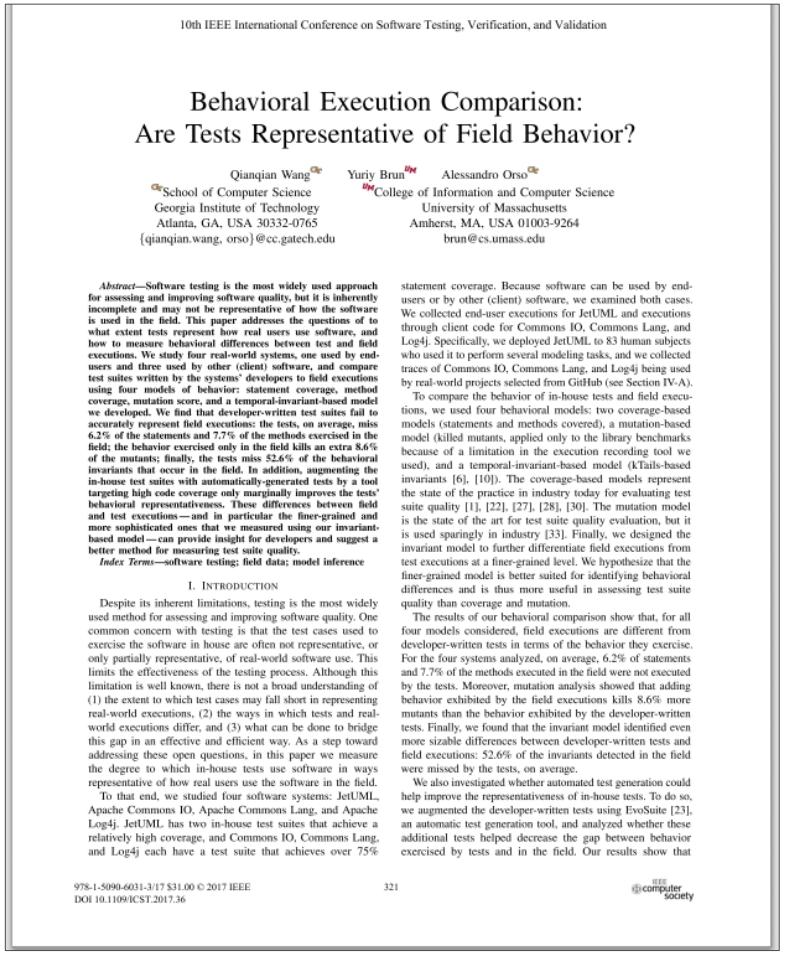
SO, MAYBE ONE TECHNIQUE ALONE IS NOT GOOD ENOUGH

- What can we do?
- Well, let's assume we have the compute cycles (which we often do) and the money to hire testers (which we often don't)
- How do combine these efforts?
- Fundamental challenges
  - Overlap is significant, bling fuzzing is not so helpful
  - Differences are hard to hit – for example, how do we hit a specific code execution path to get closer to 100% path coverage? Symbolic execution is a heavy-weight, less-than-scalable answer



DEVELOPER-WRITTEN TESTS VS. IN-THE-FIELD EXECUTION

- Study four large open-source Java projects
- We find that developer-written test suites **fail to accurately represent field executions**: the tests, on average, miss 6.2% of the statements and 7.7% of the methods exercised in the field;
- The behavior exercised only in the field kills an extra 8.6% of the mutants; finally, the tests miss 52.6% of the behavioral invariants that occur in the field.



10th IEEE International Conference on Software Testing, Verification, and Validation

Behavioral Execution Comparison:
Are Tests Representative of Field Behavior?

