Deploying Search Based Software Engineering with Sapienz at Facebook

Nadia Alshahwan, Xinbo Gao, Mark Harman(✉), Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin

Facebook, London, UK
{markharman,kemao}@fb.com

Abstract. We describe the deployment of the Sapienz Search Based Software Engineering (SBSE) testing system. Sapienz has been deployed in production at Facebook since September 2017 to design test cases, localise and triage crashes to developers and to monitor their fixes. Since then, running in fully continuous integration within Facebook's production development process, Sapienz has been testing Facebook's Android app, which consists of millions of lines of code and is used daily by hundreds of millions of people around the globe.

We continue to build on the Sapienz infrastructure, extending it to provide other software engineering services, applying it to other apps and platforms, and hope this will yield further industrial interest in and uptake of SBSE (and hybridisations of SBSE) as a result.

1 Introduction and Background

Sapienz uses multi-objective Search Based Software Engineering (SBSE) to automatically design system level test cases for mobile apps [49]. We explain how Sapienz has been deployed into Facebook's central production continuous integration system, Phabricator, how it collaborates with other Facebook tools and technologies: the FBLearner Machine Learning Infrastructure [38], the One World Mobile Platform [20] and Infer, the Facebook Static Analysis tooling [13]. We also outline some open problems and challenges for the SBSE community, based on our experience.

Our primary focus for this paper is the deployment of the SBSE technology, rather than the SBSE aspects themselves. We believe the deployment throws up interesting new challenges that we would like to surface and share with the SBSE community as one potential source of stimulus for on-going and future

This paper was written to accompany the keynote by Mark Harman at the 10^{th} Symposium on Search-Based Software Engineering (SSBSE 2018), Montpellier September 8–10, 2018. The paper represents the work of all the authors in realising the deployment of search based approaches to large-scale software engineering at Facebook. Author name order is alphabetical; the order is thus not intended to denote any information about the relative contribution of each author.

T. E. Colanzi and P. McMinn (Eds.): SSBSE 2018, LNCS 11036, pp. 3–45, 2018.
https://doi.org/10.1007/978-3-319-99241-9_1

work on deployable SBSE. The details of the SBSE algorithms, the SBSE approach adopted by Sapienz and its evaluation against state-of-the-art and state-of-practice automated mobile test techniques can be found elsewhere [49].

Sapienz augments traditional Search Based Software Testing (SBST) [33,55], with systematic testing. It is also designed to support crowd-based testing [48] to enhance the search with 'motif genes' (patterns of behaviour pre-defined by the tester and/or harvested from user journeys through a system under test [50]).

Many SBSE testing tools, both more and less recent instances [26,45,71], tend to focus on unit level testing. By contrast, Sapienz is a system-level testing tool, in which the representation over which we design a test is the event sequence at GUI level, making Sapienz approach more similar to the Exsyst[1] approach [30] than it is to other more unit-testing orientated approaches. Such system-level SBSE testing has been found to reduce the false positives that plague automated test data generation at the unit level [30]. However, it does pose other challenges to deployment, particularly on a mobile platform at scale, as we shall discuss.

In September 2017 the Sapienz system first went live at Facebook, deployed on top of Facebook's FBLearner machine learning infrastructure [38] and drawing on its One World Platform which is used to supply mobile devices and emulators [20]. Since then, Sapienz has run continuously within Facebook's Continuous Integration platform, testing every diff that lands into the Facebook Android app's code base. A 'diff' refers to a code commit submitted to the repository by an engineer. Since February 2018, Sapienz has additionally been continuously testing every smoke build of diffs, as they are submitted for review.

The Facebook Android app is one of the largest Android apps available and is one of the most widely used apps in production at the time of writing. It supports social networking and community building for hundreds of millions of users world wide. Since April 2018, we extended Sapienz to test the Messenger app for Android, another large and popular app, with hundreds of millions of users world wide, who use it to connect and communicate with people and organisations that matter to them.

These two apps are not the only communications products available, nor the only large apps for which testing is needed. Nevertheless, the challenges of scale and deployment are likely to be similar for other apps, and so lessons learned will hopefully generalise to other apps and also to the issues associated with SBSE deployment into many other Continuous Integration scenarios.

We are currently extending Sapienz to iOS and to other apps in the Facebook app family. With these extensions we aim to widen the benefits of automated test design from the hundreds of millions who currently use the Android Facebook social media app to the entire Facebook community. At the time of writing, this community numbers more than 2.2 billion monthly active users world wide, thereby representing a significant route to research impact for software engineering scientific research communities.

The 'debug payload' delivered to the engineer by Sapienz, when it detects a crashing input sequence, includes a stack trace, various reporting and

[1] http://exsyst.org/.

cross-correlation information and a crash-witness video (which can be walked through under developer control and correlated to activities covered), all of which combine to ensure a high fix rate, an approximate lower-bound on which is 75% at the time of writing.

Determining a guaranteed true fix is a challenging problem in itself, so we use a conservative mechanism to give a lower bound, as explained in Sect. 3.2. The true fix rate is likely higher, with remaining crashes reported being believed to be either unimportant to users or false positives (See Sect. 7.8).

A video presentation, by Ke Mao and Mark Harman from the Sapienz team, recorded at Facebook's F8 developer conference on 1st May 2018 is available[2]. There is also a video of Ke Mao's presentation at the conference FaceTAV 2017[3], on the Sapienz deployment at Facebook[4].

SBSE is now starting to achieve significant uptake in the industrial and practitioner sectors and consequent real world impact. This impact is being felt, not only at Facebook, but elsewhere, such as Ericsson [2], Google [77], and Microsoft [70] as well as the earlier pioneering work of Wegener and his colleagues at Daimler [72]. SBSE has been applied to test embedded systems software in the Automotive domain, for example with industrial case studies involving the Ford Motor Company, Delphi Technologies and IEE S.A. [1,54,61], and the space domain at SES S.A. [69]. It has also been deployed in the financial services sector for security testing [42] and in the maritime sector at Kongsberg Gruppen for stress testing [3].

Such industrial applications allow us to evaluate in both laboratory and in industrial/practitioner settings [52]. As has been argued previously [32], both forms of evaluation are important in their own right and each tends to bring to light complementary evaluation findings. That is, laboratory evaluations tend to be more controlled, but less realistic, while industrial/practice evaluations tend to be more realistic, but less controlled.

However, despite the widespread uptake of SBSE, indeed arguably because of it, we now have an even richer set of exciting scientific problems and intellectual challenges to tackle. In our own deployment work in the Sapienz Team at Facebook, we encountered many open problems, some of which are currently the subject of some scientific study, but some of which appeared to be largely overlooked in the literature.

As we moved from the research prototype of Sapienz to a fully scaled-up version, deployed in continuous production, we took the conscious decision to document-but-not-solve these research challenges. Our goal was to focus on deployment first, and only once we had a fully scaled and deployed automated test design platform, did we propose to try to address any open research problems.

[2] developers.facebook.com/videos/f8-2018/friction-free-fault-finding-with-sapienz/.
[3] facetavlondon2017.splashthat.com/.
[4] Sapienz presentation starts at 46.45 in this video: www.facebook.com/andre.steed.1/videos/160774057852147/.

We were not surprised to discover that, after 15 months of intensive deployment focus, we had already accrued more exciting and interesting research challenges than we could hope to solve in reasonable time. In this paper we set out some of these challenges and open problems in the hope of stimulating interest and uptake in the scientific research community. We would be interested to partner and collaborate with the academic community to tackle them.

2 Sapienz at Facebook: Overview

Sapienz is deployed using FBLearner, Facebook's Machine Learning Infrastructure. In particular, Sapienz uses the FBLearner flow operators and workflows for continuous deployment and availability. The Sapienz infrastructure also supports sophisticated and extensive facilitates for experimentation, statistical analysis and graphic reporting (see Sect. 5). This section outlines the principal components of the deployment.

2.1 Top Level Deployment Mode

The overall top level depiction of the deployment of Sapienz at Facebook is presented in Fig. 1.

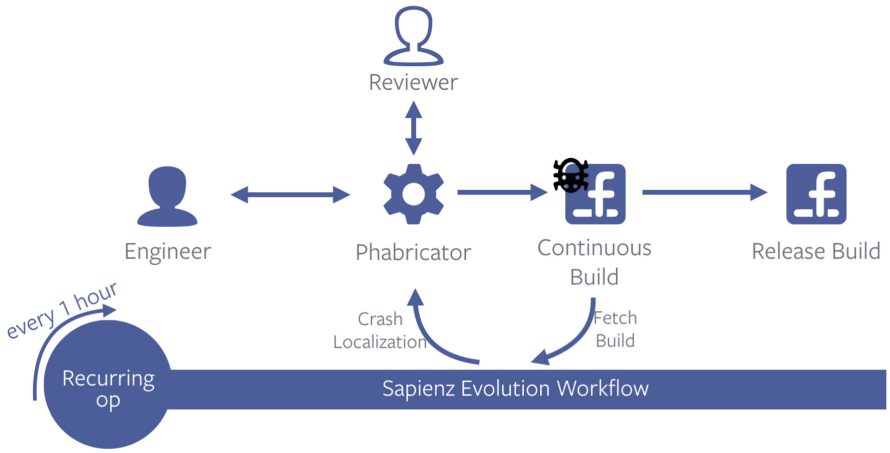

Fig. 1. Overall deployment mode for Sapienz at Facebook

2.2 Phabricator

Phabricator is the backbone Facebook's Continuous Integration system[5]. It is used for modern code review, through which developers submit changes (diffs)

[5] http://phabricator.org.

and comment on each others' diffs, before they ultimately become accepted into the code base (or are discarded). More than 100,000 diffs are committed to the central repository every week at Facebook, using Phabricator as a central gatekeeper, reporting, curating and testing system [37]. In 2011 Facebook released Phabricator as open source.

Sapienz reports the test signals it generates directly into the Phabricator code review process. It has been found with earlier tool deployment, such as the previous Infer deployment [13,37], that the code review system is an ideal carrier for the signals that originate in testing and verification technologies. The Sapienz deployment therefore followed a similar pattern to the Infer static analysis tool, which also deployed through Phabricator, commenting on developers' diffs. Infer also originated in the research community via a London-based start up Monoidics [13] in 2013, while Sapienz came from the London-based start up Majicke [24] in 2017.

2.3 Diff Time and Land Time Testing

Sapienz comments on diffs at two different points in the code review process: diff submit time, and post land time. A diff is first submitted by a developer, to be reviewed by other developers, and cannot land into the code base until it has passed this review stage, so diff submit time always occurs strictly earlier in the code development life cycle than land time. The post-land stage is the point at which diffs may be encountered by dogfooding, allowing Sapienz to cross-check whether crashes it has found have also been witnessed in pre-production dogfooding of release candidates.

At diff submit time, Sapienz receives smoke builds from Facebook's 'Sandcastle' test infrastructure, using these to test individual diffs as they are submitted by developers (and batches of such diffs where possible, for efficiency reasons). The aim of diff time testing is to run a selection of tests at least once per diff, selected from those previously-generated by Sapienz. This selection is run as soon as possible after the diff is submitted, in order to give early signal to the developers as they submit their changes to be reviewed. Often, through this mode, Sapienz is able to comment on a crashing diff before a human reviewer has had time to comment, thereby saving precious human reviewer effort. Furthermore, as has been widely-observed in the software testing literature [10,11], the earlier we can detect faults, the cheaper and quicker they can be corrected.

In order to comment at diff time, it is necessary for Sapienz to be able to test the diff quickly. This requires a scheduling system that prioritises recently-submitted diffs, and fast selection of a likely fault-revealing subset of test sequences. We are continuing to work on both the scheduling process, and smarter sampling of appropriate tests; a problem well-studied in the research community [5,16,22,59,67,74] for over two decades. At this stage we use information retrieval approaches, popular elsewhere in software engineering, such as Term Frequency–Inverse Document Frequency (TF*IDF) [68] to promote diversity and relevance of test sequences to diff under test.

When Sapienz executes on diffs that have been landed into the debug build of the code base, the primary role, here, is to *generate* a good set of test sequences, rather than to test the app (although testing does occur, and any crashes found and triaged, will be reported and their fixes tracked). More importantly, by using the debug build as a representative cut of the current user interface of the app, Sapienz is able to maintain, update and curate the Activity Transition Graph (ATG), that captures Sapienz's inferred representation of the GUI structure of the app. This testing phase is where the SBSE algorithms are used to generate new test sequences.

2.4 FBLearner

FBLearner is a Machine Learning (ML) platform through which most of Facebook's ML work is conducted [38]. Sapienz is one many tools and services that is built on top of the FBLearner framework. Facebook uses FBLearner for many other problems, including search queries for videos, photos, people and events, anomaly detection, image understanding, language translation, and speech and face recognition.

