Overview of Research (2016-2019)
Koushik Sen

Programming is hard, and it is mostly done manually. The situation will get worse for future software that demands data-driven, distributed programs that run efficiently on heterogeneous hardware. These modern programming challenges call for a new class of programming tools that will help automate programming. The goal of our research group is to develop novel and practical techniques that automate various phases of software development, such as coding, testing, and debugging. Our research methodology is to design fundamental techniques and to experimentally validate the effectiveness of these techniques on real software systems. We implement our techniques as open-source software, and they are widely used both in academic research and in commercial software products. In the next sections, we will describe the most impactful research projects we have worked on in recent years. The following are the highlights of the research in our group:

- My research group developed a programming-by-example program synthesis technique for the popular Pandas data-science library using dynamic program analysis and deep learning [MLForSystems'18, OOPSLA'19a]. This is the first technique that can handle a real-world API containing at least 119 data transformation functions. The technique, for the first time, enables any developer to build a program synthesis engine for their API without requiring any background in formal methods and program semantics. The project has been highlighted by news channels and blogs.

- Testing is the primary technique for improving the reliability of software at scale. My research group developed automated test generation techniques [ASE'18, ISSTA'18, ISSTA'19a, ICCAD'18a, ISSTA'19b, OOPSLA'19b, UsenixSecurity'20] that can find deep correctness bugs as well as pathological performance and resource usage bugs. These techniques use evolutionary algorithms, dynamic program analysis, and high-level programmer insights in the form of annotations to make fuzz testing significantly better. Fuzz testing is the most popular and widely-used testing technique used by software companies to find security vulnerabilities, correctness bugs, and software crashes in software systems such as operating systems, browsers, and web servers. Our research contributions have made fuzz testing smarter and significantly more effective for real-world software. This research has won three ACM SIGSOFT Awards and has already made its way to the software industry.

- Techniques for generating many random solutions given a set of logical constraints are used in constrained-random verification of hardware, directed automated random testing of software, and design-space exploration. My research group developed fast algorithms for sampling [ICSE'18a, ICCAD'18b, FMCAD'19] a large number of solutions from logical constraints such as Boolean formulas (SAT) and satisfiability-modulo theories (SMT) formulas. We demonstrated that these techniques generate a diverse set of test inputs at a speed, which is $100X$–$1000X$ faster than existing techniques.

- In collaboration with the "Big Code" team of Facebook, we developed techniques for code search, and recommendation using natural language queries [MAPL'18, ESEC/FSE'19] or partial code snippets as queries [OOPSLA'19c]. These techniques leverage large code corpus available from github.com and use a combination of compiler techniques, word-embedding, TF-IDF weighting, and efficient higher-dimensional search. The paper on code recommendation won an ACM SIGPLAN Distinguished Paper Award at OOPSLA'19.

- My research group developed a static bug detection technique [OOPSLA'18] for JavaScript programs, which automatically learns name-based bug detectors instead of manually writing them. A novel insight behind this technique is that learning from artificially seeded bugs yields bug detectors that are effective at finding bugs in real-world code. Applying the technique to a corpus of 150,000 JavaScript files yielded bug detectors that have a high accuracy (between 89% and 95%) are very efficient (less than 20 milliseconds per analyzed file), and reveal 102 programming mistakes (with 68% true positive rate) in real-world code.

- In addition to these major projects, we have continued working on our past projects. In the past, we developed automated testing techniques [OOPSLA'13, OOPSLA'14] for graphical user interfaces (GUI) of Android apps using machine learning and program analysis. Recently, my research group developed a technique called DETREDUCE [ICSE'18b] to significantly reduce the size of test-suites generated by such automated testing techniques. In the past, I developed a customizable dynamic analysis framework for JavaScript programs,

called Jalangi [ESEC/FSE'13]. Recently we have built several dynamic analyses on top of Jalangi: *trace typing* [ECOOP'16], a technique for automatically and quantitatively evaluating variations of a retrofitted type system on JavaScript programs; *Travioli* [ICSE'17a], a technique for detecting and visualizing data-structure traversals, for manually generating performance regression tests, and for discovering performance bugs caused by redundant traversals; *EventRaceCommander* [ICSE'17b], a technique for automated repair of event race errors in JavaScript web applications; a platform-independent dynamic taint analysis technique [TSE'18] for JavaScript. Besides, my research group developed a dynamic program analysis approach [HPCAsia'18] to optimize communication overlap in scientific applications. Using the technique, we have reduced the time spent in communication by as much as 64% for several applications that were already aggressively optimized for overlap.

