**A Framework for Automated Test Mocking of Mobile Apps**

Most software systems interact with their environment extensively. This is especially true for mobile apps, whose behavior often depends on sensors, external services, and inter-process communications. These interactions can complicate testing activities, as test cases may need a complete environment to be executed. They can also cause issues such as flakiness, for example when the environment behaves in non-deterministic ways. For these reasons, it is common to create test mocks that can eliminate the need for (part of) the environment to be present during testing. Manual mock creation, however, can be extremely time consuming and error-prone. Moreover, the generated mocks can typically only be used in the context of the specific tests for which they were created. To address these issues, we propose MOKA, a general framework for collecting and generating reusable test mocks in an automated way. MOKA leverages the ability to observe a large number of interactions between an application and its environment and uses an iterative approach to generate mocks with different reusability characteristics—advanced mocks generated through program synthesis and basic record-replay-based mocks. In this paper, we describe the new ideas behind MOKA, its main characteristics, a preliminary study, and a set of possible applications that would benefit from our framework.

**A Hybrid Analysis to Detect Java Serialisation Vulnerabilities**

Serialisation related security vulnerabilities have recently been reported for numerous Java applications. Since serialisation presents both soundness and precision challenges for static analysis, it can be difficult for analyses to precisely pinpoint serialisation vulnerabilities in a Java library. In this paper, we propose a hybrid approach that extends a static analysis with fuzzing to detect serialisation vulnerabilities. The novelty of our approach is in its use of a heap model to direct fuzzing for vulnerabilities in Java libraries. The advantage is that the analysis guides fuzzing to quickly and effectively produce results, which may also automatically validate static analysis reports.

**BugPecker: Locating Faulty Methods with Deep Learning on Revision Graphs**

Given a bug report of a project, the task of locating the faults of the bug report is called fault localization. To help programmers in the fault localization process, many approaches have been proposed, and have achieved promising results to locate faulty files. However, it is still challenging to locate faulty methods, because many methods are short and do not have sufficient details to determine whether they are faulty. In this paper, we present BugPecker, a novel approach to locate faulty methods based on its deep learning on revision graphs. Its key idea includes (1) building revision graphs and capturing the details of past fixes as much as possible, and (2) discovering relations inside our revision graphs to expand the details for methods and calculating various features to assist our ranking. We have implemented BugPecker, and evaluated it on three open source projects. The early results show that BugPecker achieves a mean average precision (MAP) of 0.263 and mean reciprocal rank (MRR) of 0.291, which improve the prior approaches significantly. For example, BugPecker improves the MAP values of all three projects by five times, compared with two recent approaches such as DNNLoc-m and BLIA 1.5.

#### ****Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative About Undefined Behaviour****

Methods for randomized testing of compilers to find miscompilation bugs typically require a way to generate programs that are free from undefined behaviour (UB). Tools such as Csmith achieve UB-freedom by heavily restricting the form of generated programs. This leads to highly idiomatic programs, and we hypothesise that this limits the thoroughness with which compilers are tested. Our idea is that researchers should investigate ways to generate less restricted programs that are still UB-free—programs that get closer to the edge of undefined behaviour, but that do not quite cross the edge. We present experiments investigating one instance of idea via a prototype tool, CsmithEdge, that uses a simple dynamic analysis to detect where Csmith has been too conservative in its use of “safe math” wrappers that guarantee UB-freedom for arithmetic operations, eliminating redundant wrappers. By reducing the use of safe math wrappers, CsmithEdge was able to discover two new miscompilation bugs in GCC that could not be found via intensive testing using regular Csmith, as well as achieving substantial differences in code coverage on GCC compared with regular Csmith.

#### ****Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing****

Software testing is an important and time-consuming task that is often done manually. In the last decades, researchers have come up with techniques to generate input data (e.g., fuzzing) and automate the process of generating test cases (e.g., search-based testing). However, these techniques are known to have their own limitations: search-based testing does not generate highly-structured data; grammar-based fuzzing does not generate test case structures. To address these limitations, we combine these two techniques. By applying grammar-based mutations to the input data gathered by the search-based testing algorithm, it allows us to co-evolve both aspects of test case generation. We evaluate our approach by performing an empirical study on 20 Java classes from the three most popular JSON parsers across multiple search budgets. Our results show that the proposed approach on average improves branch coverage for JSON related classes by 15% (with a maximum increase of 50%) without negatively impacting other classes.

#### ****Making Fair ML Software using Trustworthy Explanation****

Machine learning software is being used in many applications (finance, hiring, admissions, criminal justice) having a huge social impact. But sometimes the behavior of this software is biased and it shows discrimination based on some sensitive attributes such as sex, race etc. Prior works concentrated on finding and mitigating bias in ML models. A recent trend is using instance-based model agnostic explanation methods such as LIME[1] to find out bias in the model prediction. Our work concentrates on finding shortcomings of current bias measures and explanation methods. We show how our proposed method based on K nearest neighbors can overcome those shortcomings and find the underlying bias of black-box models. Our results are more trustworthy and helpful for the practitioners. Finally, We describe our future framework combining explanation and planning to build fair software.

#### ****On Benign Features in Malware Detection****

This paper investigates the problem of classifying Android applications into malicious and benign. We analyze the performance of a popular malware detection tool, Drebin, on malware datasets commonly used in an academic setup and show that the high detection accuracy often stems from learning benign rather than malicious indicators. That, effectively, turns the malware detection tools into benign app detectors. Yet, in practice, malware samples are often larger and can exhibit many behaviors similar to those of benign apps. Under such a challenging setup, looking for benign indicators becomes ineffective and the ability of the tools to detect malware degrades substantially.

In this paper, we propose an approach for identifying malicious portions of an app in the presence of numerous benign features, effectively eliminating “noise” and focusing the detection on truly malicious indicators. We also propose a novel metric estimating the “reasons” for correct malware classification, i.e., whether it is based on the presence of malicious indicators or the absence of benign ones. We show that our proposed approach is effective in both increasing the “standard” classification accuracy and in making more “justifiable” classification decisions.

#### ****Proving Termination by k-Induction****

We propose a novel approach to proving the termination of imperative programs by k-induction. By our approach, the termination proving problem can be formalized as a k-inductive invariant synthesis task. On the one hand, k-induction uses weaker invariants than that required by the standard inductive approach. On the other hand, the base case of k-induction, which unrolls the program, can provide stronger pre-condition for invariant synthesis. As a result, the termination arguments of our approach can be synthesized more efficiently than the standard method. We implement a prototype of our k-inductive approach. The experimental results show the significant effectiveness and efficiency of our approach.

#### ****SRRTA: Regression Testing Acceleration via State Reuse****

Regression testing is widely recognized as an important but time-consuming process. In the literature, researchers have put dedicated efforts in test selection, reduction, and prioritization, to alleviate this cost issue. These techniques share the commonality that they improve regression testing by optimizing the execution of the whole test suite. In this paper, we attempt to accelerate regression testing from a totally new perspective, i.e., skipping some execution of a new program by reusing program states of an old program. Following this intuition, we propose a state-reuse based acceleration approach SRRTA, which consists of state storage and loading. With the former, SRRTA collects some program states during the execution of an old version through three heuristic-based storage strategies; with the latter, SRRTA loads the stored program states with efficiency optimization strategies. Finally, we conduct a preliminary study on \emph{commons-math} and find that SRRTA reduces 80.3% of the regression testing time, indicating it is very promising in accelerating regression testing.