Qianqian Wang[Gr]
[Gr]School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA 30332-0765
{qianqian.wang, orso}@cc.gatech.edu

Yuriy Brun[Br]
[Br]College of Information and Computer Science
University of Massachusetts
Amherst, MA, USA 01003-9264
brun@cs.umass.edu

Alessandro Orso[Gr]

***Abstract*—Software testing is the most widely used approach for assessing and improving software quality, but it is inherently incomplete and may not be representative of how the software is used in the field. This paper addresses the questions of to what extent tests represent how real users use software, and how to measure behavioral differences between test and field executions. We study four real-world systems, one used by end-users and three used by other (client) software, and compare test suites written by the systems' developers to field executions using four models of behavior: statement coverage, method coverage, mutation score, and a temporal-invariant-based model we developed. We find that developer-written test suites fail to accurately represent field executions: the tests, on average, miss 6.2% of the statements and 7.7% of the methods exercised in the field; the behavior exercised only in the field kills an extra 8.6% of the mutants; finally, the tests miss 52.6% of the behavioral invariants that occur in the field. In addition, augmenting the in-house test suites with automatically-generated tests by a tool targeting high code coverage only marginally improves the tests' behavioral representativeness. These differences between field and test executions — and in particular the finer-grained and more sophisticated ones that we measured using our invariant-based model — can provide insight for developers and suggest a better method for measuring test suite quality.**

***Index Terms*—software testing; field data; model inference**

I. Introduction

Despite its inherent limitations, testing is the most widely used method for assessing and improving software quality. One common concern with testing is that the test cases used to exercise the software in house are often not representative, or only partially representative, of real-world software use. This limits the effectiveness of the testing process. Although this limitation is well known, there is not a broad understanding of (1) the extent to which test cases may fall short in representing real-world executions, (2) the ways in which tests and real-world executions differ, and (3) what can be done to bridge this gap in an effective and efficient way. As a step toward addressing these open questions, in this paper we measure the degree to which in-house tests use software in ways representative of how real users use the software in the field.

To that end, we studied four software systems: JetUML, Apache Commons IO, Apache Commons Lang, and Apache Log4j. JetUML has two in-house test suites that achieve a relatively high coverage, and Commons IO, Commons Lang, and Log4j each have a test suite that achieves over 75% statement coverage. Because software can be used by end-users or by other (client) software, we examined both cases. We collected end-user executions for JetUML and executions through client code for Commons IO, Commons Lang, and Log4j. Specifically, we deployed JetUML to 83 human subjects who used it to perform several modeling tasks, and we collected traces of Commons IO, Commons Lang, and Log4j being used by real-world projects selected from GitHub (see Section IV-A).