I next provide detailed high-level descriptions of some of these research projects.

1 Programming using Synthesis, Search, and Recommendation

Programmers of modern software employ various approaches to tackle complex coding tasks: they perform natural language searches on Google, they ask questions on StackOverflow.com, and they use Integrated Development Environments (IDEs) such as IntelliJ or Pycharm which help them to autocomplete code. However, these approaches are far from being fully automated. I next describe the research efforts we made to automate complex coding tasks.

AutoPandas: Neural-backed Generators for Program Synthesis. Developers nowadays have to contend with a growing number of APIs for writing software. While in the long-term, they are very useful to developers, many modern APIs have an incredibly steep learning curve, due to their hundreds of functions handling many arguments, obscure documentation, and frequently changing semantics. For APIs that perform data transformations, novices can often provide an input-output example demonstrating the desired transformation but may be stuck on how to translate it to the API. A programming-by-example synthesis engine that takes such input-output examples and directly produces programs in the target API could help such novices. Such an engine presents unique challenges due to the breadth of real-world APIs, and the often-complex constraints over function arguments.

We developed a generator-based synthesis approach to synthesize Python Pandas program using input-output examples. The approach uses a *program candidate generator*. A program candidate generator is a Python program consisting of two main components: 1) core code which constructs the program in the target language, capturing the known constraints on programs in the target language, and 2) non-deterministic operators which specify the space of possible decisions at the points where more than one decision can result in a valid program in the target language. For example, suppose in the target language only previously-defined variables can be used. The program candidate generator can keep track of all the variables declared in the target program it has generated so far, and then when using new variables, use one of these names; this is the constraint in the core code. If there are multiple pre-declared variable names, all of them are valid for use, so the generator-writer can use a non-deterministic operator to select over these. The key idea of separating the constraints from these non-deterministic operators is that we can learn to control the non-determinism of the operators to speed up synthesis while leveraging the constraints to narrow the search space. To control the non-determinism effectively, we use a neural network backend for each non-deterministic operator, instead of relying on domain-specific heuristics.

We implemented this technique for the Python Pandas library in AUTOPANDAS. AUTOPANDAS supports 119 Pandas dataframe transformation functions. We offer AUTOPANDAS as a web-based service at `https://autopandas.io/`, where programmers can provide input-output examples and get the desired Pandas program. Although the project is in its early stages, we have received very positive reviews on our paper at OOPSLA'19, the premier conference at the intersection of software engineering and programming languages. This project has also gotten media coverage on blogs[1] and news channels[2] [3]. *The* AUTOPANDAS *project demonstrated that a regular developer could design a program synthesis engine without a background in formal methods and program semantics.* We are working on applying the same technique to other popular APIs, such as Numpy and TensorFlow.

[1] `https://practicalquant.blogspot.com/2019/07/riselabs-autopandas-hints-at-automation-tech-that-will-change-the` `html`

[2] `https://www.datanami.com/2019/07/08/program-synthesis-moves-a-step-closer-to-reality/`

[3] `https://www.enterpriseai.news/2019/07/08/help-coming-for-beleaguered-programmers/`

Neural Code Search. While practical program synthesis is an ambitious research goal, we have worked on projects such as natural language code search and code recommendation, which can immediately benefit developers with coding tasks. For example, developers nowadays extensively use StackOverflow.com and similar question/answer websites to search relevant code snippets using natural language queries. However, as resourceful as StackOverflow.com is, it does not contain solutions to all possible questions. Moreover, questions specific to code and APIs proprietary to a company are not discussed in StackOverflow.com, and comparable alternate Q&A forums may not be available.

We developed, in collaboration with Facebook, a technique to carry out natural language search directly over source code, i.e., without having a curated Q&A forum such as StackOverflow.com at hand. The technique uses a combination of word embedding, TF-IDF (an information retrieval technique) weighting, and efficient higher-dimensional vector similarity search. Our experimental results show that the technique can work surprisingly well in the domain of source code. With our current model, we are able to answer almost 43% of the StackOverflow.com questions we collected directly from code. We have published our results at ESEC/FSE 2019 industry track and at MAPL 2018. The system is actively used at Facebook for programming.