Tens of trillions of ML operations are executed per day; a scale that allows Facebook to deploy the benefits of machine learning in real time. For example, Facebook performs natural language translation between approximately 2000 language pairs, serving approximately 4.5 billion translated post impressions every day, thereby allowing 600 million people to read translated posts in their mother tongue [38]. This considerably lowers linguistic barriers to international communication.

Sapienz currently uses the FBLearner Flow components to deploy detection of crashing behaviour directly into the work flow of engineers, integrated with Phabricator for reporting and actioning fixes to correct the failures detected by Sapienz. The current deployment does not yet exploit the other two phases of the FBLearner infrastructure, namely the FBLearner Feature Store and the FBLearner Predictor. Nevertheless, there are many exciting possibilities for predictive modelling in software testing at scale, and this infrastructure will naturally support investigation of these possibilities and potential future research directions. The team at Facebook would be very interested to explore collaboration possibilities with those in the research community who would like to tackle these challenges.

3 The Sapienz FBLearner Workflows

Three of the more important Sapienz FBLearner workflows are depicted in Fig. 2. Each of these three workflows is explained in more detail below.

The Evolution Workflow: The evolution workflow is used to generate test inputs and record information about them and, where they crash, to report this to developers. The evolution workflow has six principal operators, which execute cyclically. At the time of writing, the periodicity is every 30 min for test

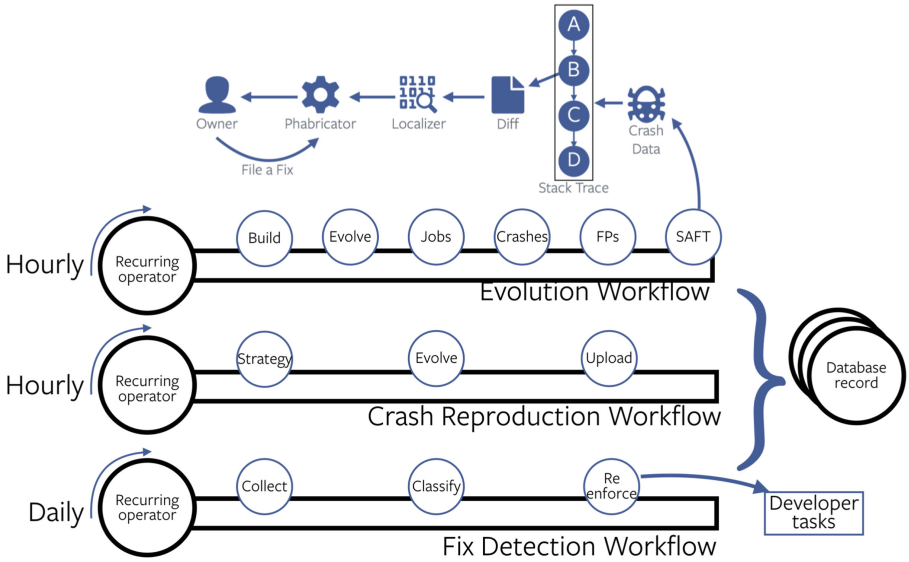

Fig. 2. The three principal Sapienz FBLearner flow workflows and their operators

generation and, for diff time test selection, every 10 min to process the queue of scheduled diff-tine requests. These repeat cycle periodicities are, of course, control parameters that remain under review and are subject to analysis and tuning.

The purpose of these operators is to test the debug build of the application file that contains all of the submitted diffs that have landed into the master build. The overall workflow is broken into 6 separate FBLearner Flow operators. The build operator builds an APK file.

The database of servers, maintained on the emulator by the APK file is updated by Sapienz with a redirection that uses a proxy server to taint the Sapienz requests so that these can be identified in production, when they hit back-end servers, and thereby diverted where they might otherwise affect production. Other than this ability to taint requests via the proxy server, the APK file has the same functionality as that downloaded onto phones by dogfooders, and ultimately, by real users, once the next cut of the debug build has been rendered into one of many release candidates.

The 'Evolve' operator runs the Sapienz evolutionary workflow, executing the Sapienz multi objective evolutionary algorithm to generate new test input sequences from the master build. The details of the evolutionary algorithm are relatively similar to those described in the previous ISSTA paper about the Sapienz research prototype [49]. The primary difference lies the additional technology required to lease and communicate with the emulators used by the MotifCore component, which executes test cases and records, *inter alia*, coverage and crashes. Many 100 s of emulators per app-under-test can be leased for

each execution of the operator, through the OneWorld platform, rather than by the more direct connection to a specific device or devices implemented in the research prototype.

The 'Jobs' operator records information about the evolutionary algorithm executions so that these can be harvested and examined later on, while the 'Crash' operator records information about crashes detected in each run. This is used in reporting and fix detection. The 'FPs' operator is inserted into the workflow to detect false positives, and remove these so that they are not reported to developers. False positives can occur, for example, because the emulator may lack (or differently implement) technical features compared to any device, for example, augmented reality features offered in advanced handsets. Where these are not present in the emulator this may lead to a crash that would not occur on a real device and clearly it would be a nuisance to developers to report these false positive crashes to them.

Finally, the Sapienz Automated Fault Triage ('SAFT') operator identifies the diff responsible (and the line of code within that diff) that is the likely root cause of each crash detected. When the SAFT operator is able to triage to a particular line of code, this is reported to the developer through the Phabricator Continuous Integration system. The developer receives a 'Debugging payload', consisting of information about the crash, the stack trace, video(s) showing steps to reproduce the crash, and pointers to collections of information about potentially related crashing behaviour, and debugging support technology.

3.1 Crash Reproduction Workflow

The problem of flaky tests means that many failing test cases will not reliably fail on every test execution, even in apparently identical circumstances. The crash reproduction workflow assesses and records information about the degree of flakiness for each crashing test sequence, using repeated execution.

Several authors have commented on the reasons for this flakiness [28,37,46, 56,62], finding that one of the most common causes lies in the prevalent use of asynchronous waits; functions make asynchronous calls to services, but may not wait sufficient time for those services to respond, producing a different result to that had they waited slightly longer.

The result of this async wait issue is that different pauses between test events can significantly affect the behaviour of the app under test. Sapienz runs a crash reproduction workflow in order to determine the level of repeatability for the failing test cases it discovers. Those found to have higher repeatability are archived and logged as such.

For diff submit time deployment of Sapienz, test flakiness is less of a pernicious problem. This is because any crash, on any occasion, even if not repeatable, has two properties that tend to promote a quick fix:

1. The crash is reported early, at diff submit time, so the developer concerned has the full context in his or her head and is able to act immediately to remediate.

2. The crash serves as a proof of existence; it is possible that this diff *can* cause the app to crash, in *some* circumstance, and this is typically sufficient signal to occasion a change at this stage in the development process.

On the other hand, when Sapienz discovers a crash that relates to a longer-standing problem, and cannot triage the crash to a recent diff submitted or landed into the code base by developer, then repeatability of tests becomes more important. A single flaky test will likely fail to give the developer sufficient signal that he or she will want to invest time effort on a fix attempt.

However, as we have argued elsewhere [37], this does not necessarily mean that flaky tests cannot be useful. Indeed, we believe that more research is needed on *combinations* of flaky tests, such that the combined signal they produce can be highly actionable.

We believe that promising research avenues may exist, for example using techniques such as information theory and other probabilistic interpretations of test outcomes [6,17,73,75]. This is important because, in many situations, particularly with automated test data generation, we may have to work in a world where it is safer to assume that All Tests Are Flaky (ATAF) [37].

3.2 Fix Detection Workflow

It turns out that detecting whether a crash is fixed or not is an interesting challenge, and one that would benefit from further scientific investigation by the research community. This is a problem to which the Search Based Software Engineering community could contribute. The problem consists of two parts:

1. Determining when two or more crashes likely originate from the same cause. This involves grouping crashes and their likely causes.
2. The problem of 'proving a negative'. That is, how long should we wait, while continually observing no re-occurrence of a failure (in testing or production) before we claim that the root causes(s) have been fixed? Since absence of proof is not proof of absence, we can really only more precisely speak of 'apparent fixes' or 'failure symptom non-reoccurrence'. Addressing this question requires a fix detection protocol.

Grouping Crashes and Their Likely Causes. The starting point for this problem is the difference between faults and failures, a problem well-known to the testing literature [10]. While a single fault may lead to multiple failures, a particular failure may also be caused by multiple faults. Therefore, there is a many-to-many mapping between faults and failures. An automated dynamic test design technology such as Sapienz is only able to directly detect a failure (not a fault), and has to use further reasoning to indirectly identify the likely candidate fault (or faults) that may have caused the failure.

In order to identify a particular failure, and distinguish it from others, we need some kind of 'failure hash', or ID, that uniquely identifies each individual failure. However, what we are really interested in, is the fault(s) that lead to these

failures. The failure hash ID is thus an attempt to capture fault(s), through these symptoms observed as a failure.

Inevitably, how ever we choose this 'failure hash ID', we are effectively grouping failures that we believe, ultimately, may share a similar cause. This 'failure signal grouping problem' is one that has been known for some time [66]. If the grouping is too coarse grained, then we over approximate, with the result that we falsely group together multiple distinct failures with distinct (unrelated) causes. Conversely, if the granularity is too fine, we separate two or more different failure observations that originate in the same root cause (false splitting).

Whatever approach we choose, there will likely be some false grouping and also some false splitting. We can bias in favour of one or the other, but it is unlikely that we shall find an approach that guarantees freedom from both, since it is a challenge to be sure of root causes. Indeed, even the very concept of causality itself, can become somewhat 'philosophical' and, thereby, open to interpretation.

Hedging in favour of finer granularity, we could consider the identity of a failure to be the precise stack trace that is returned when the failure occurs. However, different executions may follow slightly different paths, leading to slightly different stack traces, while ultimately denoting the same failure and, more importantly, the same root cause (in a fault or set faults). Therefore, using the precise sequence of calls in a stack trace for a failing execution is too fine-grained for our purpose.

Instead, we use a 'message identifier', or mid, which identifies the principal method call in a stack trace that is used as the 'hash' for the failure. In so-doing we err on the side of a coarser granularity, though we cannot guarantee that there is no false splitting. Nevertheless, the use of mids does tend to reduce the cognitive burden on developers who might otherwise be spammed by hundreds (or perhaps thousands) of messages concerning ultimately the same fault.

However, it does raise the problem of false grouping, in which two entirely independent faults can become grouped together by the same mid, and thereby appear to contribute to the same failure. False grouping poses problems for fix detection, because the developer may respond to a signal from the tool, and fix the fault that leads to failure, yet the false grouping of this failure with another, otherwise independent fault, leads to the testing tool apparently re-witnessing the failure. As a result of this apparent re-witness of the failure, the test tool will inadvertently infer that the bug has not yet been fixed, when in fact it has.

Our motivation for adopting the more coarse-grained failure hashing approach derives from the trade off in dis-benefits: We did not wish to over-claim the fix rate for our technology, preferring to be conservative, giving a lower bound to the claimed fix rate. We also cannot afford to spam developers or we risk losing their trust in, and consequent engagement with, our technology. We therefore chose to suffer this false grouping problem rather than the more pernicious problem of giving developea of spammy signal.

Although we use a mid at Facebook, similar problems would occur in any real-world testing system in which we need to identify and distinguish different

failures. There is always a trade-off between the fine-grained representation and the risk of a spammy signal, contrasted to more coarse-grained representation which may suffer from the false grouping problem. We continue to refine and optimise for this trade-off, but it is likely to remain a trade-off, and therefore the challenge is to find ways to balance the two competing constraints of not spamming developers, and not falsely grouping; a problem well-fitted to the SBSE research community.

This crash ID problem is compounded by the nature of the continuous integration system and Internet deployment. Continuous integration results in high levels of code churn, in which the particular identity of the method may change. This further fuzzes the information available to the testing technology in stack traces, over time, since code changes may introduce apparent differences in two otherwise identical stack traces. The deployment of Internet-based systems also tends to elevate the degree of flakiness of test cases, since productionised tests will tend to rely on external services, that lie outside the testers' control.

Fix Detection Protocol. Figure 3 depicts an example Fix Detection Protocol with which we have recently been experimenting within the Sapienz Team at Facebook. We do not claim it is the only possible protocol, nor that it is best among alternatives. Rather, we present it here to illustrate the subtleties that arise when one attempts to automate the process of detecting whether a fix can be said to have occurred in a Continuous Integration and Deployment environment.

We are not aware of any research work on the problem of automated fix detection for Continuous Integration and Deployment in the presence of flaky tests. We would like to suggest this as an important open problem for the research community and hope that this section adequately motivates this as an interesting open research problem.