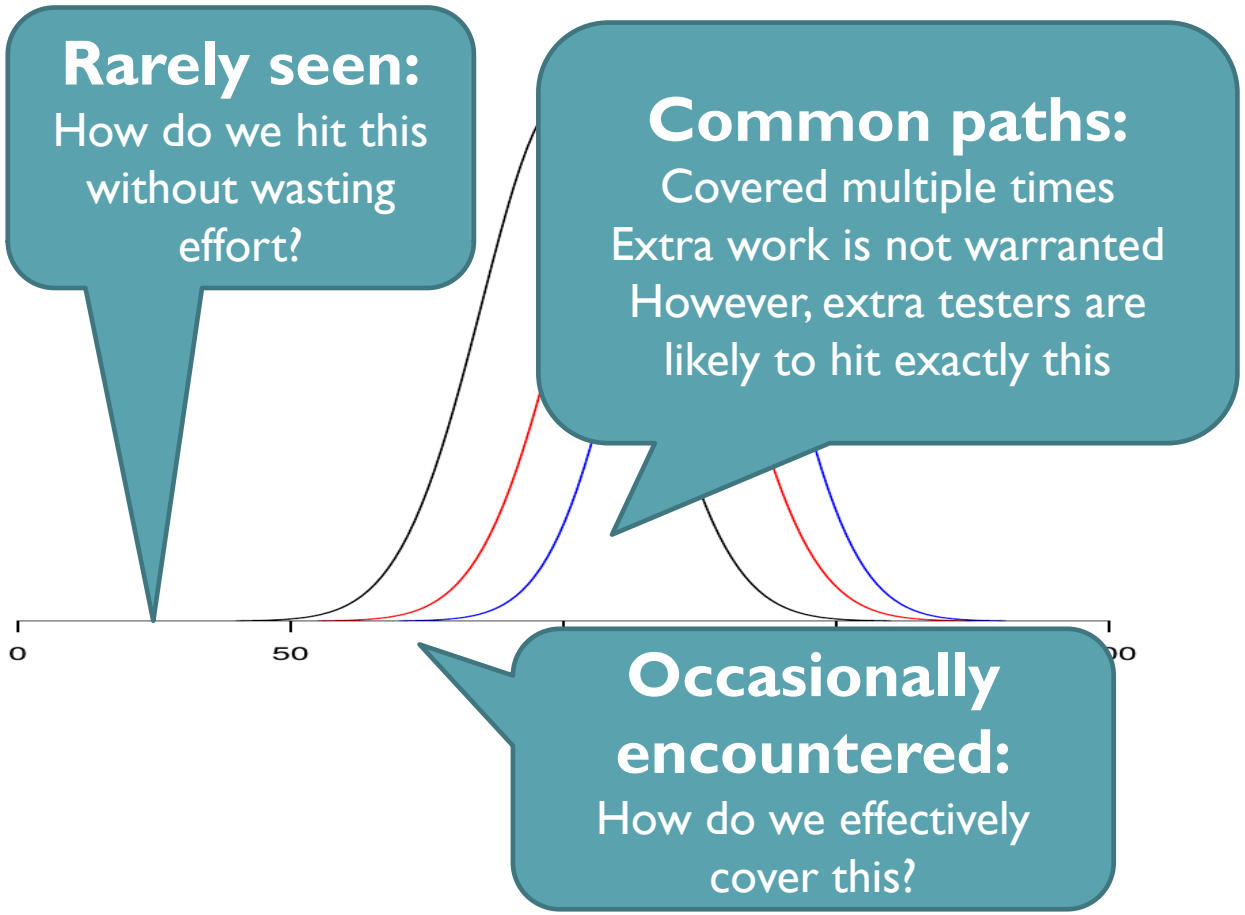
To compare the behavior of in-house tests and field executions, we used four behavioral models: two coverage-based models (statements and methods covered), a mutation-based model (killed mutants, applied only to the library benchmarks because of a limitation in the execution recording tool we used), and a temporal-invariant-based model (kTails-based invariants [6], [10]). The coverage-based models represent the state of the practice in industry today for evaluating test suite quality [1], [22], [27], [28], [30]. The mutation model is the state of the art for test suite quality evaluation, but it is used sparingly in industry [33]. Finally, we designed the invariant model to further differentiate field executions from test executions at a finer-grained level. We hypothesize that the finer-grained model is better suited for identifying behavioral differences and is thus more useful in assessing test suite quality than coverage and mutation.

The results of our behavioral comparison show that, for all four models considered, field executions are different from developer-written tests in terms of the behavior they exercise. For the four systems analyzed, on average, 6.2% of statements and 7.7% of the methods executed in the field were not executed by the tests. Moreover, mutation analysis showed that adding behavior exhibited by the field executions kills 8.6% more mutants than the behavior exhibited by the developer-written tests. Finally, we found that the invariant model identified even more sizable differences between developer-written tests and field executions: 52.6% of the invariants detected in the field were missed by the tests, on average.

We also investigated whether automated test generation could help improve the representativeness of in-house tests. To do so, we augmented the developer-written tests using EvoSuite [23], an automatic test generation tool, and analyzed whether these additional tests helped decrease the gap between behavior exercised by tests and in the field. Our results show that

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.36

LET'S FOCUS ON EXECUTION PATHS



- Need to coordinate our testing efforts
- Gap testing principles
  - Avoid **repeated**, wasteful work
  - Find ways to **hit** methods/statements/basic blocks/paths that are not covered by other methods