Aroma: Code Recommendation via Structural Code Search. I collaborated with Facebook in a project where we developed a technique that can be used to search code snippets using a partial code snippet as a query. The technique is based on the observation that developers often write code that is similar to some other existing code. A tool that allows programmers to search for similar code would help programmers to: extend partially written code snippets to completely implement necessary functionality, discover extensions to the partial code which are commonly included by other programmers, cross-check against similar code written by other programmers, or even add extra code to fix common mistakes and errors. Thus, such a structured code search is immensely valuable.

We have developed *Aroma*, a tool, and technique for code recommendation via structural code search. Aroma first indexes a huge code corpus, including thousands of open-source projects. Then, when it takes a partial code snippet as input, it searches the corpus for method bodies containing the partial code snippet, and clusters and intersects the results of the search to recommend a small set of succinct code snippets which both contain the query snippet and appear as part of several methods in the corpus. We evaluated Aroma on 2000 randomly selected queries created from the corpus, as well as 64 queries derived from code snippets obtained from StackOverflow.com, a popular website for discussing code. We implemented Aroma for 4 different languages and developed an IDE plugin for Aroma. Furthermore, we conducted a study where we asked 12 programmers to complete programming tasks using Aroma, and collected their feedback. Our results indicate that Aroma is capable of retrieving and recommending relevant code snippets efficiently. A blog post from Facebook[4] describes how they are using Aroma internally. *A paper on Aroma at OOPSLA'19 won the ACM SIGPLAN Distinguished Paper Award.*

2 Smart Fuzzing for Automated Software Testing and Security Vulnerability Discovery

Testing and debugging consumes a significant amount of time (i.e., 50% to 80% of total time) in software development. There is a dire need to automate testing and debugging. In the past, I developed an automated test generation technique called concolic testing (also known as dynamic symbolic execution.) Concolic testing has inspired the development of several industrial and academic automated testing and security tools such as PEX, SAGE, YOGI, and Vigilante at Microsoft, Apollo at IBM, ConBol and Jalangi at Samsung, CATG at NTT Laboratories, and SPLAT, BitBlaze, jFuzz, Oasis, and SmartFuzz in academia. *Our key paper on this topic recently won the ACM SIGSOFT Impact Award at ESEC/FSE'2019.* While we were working on concolic testing, another light-weight automated test generation technique called *fuzz testing* began gaining traction as it found vast numbers of bugs and security vulnerabilities in real-world software. In recent years, our research has focused on making fuzz testing smarter and more effective for real-world software. I next describe our recent contributions in the area of fuzz testing.

The recent success in fuzz testing has come from the *coverage-guided greybox fuzzing* (CGF) algorithm. Its bug-finding power has gained attention both in practice and in the research community. CGF starts with a set of user-provided seed inputs. It *mutates* the seed inputs randomly at byte-level, runs the program under test on these mutated inputs, and collects program coverage information. It saves the mutated inputs, which are *interesting* according to the coverage information—i.e., the ones that discover new coverage. It continually repeats the process but starting with

[4] https://ai.facebook.com/blog/aroma-ml-for-code-recommendation/

these interesting mutated inputs instead of the user-provided inputs. One of the leading CGF tools, American Fuzzy Lop (or simply AFL) [5] developed by Google, has found vulnerabilities in a broad array of programs (e.g., Firefox, Internet Explorer, OpenSSH, PCRE, GCC).

In spite of its success in finding correctness bugs and security vulnerabilities, fuzz testing has several limitations.

1. Fuzz testing mostly finds shallow bugs in the input-processing stages of a program—it fails to find deep bugs in the key program functionalities. For example, AFL-generated tests did not cover a large variety of packet structures in the `tcpdump` program and in the semantic checking phases of Google's Closure compiler for JavaScript.

2. Fuzz testing does not provide any mechanism with which developers can provide insights about their software.

3. Fuzz testing does not exploit the fact that real-world software systems are developed incrementally. Every time a software system is modified, fuzz testing is run from scratch, discarding the fuzzing efforts made in the previous versions of the software.

4. Fuzz testing mostly focuses on finding correctness bugs and crashes. It does not try to find performance, resource allocation, and other non-correctness bugs.

We have developed smart fuzzing techniques which have tried to address these limitations.

JQF+Zest and FuzzFactory: Human-in-the-loop Fuzzing. Creators of fuzz testing have focused their efforts on push-button techniques. The developer provides the software under test, presses the magic button, and the fuzz testing runs automatically for several hours and finds bugs and security vulnerabilities. As such fuzz testing searches the space of inputs blindly without exploiting developer insights, which could help fuzz testing to increase code coverage efficiently and to find deep bugs effectively.

