From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis

Mark Harman*, Peter O'Hearn*
*Facebook London and University College London, UK

***Abstract*—This paper[1] describes some of the challenges and opportunities when deploying static and dynamic analysis at scale, drawing on the authors' experience with the Infer and Sapienz Technologies at Facebook, each of which started life as a research-led start-up that was subsequently deployed at scale, impacting billions of people worldwide.**

The paper identifies open problems that have yet to receive significant attention from the scientific community, yet which have potential for profound real world impact, formulating these as research questions that, we believe, are ripe for exploration and that would make excellent topics for research projects.

I. Introduction

How do we transition research on static and dynamic analysis techniques from the testing and verification research communities to industrial practice? Many have asked this question, and others related to it. A great deal has been said about barriers to adoption of such techniques in industry, and the question and variations of it form the basis for the perennial panel sessions that spring up at countless international conferences.

In this paper, we wish to make a contribution to this discussion informed by our experience with the deployment of the static analysis technique Infer [23], [28], [29], [109], and the dynamic analysis technique Sapienz [45], [89], [91], at Facebook.

We do not claim to have all, nor even many answers, and our experience may be somewhat specific to continuous deployment in the technology sector in general and, perhaps in some places, to Facebook alone in particular. Nevertheless, we believe that our relatively unusual position as both professors (in Programming Languages and Software Engineering) and also engineering managers/engineers in a large tech sector company, may offer us a perspective from which we can make a few useful contributions to this ongoing 'deployment debate'.

We explain some of the myths, prevalent in Programming Languages and Software Engineering research communities, many of which we have, ourselves, initially assumed to be merely 'common sense', yet which our more recent experience in industry has challenged. We also seek to identify attributes of research prototypes that make them more or less suitable to deployment, and focus on remaining open problems and research questions that target the most productive intersection we have yet witnessed: that between exciting, intellectually challenging science, and real-world deployment impact.

Many industrialists have perhaps tended to regard it unlikely that much academic work will prove relevant to their most pressing industrial concerns. On the other hand, it is not uncommon for academic and scientific researchers to believe that most of the problems faced by industrialists are either boring, tedious or scientifically uninteresting. This sociological phenomenon has led to a great deal of miscommunication between the academic and industrial sectors.

We hope that we can make a small contribution by focusing on the intersection of challenging and interesting scientific problems with pressing industrial deployment needs. Our aim is to move the debate beyond relatively unhelpful observations we have typically encountered in, for example, conference panels on industry-academia collaboration. Here is a list of such (perhaps uncharitably paraphrased) observations:

1) **Irrelevant**: 'Academics need to make their research more industrially-relevant',
2) **Unconvincing**: 'Scientific work should be evaluated on large-scale real-world problems',
3) **Misdirected**: 'Researchers need to spend more time understanding the problems faced by the industry',
4) **Unsupportive**: 'Industry needs to provide more funding for research',
5) **Closed System**: 'Industrialists need to make engineering production code and software engineering artifacts available for academic study', and
6) **Closed Mind**: 'Industrialists are unwilling to adopt promising research'.

While some of these observations may be true some of the time, focussing any further attention on them leads both communities into an unproductive dead end; none of these paraphrased quotations provides much practical actionable guidance that can be used by the scientific research community to improve the deployability of research prototypes in industry.

We believe (and have found in practice) that practitioners are, in the right circumstances, open minded and supportive and willing to adopt and consider academic research prototypes. More specifically,

> **With the right kind of scientific evidence in the evaluation, industrialists are very willing to adopt, deploy and develop research.**

[1]This paper accompanies the authors' joint keynote at the 18th IEEE International Working Conference on Source Code Analysis and Manipulation, September 23rd-24th, 2018 - Madrid, Spain

In this paper we try to elucidate how the research community might further and better provide such scientific evidence. The research prototypes need not be immediately deployable. This would require significant additional engineering effort that the academics could not (and should not) undertake. However, with the right kind of research questions and consequent evidence in the scientific evaluation, researchers can demonstrate an elevated likelihood that, with such further engineering effort from industry, there will exist effective and efficient deployment routes.

Even where the immediate goal of the research is *not* deployment, we believe that consideration of some of these research questions and scientific evaluation criteria may improve the underlying science. For example, by clarifying the intended deployment use case, the researcher is able to *widen* the number of techniques that become plausible thereby ensuing that the community retains promising techniques that might happen to fall foul of some, in vogue evaluation criterion.

In the paper we provide a set of open problems and challenges for the scientific research community in testing and verification. We believe that these open problems are, and will remain, pertinent to software deployment models that exhibit continuous integration and deployment.

Although this constitutes a large section of the overall industrial sector, we do not claim that these challenges necessarily retain their importance when extended beyond continuous integration and deployment to other parts of the Software Engineering sector. We would be interested to explore collaboration opportunities that seek to tackle these open problems.

The paper is structured as follows:

- **Background**. Section II: Our background lies in the scientific community, while the principal focus of our current work is in industrial deployment of static and dynamic analysis. Section II provides a brief overview of this industrial deployment to set this paper in the context of our recent industrial experience.
- **ROFL Myth**. Section III: We initially fell into a trap, which we characterise as believing in a myth, the 'Report Only Fault/Failure List' (ROFL) myth, which we describe in Section III. We want to surface this myth in Section III, because we believe it may still enjoy widespread tacit support in the approach to research adopted by the community. We have experienced the pernicious effect that misplaced ROFL belief can have on the deployability of research work.
- **Compositionality and Incrementality**. Sections IV and V: We have found that the related, but distinct, properties of compositionality and incrementality are important to the scalability of testing and verification through static and dynamic analysis.
- **Assume Tests Are Flaky**. Section VI: A flaky test is one for which a failing execution and a passing execution are observed on two different occasions yet for both executions, all environmental factors that the tester seeks to control remain identical. We believe more work is needed on flakiness of tests. We discuss this problem in Section VI, where our focus is on moving beyond identifying and reducing flakiness, to coping with and optimising for flaky tests. We characterise this as the need to recognise that the world into which we deploy testing and verification techniques is increasingly one where we could achieve greater impact by assuming that *all tests are flaky*.
- **TERF Ratio**. Section VII. The Test-Execution-to-Release-Frequency (TERF) Ratio is changing. We surface the issue of accounting for (and reducing) test execution cost in terms of this ratio in Section VII.
- **Fix Detection**. Section VIII: We had thought, before we migrated to our industrial roles, that the problem of determining whether a bug is fixed was a relatively trivial and entirely solved problem. That was another misconception; the problem of fix detection is a challenging one and, we believe, has not received sufficient attention from the research community.
- **Testability Transformation**. Section IX: As a community, we regularly use code transformation in order to facilitate testing and verification, for example, by mocking procedures and modeling subsystems for which code is unavailable. Sadly, this practice lacks a fully formal foundation, a problem that we briefly touch on in Section IX.
- **Evaluation**. Section X: We would like to suggest some possible dimensions for scientific evaluation of research prototypes, results along which we believe would provide compelling evidence for likely deployability. These evaluation criteria, discussed in Section X, are not an additional burden on researchers. Rather, they offer some alternative ways in which research work may prove actionable and important, even if it might fail to meet currently accepted evaluation criteria.
- **Deployability**. Section XI: Our experience lies primary in the challenges of deployment within continuous integration environments, which are increasingly industrially prevalent [101]. In Section XI we describe some of the lessons we learned in our efforts to deploy research on testing and verification at Facebook, focusing on those lessons that we believe to be generic to all continuous integration environments, but illustrating with specifics from Facebook's deployment of Infer and Sapienz.
- **Find, Fix and Verify**. Section XII: Finally, we could not conclude without returning to a grand challenge. The challenge combines the two authors' research interests and, we believe, has the potential for profound and lasting impact on both the scientific and practitioner community. The challenge is the find-fix-verify challenge (FiFiVerify), which we believe is within the grasp of the scientific community. Achieving FiFiVerify would lead to practical deployed systems that are able to find failing functionality and poor performance, automatically fix them, and automatically verify the fixes' correctness and the absence of regressions as a result of fixes. The FiFiVerify challenge has been described previously elsewhere [68]. In Section XII, we generalise it to the FiGiVerify problem of finding (Fi) functional or non-functional issues, (genetically) improving (Gi) them and verifying the improvement.