TWO EXAMPLES OF MORE TARGETED TESTING

Crowd-based UI testing
aiming for 100% coverage

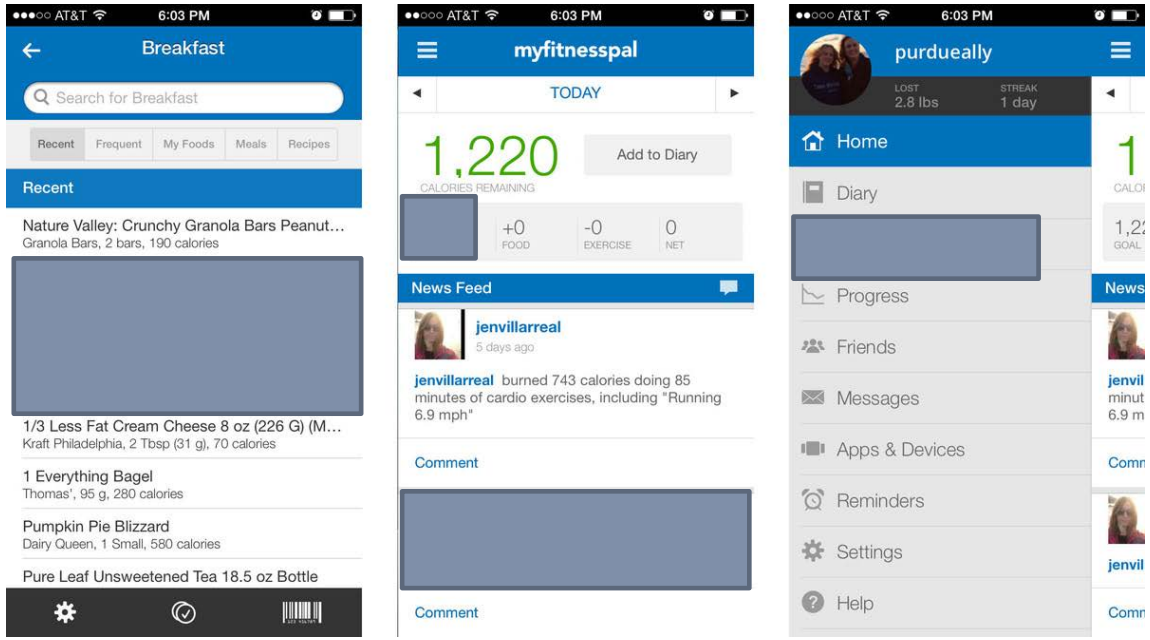
Targeted symbolic execution
aiming to hit interesting parts of the code

GAP TESTING FOR UI

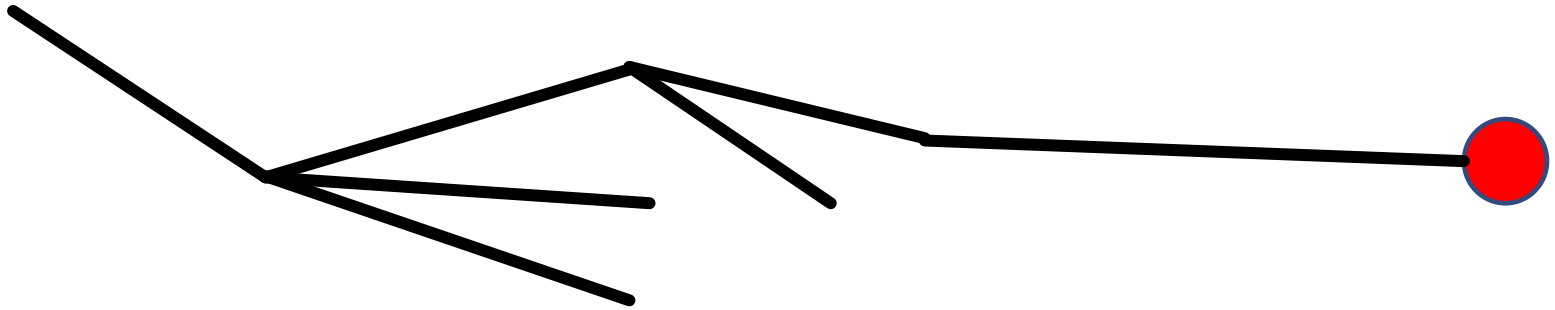
- Testing Android apps
- Goal: to have 100% UI coverage
- How to define that is sometimes a little murky
- But let's assume we have a notion of screen coverage