We have developed domain-specific language mechanisms and associated algorithms that enable a developer of software to quickly and explicitly encode domain-specific knowledge to guide the search heuristics of coverage-guided fuzz testing. In one project, called JQF+Zest, we developed a smart generator-based fuzzing approach. Generator-based fuzzing allows users to write generator programs for producing inputs that belong to a specific type or format. At each execution, these generators produce a new test input that conforms to some structure expected by the application under test, e.g., XML documents, JSON files, network packets, or database tables. Many commercial black-box fuzzing tools, such as Peach, beSTORM, Cyberflood, and Codenomicon use generators. Generators restrict the space of inputs to be generated but could generate redundant and repetitive test inputs. Unfortunately, manually writing a good generator that avoids redundant inputs can be time-consuming, tedious, and error-prone. For example, the developers of CSmith, a generator-based fuzzing tool for C compilers, spent a few years to develop a capable generator for C programs.

Instead, in JQF+Zest, we allow the users to write simple generators which restrict the input space approximately. Such a generator usually takes less than a few hours to write. Zest then performs online tuning of the generator so that the generator generates a valid and diverse set of inputs. To do so, Zest converts random-input generators into deterministic *parametric generators*. The key insight behind Zest is that mutations in the untyped parameter domain map to structural mutations in the input domain. Zest then uses program feedback in the form of code coverage and input validity to perform *coverage-guided fuzzing* of the parameter space. JQF+Zest has enabled the discovery of over 40 previously unknown bugs (including four CVEs) in widely used open-source software, including bugs in the key functionalities of a couple of JavaScript compilers and build tools. A couple of papers describing the technique appeared at ISSTA'19. *One of the papers won an ACM SIGSOFT Distinguished Artifact Award, and the other won an ACM SIGSOFT Tool Demonstration Award.* Since its public release, JQF+Zest has been adopted by several companies, such as Netflix, ModZero AG, for effective testing of their software. Zest has also been commercialized by a startup (Fuzzit.dev) that provides cloud-based dynamic analysis as part of continuous integration. More recently, we found that the tuning of the generator can be done more effectively using machine-learning algorithms. We are actively working on this approach.

In order to address the limitation of CGF that it only focuses on finding correctness bugs, recently researchers have introduced various specializations CGF for different domain-specific testing goals, such as finding performance

[5] http://lcamtuf.coredump.cx/afl/

bottlenecks (e.g., our PerfFuzz tool), finding memory bottlenecks, generating valid inputs, handling magic-byte comparisons, and testing patches generated during incremental software development. Each such solution can require non-trivial implementation effort from six months to a year. We observed that many of these domain-specific solutions follow a common solution pattern. We developed FuzzFactory, a framework, and a domain-specific language for rapidly developing domain-specific fuzzing applications without requiring changes to the core algorithms of CGF. FuzzFactory allows users to specify the collection of dynamic domain-specific feedback during test execution, as well as how such feedback should be aggregated. FuzzFactory uses this information to selectively save intermediate inputs, called waypoints, to augment coverage-guided fuzzing. Such waypoints always make progress towards domain-specific multi-dimensional objectives. We instantiated six domain-specific fuzzing applications using FuzzFactory: three re-implementations of prior work and three novel solutions, and evaluate their effectiveness on benchmarks from Google's fuzzer test suite. We showed how multiple domains could be composed to perform better than the sum of their parts. For example, we combined instrumentation of comparison operations (e.g., memcmp, strcmp) with that of dynamic memory allocations (e.g., malloc, calloc) to automatically synthesize LZ4 bombs and PNG bombs: tiny inputs that lead to excessively large allocations beyond the maximum theoretical compression ratio in libarchive and libpng respectively (both are widely used C libraries). A paper describing the technique will appear at OOPSLA'19. A domain-specific fuzzer that we built in collaboration with Samsung for ARM TrustZone [USENIX-Security20] unearthed more than 40 security vulnerabilities across Trusted Execution Environments deployed by Qualcomm, Samsung, Linaro, and Trustonic on over 2 billion Android devices. A paper based on describing the results will appear at USENIX Security'20, one of the top two conferences in computer security.