In Fig. 3, when printed in colour, green edges represent the absence of a failure observation, denoted by mid M, within a given time window, while the red edges denote the observation of mid M, within a given observation window. We distinguish two observation windows; the cooling window and the resurrection window. When printed in black and white, these colour-augmentations may be lost, but this should not unduly affect readability.

The cooling window is the period we require the protocol to wait before we initially claim that the mid M is deαd. When it is not observed for this cooling window duration, we claim that the bug is 'α-fixed'; it initially appears to be fixed. We mark the mid M as deαd and increase our count of α-fixes, but the mid is not yet 'buried'.

If a deαd mid is re-observed during the resurrection window, then the mid is said to be 'undead', whereas those mids that remain undetected during the cooling window and the subsequent resurrection window are claimed to be 'dead and buried', and we call this a 'β-fix'. A mid, M which is dead and buried is one for which we have high confidence that, should M subsequently be re-observed after the resurrection window, then this is a 'recycled' mid; one that has re-

entered our protocol as a result of new, previously unwitnessed, failure and its corresponding cause(s). A recycled mid is a special instance of false grouping in which the resurrection window allows us to accrue evidence that the mid is, indeed, falsely grouped.

We count the overall number of α and β fixes. The aim in choosing the window durations is to have a reasonable chance of the true fix count lying between these two numbers, without making them so wide as to become impractical.

The five nodes along to top of the state machine protocol follow the a relatively simple path from initialisation of the protocol through detection of the first observation of a brand new (never before observed) mid M, its subsequent non-observation during both cooling and resurrection window periods and the ultimate declaration of our claim that M is dead and buried, with consequent increments to both the α and β fix counters along the way. This path through the state machine represents the ideal scenario that we would *like* to witness for *all* failures, in which we detect them, they get fixed, and that is all there is to be said about them.

The remainder of the state machine denotes the complexities involved in fix detection, arising from the problems of flaky tests, false grouping/splitting, and the inevitable attempt to 'prove a negative' inherent in fix detection. Moving from left to right, when we first encounter a mid M after some period, it may turn out to be one that we previously believed was dead and buried, in which case we claim a false grouping (of faults and their failures) has occurred because, by definition (of buried), M has not been observed for, at least, the duration of the cooling and resurrection windows combined. The combination of these two windows can be thought of as the duration after which we believe any mid to be 'recycled' should it re-occur. The duration denoted by these two windows in sequence therefore represents that time after which we believe it is sufficiently safe to assume that any new mid M observation arises due to some newly-introduced root cause.

Another possibility is that the mid M is observed for a new revision, R in which case the mid M is already at some stage in the protocol (prior to being dead and buried). This observation also occasions a false grouping claim, because we know the mid arises from a different code change to the previous observation of the same mid through testing a different revision. For this reason, our protocol effectively treats the 'crash hash' for Sapienz-detected crashes as the combination of the failure hash identifier M (the mid), and the revision, R (the testing of which encounters M). For mids observed in production we may do not always know the corresponding revision, but for mids reported to developers by Sapienz we only report those that we can triage to a specific diff, so this pair is well-defined for all Sapienz crashes reported.

When we detect a (mid, revision) pair (M, R), we initialise a count of the number of occasions on which M has cycled from being deαd to undead, MUD_M. Mid UNdead (MUD) is an appropriate acronym for this counter since the status of mids that cycle between deαd and undead statuses is somewhat muddy itself, and denotes a primary source of uncertainty in our protocol.

If M was previously buried, we 'dig it up' and note that M has been recycled whereas, if it was not buried previously, we do not treat it as recycled. In either case we believe a false grouping has occurred and claim it. Of course, we cannot be *sure*, so this false grouping claim, like our other claims, is just that; a claim.

Once we have determined whether a newly-observed mid is a brand-new (never before seen) mid, or one of these two categories of falsely-grouped mids, we enter the main sequence of the protocol for the mid M and corresponding revision R. At this stage, we wait for a duration determined by the cooling window. This is a potentially infinite wait, since the cooling window is a sliding window protocol. A mid therefore remains unfixed until it passes the cooling window non-observation criterion.

If it does survive unobserved through the cooling window, we claim it is deαd and increment the α-fix count. We then start waiting again, to see if the mid re-occurs. If the mid re-occurs within the (re-initialised) cooling window, it becomes undead, from which status it can either return to being deαd (if it does not become re-observed during the cooling window), can remain undead, can oscillate between deαd and undead states, or may finally exit the protocol as a 'dead and buried' mid.

Overall, the protocol ensures that no mid can be claimed to be dead and buried unless it has undergone both the cooling window and subsequent resurrection window. The undead status is recorded for subsequent analysis, since it may have a bearing on the false grouping problem, the test flakiness problem, and the determination of suitable durations for the cooling window and resurrection window. However, the undead status plays little role in the external claims made by Sapienz about fixes; it is merely recorded as an aid to further 'healthiness' analysis for our window duration, mid groupings, and test flakiness.

The external claims made by the protocol concern the number of α- and β-fixes, and the claims concerning the deαd, and the dead and buried statuses of the mids observed by the fix detection protocol.

3.3 The Evolve FBLearner Flow Operator

The 'Evolve' operator in the evolution workflow, is the core Sapienz FBLearner operator. It is depicted in Fig. 4. This architecture of the Sapienz operator remains similar to that envisaged for the Sapienz Research prototype [49], as can be seen.

The Evolve operator is used to generate test inputs. These are used for testing; if they find a failure then it is reported through the evolution workflow. However, perhaps more importantly, the test inputs are archived and curated into an Activity Transition Graph (ATG). This pool of tests is then used as a source of pre-computed test inputs for diff time testing, so that Sapienz can quickly find a set of pre-vvovled test sequences to apply to each diff as it is submitted, using its smoke build, or a batch of such builds combined.

The Sapienz operator workflow starts by instrumenting the app under test, and extracting statically-defined string constants by reverse engineering the APK. These strings are used as inputs for seeding realistic strings into the app, a

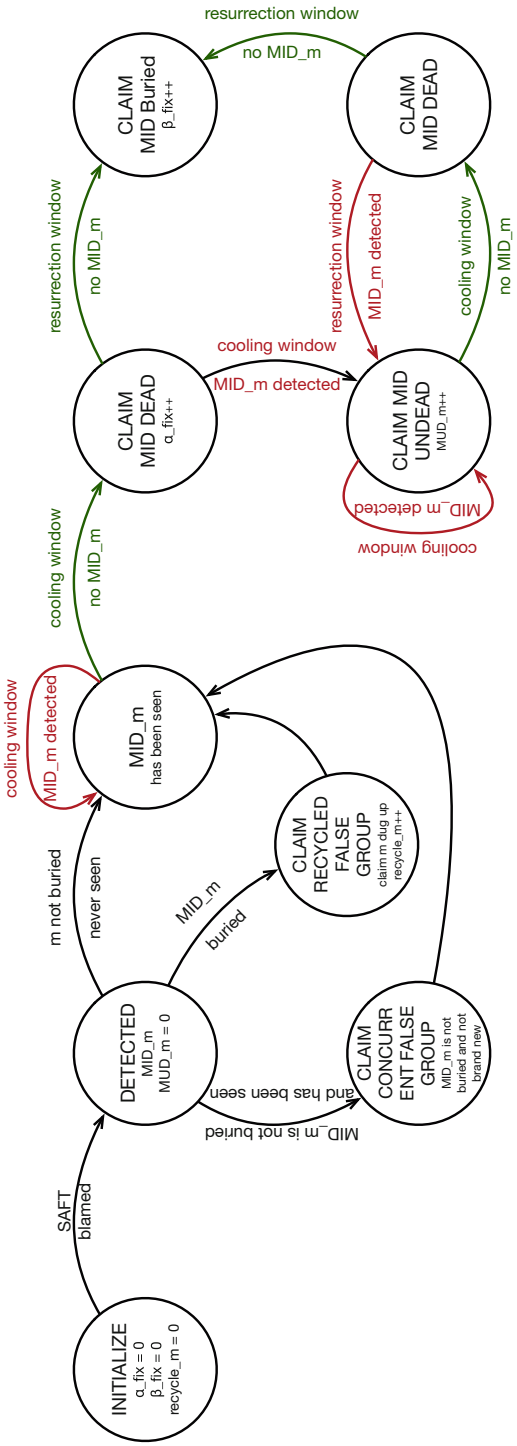

Fig. 3. An example of an FDP (Fix Detection Protocol) with which we are experimenting. In general, much more research is needed on the problem of automated fix detection; a largely overlooked, yet highly intellectually stimulating problem. (Color figure online)

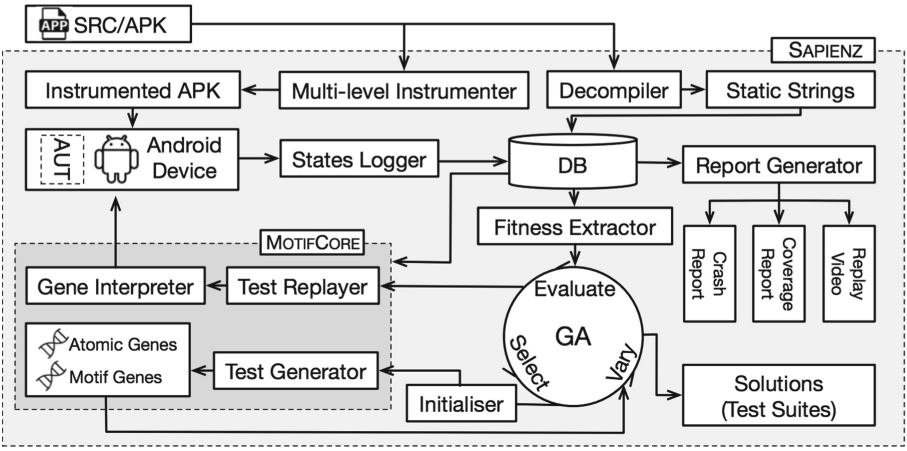

Fig. 4. Sapienz evolution operator workflow (Taken from the ISSTA 2016 paper [49])

technique that has been found to improve the performance of other search based software testing techniques too [4,27].

Sapienz's multi-objective search algorithm initialises the first population via the *MotifCore* component which runs on the device or emulator. On android, when evaluating individual fitnesses, Sapienz communicates with the *App Exerciser* via the Android Debugging Bridge (ADB) and monitors the execution states, returning measurement data, such as Android activities covered to the fitness evaluator.

Sapienz uses a Search Based Software Engineering (SBSE) approach to optimise for three objectives: code coverage, sequence length and the number of crashes found. Its computational search is a pareto-optimal approach, based on NSGA-II, a widely-used multi-objective evolutionary search algorithm, popular in SBSE research [35], but hybridised with other search techniques, such as localised systematic search on devices/emulators. More details on the Sapienz algorithms and their evaluation against the state of the art and the state of practice can be found in the ISSTA paper [49] and in Ke Mao's PhD dissertation [47].

4 Integration with Other Facebook Tools and Systems

In this section we briefly explain some of the ways in which the Sapienz infrastructure inter-operates with other tools, systems and services at Facebook.

4.1 Collaboration Between Static and Dynamic Analysis: Infer and Sapienz

Facebook has a static analysis technology, Infer [13], for scalable compositional static analysis. Infer reviews potential faults, pinpointing the line of code concerned, and reporting the fault candidate to the developer with a suitable message. For example, Infer statically analyses program flows to highlight potential

Null Pointer Exceptions and reports these to developers through the Phabricator code review interface.

Naturally, it can happen that the developer decides that this fault reporting is inaccurate, deeming a reported fault to be a false positive. The developer has domain knowledge and may believe, for example, that execution cannot cause this statically-inferred issue to arise in practice. Occasionally, the developer maybe correct, but also, on other occasions he or she might be mistaken.

When the developer is incorrect in setting aside advice from Infer, it would be highly useful to use dynamic analysis to provide a positive existence proof that the fault can, indeed, lead to a failure. Sapienz seeks to provide just such an existence proof, by checking whether a crash can be triaged to a line of code on which Infer as previously commented. When such a failure is traced to a line of code on which Infer has previous commented, a specific task is created and reported to the developer to fix the fault, augmented with the additional signal that this fault does really lead to a demonstrable failure (a constructive existence proof), including a witness video, stack trace and line at which the crash occurs.

At the time of writing, the fix detection workflow has detected an almost 100% fixed rate for such Sapienz-Infer ('SapInf') faults. This is evidence that, when static and dynamic analysis techniques can *collaborate* to provide a combined (and thereby strengthened) signal to developers, it is highly likely that such faults are likely to be true positives, and also that they are actionable and, thereby, tend to get fixed. We believe are many more ways in which static and dynamic analysis could collaborate and so we set out this collaboration as a remaining open problem that requires further research work (see Sect. 7.7).