II. Background

The authors' industrial experience arises primarily from their work at Facebook, where they are involved in the development of the static analysis tool Infer, and the dynamic analysis tool Sapienz. This section provides a brief overview of these two technologies, as the background to the observations made subsequently about scalable static and dynamic analysis. Naturally, we hope that our observations will extend and apply more widely than Facebook. Our observations are sufficiently general in nature that we believe they will be found applicable to any tech sector organisation that deploys software into the Internet and/or mobile application market, based on continuous integration and deployment [46], [101].

There are millions of lines of code in Facebook's Android app alone, while there are also hundreds of thousands of commits per week to the core software repositories maintained by the company in its continuous integration and deployment framework. The technical challenges raised by continuous integration and deployment are felt, not only at Facebook, but across the sector as a whole. For example, Memon et al. have commented on these challenges in the context of scaling testing at Google, in their excellent ICSE-SEIP paper [101].

The combination of code size and change frequency that comes with continuous integration and deployment puts us, as research scholars making a transition to industrial deployment and practice, in a very fortunate and privileged position. Working at Facebook has given us opportunities to deploy the Infer and Sapienz static and dynamic analysis techniques at scales that are possible in few other environments. We have benefited greatly from the considerable support, understanding (and occasionally from the necessary forbearance) of the Developer Infrastructure community and leadership at Facebook.

The Facebook culture of move fast, fail fast, bold experimentation and explore within an open, collaborative and technically measurable and accountable environment has meshed perfectly with our research and scientific instincts and *modus operandi*. We believe that the relationship between academic research and the tech sector is changing, much for the better. Indeed, continuous integration and deployment is, in essence, nothing more than an enormous, continuous and highly exploratory scientific experiment.

A. Infer: Static Analysis at Scale

Infer is a static analysis tool applied to Java, Objective C and C++ code bases at Facebook. It grew out of academic work on Separation Logic [108], [109], which attempted to scale algorithms for reasoning about memory safety of programs with embedded pointers – one of the most pressing verification challenges of the 2000s – from 1,000s LOC [133] to multiple 1,000,000s LOCs [31]. Infer arrived at Facebook with the acquisition of the program proof startup Monoidics in 2013, and its deployment has resulted in tens of thousands of bugs being fixed by Facebook's developers before they reach production. Infer is open source [29] and is used as well at a number of other companies, including AWS, Mozilla, JD.com and Spotify.

Fig. 1. Continuous Development and Deployment. Diff time is the time when a developer submits a Diff (a code change) to the system for review. Post land is the period after a diff has been incorporated into the master build of the system. CI refers to the Continuous Integration of diffs into the code base subject to code review.

Infer is notable for performing a 'deep' static analysis of source code. It uses inter-procedural analysis and follows pointer chains, yet still scales to large code bases. In comparison, popular open-source tools such as Findbugs and Clang Static Analyzer do indeed scale, but limit their reasoning, typically to a single file. A 2017 study of 100 recent fixes, committed in response to Infer reports, in several bug categories, found several categories for which the majority of the bugs were inter-procedural [22], confirming that analysis beyond intra-procedure reasoning can produce value. On the other hand, while many research tools may often offer inter-procedural analysis, they typically require a sophisticated whole-program analysis, which may be precise, but sadly cannot scale to 1,000,000s of lines of code.

Infer scales by implementing a novel compositional program analysis, where the analysis result of a composite program is computed from the analysis results of its parts [31]. Compositionality presupposes that 'analysis result of a part' is meaningful without having the whole program, and allows for an incremental deployment which fits well with Facebook's software development model.

Facebook practices *continuous development* where a shared code base is altered by thousands of programmers submitting 'diffs' (code modifications). A programmer prepares a diff, and submits it to the code review system. Infer participates as a bot, writing comments for the programmer and other human reviewers to consider. Figure 1 shows a simplified picture of this process. The developers share access to a single codebase and then land, or commit, a diff to the codebase after passing code review. Infer is run at diff time, before land, while longer-running perf and other tests are run post-land. Post-land is also where employee dogfooding occurs.

When a diff is submitted, an instance of Infer is sparked up in Sandcastle, Facebook's internal *continuous integration system*. Because of compositionality, Infer does not need to process the entire code base in order to analyze a diff and, as a result, it is very fast. For example, a recent sampling of Infer's diff analysis within Sandcastle for Facebook's Android App found that it was delivering comments to developers in 12 minutes (on average) whereas, were Infer to perform a whole-program analysis on the entire app it would take over 1 hour. There is a research paper describing Infer's deployment as of 2015 [28], and a video of a talk on our experience with static

analysis at scale from the CurryOn 2016 conference[2].

Thus, Infer has its roots in program verification research, but it deploys proof technology in a novel way, in what might be termed *continuous reasoning* [107]. Proof technology is not used to show that a complete program is absent of all errors, but rather, to provide static reasoning about the source code to cover many paths through an app at once. It is used on diffs to prevent regressions, implementing a form of what is sometimes referred to as 'differential analysis' [81], but there is no claim that *all* regressions are caught (except sometimes up to certain assumptions).

Remarkably, although program proving techniques have traditionally been considered to be expensive, Infer is seen internally as fast; deployed at the same place as unit tests in continuous integration, and before human-designed end-to-end tests.

Infer started as a specialized analysis based on Separation Logic [108] targeting memory issues, but has now evolved into an analysis framework supporting a variety of sub-analyses, including ones for data races [23], for security (taint) properties, and for other specialized properties. These sub-analyzes are implemented as instances of a framework Infer.AI for building *compositional abstract interpreters*, all of which support the continuous reasoning model.