PerfFuzz: Automatically Generating Pathological Inputs. Most CGF techniques developed so far have focused on finding correctness bugs and crashes. We developed a novel CGF technique, called PerfFuzz, that focuses on finding performance problems in software which can arise unexpectedly when programs are provided with inputs that exhibit worst-case behavior. PerfFuzz automatically generates inputs that exercise pathological behavior across program locations, without any domain knowledge. Unlike previous CGF approaches that attempt to maximize the coverage of various program locations, PerfFuzz uses multi-dimensional feedback and independently maximizes execution counts for all program locations. This enables PerfFuzz to (1) find a variety of inputs that exercise distinct hot spots in a program, and (2) generate inputs with higher total execution path length than previous approaches by escaping local maxima. PerfFuzz is also effective at generating inputs that demonstrate algorithmic complexity vulnerabilities. We have implemented PerfFuzz on top of AFL, a popular coverage-guided fuzzing tool, and evaluated PerfFuzz on several real-world C programs typically used in the fuzzing literature. We found that PerfFuzz outperforms prior work by generating inputs that exercise the most-hit program branch $5\times$ to $69\times$ times more, and result in $1.9\times$ to $24.7\times$ longer total execution paths. A paper describing the technique won an *ACM SIGSOFT Distinguished Paper Award* at ISSTA'18, the top conference in software testing and analysis.

FairFuzz: A targeted mutation strategy for increasing fuzz testing coverage. A limitation of CGF is that it cannot find deep bugs since it simply does not cover large regions of code. We developed a two-pronged approach to increase the coverage achieved by CGF. First, the approach automatically identifies branches exercised by few CGF-produced inputs (*rare* branches), which often guard code that is empirically hard to cover by naïvely mutating inputs. The second part of the approach is a novel *mutation mask* creation algorithm, which allows mutations to be biased towards producing inputs hitting a given rare branch. This mask is dynamically computed during testing and can be adapted to other testing targets. We implemented this approach on top of AFL in a tool named FAIRFUZZ. We conducted an evaluation on real-world programs against state-of-the-art versions of AFL. We found that on these programs FAIRFUZZ achieves high branch coverage at a faster rate than state-of-the-art versions of AFL. Besides, on programs with nested conditional structure, it achieves sustained increases in branch coverage after 24 hours (average 10.6% increase). A paper based on this technique appeared at ASE'19, a top-tier conference in software engineering.

RFuzz: Coverage-Directed Fuzz Testing of RTL on FPGAs. Dynamic verification is widely used to increase confidence in the correctness of RTL circuits during the pre-silicon design phase. Despite numerous attempts over the last decades to automate the stimuli generation based on coverage feedback, *coverage directed test generation (CDG)* has not found the widespread adoption that one would expect. Based on new ideas from the software testing community around coverage-guided fuzz testing, we developed a new approach to the CDG problem, which requires minimal setup and takes advantage of FPGA-accelerated simulation for rapid testing. We provide test input and coverage defi-

nitions that allow fuzz testing to be applied to RTL circuit verification. Besides, we proposed and implemented a series of transformation passes that make it feasible to reset arbitrary RTL designs quickly, a requirement for deterministic test execution. We have publicly released RFuzz, fully-featured implementation of our testing methodology. An empirical evaluation of RFuzz shows promising results on archiving coverage for a wide range of different RTL designs ranging from communication IPs to an industry scale 64-bit CPU. Our ICCAD'18 paper describes the technique.

3 Fast Sampling of Solutions from Logical Constraints

In automated software and hardware testing and verification, a fundamental technical challenge is to generate a set of random solutions to a given set of constraints. If the constraints represent the preconditions that a test input needs to satisfy, sampling solutions to the constraints allows the generation of a large number of stimuli that can be used to exercise the program or circuit under test. This idea can be combined with different testing and verification techniques, such as constrained-random verification, dynamic symbolic execution, and fuzz testing. By generating a large number of high-quality inputs, we can improve coverage and help uncover new bugs.

However, despite its importance, the problem of sampling a diverse set of solutions efficiently is still challenging today. There are some formal approaches that can provably sample from one desired distribution but are very expensive in practice. Other more heuristic approaches tend to generate biased samples or do not perform well on large and complex formulas.