4.2 Combining with Feedback from Field Trials

It can happen that a crash hits real world users of the app, who experience the crash in production. Clearly we try to avoid this where possible. In fact, the 'real world' user may well, in the first instance, be a Facebook dogfooder who is running the app on their own device. We use such dogfooding as a line of defence in preventing crashes hitting our end users.

When any user, dogfooder or end user, experiences a crash, the crash can be logged and used to help improve testing and fault remedy. Sapienz has a continuously running workflow which tracks the crashes it has reported to developers against the real world crashes found in production. Sapienz files a task for the developer when this happens. Once again, this production-firing evidence provides a stronger signal to the developer that a crash really is a true positive.

4.3 Bug Severity Prediction

Sapienz uses a simple tree-based machine learning classification technique (C4.5) to predict whether the crashes it detects are likely to have a high number of real world affected users. This is a measure of bug severity. After some experimentation, we determined that the simple C4.5 classifier produced a high prediction accuracy (of approximately 75%), and therefore adopted for this approach.

The classification algorithm takes as input, drivers of bug severity, such as:

1. **Crash Metadata Features:** e.g., new mid, number of hits, has the mid been seen in prod;
2. **Crash Stack Trace Features:** e.g., crash type, stack depth;
3. **Diff Features:** duration to find crash, code churn metrics, reviewer comment metrics, location in file tree.

5 Automated Scientific Experimental Reporting (ASER) Workflow

The Automated Scientific Experimental Reporting (ASER) seeks to achieve automated empirical software engineering, at scale, and fully in production, using best-practice recommended inferential statistical evaluation and reporting for empirical SBSE [7,36]. The ASER is an example of a broader company-wide (and increasingly sector-wide) approach to evidence-based decision making. For all the challenges of Continuous Integration and Deployment, such demanding modes of deployment do also have the exciting potential to facilitate such evidence-based decision making, through support for automated experimentation.

As well as ensuring the team retains its scientific roots and culture, the ASER also allows the team to collaborate efficiently and effectively with other researchers from the academic and scientific community; new ideas, concerning SBSE algorithms and more general automated Software Testing techniques, originating from outside the team, can be empirically evaluated quickly and systematically on a level playing field.

The ASER has four phases, starting with an initial proof of concept experiment, through to full evaluation in parallel with (and evaluated against) the production release of Sapienz. The motivation for these four phases is to allow the experimenter to devote increasingly larger levels of resource to evaluating a proposed new approach to test generation. As a result, promising techniques are placed under increasingly strong scrutiny in order to evaluate them against ever-increasingly demanding criteria, while less promising approaches can be quickly identified as such and discarded.

Our aim is to explore and experiment, but also to 'fail fast' with those less promising approaches, so that we can divert human and machine resources to the most promising avenues of research and development. At each phase of the ASER, metrics reporting on the performance of the approach are collected and used as the inputs to inferential statistical analyses. These analyses are performed automatically and the results of confidence intervals, significance tests, effect sizes and the corresponding box plots and other visualisations are automatically computed and returned to the experimenter and the overall team, for inspection and assessment.

The overall flows through the four phases from initial proof of concept to full deployment are depicted in Fig. 5. The migration through each of the four

phases of evaluation is placed under human control. Movement through the ASER is informed by inferential statistical analysis; should the newly-proposed approach significantly outperform a baseline benchmark version of the deployed technology, the newly-proposed technique moves from the initial proof of concept to a more thorough evaluation against the production deployment.

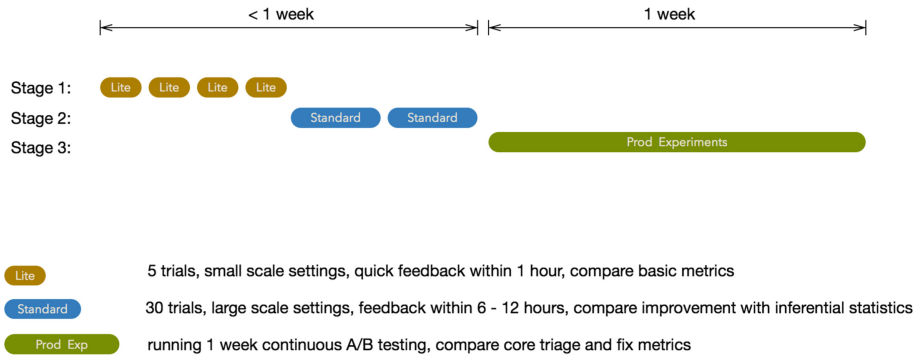

Fig. 5. Experimental workflow into production

The first three phases are christened 'Lite', 'Standard' and 'ProdExp'. The fourth phase is a deployment into production for up to one month alongside production deployment so that we can fully assess the number of faults triaged and fixed by the newly-proposed technique. The Lite phase is designed to give proof-of-concept feedback within an hour. It uses only five emulators on a choice of search based parameter settings that ensures the algorithms terminate quickly.

Ideally, for any testing or verification technique, we should evaluate the signal-to-noise ratio of the technique, rather than give an arbitrary time limit, or we may fall in to the Cherry-Picked Budget (CTB) trap [37]. Using our Lite experiment, it is possible that the reliance on CTB for the Lite phase can cause techniques with a long startup time to be prematurely discarded.

Fortunately, at this stage of the deployment of Sapienz, we are blessed by the large number of potentially promising ideas to explore. Therefore, it is more important that we are able to quickly dismiss those that appear unpromising (albeit initially with a CTB that favours early-stage effectiveness). As the technology matures, we will revisit these assumptions that underpin the ASER.

Newly proposed techniques that pass the initial Lite phase move into the 'Standard' ASER experimental phase. The Standard phase of the ASER deploys best practice [7,36] (thus 'standard') inferential statistical experimental assessment of effectiveness using 30 emulators. This number balances the efficiency of time and computational resources against statistical test reliability.

We believe it sufficient to yield meaningful inferential statistical feedback on the relative performance of the newly-proposed technique against a productionised alternative. We tend balance in favour of avoiding Type I errors (incorrectly rejecting the Null Hypothesis) at the expense of risking Type II errors

(incorrectly accepting the Null Hypothesis), reflecting the current opportunity-rich environment in which we seek large effect sizes to motivate further investigations. In terms of resource bounds, our aim is to obtain results within less than a week, so that a decision can be made relatively quickly about whether to move the newly proposed technique to the 'ProdExp' phase.

In the 'ProdExp' phase, the ASER deploys A/B testing, over a period of a week, comparing the number of crashes triaged against the production deployment. The final phase is a longer A/B test, against production, for a period of up to one month before the new technique is finally allowed to become part of the new production deployment. This is necessary because Fix Detection has a natural window (the cooling window; see Sect. 3.2) within which it is not possible to reliably compute a fix rate for any newly-proposed technique. As a result, we can compute the fault reporting rate (the number of crashes triaged) during ProdExp phase, but not the rate at which these crashes are fixed.

The ASER is fully parallelised for scalability, so multiple experiments can be run at any given time, each of which resides at a different stage of the ASER process. Overall, this ensures that less promising techniques are dismissed within a week and that, after one week, we have sufficient evidence to put those techniques that pass standard empirical SBSE evaluation into full A/B testing, yielding their benefit in terms of triages and fixes.

For inferential testing we use a paired Wilcoxon (non Parametric) test, and highlight, for follow up, those results with p values lower than 0.05. We do not perform any p value corrections for multiple statistical testing, since the number of tests is essentially unbounded and unknowable. This choice poses few practical problems, because the final A/B testing phase would allow us to dismiss any Type I errors before their effects hit production. The effect of Type I error is thus an opportunity lost (in terms of time that would have been better spent on other approaches), but it is not a computational cost in terms of the performance of the deployed Sapienz technology.

In the following subsections we give 3 examples of the applications of the ASER framework to evaluate new generation and selection algorithms and techniques we considered for improving fitness computation, the motif core interface and solution representations. Within the first 3 months of deployment of the ASER, we had conducted tens of different high level experimental investigations to answer our research questions, with (overall) hundreds of specific low level experimental algorithm comparisons (thousands of experimental runs), each with different parameter settings and choices.

In the subsections that follow, we chose one example that allowed us to fail fast, one that initially seemed promising, yet failed subsequent more demanding phases of the ASER, and one that made it through the full ASER process to production deployment. Each example also illustrates different aspects of the automated reporting: box plots, scatter-plots and inferential statistical analysis. We show the completely unvarnished output from ASER, exactly as it is automatically rendered to the engineer in these three examples.

5.1 ASER Example: Fail Fast Speed up Experiment

Sapienz communicates with the device or emulator using the Android Debug Bridge (ADB). We had available to us, a modified `ADB` for which others had previously reported *prima facae* evidence for high performance. Naturally, we wanted to investigate whether it could reduce the time Sapienz required for each test case.

RQ: New adb: Can the new `adb` version increase performance of the Sapienz MotifCore and thereby the overall test execution time?

We used the ASER to run an experiment comparing execution time for the modified `ADB` with the original ADB. Figure 6 illustrates the output from ASER's initial 'Lite' phase.

This Lite experiment immediately revealed that the modified `ADB` was, in fact, *slower*, for our use-case, not faster so it was discarded, quickly and without our engineers investing time building a full implementation. This example illustrates the way in which ASER allows us to move fast and fail fast, with those ideas that appear promising yet which, for a myriad of practical reasons, fail to deliver benefits in practice.

5.2 ASER Example: Longer Test Sequences

We considered dropping the test sequence length constraint of Sapienz, to allow it to search for longer test sequences that might achieve higher coverage. This challenged a core assumption in our earlier work. That is, we had initially assumed that shorter test sequences would be inherently good for efficiency and effectiveness of debugging. As a result, length formed part of the original multi objective optimisation approach.

We used a pareto optimal multi objective approach, so length could only be reduced if such reduction could be achieved without sacrificing coverage.

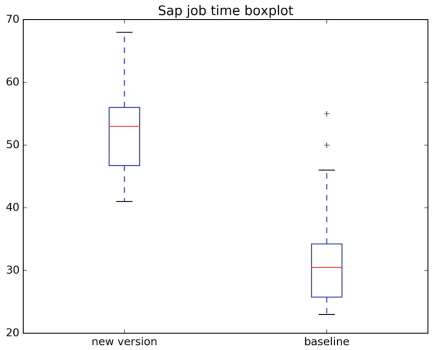

Fig. 6. Answer to RQ: New adb: 'No. The new `adb` is not faster for Sapienz test design'. An example of a box plot outcome that enabled us to Fail fast box plot. Vertical axis shows execution time per overall run, in minutes.

To challenge the assumption that length should be an objective, we created a bi-objective version (crashes and coverage only) of our original tri-objective algorithm, as formulated in the 2016 ISSTA paper [49].

RQ: Length: What is the efficiency and effectiveness of a bi-objective SBSE algorithm compared to the production tri-objective algorithm?

Initial results from ASER's 'Lite' phase were promising, as were inferential statistical results from ASER's subsequent 'Standard' phase; maybe our test sequence length objective should be dropped. The results, as obtained directly and automatically by ASER, are presented in Fig. 7. The figure shows the bi-objective runs (red plots with data points marked as '+') maintaining and slightly improving the activity coverage as the population evolves.

The bi-objective version found 90 mids, while production found 57 with 40 overlaps). The bi-objective version also covered 130 Android activities, while prod covered fewer (114, with 107 overlaps). Inferential statistical analysis from ASER's 'Standard' phase also gave cause for optimism: The unique crash count improved by 109% ($p = 0.003$), while the unique activity coverage count improved 31% ($p < 0.001$). However, set against this, run time slowed down by 21% ($p < 0.001$), largely because the bi-objective version's mean test sequence length was longer.

Based on the inferential statistical analysis from ASER's 'Standard' phase, we were highly optimistic for the bi-objective version of our algorithm: surely a modest test system slow down of 21% was a small price to pay for extra coverage. However, when we deployed the bi-objective version alongside production Sapienz in a full A/B test, we found that the bi-objective version found relatively few extra crashes in practice compared to production and reported fewer per unit time.

This illustrates the importance of the final A/B testing phase, rather than merely relying on (purely) laboratory-condition experimental results alone. The result also underscored, for us, the importance of time-to-signal; bugs-per-minute, being more practically important than coverage or total bugs found, in terms of their immediate impact on our developers, something that has been noted elsewhere [37].

5.3 ASER Example: ATG Test Selection

As described in Sect. 2.3 we use a generate-and-select approach in which we generate a set of tests from the debug build of the app (reporting any errors this uncovers) and use the generated tests as a pool from which we subsequently select tests to be run against each submitted diff. The generate and select approach was first trialed using the ASER against the then production version (which solely used generation). The research question we addressed using the ASER was thus:

RG: ATG: What is the efficiency and effectiveness of an Activity Transition Graph (ATG) generate-and-select algorithm compared to a purely generation-based algorithm?