B. Sapienz: Dynamic Analysis at Scale

Sapienz is a multi-objective automated test case design system, that seeks to maximize fault revelation while minimizing the debug effort to fix. The current version aims to reduce the debug effort by seeking to simultaneously minimise sequence length while maximising coverage, to ensure actionability of fault-revealing test sequences.

Sapienz is based on Search Based Software Testing (SBST) [68], [70], [96], but it augments SBST with systematic testing and is also designed to support crowd based testing [90] extensions through its use of motif genes, which can subsequently draw on patterns of behaviour harvested from user journeys through a system under test [92].

Sapienz was developed as a research prototype (and made publicly available as such[3]). It was subsequently briefly offered as part of the tools and services of the Android testing start-up Majicke, before being acquired by Facebook[4] in February 2017. Since March 2017, the Sapienz technology has been developed and deployed to test Facebook's Android app[5], while work has already begun to extend Sapienz to iOS.

Sapienz is currently deployed to run continuously, testing the most recent internal builds of the Facebook apps, using FBLearner (Facebook's Machine Learning infrastructure [72]) and Facebook's OneWorld platform[6] to support scalable deployment on an arbitrary number of emulators. Currently, it is deployed on (of the order of) 1,000 emulators, at any given time, to test the Android app alone.

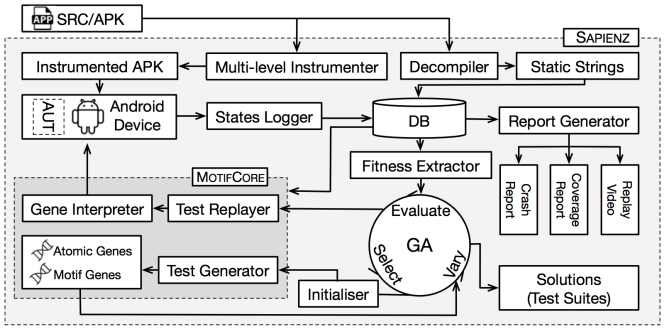

Fig. 2. Sapienz workflow (Taken from the ISSTA 2016 paper [91])

.

There is a (high quality) video presentation[7] from FaceTAV 2017 by Ke Mao, outlining the initial Sapienz deployment at Facebook[8]. The SSBSE 2018 keynote paper [45] describes the current deployment of Sapienz in more detail, while the ISSTA paper [91] contains more details of the underlying technology and scientific evaluation of the research prototype.

FBLearner is a machine learning platform designed to support machine learning at a global scale, through which most of the ML training at Facebook runs [72]. Sapienz is merely one of hundreds of services deployed on this framework.

Using the FBLearner infrastructure, Facebook applies machine learning to a wide range of problems. These include the determination of which ads to display, and the performance of distinct searches along verticals in response to search queries, that specialize for different forms of content, such as videos, photos, people and events. The machine learning infrastructure is also used to support anomaly detection, image understanding, language translation, and speech and face recognition [72].

All of these demanding ML tasks need to be performed at a scale that supports, for example, ML inference phase executions run into the tens of trillions per day [72]. This scalability allows Facebook to globally deploy the benefits of machine learning in real time, supporting translations between more than 45 languages (2000 translation language pairs), which serve approximately 4.5 billion translated post impressions per day. This language translation service alone effects hundreds of millions of people, who see translated posts in their news feeds, thereby lowering linguistic barriers to communication.

Sapienz currently uses the FBLearner Flow component 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 Sapienz automated test design work flow is depicted in Figure 2. The tool 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 technique that has

[2]https://www.youtube.com/watch?v=xc72SYVU2QY

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

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

[5]https://arstechnica.com/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

[6]https://code.facebook.com/posts/1708075792818517/managing-resources-for-large-scale-testing/

[7]https://facetavlondon2017.splashthat.com/

[8]Sapienz presentation starts at 46.45 in this video: https://www.facebook.com/andre.steed.1/videos/160774057852147/

been found to improve the performance of other search based software testing techniques [4], [49].

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. The crashes found are reported to a relevant engineer through Facebook's Phabricator Continuous Integration system[9], using smart fault localisation, developed in-house to triage the crash to a specific line of code.

The 'debug payload' delivered to the engineer includes a stack trace, various reporting and cross-correlation information, crash-witness video (which can be walked through under developer control and correlated to Android activities covered) all of which combine to ensure a high fix rate, a lower-bound of which[10], at the time of writing, stands at 75%. Also, at the time of writing (March 2018), work is underway to extend Sapienz to iOS and other apps in the Facebook family of apps.

Sapienz uses a Search Based Software Engineering (SBSE) approach to optimise for three objectives: code coverage, sequence length and the number of crashes found. Work is also underway to hybridise Sapienz with different search strategies and other techniques to elevate coverage and fault revelation.

We also augmented the production version of Sapienz with an 'Automated Scientific Experimentation (ASE)' FBlearner workflow. The ASE workflow automatically runs different candidate techniques on reference (benchmark) versions of the Android app, collecting standard inferential statistical results, widely recommended for use in SBSE [8], [71] and presents these and graphical representations such as box plots.

The ASE workflow allows the Sapienz team to quickly spin up experiments with new approaches, variations and parameter choices, and to collect scientific evidence for their performance, relative to the current productionised choices. This allows us to fulfill the mantra of 'move fast', and 'fail fast'; we continually experiment with new modes of deployment. The ASE workflow can also support researchers to work alongside the team's engineers to quickly experiment with different techniques and approaches to search. More details on the ASE workflow can be found in the SSBSE 2018 keynote paper about Sapienz deployment [45].

When applied to the top 1,000 Google Play apps (for 30 minutes each), in an evaluation conducted in 2016 using the research prototype version [91], Sapienz found 558 unique, previously unknown, crashes on these 1000 apps.

III. Making bug reports more actionable; moving beyond the ROFL myth

There is an implicit assumption, the ROFL (Report Only Failure List) assumption which assumes that all that a testing technology need do is to report a list of failures to the engineer, in order for these to be fixed. Many very valuable scientific contributions complete their evaluation with claims constructed in terms of the number of faults found by the technique.

The implicit mode of deployment for such studies is thus a list of failures reported by the technique; the list of test inputs for which the execution clearly fails. Such failures are deemed to 'fail' either with reference to an implicit oracle [14], by deviating from an agreed specification, or by distinguishing the behaviour of a correct and a known-faulty version of the program.

The ROFL assumption is also made by static analysis researchers (ourselves included) when they assume that they merely need to report a list of Faults (ROFL = Report Only Fault List) with a low false positive rate. Thus ROFL applies to both testing and verification equally: assuming that all the user requires is a list of faults or failures is one sure way to skip over all the interesting intellectual and scientific challenges posed by deployment and will, thereby, also likely limit the ultimate impact of the research.