- Move away from covered **screens**
- By shutting off parts of the app
- Aim is to to get as close as 100% coverage by guiding crowd-sourced testers

CROWD OF TESTERS WITH THE SYSTEM GUIDING THEM TOWARD UNEXPLORED PATHS



GUIDING SYMBOLIC EXECUTION

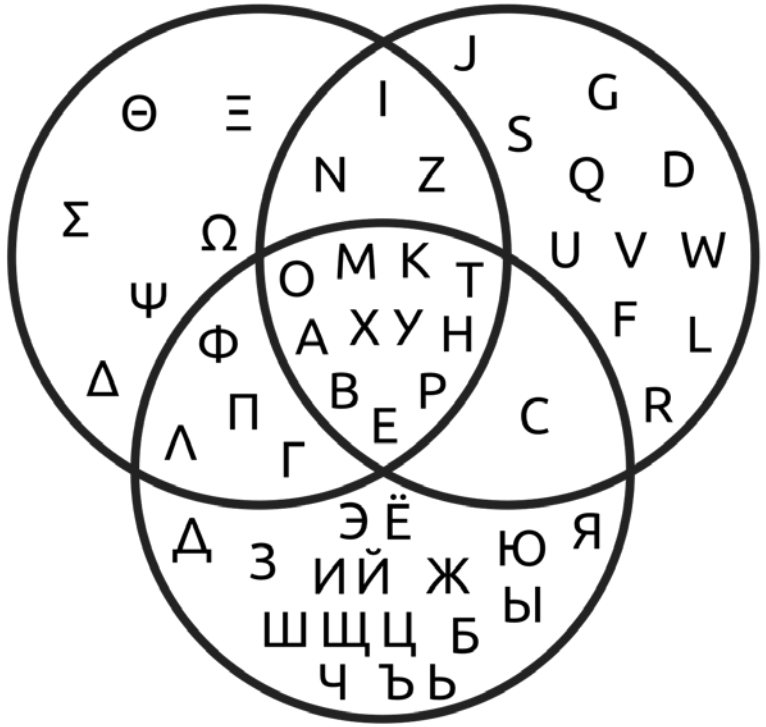


- Continue exploring the program until we find something "interesting'
- That may be a crash or an alarm from a tool such as AddressSanitizer, ThreadSanitizer, Valgrind, etc.
- Suffers from exponential blow-up issues and solver overhead

- If we instead know **what** we are looking for, for example, a method in the code we want to see called, we can direct our analysis better
- **Prioritize** branch outcomes so as to him the target

ULTIMATE VISION

- A portfolio of testing strategies that can be invoked on demand
- Deployed together to improve the ultimate outcome
- Sometimes, manual testing in the right thing, sometimes it's not
- We've seen some examples of **complimentary testing strategies**
- The list is nowhere close to exhaustive…



OPTIMIZING TESTING EFFORTS

- How to get the most out of your portfolio of testing approaches, **minimizing** the time and money spent
- It would be nice to be able to estimate the **efficacy** of a particular method and the cost in terms of time, human involvement, and machine cycles
- That's actually possible with machine learning-based predictive models, i.e. mean time to the next bug found is something we can train predictors for

THE END.

GAP TESTING: COMBINING DIVERSE TESTING STRATEGIES FOR FUN AND PROFIT

We have seen a number of testing techniques such as fuzzing, symbolic execution, and crowd-sourced testing emerge as viable alternatives to the more traditional strategies of developer-driven testing in the last decade.

While there is a lot of excitement around many of these ideas, how to property combine diverse testing techniques in order to achieve a specific goal, i.e. maximize statement-level coverage remains unclear.

The goal of this talks is to illustrate how to combine different testing techniques by having them naturally complement each other, i.e. if there is a set of methods that are not covered via automated testing, how do we use a crowd of users and direct their efforts toward those methods, while minimizing effort duplication?

Can multiple testing strategies peacefully co-exist? When combined, can they add up to a comprehensive strategy that gives us something that was impossible before, i.e. 100% test coverage?