Figure 8 presents the results obtained from ASER for an experiment on the Activity Transition Graph (ATG) approach, selecting from the ATG, the top activity-covering sequences, rather than simply running in production to generate tests on each and every occasion.

The results were very promising for ASER's 'Standard' phase, as can be seen from Fig. 8. In this case, when we deployed the ATG selection approach alongside prod, in A/B testing the optimism derived from the inferential statistical analysis was fully borne out in practice. As a result of the successful final A/B phase, we moved from purely generation-based testing to a generate-and-select approach.

6 DevOps Results Monitoring

The results obtained from deployment of Sapienz include many of those that we would typically wish to collect for scientific evaluation. That is, in common with any scientific evaluation of a research prototype, we collect information about the key performance indicators of the Sapienz deployment. These include the number of crashes triaged to developers, and breakouts of this data, by Android API level, application under test, and so on. These data are plotted, typically, as timeseries data, and are available on dashboards.

In essence, this can be thought of as a continuous empirical software engineering experiment of efficiency and effectiveness of the production deployment against a sequence of recent debug builds and submitted diffs. The comparatively rapid rate at which diffs are submitted ensures that regressions are detected quickly.

In addition to the continuous monitoring of efficiency and effectiveness, the Sapienz infrastructure also needs to collect a large number of DevOps infrastructural 'health' monitoring metrics. These health metrics help engineers to detect any issues in continuous deployment.

Facebook provides much infrastructural support to make it easy to mash up such dashboards, data analysis, and inferential statistical analysis. This provides the deployment of Sapienz with detailed and continuous empirical evaluation feedback. We developed the Automated Scientific Experimental Reporting (ASER) framework, described in Sect. 5, to allow us to use these data science infrastructural features to quickly prototype new experiments and ideas. We use the same infrastructural support to monitor the ongoing performance and health of the Sapienz fault triage and reporting system and its fault detection protocols and workflows.

Many continuous integration and deployment organisations use a so-called 'DevOps' approach such as this, in which system deployment is continuously monitored by engineers. This DevOps process is supported by key 'health metrics' reporting, which we briefly describe in this section. We also briefly illustrate the kind of performance indicators that we continually monitor to understand the signal that the Sapienz deployment gives to our user community of developers.

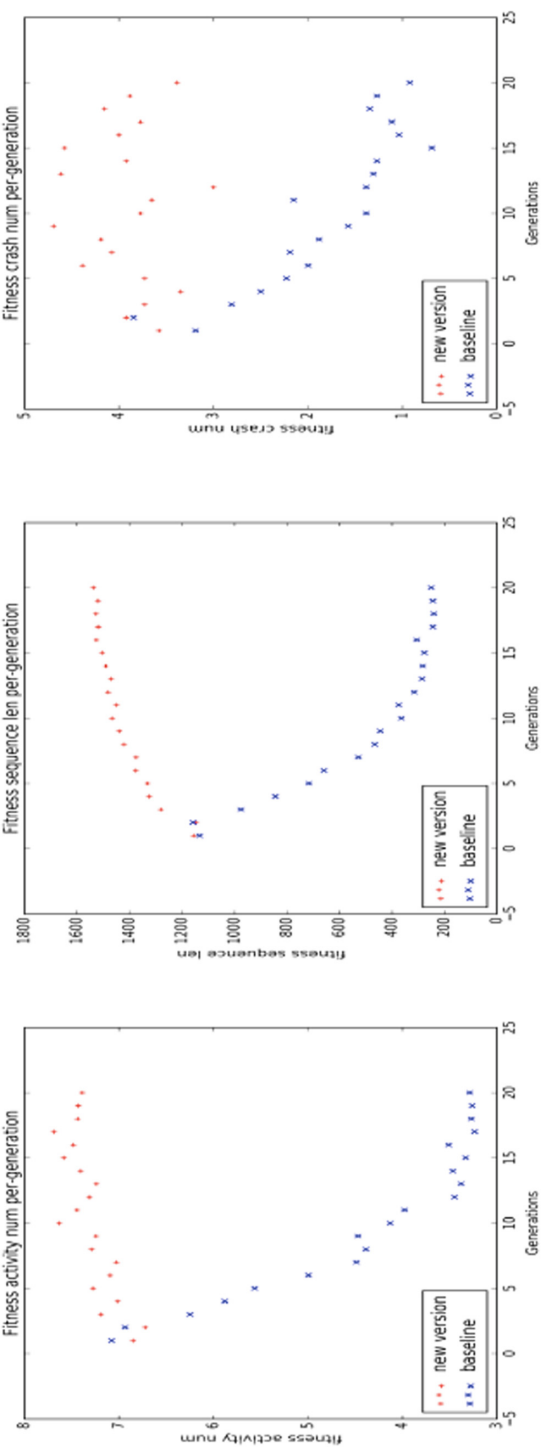

Fig. 7. Sample ASER output: Scatter plots. The upper (red) scatter plot (smaller '+' points) shows the result of Bi-Objective Algorithms. The lower plot (larger 'X' points) is the original tri-objective algorithm. The bi-Objective algorithm appears to be superior to the tri-objective algorithm; an observation re-enforced by the inferential statistical analysis of effectiveness (but not efficiency). In subsequent A/B testing, the tri-objective algorithm's faster execution proved pivotal for bug detection *rate*. This demonstrated that the modest efficiency hit suffered by the bi-objective approach proved to be pivotal in practice, when A/B tested against the tri-objective version. (Color figure online)

Name	Baseline	New Version	%Diff (p_value)
mid_count	2	11	+450.00% (p = 0.000)
nonunique_crash_count	22	68	+209.09% (p = 0.000)
sap_job_time	81	90	-11.11% (p = 0.201)
unique_activity_count	38	58	+52.63% (p = 0.000)
unique_crash_count	22	48	+118.18% (p = 0.002)

Fig. 8. RG: ATG: sample ASER output: inferential statistics

6.1 Health

As explained in Sect. 3, Sapienz deployment contains many 'moving parts', and such large-scale infrastructure typically cannot be expected to run continuously without individual components occasionally failing, due to timeouts, temporary interoperability mismatches and other resource-based exigencies.

The role of Sapienz health metrics is to understand whether these are routine, temporary incursions into deployment, or whether a more serious problem has occurred which needs engineer intervention. Supporting the Sapienz team in making these decisions are a number of automated monitoring and control systems, that report continually to a dashboard monitored by the team member who is the designated 'on-call' for each week.

These dashboards report on many aspects of the health of the deployment, including the

1. number of failed production worflow runs,
2. activity coverage,
3. number of tested smoke builds currently deployed on Sapienz testing tasks,
4. number of crashes detected,
5. number of requests sent to server,
6. logging of various internal soft error warnings,
7. numbers of replication of production failures,
8. the number and proportion of reproducibility (non flakiness) of test cases.

The DevOps reporting also includes a suite of data concerning the performance of the triage, and the response of developers to the signal provided to them.

As an illustration, consider Fig. 9, which depicts a graph plot for six days in April 2018, showing activity coverage. The vertical axis is not shown and is not necessary for this illustration. As can be seen from the figure, activity coverage retains an apparently consistent level, which we regard as 'healthy', but on the 29th April a sudden drop is noticed. This points to potential problems in the deployment, occasioning further investigation, as necessary.

Fortunately, in this case, the sudden drop proved to be merely the result of a temporary quota limit on emulators being reached and within a few minutes, normal behaviour resumed. This example is included as an illustration of the

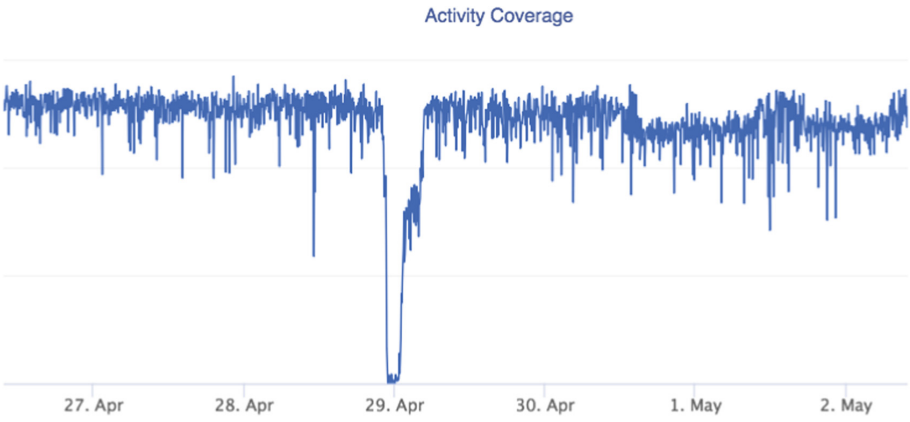

Fig. 9. Example DevOps monitoring: activity coverage over three days' worth of production runs in April 2018 (vertical axis deliberately occluded).

way in which a DevOps approach is used to tackle availability and resilience of the Sapienz infrastructure. Naturally, an interesting challenge is to automate, to the greatest extent possible, this resilience, so that little human intervention is required to maintain healthy operation.

6.2 Key Performance Indicators

Figure 10 shows the two key performance indicators of crashes triaged to developers by Sapienz, and fixes detected by the automated fix detection protocol described in Sect. 3.2. We are interested in fixes detected as a proxy for assessing a bound on the likely false positive rate (more precisely, the signal-to-noise ratio [37]) from Sapienz. Currently Sapienz enjoys a fix rate of approximately 75%, indicating that the signal carries low noise. As explained in Sect. 3.2, this is both an estimate on fix rate and a likely lower bound.

The figure covers the period from the first minimal viable product, in the summer of 2017, through deployment, in full production, at the end of September 2017, to the end of May 2018. Up to the end of September 2017, all crashes triaged to developers (and consequent fixes detected), were implemented by hand, as a result of experiments with the initial minimal viable product. Since the end of September 2017, after successful experimentation, the Sapienz automated fault triage system went live, and Sapienz started commenting, automatically, on diffs that had landed into the debug build of the Facebook android app.

In February 2018, the Activity Transition Graph (ATG) diff time generate-and-select approach, described in Sect. 2.3, was deployed, following successful experimentation with the ASER scientific experimentation framework (described in Sect. 5.3). As can be seen from the figure, this produced a significant uptick in the number of crashes detected and fixed.

Facebook follows the DevOps approach, in which developers are personally responsible for the code they deploy, but also supports developers in maintaining their work-life balance, and thereby respects its own responsibility to developers:

> "The flip side of personal responsibility is responsibility toward the engineers themselves. Due to the perpetual development mindset, Facebook culture upholds the notion of sustainable work rates. The hacker culture doesn't imply working impossible hours. Rather, engineers work normal hours, take lunch breaks, take weekends off, go on vacation during the winter holidays, and so on" [25].

As can be seen, Fig. 10 reveals a relatively 'quiet time' for developers around the end of the year, which corresponds to the winter holiday vacation period. Looking more closely, one can also see a roughly weekly cyclical periodicity, post February (when Sapienz was deployed at diff submit time) which is accounted for by weekends off.

7 Open Problems and Challenges

In this section, we outline a few interesting research challenges we have encountered during our attempts to improve the deployment of Sapienz at Facebook. Some of these problems have been partially tackled, but we believe all of them would benefit from further research work.

We eagerly anticipate results from the scientific research community on these open research challenges and problems. We believe that progress will likely impact, not only the Sapienz deployment, but also other automated test design initiatives elsewhere in the software engineering sector.

7.1 Flaky Tests

As previously observed [37], it is better for research to start from the assumption that all tests are flaky, and optimise research techniques for a world in which failing tests may not fail reliably on every execution, even when all controllable variables are held constant. This raises a number of research challenges, and provides rich opportunities for probabilistic formulations of software testing, as discussed in more detail elsewhere [37].