Developers are, in practice, not short of lists of fault and failure reports from which they might choose to act. Many systems, such as those typically deployed through app stores, for example [94], have ample mechanisms for users to report bugs, and most practicing developers are typically overwhelmed by such lists; their most pressing problem is not simply finding more bugs; their problem is more nuanced. They need to find bugs with:

1) **Relevance**: the developer to whom the bug report is sent is one of the set of suitable people to fix the bug;
2) **Context**: the bug can be understood efficiently;
3) **Timeliness**: the information arrives in time to allow an effective bug fix.
4) **Debug payload**: the information provided by the tool makes the fix process efficient;

These four properties would likely remain relevant, even were the human to be replaced by an automated repair tool such as, for example, GenProg [85]. Therefore, whether or not it is human, machine or hybrid that attempts the fix, we believe that research work needs to move beyond the ROFL assumption, to produce, not only lists of faults and/or failures, but to also focus on the actionability of the bug reports.

Relevance: Actionability is not merely a property of a particular bug report, but it is also a property of the context in which it is delivered to the developer.

A first problem is *relevance*. When a (static or dynamic) program analysis issue is found, a question that arises is

"To whom should this bug report be directed?"

It is not only a question of responsibility, but also one of finding someone who knows the code related to the issue sufficiently well to act upon it; the person best placed to judge a fix.

At Facebook, when an analysis is running post-land (see again Figure 1), a number of heuristics are used to find the most appropriate developer to whom a bug report should be sent. One heuristic performs a binary search of diffs to find the one that 'caused' the issue.

[9]http://phabricator.org

[10]Precisely determining a guaranteed 'fix' is, itself, a significant challenge as we discuss in Section VIII. This is why we prefer to give a (conservative) lower bound on fix rates, thereby avoiding over-claiming.

However, such problems are fundamentally heuristic and there is no perfect answer. For instance, the developer who 'caused' the problem might have left the team, or even left the company. Developing effective recommender systems for bug reports remains an open problem, and is increasingly important in open source and crowd-sourced software development scenarios [93].

Context: Naturally, developers tend to find bug reports more actionable when they provide sufficient context to allow them to quickly determine causes and remedial actions.

However, even a perfect bug-assignment mechanism that achieves full relevance, and with suitable context (were it ever to exist), would run into the *human cost of context switching*. If a developer is working on one problem, and then they are confronted with a report from a program analyzer on a separate problem, then they must swap out the mental context of the first problem and swap in the second. It is well-known from psychological studies that such context switching can be an expensive operation, not only for software engineers, but for their users too [76].

Timeliness: Running an analysis at diff submit time rather than post-land time elegantly, although only partially, resolves some of the problems of context and relevance: If an analysis participates as a bot in code review, then the developer's mental state is already 'swapped in' (helping alleviate context swapping), because the developer is already expecting to be discussing the code with the human reviewers. Furthermore, if the issue concerns a change in a newly submitted diff, then there is an increased chance that it is relevant to the developer. In this way timeliness can go some of the way to addressing context and relevance issues.

However, relevance is not fully solved by diff submit time analysis alone. A developer might be refactoring code, and the analysis tool flags a number of pre-existing issues (and the fact that they are pre-existing is fooled by the refactoring). These issues are not relevant to the purpose of the diff.

A warning signal might involve a trace that starts from code in the diff, for example, in product code, but uncovers a problem in distant framework code. The product developer might be ill-advised to go into the framework code to fix the issue (and in some organisations may be prevented from doing so by code access privileges). In general, the relevance problem is very important to the effectiveness of a program analyzer, and is deserving of research attention.

Even with these caveats, the benefits of diff submit time deployment were highlighted by our experience with Infer: The first deployment we considered was a post-land, batch-mode, ROFL deployment of the analysis, together with a manual (rather than automated bisect-based) bug assignment. The fix rate for this deployment mode (the rate at which developers chose to resolve the issues) was close to 0%.

However, when we switched to a diff submit time deployment, the fix rate rocketed to over 70%. We learned from bitter experience, that an identical technical analysis, deployed in two different modes that differed only in terms of the point within the development lifecycle at which they reported, could have fundamentally different response rates from developers.

It therefore seems critical that researchers should attempt to consider, report on and evaluate the specific modes of deployment through which they envisage their research techniques might best be deployed; those that would find most traction with developers. Failure to do so might invalidate the scientific findings of the study, so it poses a powerful threat to validity that needs to be considered in any empirical analysis.

Failure to consider a wide range of deployment modes, and report on and evaluate the best-suited to the research technique introduced is also important to ensure that good ideas are not overlooked. Otherwise promising research techniques might be abandoned by researchers or rejected by referees, simply because they fail to suit the implicitly-supposed default deployment mode.

For improved timeliness, we have found it best to deploy as many analyses as possible at diff submit time rather than post land. Not only is the signal more likely to be relevant and timely, but fixing bugs early is well-known to be (dramatically) less costly than fixing them later on [15].

Sapienz was initially deployed post-land, yet achieved fix rates approaching 75%. It was initially deployed post-land, simply for efficiency and scalability reasons (it can test multiple diffs in a single Android APK file). However, more recently, in March 2018, we deployed a lightweight version of Sapienz at diff commit time.

Deploying Sapienz at diff commit time further elevated the fix rate. This highlights a difference between static and dynamic analyses: While static analysis reports likely faults, dynamic analysis reports likely failures. Failures are, perhaps, inherently more compelling for engineers, since they find it hard to ignore the evidence that there is a problem that needs addressing.

We combine both the Sapienz and Infer tools in our deployment at Facebook. For example, when Infer comments on a line of code that may lead to an Null Pointer Exception (NPE) and Sapienz also discovers a crash that it traces back to the same NPE at the same line of code, Sapienz files a task for the engineer to check. This combined deployment mode has a (close to) 100% fix rate at the time of writing, indicating the potential for high-impact combinations of static and dynamic analysis.

Our observations about diff submit time versus post land time, and the ROFL assumption are not unique to Facebook. Indeed we are replicating and corroborating the experiences reported by other practitioners at other companies. For example, related observations have been made at Microsoft, with Prefix (ROFL) and Prefast (similar to our diff submit time observations) [83], by Coverity [35], and by Google [101], [117].

Debug payload: Finding a failing execution is a strong signal to a developer of the need to fix the bug that causes it. As we noted, finding a true positive failure in a timely fashion has a good probability to occasion a fix. However, timely true positive failure reporting, although a valuable pre-requisite, is often insufficient on its own to ensure that a bug fix takes place. We believe that insufficient research attention is paid to the 'debug payload'; the information supplied with a failing

execution that tends to help an engineer fix the bug(s) that cause an observed failure.

Despite its practical importance, the problem of debugging support has been overlooked, and attempts to launch scientific events focusing on problems associated with debugging assistance have had an unfortunate tendency to wane, through lack of a large and strong coherent community of researchers. One example of an excellent scientific event, which has now sadly demised is the AADEBUG workshop series on automated algorithmic debugging (1993-2005).

Notwithstanding admirable efforts by highly dedicated individuals to build a 'debugging scientific community' there remains, at the time of writing, no regular international scientific event focusing on debugging support for developers. Partly as a result of this lack of an event around which a community can coalesce, technical support for debugging activities clearly lags behind other advances in program development environments, as it has continued to do for some time [59].