We have developed QuickSampler, a technique for sampling millions of solutions to an SAT formula. QuickSampler is orders of magnitude faster than previous techniques and allows the generation of a diverse set of solutions. Given an SAT formula as a constraint, QuickSampler works as follows. QuickSampler first finds a random assignment to the variables of the formula. Such an assignment may not satisfy the formula. QuickSampler then uses a MAX-SAT solver to find a solution to the formula that is close to the random satisfying assignment. It then flips the value of each variable in the solution and again uses MAX-SAT to find a close solution of the formula. The diff between the original solution and the modified solution is called *an atomic mutation*. For each variable in the formula, this generates at most one atomic mutation. A small bounded number of such atomic mutations is then applied to the original solution to generate a potential new solution. We expect that such combinations of small atomic mutations will often result in a new valid random solution. This is because each atomic mutation identifies a small set of variables that are dependent on each other. Whereas the variables from two different atomic mutations are often independent of each other. Therefore, if two such atomic mutations are combined and applied to the original solution, then the resulting solution will often be a solution of the formula. The entire process will be repeated several times. Since QuickSampler creates many of solutions by simply combining atomic mutations, it avoids making frequent solver calls—which are often the bottleneck in solver-aided techniques. This, in turn, results in a quick generation of lots of random and diverse solutions. Our evaluation of QuickSampler on large real-world benchmarks shows that it can produce unique, valid solutions orders of magnitude faster than other state-of-the-art sampling tools. We also empirically verified that the distribution of solutions is close to uniform, which was our target distribution. A paper based on this technique has appeared at ICSE'18, the top conference in software engineering.

SMTSampler is an extension of the QuickSampler technique that allows efficient sampling of solutions from Satisfiability Modulo Theories (SMT) constraints. This is important since many constraints found in practical applications are more naturally represented by SMT formulas that include theories such as arrays and bit-vectors. By working over SMT formulas directly, without encoding them into Boolean (SAT) constraints, SMTSampler is able to sample solutions more efficiently, and also achieve a better coverage of the constraint space. In our evaluation, we defined a new notion of coverage that better captures the diversity of SMT solutions, and showed that SMTSampler helps improve this coverage. SMTSampler works similarly to QuickSampler, leveraging a small number of calls to a constraint solver in order to generate up to millions of stimuli. However, SMTSampler can sample random solutions from large and complex SMT formulas with bit-vectors, arrays, and uninterpreted functions. Our evaluation on a large set of complex industrial SMT benchmarks shows that SMTSampler can handle a larger class of SMT problems, outperforming QuickSampler in the number of samples produced and the coverage of the constraint space. A paper on SMTSampler has appeared in ICCAD'18, one of the two top conferences in computer-aided design.

GuidedSampler is an extension of SMTSampler that allows coverage-guided sampling of SMT solutions by letting the user specify the desired set of coverage points that will shape the distribution of solutions. This is important because most current sampling techniques lack a problem-specific notion of coverage, considering only general goals such as uniform distribution, as in QuickSampler, or the coverage of the SMT formula, as in SMTSampler. However,

many applications would benefit from a more specific coverage definition, for example, based on coverage points specified by the hardware designer. Our tool GuidedSampler enables this greater flexibility by using the specified coverage points to guide the sampling algorithm into generating solutions from diverse coverage classes. And even for applications where a general notion of coverage suffices, our evaluation shows that the coverage-guided sampling approach is more effective at achieving this desired coverage. GuidedSampler is thus able to efficiently generate high-quality stimuli for constrained-random verification, by sampling solutions to SMT constraints that also cover a large number of user-defined coverage classes. A paper on GuidedSampler has appeared at FMCAD'19, one of the two top conferences in computer-aided design.

4 DeepBugs: A Learning Approach to Name-based Bug Detection

Natural language elements in program source code, e.g., the names of variables and functions, convey useful information. However, most existing bug detection tools ignore this information and therefore miss some classes of bugs. The few current name-based bug detection approaches reason about names on a syntactic level and rely on manually designed and tuned algorithms to detect bugs. We have developed DeepBugs, a deep-learning-based approach to name-based bug detection, which reasons about names based on a semantic representation and which automatically learns bug detectors instead of manually writing them. We formulate bug detection as a binary classification problem and train a classifier that distinguishes correct from incorrect code. To address the challenge that effectively learning a bug detector requires examples of both correct and incorrect code, we create likely incorrect code examples from an existing corpus of code through simple code transformations. A novel insight learned from our work is that learning from artificially seeded bugs yields bug detectors that are effective at finding bugs in real-world code. We implemented our idea into a framework for learning-based and name-based bug detection. Three bug detectors built on top of the framework detect accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations. Applying the approach to a corpus of 150,000 JavaScript files yields bug detectors that have a high accuracy (between 89% and 95%) are very efficient (less than 20 milliseconds per analyzed file) and reveal 102 programming mistakes (with 68% true positive rate) in real-world code. A paper describing the technique appeared at OOPSLA'18.