7.2 Fix Detection

As we described in Sect. 3.2, it remains challenging to determine whether a fix has occurred, based solely on the symptoms of a fault, witnessed/experienced as a failure. More research is needed to construct techniques for root cause analysis, allowing researchers and practitioners to make more definite statements about fix detection. Given the assumption that tests are flaky (described in the previous section), it seems likely that statistical inferences about causal effects are likely

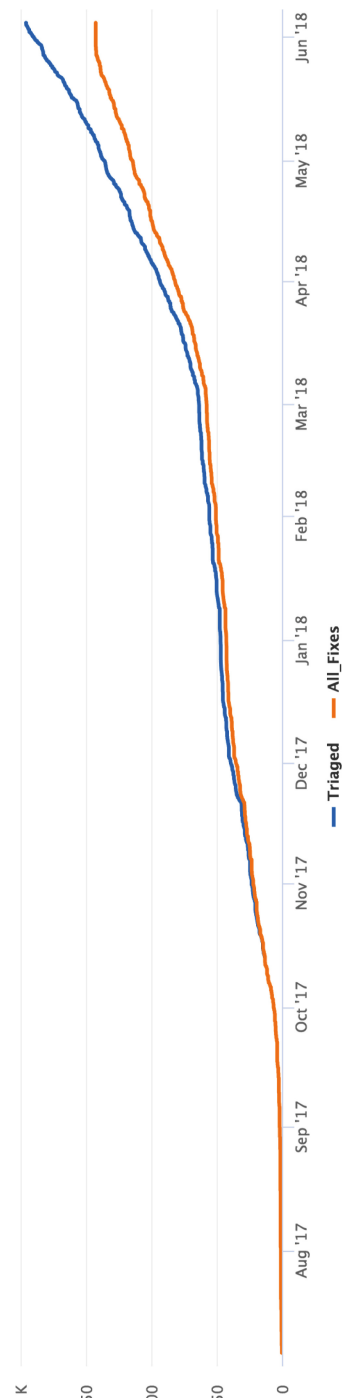

Fig. 10. Example Key Performance Indicators: the first 1k crashes triaged and (detected as) fixed by Sapienz since deployment for the Facebook Android App. The upper curve depicts crashes triaged to a developer; lower line depicts those detected as fixed. As with scientific evaluation and reporting, a key concern is the rate of detection of failures and the proportion that get fixed (a proxy indicator of true positives). The period before the end of September 2017 involved only initial experiments on (by-hand) triage. The effect of deploying, in full production mode, at the end of September 2017 can clearly be seen from the figure. The quieter period around the end of the year can also be seen in the data. The tapering of the fix rate is a consequence of the need for a cooling window, during which no fix can be detected, and also recency effects; crashes take some non-zero time to fix, so older crashes are more likely to be detected as fixed. Finally, the noticeable step change in performance in February 2018 resulted from the successful deployment of the Activity Transition Graph Approach at diff submit time (See Sect. 2.3).

to play a role in this work, due to the probabilistic nature of testing continuously deployed Internet-based systems.

Fortunately, there has been much recent progress on causal inference [63], which has seen applications elsewhere in software engineering [53], as well as in defect prediction [14]. Therefore, the opportunity seems ripe for the further development and exploitation of causal analysis as one technique for informing and understanding fix detection. Empirical software engineering research is also needed to understand whether different classes of fault have different fix detection characteristics, and whether different approaches to fixing faults could lead to different fix detection characteristics.

In general, the problem of fix detection can shed light on a collection of interrelated software testing problems, such as the mapping between faults and failures, the flaky test problem, the cost benefit trade-offs in testing, fault severity, debugging and social aspects of software testing and repair (whether automated or otherwise).

Part of the fix detection problem arises from the subproblem of tackling the mapping between faults and failures. We need techniques for inferring this mapping from observed failures. We need techniques that can use plausible reasoning and inference to identify likely groupings of failures that originate with the same cause, minimizing false grouping and false splitting according to their likely root causing fault(s). Research might also develop techniques for adaptive testing that could be used to drive the search for test cases that help to distinguish such falsely grouped and/or falsely split crashes.

7.3 Automated Oracle

In our work on Sapienz deployment, we opted for a simple and unequivocal implicit oracle [9]; any test which exposes crashing behaviour is a test that is deemed to lead to a failure. Furthermore, if it is possible for a test to witness a crash only *once*, and even if this test is flaky, this is a strong signal to the developer that action is needed:

> *A Sapienz-detected crash is, essentially, a constructive existence proof; it proves that there does exist a configuration in which the app can crash on the input sequence constructed by Sapienz.*

This use of an implicit oracle was important, both for us to be able to fully automate deployment and to increase the actionability of the signal Sapienz provided the developers. However, we believe it is merely a first step, with an obvious starting point, using an implicit oracle.

Much more work is needed to find techniques to automate more sophisticated and nuanced test oracles. Naturally, if the developers use assertions or exception handling, then these can lead to soft errors that can be exploited by an automated search-based testing technique.

Nevertheless, it remains an open question how to either augment or improve the existing test oracles provided by developers [41]. It is also important to

find techniques that generate, from scratch, likely test oracles using techniques such as assertion inference [23]. As test input generation becomes more mature, and more widely-deployed in industry, we can expect a natural migration of the research challenges from the problems of automatically generating test inputs, to the problems of automatically generating test oracles.

Ultimately, if we can automate the generation of oracles *and* the generation of repairs, then we are very close to the grand challenge of FiFiVerify [34]; automatically finding, fixing and verifying software. By tackling the FiFiVerify challenge, we would have practical deployed approaches that would take an existing software system (which may be buggy), and return a new version of the software, guaranteed to be free of certain classes of bugs, *entirely automatically.*

7.4 Fitness Evaluation Resource Requirements

One of the practical problems in deploying search-based software testing lies in the resources required for fitness evaluation. This problem falls inbetween engineering detail and research question. Undoubtedly, some of the solution involves specific platforms and their characteristics and therefore is more a matter of engineering implementation excellence than it is for scientific research.

Nevertheless, there is insufficient guidance in the research literature, and insufficient evaluation in the empirical software engineering literature, of techniques for *reducing* time spent on fitness evaluation. Fitness evaluation reduction techniques essentially trade the computing resources needed for individual fitness evaluation, against the quality of signal returned by fitness evaluation. The ultimate efficacy of fitness evaluation optimisation depends upon the observed impact on higher-level system-wide metrics, such as fault detection rate.

A fast-but-imprecise fitness computation may significantly reduce correctness and thereby guidance provided by an individual fitness evaluation. Nevertheless, such an apparently suboptimal individual fitness evaluation, when executed many millions of times over the lifetime of an evolutionary process, may have a profound effect in reducing the execution time for the overall technique.

As a result, a relatively imperfect fitness evaluation that is fast may be preferable to a much more precise fitness evaluation. These questions naturally centre on cost-benefit trade-offs, which are at the very heart of *any* engineering discipline. In taking scientific ideas from the evolutionary computation community and turning these into practical engineering techniques for the software engineering research community, much more work is needed on the question of reducing fitness evaluation resource consumption.

7.5 Wider Search Spaces

In common with most system-level approaches to search based testing in particular, and automated test data generation in general, Sapienz considers the input to the program to consist solely of user interactions. Other approaches to search-based testing, at the unit level, typically consider a vector of values that can be presented to the unit under test.

However, there has been little work on extending the test data generation search space to include the app's user state and environment. User state and the users' device environment can play a critical role in both elevating coverage of the application under test, and in revealing faults. Some of the faults revealed through different state/environment settings may occur only in certain specific user state and environment settings.

These observations are not peculiar to Facebook, but apply to any software system in which the history of interactions of the user and other state variables and configurations can play a role in determining which path is executed. More research is needed in order to define general approaches to tackling this wider search space.

In our particular situation, we are concerned, naturally, with the 'social state' of the user. For example, a fault may not be witnessed by a test input sequence, unless the user has already responded to at least one post by another user, or has at least one connection in their social network. The user state is thus a part of the wider space in which we search for test cases using SBSE. Characterising the circumstances under which a crash occurs, in terms of this state, would yield highly actionable signal to the developer. It may also prove pivotal in helping to debug otherwise very subtle bugs.

For other apps, and other organisations, the details of the user state will clearly differ, but the general problem of characterising the user state, and the search space it denotes, and defining fitness functions on that representation remains an important, generic, and open research problem. Scientific progress on this open problem is highly likely to yield impactful and actionable research.

We distinguish the user state from the user environment. The environment is general to all applications, while the state is particular to a particular application under test. The environment will, nevertheless, have different impact on different applications. For example, for a photo sharing app, it will likely be important that the user has photos in their photo library on the device. For a map or travel application, GPS settings may prove to be important, although both apps will have access to photos and GPS settings and may use both. In most applications, the network environment will also play an important role.

As devices become more sophisticated, the environment will become ever richer, offering interesting opportunities for SBSE-based characterisations of the search space. More work is needed to characterise this environment in which users execute applications, particularly on mobile devices, to tease out notations for eloquently and succinctly defining this environment. Once characterised, techniques such as Combinatorial Interaction Testing (CIT) [43,60,64] can be used to explore interaction faults, for example.

For the SBSE community, we also need to consider different representations for those environments that promote efficient and effective search. Such work will enrich the search space and tackle several practical problems, such as device fragmentation and context-aware test case generation.

More work is also needed to provide general notations for describing the user state, such that the generic properties of state-based testing can be explored

scientifically and empirically. Such work will shed light on the nature of state-based testing for Internet-deployed, device-executed, applications. This notation may also use and benefit from work on CIT.

Research on user state and environment will have multiple benefits to the research community and to practitioners. For researchers, this work will provide a rich avenue of untapped research questions, with potential insights that may help the research community to understand different kinds of deployment mode, applications, environments and properties of states. For practitioners, this research may help us to better understand the applications we are testing, may help us to go beyond merely revealing faults, and also help us to characterise salient properties that yield deep insights into app performance, usability, and different use-case scenarios for different sub-communities of users.

7.6 Smarter, Unobtrusive and Controllable White Box Coverage

After so many years of software testing, in which instrumentation of the system under test has often played a central role, the reader could be forgiven for believing that the problem of white box coverage assessment is entirely solved. However, while it may be true that white box coverage techniques exist for most languages, platforms and systems, for search based software testing there are more stringent requirements than simply the ability to collect coverage information.

Instrumentation of a system under test changes the behaviour of the system, and these changes can impact on the test data generation technique. Search-based software testing, in particular, may be vulnerable to such influences, where the instrumentation changes timing properties, possibly occluding or revealing race conditions, and other time-based behaviours, differently in the app under the test, when compared to the app in the field.

We need smarter control of white box coverage information, that is minimally obtrusive on the execution characteristics of the app under test. Such techniques need to be smarter. That is, for effective SBSE we need a greater level of control over the parameters that affect the trade-offs between quality of white box coverage information and the impact of collecting this signal.

Some of this work is necessary engineering and implementation detail, but there are also interesting high-level scientific problems. The challenge is to tease out and empirically investigate these trade-offs between quality of signal from white box coverage, and impact of collecting any signal on the system under test.

7.7 Combining Static and Dynamic Analysis and Hybrids of SBSE

Although there has been a great deal of research on static analysis techniques, and dynamic analysis techniques, there has been comparatively less work on the combination of static and dynamic analysis. This ‘blended’ analysis (as it has been called [19]), has the potential for practical impact, since we can leverage the strengths of both techniques to overcome some of the weaknesses of the other.

Equally importantly, such combinations of static and dynamic analysis may yield insights into fundamental questions of computation, touching on the limits imposed by decidability constraints and the connections between statistical and logical inferences.

7.8 False Positives and Pseudo False Positives

It is sometimes claimed, partly as an aphorism [18], that static analysis uses a conservative over approximation, thereby avoiding false negatives (at the expense of some false positives), while dynamic analysis suffers many false negatives, but does not suffer from false positives because it is an under approximation.

This claim is based on an attractive (but misplaced) assumption that the symmetries of over and under approximation, inherent in the theoretical characterisation of static and dynamic analysis, carry over into practice. They do not [37]. Both static *and* dynamic analysis, whether over or under approximating their respective models of computation in production, suffer from both false positives *and* false negatives.

It is well known that static analysis yields false positives when it seeks to offer a conservative over-approximation. However, static analysis, even when construed as a conservative (i.e., 'safe') approach, can also yield false negatives. For example, a static slicing algorithm is only conservative with respect to a set of assumptions, and these assumptions always allow *some* false negatives; dependencies that exist between elements of real systems, yet which go undetected by the slicing algorithm [12].

Dynamic analysis is well-known to suffer from false negatives, due to the inability to exhaustively test (apart from in special circumstances and with respect to strong assumptions, such as integration testing with stream X-machines [39,40]).

At the unit level, dynamic analysis has also been shown to suffer from false positives [30]. However, *even* at the system level, dynamic analysis also suffers from false positives. System level false positives occur in dynamic analyses, such as the testing deployed by Sapienz. We generate tests on a version of the system as close to production as possible. Nevertheless, since the testing tool is not the real end user, there can be differences in behaviour that will cause the testing tool to detect crashes that no real user will ever encounter.

This occurs, for example, due to differences in the device used for testing, and devices used by end users. False positives are also caused by differences in the abilities of the automated testing tool compared to real user abilities; the test tool has an arbitrary number of 'fingers' at its disposal. Finally, due to differences in the deployment environment for the test infrastructure and the production infrastructure used by the end users can also cause false positives.

Sapienz uses emulators to perform test execution. We have found that, because Facebook has 2.2 billion monthly active users (at the time of writing), this means that almost *any* crash we can find with an emulator can be found on *some* device in the real world. Therefore, we have *not* experienced a large number of false positives, simply due to our use of emulators, rather than real devices.