We hope that Automated Program Repair [84], Genetic Improvement [112], recent approaches to code synthesis and other related automated code improvement techniques that have recently witnessed an upsurge in scientific activity, may alleviate some of the pressure on human debugging effort. Nevertheless, it is undoubtedly the case that one of the surest routes to real world impact for program analysis research lies in the area of debugging assistance.

IV. Compositional testing and verification

The concept of compositionality comes from language semantics, and is often associated with Frege: a semantics is compositional if the meaning of a composite term is defined in terms of the meanings of its parts. Likewise, a program analysis is compositional if the analysis result of a composite program is computed from the results of its parts [32].

In the program analysis context we are discussing here, we are concerned with *automatic* compositionality, where the analysis algorithm decomposes the problem into parts and recomposes them as needed. This contrasts to *manual* compositionality where the human specifies the interfaces of procedures or modules (thus enabling local analysis).

Compositionality pre-supposes that the 'result of the part' makes sense without having all of the surrounding context. This means that a compositional analysis does not need the entire program: it is in a sense the opposite of traditional 'whole program analysis' [10].

Compositionality also supports diff submit time analysis. For example, through compositionality Infer is able to achieve a 'begin anywhere' formulation of its analysis; in principle, the analysis may start at an arbitrary line of code. This opens the way to analyzing program components or libraries before they are run, and that sort of use case was, in fact, a key part of the application of Infer to multi-threading in Facebook's Android Newsfeed [23].

While the basic principles of compositional static analysis are well known [40], most research in the area focusses on whole programs. Further development of automatic compositional analysis is a key research direction. An important part of this involves *effective signal*. It is not always obvious when or where errors should be reported in a compositional analysis (it is easier to imagine for whole program). Fundamental work is needed here.

Note that the word 'compositional' is used in some works on testing (e.g. [52]), in a way that is somewhat different from Frege's original sense (which presupposes getting a testing result without having the complete program); rather, the idea of procedure summary from static program analysis is used in a testing scenario to help scale the analysis by avoiding recomputing information. This might be more directly related to what we term an 'incremental' approach (see below), where the subsequent release is considered to be a composition of the previous release and a change. Perhaps methods that mix static and dynamic analysis would address the compositional testing problem [26]

Mocking of as-yet unimplemented procedures is one practical example where testing does already allow for compositionality; it allows for testing to be performed when the whole program is not known, in such a way that testing results remain valid for any instantiation of the mocked out procedure. This is a form of what we referred to above as *manual* compositionality. Fully *automatic* compositional testing would, for example, allow us to decompose system tests to tackle the false positive problem for unit tests [55], using a fragment of a system, and a corresponding fragment of a system test as a unit test in the knowledge that the unit test is realistic and cannot yield a false positive. It would therefore be interesting and exciting to see more research on composition testing and verification.

A. Research Questions that Address Compositional Testing and Verification

CVF **Compositional Verification Formulation**: How do we best formulate a compositional version of verification approaches, such as CEGAR, interpolation, numerical abstract domains, so that we can 'begin-anywhere' in the code with the stataic analysis and demonstrate effective signal, for example, in terms of fix rate.

CTF **Compositional Testing Formulation**: How do we best formulate testing so that we can start system level (end to end) test case design, from any point in the program under test and in an arbitrarily determined state.

V. Incremental Testing and Verification

Compositionality naturally gives rise to incremental analysis, where changing a part of a system does not necessitate re-analyzing the whole system. Nevertheless, compositionality is not necessary for incrementality: it is possible in principle to do a whole program analysis, to store summaries of the analysis results for program components, and to only re-analyze changed parts, re-using the stored summaries. This sort of incremental analysis does not presuppose that a part can be analyzed without having the whole (which is the hallmark of compositionality).

A key technique in inter-procedural static analysis has been to use procedure summaries to avoid re-computing information. If an abstract state at a call site matches one already seen,

the summary can be used without re-analyzing the procedure. This is the basis of the fundamental RHS algorithm [115], and the idea is used in many static analyzers. However, summary-based interprocedural analysis is used mostly to attack the scaling problem for a program analysis, to make it work for large codebases. Comparatively less work has targeted incremental analysis of code changes. We acknowledge that valuable work has been done in this direction (e.g., [41], [119]), but much more is needed.

The direction of incremental dynamic analysis is even less developed, and yet could be even more impactful. Potentially, a host of currently-expensive dynamic analyses could be moved from post-land to diff time, where the reports are more easily actionable.

The work on SMART (Scalable DART, [6], [52]) works by importing the concept of procedure summary from static analysis into a dynamic analysis context. As far as we are aware, procedure summaries have been used in SMART to address the scalability challenge, not to move an analysis from a post-land to a diff-time, incremental deployment. The general idea to use symbolic procedure summaries in testing may extend to the problem of making automatic incremental testing.

Most current research on automated software testing considers the system under test to be a monolithic piece of code. Although there may be different levels of testing, such as unit level through to system level, the piece of code tested is typically treated as an atomic monolith. A more realistic scenario, better-suited to a world of low TERF (Test-Execution-to-Release-Frequency) ratios, takes account of the development history, thereby supporting incremental testing to collaborate with incremental development and continuous deployment. Sadly there is very little work on incremental automated testing in the research literature.

By 'incremental software testing' we mean automated test case design that constructs new test cases, that is aware of (and exploits) the history of changes to the system and previous tests, thereby increasing the efficiency of automated test case design. Incremental testing should have the property that the time to execute a system-level test is proportional to the execution time for the changed code alone (rather than the execution time of the whole system into which the diff is deployed); system testing execution benefits for unit test costs. Similarly, the time to automatically design the test case should be proportional to the size of the diff to be tested, rather than the code base into which it will be deployed.

While there is work on selective regression testing [54], [136], seeding of test cases [4], [9], [49] and test case regeneration and augmentation [118], [138], we have found little work on automated test case design that exploits knowledge of previous tests [33], the mapping between these tests and changes to the system, and their outcomes, in order to improve the efficiency of ongoing testing in a continuous integration environment.

Such research might, for example, exploit test case caching, and reuse partial test sequences and the resulting system states in subsequent compositions of previous test fragments.

A. Research Questions that Address Incremental Testing and Verification

We outline a few possible research questions that are naturally suggested by continuous integration and deployment:

- IVF **Incremental Verification Formulation**: How do we best formulate an incremental version of verification approaches that demonstrate efficiency on diffs proportional to the size of the diff rather than the size of the code base into which the diff is deployed?
- ITF **Incremental Testing Formulation**: How do we best formulate testing so that test design and execution time are proportional to the size of a diff (and/or diff execution time) rather than to the size (respectively execution time) of the code into which the diff is deployed?

IVF would allow, for example CEGAR [37], interpolation [95], and numerical abstract domains [77] to scale to millions of lines of code. We believe progress in this direction would be very exciting because this 'stretch goal' is far beyond the current state of the art. ITF could be even more impactful, as it could conceivably transform computationally expensive end-to-end testing techniques to where they could be run at diff time.