5 DetReduce: Minimizing Android GUI Test Suites for Regression Testing

In recent years, several automated GUI testing techniques for Android apps have been proposed, including our tools SwiftHand and EventBreak. These tools have been shown to be effective in achieving good test coverage and in finding bugs without human intervention. Being automated, these tools typically run for a long time (say, for several hours), either until they saturate test coverage or until a testing time budget expires. Thus, these automated tools are not good at generating concise regression test-suites that could be used for testing in the incremental development of the apps and in regression testing.

We developed a heuristic technique that helps create a small regression test suite for an Android app from a large test suite generated by an automated Android GUI testing tool. The key insight behind our technique is that if we can identify and remove some common forms of redundancies introduced by existing automated GUI testing tools, then we can drastically lower the time required to minimize a GUI test suite. We have implemented our algorithm in a prototype tool called DetReduce. We applied DetReduce to several Android apps and found that DetReduce reduces a test-suite by an average factor of $16.9\times$ in size and $14.7\times$ in running time. We also found that for a test suite generated by running SwiftHand and a randomized test generation algorithm for 8 hours, DetReduce minimizes the test suite in an average of 14.6 hours. A paper describing the technique appeared at ICSE'18.

6 Trace Typing: An Approach for Evaluating Retrofitted Type Systems.

Recent years have seen growing interest in the retrofitting of type systems onto dynamically-typed programming languages, such as JavaScript and Python, in order to improve type safety, programmer productivity or performance. In such cases, type system developers must strike a delicate balance between disallowing certain coding patterns to keep the type system simple, or including them at the the expense of additional complexity and effort. Thus far, the

process for designing retrofitted type systems has been largely ad hoc, because evaluating multiple variations of a type system on large bodies of existing code is a significant undertaking.

In collaboration with Samsung, we have developed *trace typing*, a framework on top of Jalangi (a customizable dynamic analysis framework for JavaScript programs which we previously developed in our research group) for automatically and quantitatively evaluating variations of a retrofitted type system on large codebases. The trace typing approach involves gathering traces of program executions, inferring types for instances of variables and expressions occurring in a trace, and merging types according to merge strategies that reflect specific (combinations of) choices in the source-level type system design space. We evaluated trace typing through several experiments. We compared several existing variations of type systems retrofitted onto JavaScript, measuring the number of program locations with type errors in each case on a suite of over fifty thousand lines of JavaScript code. We also used trace typing to validate and guide the design of a new retrofitted type system that enforces a fixed object layout for JavaScript objects. Finally, we leveraged the types computed by trace typing to automatically identify tag tests—dynamic checks that refine a type—and examined the variety of tests identified. This work appeared in Europe's leading conference on programming languages ECOOP'16.

7 ChocoPy

In 2018, we designed ChocoPy for teaching undergraduate Compilers course at UC Berkeley. ChocoPy is a statically typed, restricted subset of Python 3.6. ChocoPy is fully specified using formal descriptions of syntax, typing, and operational semantics. Students taking CS164 learn to reason about such formalisms, consider alternatives, and extend them to support additional features of Python. Students work in teams to develop a full compiler for ChocoPy, targeting the RISC-V architecture. We provide students a language reference manual, a RISC-V implementation guide, and a skeleton code for developing a modular ChocoPy compiler in Java. Additional resources include an instructor-provided reference compiler, a web-based ChocoPy IDE, a web-based RISC-V simulator with step-through assembly debugging, and an auto-grader. The ChocoPy compiler project is developed in three modular stages that can be tested and graded independently of each other. The ChocoPy project is designed to be portable. All ChocoPy resources are available at `https://chocopy.org`.

ChocoPy's syntax is a simplified subset of Python syntax. One huge advantage of this fact is that we get syntax highlighting for free in almost every code editor. The semantics of ChocoPy programs have been carefully designed so that the execution of a valid ChocoPy program results in the same observable output as the execution of the same program in a standard Python interpreter. We soon discovered an additional advantage of basing a compiler project around a type-safe subset of a highly dynamic language such as Python. On non-trivial benchmarks, student-implemented compilers can easily outperform the official Python implementation!