Nevertheless, we have witnessed a kind of pseudo false positive due to implementation details concerning emulators, such as inadequate emulation of Augmented Reality (AR) features in earlier instances of the Android emulator API (e.g., API 19, which lacks the relevant AR library support). This leads to crashes which are 'true' positives, strictly speaking (since the system should not crash if the AR library is not present). However, we need to treat such a crash like a 'pseudo false positive', since it instantly crashes the app and thereby prohibits further testing, yet it is unlikely to be a priority for fixing (since such a crash tends not to fire in the field, although it *could* in *theory*).

This observation of 'pseudo' false positives suggests a spectrum in which a completely false positive lies at one extreme, but for which a pseudo false positive, that is exceptionally unlikely to occur in practice, lies close to the 'fully false' positives; it shares many of the practical characteristics of 'fully false' positives.

In deployment scenarios where there are only relatively few end users, and these end users only use a strict subset of the available Android devices available, deployment of automated testing techniques, like Sapienz, may also yield further pseudo false positives (which we do not tend to witness at Facebook) due to the differences in test devices and end-user devices.

The degree of 'pseudo falseness' of a crash signal is, effectively, a function of the number of likely end users, since this is a determinant of the probability that a test-time crash will be found in production. As testers at Facebook, we are thus blessed by the relatively large number of end users we serve, because of the way in which this number tends to reduce pseudo false positiveness to a minimum; almost any signal concerning crashes is treated as a true positive by our developer community.

End user abilities may also differ from those of the automated testing system. The primary difference we have noticed lies in the speed with which the automated test sequence can be executed on an emulator; potentially far faster than that achieved by any human. Once again, this has led to fewer false positives than we initially expected. The wide variety of different Android devices in circulation means that test sequences executed by a user on a slower device may have similar characteristics to a faster-executed test sequence on a higher-end device. Nevertheless, some false positives can occur due to speed of test sequence execution.

A further source of difference between test user ability and real user ability, lies in the exceptionally dextrous nature with which an automated test technique can interact with the device or emulator. Effectively, the test user is not hampered by physical constraints imposed by the number fingers and thumbs on a typical human hand, and their varying degrees of freedom to articulate. Previously, it was proposed to use robotic testing to achieve fully black box testing, thereby avoiding this potential source of false positives [51]. This is not something we have currently deployed, but it remains an option, should the false positive problem ever become more pernicious.

7.9 Unit Tests from System Tests

Sapienz generates system-level tests, that test the application, end to end, imitating as closely as possible the behaviour of real users. This system-level approach significantly reduces the propensity of automated test design to produce false positives, that have been widely reported to occur for more lower-level, unit level, testing [30]. However, an interesting research problem lies in converting system level test sequences into corresponding unit level test information, thereby circumventing the unit level false positive problem, while simultaneously facilitating unit testing through automated test design.

One possibility lies in generating a set of system-level tests [21], instrumented to witness the pre- and post-condition state for some unit under test, and the subsequent use of likely invariant inference, such as Daikon [23], to infer the constraints that apply at the unit level. With these inferred constraints in hand, automated test design (at the unit level) can now proceed within the constrained search space, thereby reducing the incidence of unit-level false positives.

7.10 Combining Human- and Machine- Designed Tests

Humans have designed tests for many years. Automated test design techniques, like Sapienz, might reduce human effort and thereby minimize the 'friction' of the test design process, but they are unlikely to fully replace humans. After all, humans have domain knowledge and engineers can link this domain knowledge to specific aspects of code. There is a productive benefit in finding hybrids that can combine human- and machine-designed tests, but this remains an open research challenge.

One possibility is to extract assertions from human-designed tests and re-use them as partial oracles for machine-designed test cases. Humans' domain knowledge is an important resource, while automating the test oracle design process is non-trivial. Perhaps human-written tests can be mined for re-usable test oracle information in the form of assertions extracted from human-designed test cases.

Another possibility would be for the human test to act as a prefix to the machine-designed test. Perhaps the human test might move the system into a state that is hard to reach, but important, or it may simply do so more efficiently than a machine-designed test. Perhaps a human-designed test prefix might establish a state of interest or set up a particular environmental configuration that enables machine-designed tests. For all these reasons, it makes sense to use a human-designed test as a prefix for a Sapienz (or other automatically designed) test. More research is needed on techniques to best combine human-designed and machine-designed test cases.

7.11 Enhancing the Debug Payload

Far too little research is undertaken on the important problem of debugging [37]. Many problems in software debugging can be characterised in terms of multi-objective search. Therefore, we believe the SBSE community has a role to play in tackling this important and under-researched set of challenges.

In some cases, as much as 50% of engineers' time spent on programming may be devoted to the problem of debugging in its various forms. It is therefore surprising (and disappointing) that there is not a single dedicated annual international academic conference, nor any scientific journal dedicated to the perennially important problem of automated support for debugging.

We would like to encourage the software engineering community, more generally, and the Search Based Software Engineering community, in particular, to renew research interest and activity on debugging. Even if automated software repair were ultimately able to remove the need for human debugging effort, the problem of automated debugging would remain a pressing one. That is, techniques that supply additional context and guidance to a human concerned with complex debugging tasks, would also be likely to provide useful input to improve the efficiency and effectiveness of automated program repair. This potential dual use of debugging support, for both human-based debugging activity and automated program repair, makes it all the more important that we should see significant attention and energy devoted to techniques to support debugging activity, whether that activity be by machine or by human hand.

7.12 Search in the Presence of Inherent Flakiness

Search-based software engineering is well-adapted to tackle the problems of test flakiness [37,46,57,62]. We envisage a bright future for probabilistic approaches to testing, and believe that the SBSE community has an important role to play here. By making flakiness of first class property of test cases, and test suites, we can optimise for this property. Furthermore, by measuring the signal and signal-to-noise ratio [37] produced by automated testing tools, we can define these evaluation criteria, and potentially also surface them as fitness functions.

Addressing the Assume Tests Are Flakey (ATAFistic) world [37], we may construct a fully probabilistic formulation of software testing and verification. We hope to formulate and investigate new concepts of correctness, better-fitted to Internet-based deployment than their precursors that were generally initially constructed in the era of mainframes and stand alone desk tops with low numbers of inter-connections.

Probabilistic testing and verification is nascent in the work in information theory for testing [73,75] and probabilistic model checking [15,44], but more work is required on theoretical foundations to unify testing and verification within a fully probabilistic framework. More work is also required to develop the fruits of such foundations in practical, scalable and deployable techniques for probabilistic testing and verification.

7.13 New Search Algorithms that Fully Realize Efficiently Deployable Parallelism at Scale

Search-based software testing systems have rested on evolutionary computing as one of the primary search techniques to explore the space of all candidate inputs to the system under test [33,55]. While there has been some work on

parallelisation to achieve the, oft-stated but seldom witnessed, 'embarrassing parallelism' of SBSE [8,58,76], there has been little work on the formulation of SBSE algorithms to better fit modern distributed computational resources.

The bottleneck for most search based testing (and much of SBSE, more generally), lies in the computation of fitness for a candidate test input sequence (more generally, of a software engineering artefact). The distribution of computation times for SBSE fitness inherently involves a high degree of variance, due to the highly stochastic nature of software deployment for Internet-based computing.

Therefore, any approach based on iterative generations of the population is inherently inefficient when we require that all members of the population have to be evaluated in lockstep. We believe there is further work to be done on extending, rethinking, and redefining the underlying evolutionary algorithms. We need to fully decouple unnecessary interdependence between fitness computations, so that maximal parallelism can be achieved; algorithms that can fully exploit asynchronous fitness evaluation will scale well with available parallel computation resources.

Furthermore, closer integration of evolutionary algorithm technology with predictive modelling and machine learning is required in order to better use computational resources for static fitness estimation. For example, predicting likely fitness outcomes and maximising the quantity of information derived from each fitness outcome are both important concerns to maximise the impact of Search Based Software Engineering.

7.14 Automated Fixing

Automated software repair remains an active topic in the research community [29]. At Facebook we find ourselves in an excellent position to act as both a producer and consumer of research and development on deployable automated program repair; as this paper explains we have infrastructure in place for automated testing and for fix detection. We would be particularly interested to collaborate with the academic research community on this topic.

7.15 Automated Performance Improvement

Generalising from automated repair to genetic improvement [65], and related topics in program synthesis [31], we also see great potential for research in automating program improvement, particularly for non-functional properties, such as performance-related behaviours and resource consumption characteristics [34].

We would also be very interested to collaborate with the academic community on scalable and deployable automated program improvement techniques. With Sapienz now deployed at Facebook, we are in a good position to provide the automated test input generation infrastructure that would be a natural prerequisite for the deployment of test-based program improvement and synthesis techniques such as genetic improvement.

Facebook also has static analysis and verification technology in the form of Infer [13], as well as dynamic test design infrastructure (Sapienz) and manually-designed automated execution frameworks (such as the Buddy end-to-end testing system). We are therefore also in an excellent position to offer collaborative support to academics seeking to tackle the FiGiVerify challenge; Finding issues (bugs, performance, resource-related), Genetically Improving them (to synthesize improvements), and Verifying the improvements' correctness [37].

8 The History of Sapienz Deployment to Date

Sapienz was developed as a research prototype, which was initially proposed by Ke Mao and grew out of his PhD work [47]. The first version of Sapienz was described in the Ke's thesis [47] and at ISSTA 2016 [49], the International Symposium on Software Testing and Analysis (ISSTA 2016). This version was made publicly available as a research prototype[6]. The research prototype found 558 unique crashes among the top 1,000 Android apps, several of which were reported and fixed [49].

The three authors of the ISSTA 2016 paper (Ke Mao and his two PhD supervisors, Mark Harman and Yue Jia) launched an Android testing start-up, called Majicke Ltd., in September 2016, with Ke at the CTO, Yue as CEO and Mark as scientific advisor. Majicke's technical offering was based on Sapienz. The three subsequently moved to Facebook[7] on February 6th 2017, where they founded the Sapienz team at Facebook London, with Ke moving into the role of technical lead, Yue focusing on long term technical issues (the vital 'important but not urgent') and Mark taking up the role of team manager.

The Facebook Sapienz Team's role is to deploy, develop and research SBSE-related techniques for automated test case design so that we can have Friction-Free Fault Finding and Fixing. The team has been strongly supported by Facebook's Developer Infrastructure team (DevInfra).

The Sapienz team has grown significantly since then and now includes (or has included) the authors of this paper. Taijin Tei subsequently moved to work for another team, while Alexander Mols has worked and continues to work part time on Sapienz and other projects. The remaining authors of this paper have worked full time, continuously, on Sapienz since starting work at Facebook. Many others at Facebook have helped with support, advice and other contributions and we thank them in the acknowledgements of this paper.

The Sapienz team's many partners, collaborators and supporters in the Facebook developer community have also provided a wealth of support, advice and collaboration. Their willingness to try new technologies and to explore and experiment was evident from the very outset. Facebook's open engineering culture has greatly accelerated the deployment of SBSE at Facebook.

[6] https://github.com/Rhapsod/sapienz.

[7] http://www.engineering.ucl.ac.uk/news/bug-finding-majicke-finds-home-facebook/.

9 Conclusions

We have outlined the primary features of the deployment of the Sapienz Search Based Testing system at Facebook, where it is currently testing Facebook's Android social media and messaging apps. These are two of the largest and most widely-used apps in the overall international Android ecosystem. Work is under way to extend to other apps and platforms and to improve the algorithms and technology on which Sapienz relies.

To achieve rapid development of research into production, we use an Automated Scientific Experimental Reporting (ASER) framework, which automates experiments from proof of concept, through inferential statistical testing to full experiment-to-prod A/B testing.

We also outline some of the challenges and open problems that we believe are suitable for tackling by the automated testing and SBSE research communities, based on our experience from this Search Based Software Testing deployment work.

Acknowledgement. Thanks to all our colleagues at Facebook and in the scientific research, open source and developer communities for their support, both technical and non-technical, that has allowed us to so-rapidly deploy search based system-level testing into regular production. Many people at Facebook have helped with the deployment work reported on here in this keynote paper. We would like to thank these colleagues who gave of their time and support while at Facebook, including Sharon Ayalde, Michelle Bell, Josh Berdine, Kelly Berschauer, Andras Biczo, Megan Brogan, Andrea Ciancone, Satish Chandra, Marek Cirkos, Priti Choksi, Wojtek Chmiel, Dulma Churchill, Dino Distefano, Zsolt Dollenstein, Jeremy Dubreil, Jeffrey Dunn, David Erb, Graham French, Daron Green, Lunwen He, Lawrence Lomax, Martino Luca, Joanna Lynch, Dmitry Lyubarskiy, Alex Marginean, Phyllipe Medeiros, Devon Meeker, Kristina Milian, Peter O'Hearn, Bryan O'Sullivan, Lauren Rugani, Evan Snyder, Don Stewart, Gabrielle Van Aacken, Pawel Wanat, and Monica Wik. We sincerely apologise to any who we omitted to mention here.