See [107] for further discussions on compositional and incremental testing and verification.

VI. Survive and thrive, even when we Assume Tests Are Flaky (ATAF)

An important difference between verification and testing derives from the way in which execution has increasingly become nondeterministic. Whereas programming languages have included new semantic features that have posed novel challenges for verification research and practice, the dramatic increase in the stochastic behaviour of most deployed systems, over a similar period, has posed challenges for software testing; tests are 'flaky' [50], [87], [100]. In this section we explore this test-specific challenge, which can be summarised with the aphorism: Assume all Tests Are Flaky (ATAF).

Flakiness challenges implicit assumptions at the very heart of much of the research on software testing. We might like to think of a test T for a program p as a boolean:

$$T_p : Bool$$

This is, essentially, the simplest form of test Oracle [14]; one in which there is a simple boolean outcome that determines whether the test has passed or failed. The nomenclature of most research literature on software testing is imbued with this notion of tests as deterministically and reliably either 'passing' or 'failing', with an implicit 'Law of the excluded middle', that allows for no other possible outcome from the application of a test input to a system under test.

However, this attractively deterministic world is highly unrealistic, and by relying on the assumption that few tests are flaky, we miss many research opportunities, and important avenues for greater research impact.

The reality of testing, in most continuous integration environments, is that it is safer to *Assume Tests Are Flaky (ATAF)*. That is, all tests that can fail, will also sometimes pass,

in apparently identical test scenarios. A flaky test, T, when applied to a program p may thereby yield different outcomes on different occasions, such that it is better to think of the test, not is having a boolean outcome, but as having a probabilistic outcome:

$$T_p : [0..1]$$

At first sight, this may seem like a problem of context: surely if we had sufficient context on the execution environment of the system under test, we might be able to reduce or remove sources of non-determinism in software testing? Unfortunately, the search for such a wider context, even if it were theoretically possible, is certainly practically impossible; there are simply too many external components, services and resources upon which the execution of the system under test depends, and which cannot be controlled by the tester.

Rather than resisting any move from the comforting world of $\{0, 1\}$ to the non-deterministic world of $[0..1]$, we believe that the software testing research community should embrace flakiness; start from the assumption that we 'Assume Tests Are Flaky' (ATAF): how does an 'ATAFistic world' change the research perspective and what new opportunities emerge for novel research problems? Here are some examples of how the ATAFistic world changes testing theory and practice:

Regression testing: In regression test optimisation [44], [137] it is typical to assume that we should optimize according to objectives such as execution time, coverage, and resource consumption [60]. However, in the ATAFistic world, we have an additional objective to consider: test case prioritization might usefully favour more deterministic tests over those that are less deterministic (more flaky), for instance, so that more certainty can be achieved sooner in the test process. Such an ATAFistic regression testing model seems well suited to testing models underpinned by information theory [36], [139].

It might prove more subtle than merely choosing to prioritise for early determinism overall. Rather, the tester may have a property of interest for which he or she would like to receive an early signal from testing. In this scenario, the tester would like test case prioritisation to favour early execution of test cases that contribute most to reducing uncertainty *with respect to this property of interest*; surely a paradigm well-suited to solutions grounded in information theory [120] (especially where it is already known to be applicable to testing [7], [36], [134], [139]). We believe that this is a novel research area that, hitherto, remains untackled by the growing research community working on information theory for software engineering.

Mutation testing: In mutation testing [78], fundamental concepts such as the mutation score need to be adapted to cater for an ATAFistic world. This opens up new research possibilities, such as the construction of mutation operators that tend to reduce test flakiness (while accepting that it is impossible to eliminate it).

In general, it seems unrealistic to expect a one-size-fits-all approach to mutation testing to be successful; mutants are concerned with simulating real faults, but different systems and different scenarios and workflows occasion very different kinds of faults. Therefore, it seems natural to expect that mutation testing research will evolve to cater for specific contexts and scenarios, leading to more tailored mutant design [3], [67] and mutant selection policies tailored to the program under test [2].

However, we need to go further than simply tailoring mutants to the program under test, but also to the 'test question' for which we are seeking some 'answer signal'; different test objectives will require different kinds of mutant. Previous research has investigated specific types of mutants for revealing security issues [24], interface interactions (such as feature interactions on integration) [43], and memory faults [132], for example, yet there appears to be no work on tailoring mutants to questions that are unavoidably probabilistic, due to the inherent flakiness of the tests (ATAF). This challenge of tailoring mutants to maximise signal in the presence of flaky tests, therefore, remains open and, we believe, potentially would have significant impact on the deployment of practical mutation testing systems.

Foundations: The theoretical foundations of software testing need to be revisited, and reconstructed for an ATAFistic world. Even fundamental concepts such as coverage, need to be refined, since the program elements that are covered by a given test case may differ on different executions. We noticed this phenomenon when testing a web-based weather-reporting application, for which statement coverage for a given test suite depended on the prevailing weather conditions [4]. Not every aspect of software testing foundations need necessarily change. For example, if a particular branch is non-deterministic, then the execution of the statements it controls will be consequently also non-deterministic, suggesting that there will remain a subsumption relationship between branch coverage and statement coverage, even in an ATAFistic world.

1) Research Questions that Address the ATAF Assumption: The Assume Tests Are Flaky (ATAF) assumption may provide a useful point of departure for an intellectually rich landscape of research possibilities.

Here we sketch a few open research questions that flow from this ATAFistic starting point:

TFA **Test Flakiness Assessment** Can we find quick and effective approximate measures of the flakiness of a test case?

TFP **Test Flakiness Prediction** Can we find predictive models that will allow us to predict the degree of flakiness of a test case?

TFA **Test Flakiness Amelioration** Can we find ways to transform test goals so that they yield stronger signals to developers in the presence of test flakiness?

TFR **Test Flakiness Reduction** Can we find ways to reduce the degree of test flakiness or to transform a flaky signal into a more abstract, interpolated or otherwise transformed version that makes flakiness less of an issue? Perhaps it would be intellectually rewarding to construct an Abstract Interpretation [39] for Testing and Verification, such that the elevated level of abstraction tends to reduce unwanted variability due to flakiness; we can say more definite things about more abstract properties of the System Under Test.

RTT **Reformulate Test Techniques for an ATAFistic world** Can we find new formulations of, for example, Regression Testing, Mutation Testing, Search Based Testing, Dynamic Adaptive Testing, Model Based Testing, etc, that place at their heart, the assumption that All Tests Are Flaky (ATAF) to some degree, and which cater for this a natural way? This research agenda is in stark contrast to merely seeking, generic or test-technique-specific, approaches to reduce flakiness in order that existing formulations become applicable, once again.

A. Prior work on the Flakiness Problem

There has been previous work on the flakiness problem; understanding what makes test cases flaky and how to minimise the pernicious effects of flakiness on testing methods that, implicitly or explicitly, assume tests to be deterministic. We briefly review this literature here, explaining why we believe it is necessary, yet not sufficient, because we need to move beyond the control of flakiness to acceptance and even optimisation *for* flakiness: assume all tests are flaky.