A paper on ChocoPy appeared at the 2019 ACM Symposium on *SPLASH-E*, a venue for research on education in programming languages and software engineering. An article on ChocoPy's success story was published by TechRepublic, an online media hub[6]. ChocoPy also made it to the front-page of the popular HackerNews portal[7]. I am currently teaching the third edition of a ChocoPy-based course in Fall 2019, and we are continuously improving the project based on student feedback. The ChocoPy project has created a renewed interest in the compilers course at UC Berkeley.

References

[MLForSystems'18] Rohan Bavishi, Caroline Lemieux, Neel Kant, Roy Fox, Koushik Sen, and Ion Stoica. Neural Inference of API Functions from InputOutput Examples. In *Workshop on ML for Systems at NeurIPS*. 2018.

[OOPSLA'19a] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: Neural-Backed Generators for Program Synthesis. *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (OOPSLA'19)*, 3(OOPSLA), October 2019.

[6] `https://techrepublic.com/article/how-chocopy-uses-python-and-risc-v-to-teach-compiler-creation`
[7] `https://news.ycombinator.com/item?id=20957420`

[ASE'18] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *33rd IEEE/ACM International Conference on Automated Software Engineering*. 2018.

[ISSTA'18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically Generating Pathological Inputs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, 2018. ACM SIGSOFT Distinguished Paper Award.

[ISSTA'19a] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'19. 2019. doi:10.1145/3293882.3330576. ACM SIGSOFT Distinguished Artifact Award.

[ICCAD'18a] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *International Conference On Computer Aided Design*. 2018.

[ISSTA'19b] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19. 2019. doi:10.1145/3293882.3339002. ACM SIGSOFT Tool Demonstration Award.

[OOPSLA'19b] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (OOPSLA'19)*, 3(OOPSLA):174:1–174:29, October 2019.

[UsenixSecurity'20] Lee Harrison, Hayawardh Vjayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 2020.

[ICSE'18a] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient Sampling of SAT Solutions for Testing. In *40th International Conference on Software Engineering (ICSE'18)*. IEEE, 2018.

[ICCAD'18b] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints. In *International Conference On Computer Aided Design*. 2018.

[FMCAD'19] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. GuidedSampler: Coverage-guided Sampling of SMT Solutions. In *Formal Methods in Computer Aided Design (FMCAD'2019)*. 2019.

[MAPL'18] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL'18)*. 2018.

[ESEC/FSE'19] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When Deep Learning Met Code Search. In *Industry Track of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, pages 964–974. ACM, 2019.

[OOPSLA'19c] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code Recommendation via Structural Code Search. *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (OOPSLA'19)*, 3(OOPSLA), October 2019. ACM SIGPLAN Distinguished Paper Award.

[OOPSLA'18] Michael Pradel and Koushik Sen. DeepBugs: A Learning Approach to Name-based Bug Detection. In *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (OOPSLA'18)*. 2018.

[OOPSLA'13] Wontae Choi, George Necula, and Koushik Sen. Guided GUI Testing of Android Applications with Minimal Restart and Approximate Learning. In *Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'13)*, pages 623–640. ACM, 2013.

[OOPSLA'14] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation. In *Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'14)*. ACM, 2014.

[ICSE'18b] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. Minimizing Android GUI Test Suites for Regression Testing. In *40th International Conference on Software Engineering (ICSE'18)*. IEEE, 2018.

[ESEC/FSE'13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*. ACM, 2013.

[ECOOP'16] Esben Andreasen, Colin S. Gordon, Satish Chandra, Manu Sridharan, Frank Tip, and Koushik Sen. Trace Typing: An Approach for Evaluating Retrofitted Type Systems. In *European Conference on Object-Oriented Programming (ECOOP'16)*. 2016.

[ICSE'17a] Rohan Padhye and Koushik Sen. TRAVIOLI: A Dynamic Analysis for Detecting Data-Structure Traversals. In *39th International Conference on Software Engineering (ICSE'17)*. IEEE, 2017.

[ICSE'17b] Christoffer Quist Adamsen, Anders Mller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing Event Race Errors by Controlling Nondeterminism. In *39th International Conference on Software Engineering (ICSE'17)*. IEEE, 2017.

[TSE'18] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2018. doi:10.1109/TSE.2018.2878020.

[HPCAsia'18] Emmanuelle Saillard, Koushik Sen, Wim Lavrijsen, and Costin Iancu. Maximizing Communication Overlap with Dynamic Program Analysis. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, Chiyoda, Tokyo, Japan, January 28-31, 2018*, pages 1–11. 2018. Best Paper Finalist.