References

1. Abdessalem, R., Nejati, S., Briand, L., Stifter, T.: Testing vision-based control systems using learnable evolutionary algorithms. In: 40th International Conference on Software Engineering (ICSE 2018) (to appear)
2. Afzal, W., Torkar, R., Feldt, R., Wikstrand, G.: Search-based prediction of fault-slip-through in large software projects. In: Second International Symposium on Search Based Software Engineering (SSBSE 2010), Benevento, Italy 7–9 September 2010, pp. 79–88 (2010)
3. Alesio, S.D., Briand, L.C., Nejati, S., Gotlieb, A.: Combining genetic algorithms and constraint programming to support stress testing of task deadlines. ACM Trans. Softw. Eng. Methodol. **25**(1), 4:1–4:37 (2015)
4. Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, Kansas, USA, 6th–10th November 2011, pp. 3–12 (2011)

5. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: International Symposium on Software Testing and Analysis (ISSTA 2014), pp. 181–192. ACM (2014)
6. Androutsopoulos, K., Clark, D., Dan, H., Harman, M., Hierons, R.: An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, pp. 573–583, June 2014
7. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 33rd International Conference on Software Engineering (ICSE 2011), pp. 1–10. ACM, New York (2011)
8. Asadi, F., Antoniol, G., Guéhéneuc, Y.: Concept location with genetic algorithms: a comparison of four distributed architectures. In: 2nd International Symposium on Search based Software Engineering (SSBSE 2010), pp. 153–162. IEEE Computer Society Press, Benevento (2010)
9. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
10. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold (1990)
11. Bertolino, A.: Software testing research: achievements, challenges, dreams. In: Briand, L., Wolf, A. (eds.) Future of Software Engineering 2007. IEEE Computer Society Press, Los Alamitos (2007)
12. Binkley, D., Gold, N.E., Harman, M., Islam, S.S., Krinke, J., Yoo, S.: ORBS and the limits of static slicing. In: 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015), pp. 1–10. IEEE, Bremen, September 2015
13. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
14. Ceccarelli, M., Cerulo, L., Canfora, G., Penta, M.D.: An eclectic approach for change impact analysis. In: Kramer, J., Bishop, J., Devanbum, P.T., Uchitel, S. (eds.) 32nd ACM/IEEE International Conference on Software Engineering (ICSE), vol. 2, pp. 163–166. ACM (2010)
15. Chechik, M., Gurfinkel, A., Devereux, B.: ξChek: a multi-valued model-checker. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 505–509. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_41
16. Chen, Y.F., Rosenblum, D.S., Vo, K.P.: TestTube: a system for selective regression testing. In: 16th International Conference on Software Engineering (ICSE 1994), pp. 211–220. IEEE Computer Society Press (1994)
17. Clark, D., Hierons, R.M.: Squeeziness: an information theoretic measure for avoiding fault masking. Inf. Process. Lett. **112**(8–9), 335–340 (2012)
18. Dijkstra, E.W.: Structured programming (1969). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF, circulated privately
19. Dufour, B., Ryder, B.G., Sevitsky, G.: Blended analysis for performance understanding of framework-based applications. In: International Symposium on Software Testing and Analysis, ISSTA 2007, 9–12 July, London, UK, pp. 118–128. ACM (2007)
20. Dunn, J., Mols, A., Lomax, L., Medeiros, P.: Managing resources for large-scale testing, 24 May 2017. https://code.facebook.com/posts/1708075792818517/managing-resources-for-large-scale-testing/
21. Elbaum, S.G., Chin, H.N., Dwyer, M.B., Jorde, M.: Carving and replaying differential unit test cases from system test cases. IEEE Trans. Softw. Eng. **35**(1), 29–45 (2009)

22. Engström, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Inf. Softw. Technol. **52**(1), 14–30 (2010)
23. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. **27**(2), 1–25 (2001)
24. Facebook Research: Facebook Research post describing the move of Majicke to Facebook (2017). https://facebook.com/academics/posts/1326609954057075
25. Feitelson, D.G., Frachtenberg, E., Beck, K.L.: Development and deployment at Facebook. IEEE Internet Comput. **17**(4), 8–17 (2013)
26. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), pp. 416–419. ACM, 5th–9th September 2011
27. Fraser, G., Arcuri, A.: The seed is strong: seeding strategies in search-based software testing. In: Antoniol, G., Bertolino, A., Labiche, Y. (eds.) 5th International Conference on Software Testing, Verification and Validation (ICST 2012), pp. 121–130. IEEE, Montreal, April 2012. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6200016
28. Gao, Z., Liang, Y., Cohen, M.B., Memon, A.M., Wang, Z.: Making system user interactive tests repeatable: when and what should we control? In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th International Conference on Software Engineering (ICSE 2015), pp. 55–65. IEEE Computer Society, Florence, 16–24 May 2015
29. Goues, C.L., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Softw. Qual. J. **21**(3), 421–443 (2013)
30. Gross, F., Fraser, G., Zeller, A.: Search-based system testing: high coverage, no false alarms. In: International Symposium on Software Testing and Analysis (ISSTA 2012), pp. 67–77 (2012)
31. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012)
32. Harman, M., Burke, E., Clark, J.A., Yao, X.: Dynamic adaptive search based software engineering (keynote paper). In: 6th IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012), Lund, Sweden, pp. 1–8, September 2012
33. Harman, M., Jia, Y., Zhang, Y.: Achievements, open problems and challenges for search based software testing (keynote paper). In: 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), Graz, Austria, April 2015
34. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The GISMOE challenge: constructing the Pareto program surface using genetic programming to find better programs (keynote paper). In: 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Essen, Germany, pp. 1–14, September 2012
35. Harman, M., Mansouri, A., Zhang, Y.: Search based software engineering: trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012)
36. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based software engineering: techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) LASER 2008-2010. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25231-0_1

37. Harman, M., O'Hearn, P.: From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis (keynote paper). In: 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018), Madrid, Spain, 23rd–24th September 2018, to appear
38. Hazelwood, K., et al.: Applied machine learning at Facebook: a datacenter infrastructure perspective. In: 24th International Symposium on High-Performance Computer Architecture (HPCA 2018), Vienna, Austria, 24–28 February 2018
39. Hierons, R.M., Harman, M.: Testing against non-deterministic stream X-machines. Formal Aspects Comput. **12**, 423–442 (2000)
40. Ipate, F., Holcombe, M.: Generating test sequences from non-deterministic X-machines. Formal Aspects Comput. **12**(6), 443–458 (2000)
41. Jahangirova, G., Clark, D., Harman, M., Tonella, P.: Test oracle assessment and improvement. In: International Symposium on Software Testing and Analysis (ISSTA 2016), pp. 247–258 (2016)
42. Jan, S., Panichella, A., Arcuri, A., Briand, L.: Automatic generation of tests to exploit XML injection vulnerabilities in web applications. IEEE Transactions on Software Engineering (2018, to appear)
43. Jia, Y., Cohen, M.B., Harman, M., Petke, J.: Learning combinatorial interaction test generation strategies using hyperheuristic search. In: 37th International Conference on Software Engineering (ICSE 2015), Florence, Italy, pp. 540–550 (2015)
44. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46029-2_13
45. Lakhotia, K., Harman, M., Gross, H.: AUSTIN: an open source tool for search based software testing of C programs. J. Inf. Softw. Technol. **55**(1), 112–125 (2013)
46. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Cheung, S.C., Orso, A., Storey, M.A. (eds.) 22nd International Symposium on Foundations of Software Engineering (FSE 2014), pp. 643–653. ACM, Hong Kong, 16–22 November 2014
47. Mao, K.: Multi-objective Search-based Mobile Testing. Ph.D. thesis, University College London, Department of Computer Science, CREST centre (2017)
48. Mao, K., Capra, L., Harman, M., Jia, Y.: A survey of the use of crowdsourcing in software engineering. J. Syst. Softw. **126**, 57–84 (2017)
49. Mao, K., Harman, M., Jia, Y.: Sapienz: multi-objective automated testing for Android applications. In: International Symposium on Software Testing and Analysis (ISSTA 2016), pp. 94–105 (2016)
50. Mao, K., Harman, M., Jia, Y.: Crowd intelligence enhances automated mobile testing. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 16–26 (2017)
51. Mao, K., Harman, M., Jia, Y.: Robotic testing of mobile apps for truly black-box automation. IEEE Softw. **34**(2), 11–16 (2017)
52. Marculescu, B., Feldt, R., Torkar, R., Poulding, S.: Transferring interactive search-based software testing to industry. J. Syst. Softw. **142**, 156–170 (2018)
53. Martin, W., Sarro, F., Harman, M.: Causal impact analysis for app releases in Google Play. In: 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016), Seattle, WA, USA, pp. 435–446 November 2016
54. Matinnejad, R., Nejati, S., Briand, L., Bruckmann, T.: Test generation and test prioritization for simulink models with dynamic behavior. IEEE Trans. Softw. Eng. (2018, to appear)

55. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004)
56. Memon, A.M., Cohen, M.B.: Automated testing of GUI applications: models, tools, and controlling flakiness. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering (ICSE 2013), pp. 1479–1480. IEEE Computer Society, San Francisco, 18–26 May 2013
57. Memon, A.M., et al.: Taming Google-scale continuous testing. In: 39th International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP), pp. 233–242. IEEE, Buenos Aires, 20–28 May 2017
58. Mitchell, B.S., Traverso, M., Mancoridis, S.: An architecture for distributing the computation of software clustering algorithms. In: IEEE/IFIP Working Conference on Software Architecture (WICSA 2001), pp. 181–190. IEEE Computer Society, Amsterdam (2001)
59. Mansour, N., Bahsoon, R., Baradhi, G.: Empirical comparison of regression test selection algorithms. Syst. Softw. **57**(1), 79–90 (2001)
60. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 11:1–11:29 (2011)
61. Ouni, A., Kessentini, M., Sahraoui, H.A., Inoue, K., Deb, K.: Multi-criteria code refactoring using search-based software engineering: an industrial case study. ACM Trans. Softw. Eng. Methodol. **25**(3), 23:1–23:53 (2016)
62. Palomba, F., Zaidman, A.: Does refactoring of test smells induce fixing flakey tests? In: International Conference on Software Maintenance and Evolution (ICSME 2017), pp. 1–12. IEEE Computer Society (2017)
63. Pearl, J.: Causality. Cambridge University Press, Cambridge (2000)
64. Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, pp. 26–36. ACM, Saint Petersburg, August 2013
65. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Trans. Evol. Comput. (2018, to appear)
66. Podgurski, A., et al.: Automated support for classifying software failure reports. In: 25th International Conference on Software Engineering (ICSE 2003), pp. 465–477. IEEE Computer Society, Piscataway, 3–10 May 2003
67. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. **22**(8), 529–551 (1996)
68. Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval. Inf. Process. Manag. **24**(5), 513–523 (1988)
69. Shin, S.Y., Nejati, S., Sabetzadeh, M., Briand, L., Zimmer, F.: Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In: International Symposium on Software Testing and Analysis (ISSTA 2018) (to appear)
70. Tillmann, N., de Halleux, J., Xie, T.: Transferring an automated test generation tool to practice: from Pex to Fakes and Code Digger. In: 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 385–396 (2014)
71. Tracey, N., Clark, J., Mander, K.: The way forward for unifying dynamic test-case generation: the optimisation-based approach. In: International Workshop on Dependable Computing and Its Applications (DCIA), IFIP, pp. 169–180, January 1998

72. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. **43**(14), 841–854 (2001)
73. Yang, L., Dang, Z., Fischer, T.R., Kim, M.S., Tan, L.: Entropy and software systems: towards an information-theoretic foundation of software testing. In: 2010 FSE/SDP Workshop on the Future of Software Engineering Research, pp. 427–432, November 2010
74. Yoo, S., Harman, M.: Regression testing minimisation, selection and prioritisation: a survey. J. Softw. Testing Verif. Reliab. **22**(2), 67–120 (2012)
75. Yoo, S., Harman, M., Clark, D.: Fault localization prioritization: comparing information theoretic and coverage based approaches. ACM Trans. Softw. Eng. Methodol. **22**(3), Article no. 19 (2013)
76. Yoo, S., Harman, M., Ur, S.: GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards. J. Empir. Softw. Eng. **18**(3), 550–593 (2013)
77. Yoo, S., Nilsson, R., Harman, M.: Faster fault finding at Google using multi objective regression test optimisation. In: 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), Szeged, Hungary, 5th–9th September 2011. Industry Track