Luo et al. [87] categorise causes of flakiness in 201 commits made in 52 Open Source projects, for which they are able to categorise the cause in 161 cases. The must common of which are

1) Asynchronous wait (74/161; 45%), in which a test makes an async call and does not fully await the response;
2) Concurrency (32/161; 20%), in which undesirable thread interaction (other than async wait) occurs, such as race conditions, deadlocks and atomicity violation;
3) Test order Dependency (19/161; 12%), in which one test depends on the outcome of another and this test order may change dynamically (e.g. shared access), a phenomenon also studied by Zhang et al. [141] as one cause of flakiness.

These results were partially replicated in the more recent study by Palomba and Zaidman [111], who introduced the concept of refactoring of test smells to reduce test flakiness. Like Luo et al, Palomba and Zaidman also report that asynchronous wait is responsible for 45% of the flakiness they discovered, and that and currency issues are also prevalent (ranked in 4th place at 17%). However, unlike the previous work of Luo et al, Palomba and Zaidman report that (non-network-related) I/O operations are responsible for a much larger number of flakiness issues (22% vs. only 3% in the earlier study), and also found that network issues are important (10% of cases).

Flakiness is also a phenomenon that may differ at different levels of test abstraction; The Luo et al. study extracted any commits with tell-tale key words (e.g., "flak*"), whereas the Palomba and Zaidman study focuses on JUnit tests. We observe that, while flakiness undoubtedly poses problems at the unit level, it is even *more* challenging at the system level, and particularly so for automatically generated tests. In these situations, it is common to experience async wait flakiness (and harder to control for it when the test is auto generated).

Gao et al. [50] consider the related problem of understanding the impact of flakiness at different layers (user interaction layer, behavioural layer, and code layer), concluding that testers should do their best to control flakiness at each level.

All of this research has been concerned with understanding and/or removing sources of flakiness to attempt to *control* the flakiness problem, seeking to migrate testing from the uncomfortable new world of non-determinism to the more familiar (and comfortable) world in which a test either passes or fails, reliably and repeatably.

We believe that where it is possible to reduce or control flakiness this is clearly desirable, but we also would like to stress that hoping we may ever return to world of deterministic testing is quixotic in the extreme. The software industry badly needs automated system level testing, for efficiency and effectiveness of continuous integration and deployment. However, in order to have all the benefits that accrue form automated system test design, we believe it better to assume all test are flaky. Even though some will not be flaky, making this assumption in research work will prioritize the central challenge posed by flakiness.

We believe there will be dramatic real world impact if the community would undertake more research on the problem of, not only coping with flakiness, but perhaps even constructing test automation approaches that actually benefit from it. If we can reformulate testing problems such that flaky tests are merely a 'fact of life', then we may be able to, for example, smooth fitness landscapes, better adopt probabilistic testing approaches, and maybe also finally place testing properly within the framework of information theory, such that test signals can truly be thought of in an information theoretic sense.

VII. Better understanding and reduction of costs

Mobile software release cycles have become considerably shorter (compared to previously widely-used software deployment models), while continuous integration and deployment has simultaneously become the norm rather than the exception, accelerated by web-based and app store-based software deployment paradigms [94]. In more traditional software development environments, such as so-called shrink-wrapped software deployment, or Code Of The Shelf (COTS) [128], the release cycle was markedly slower. Faster release cycles mean that we now have to distinguish between two very different aspects of test efficiency:

1) **Human Test Efficiency**: human design of test cases has traditionally been (and remains) a significant bottleneck that inhibits more effective and efficient software testing.
2) **Computational Test Efficiency**: the machine time required to (possibly design and) execute test cases and to deliver a signal back to developers.

Fortunately, due to advances in automated test case design [5], [27], [68], the slow, error prone and tedious task of human test case design is gradually, finally, receding to be replaced by ever-more automated test environments. A remaining challenge for further automation and replacement of tedious human effort in test case design centres on the Oracle problem [14]. The oracle problem is more resistant to automation, since the determination of correct operating behaviour may sometimes inherently involve judgment concerning the connection

between requirements and system under test; an aspect of the development process most closely associated with human domain expertise.

In more traditional modes of deployment, in which an application was released (perhaps at most) several times per year, we have been fortunate enough to enjoy an extremely low ratio of the time to execute tests, relative to the time to release a freshly built version of the system. This Test-Execution-to-Release-Frequency (TERF) Ratio has traditionally been sufficiently low that computational test efficiency has been regarded as unimportant. However, in emerging Internet and mobile deployment scenarios, this assumption no longer holds, especially as advances in automated testing reduce problems associated with lack of human efficiency, but transfer this cost to the machine.

For instance, Google recently reported [101] that of 5.5 million test cases, only 63,000 typically failed (flakily or otherwise). Clearly, if we can be almost sure that a test will pass without needing to execute it, then it may become computationally efficient to avoid executing it. In this way, we need research that moves beyond producing test suites, to research that prioritises test suites; already a research topic of growing importance [137].

In a continuous development environment in a large scale tech sector organisation, tens to hundreds of changes may be submitted to the continuous integration system every minute, while build times for the entire 'release candidate' version of a large system might run to tens of minutes, even with large scalable infrastructure. Even in smaller (or growing) tech sector organizations, the TERF (Test-Execution-to-Release-Frequency) ratio is typically orders of magnitude higher than for previously prevalent deployment models, for which many widely-used test techniques were primarily developed. A dramatically higher TERF ratio, raises profound questions for automated testing research.

Many of our observations apply equally well to testing and verification, and by extension to static and dynamic analyses. However, focusing attention on deployability does bring some of the differences between verification and testing into sharp relief. In order to verify system correctness the verification algorithm must complete, whereas testing, being essentially a counterexample-driven approach, is fundamentally an anytime algorithm; it can be terminated at any time, yielding the current set of counterexamples discovered. Therefore, from its inception, one of the primary focuses of research on verification has been the computational efficiency of the underlying verification algorithms, and also of the practical efficiency of their implementations.

Although both verification and testing started as a human-centric activity, the verification community quickly moved away from the idea of human mathematicians performing proofs [127], to consider automated verification systems, and from there, immediately encountered the efficiency challenge: a challenge that has remained central to research in the area ever since.

By contrast, testing activity was already prevalent in industry before it became a topic of research interest. Research initially focussed on moving testing activity away from the tedious error-prone human-centred process to an automated process. As the community more successfully tackles this test automation problem, it needs to engage with the computational efficiency challenges that have already been considered in the verification community.

The testing community has made great strides in lifting the test burden from human shoulders to place it more squarely on machines [27], [68]. This automation has dramatically scaled up test effectiveness and efficiency over the painful past of human-designed and executed tests (notwithstanding recent developments in crowd-sourced testing [90]). Perhaps, in the testing research community, we have tended to implicitly assume that computational cost is thus, like early inaccurate aspirations for electric power from nuclear fission, simply 'too cheap to meter' [124].

VIII. The fix detection challenge

Testing and verification techniques have challenges in determining when a fix has occurred, but the details of the challenge are different for testing and verification. For static analysis, the challenge derives from the way in which the code base is continually changing, so traditional instrumentation and markers that identify the fault location may change. This situation can be addressed with a source code 'bug hash', but such a hash is unlikely to be perfect in the presence of a high level of code change.

Testing techniques also need to track the location of faults, where they have been able to triage a failing test to a particular fault, but they also need to track whether failure (the symptom of the fault) disappears and this requires a form of 'crash hash'; a unique identifier for a failure.

The complex nature of the mapping between faults and failures, coupled with the changing code base, make it a challenge to define suitable bug and crash hash techniques. We seek techniques that maximise the chance of detecting a fix, without introducing false positive fix identifications. Our deployment of both Infer and Sapienz at Facebook uses a conservative lower bound on fix detection. The problem of detecting fixes conservatively, yet with minimal false positives remains an interesting challenge that we consider in this section.

It is customary in software testing research to assume that the problem of detecting when a bug is fixed is relatively trivial. Indeed, many studies of software testing techniques, (including those by one of the authors), start from the apparently-reasonable assumption that we reside in a world in which it is easy to distinguish between faulty and fixed versions of the software under test. However, while the observation of a field failure remains relatively uncontroversial, detecting when the underlying cause has been addressed is far from straightforward.

This is partly a ramification of the ATAF (Assume Tests Are Flaky) observation; the field failure may temporarily appear to have been fixed, simply because the available tests are insufficiently deterministic to reliably reveal it. However, this aspect of the ATAF problem can be ameliorated by repeated test execution, and therefore simply degenerates to a problem

of test efficiency. There are other, more subtle reasons why detecting the presence of a fix is non-trivial.

A fault may lead to multiple different failures. Testing activity typically reveals the failure, not the fault and it may only reveal a proper subset of the failures caused by a given fault. Multiple faults may contribute to the same failure observation. An attempted fix may remove some of the failure manifestations of a given fault, but not all.

It has been known for some time [15], [103] that it is important to distinguish between faults and the failures they cause, and that there may exist such subtle relationships that map faults to failures. However, research also needs to take account of how this relationship is affected by a continuous deployment environment with high TERF ratio.

In such a scenario, partial coincidental correctness and failed error propagation [7], [123], [129] become paramount concerns, because testing only reveals a subset of failures due to a fault, and ongoing changes may cause further failures to emerge or disappear, independent of any attempts to fix those that testing has revealed so far.

This independent emergence and disappearance of failure manifestations is also subject to the ATAF problem, and is typically transient, due to the ongoing regular changes to the code base. In such a scenario there is a challenge in detecting when a fix can be said to have occurred.

A. Research Questions on Fix Detection

The need for better fix detection raises scientifically interesting (and industrially high impact) research questions that we would suggest are worthy of further research and study:

PFD **Partial Fix Detection Problem**. In the presence of multiple failures caused by a single fault, how can we best detect partial fixes (that remove some failures, though not necessarily all, without regression)?

NFD **Noisy Fix Detection Problem**. Given the ATAF principle, and continual changes to the code base that, without techniques to ameliorate, may otherwise partially fault mask, or confound failure signals from multiple faults, how can we detect when a fault is fixed?

TFD **Transient Fix Detection Problem**. If a fault, f_1 is fixed by a change c_1 it might mean that the failure $F(f_1)$ associated with fault, f_1, disappears (i.e. $\neg F(f_1)$). However, to be more precise, we should take account of time: it can be f_1 is fixed by a change $C(c_1, t_1)$ for time t_1 to t_2 because $\forall t.t_1 \leq t \leq t_2 \cdot \neq F(f_1, t)$. All fixes are thereby generalised to transient fixes, with the 'traditional' view of a fix as the limit (fixed for an indefinite period). This raises interesting research questions, such as

a) What other code can influence a transient fix (changing its status from fixed to unfixed)? Perhaps dependence analysis [17], [75] and mutation analysis [78] can be helpful here?
b) How can defensive code be added to improve fix resilience (thereby extending the window of transience)?
c) How can we optimise the window of transience, by choosing from available fixes or by warning against code modifications that may break previous fixes?

Note that fix detection is relevant to static as well as dynamic analysis. For example, in the static case, a tool can be fooled into thinking a fix has been achieved if, for example, a refactoring takes place which changes the "bug hash" or other method of identifying a report; an analogue of NFD above.

IX. Testability and Verifiability Transformation

One possibility, known as testability transformation [25], [66], [97], allows us to test, not the ultimate deployment candidate, but some version of it, from which useful test signal can be extracted. The goal of testability transformation is to quickly arrive at a useful version of the system under test from which a better and/or faster signal can be extracted.

The transformation to the system need not necessarily create a version that is functionally equivalent to the original [66]. For instance, expensive set up and tear down phases may be avoidable, while expensive interactions may be unnecessary for testing and thus mocked. However, the rapidly increasing TERF (Test-Execution-to-Release-Frequency) ratio creates other possibilities for new research directions that are currently under-developed, among which we wish to draw attention to the problem of fully incremental testing.

Transformation has been familiar in both testing and verification, where it is common to mock procedures that have yet to be implemented in order to test early, or to model with stubs, the behaviour of inaccessible code, such as operating system routines, in order to verify. For example, when Microsoft's Static Driver Verifier tool is applied to a Windows driver, it replaces calls from the driver to operating system functions, and even some C programming language functions, by model code that does not behave in the same way as the usual execution context of the drivers [11].

Surprisingly, given its prevalence, the provision of formal foundations of testing and verification transforms remains an open challenge [61]. Since such transformations need not necessarily preserve functional equivalence, they differ from more traditional transformation approaches [42], [51], [56] which are meaning preserving (usually by construction) in the traditional sense of functional correctness [73], [102]. It is perhaps ironic that a transformation that is used as part of the process of testing and verification need not, itself, preserve functional correctness in order to best perform this role. However, in order to rely more fully on this role, there remains a pressing need for formal underpinning of the techniques used for testability/ verification transformation [61].

A recent set of open research questions on Testability transformation, encompassing problems including semantic definitions, abstract interpretation, mutation testing, metamorphic testing, and anticipatory testing can be found elsewhere [58].

X. Evaluation Criteria

The evaluation criteria that are typically used in scientific evaluation of testing and verification tend to focus on efficiency and effectiveness, where effectiveness is often characterised by the number of faults detected by the technique under investigation. It is useful to know how many faults are detected

by a technique, particularly when this is part of a controlled experiment with a known pool of faults [34], [47], [104], but there are other important criteria to be taken into account.

The evaluation criteria we list here are neither intended to replace execution criteria, nor are we suggesting that they are all mandatory. Rather than placing an additional burden on the shoulders of scientists, with yet more evaluation obligations, we offer these as *alternative* dimensions for evaluation, along which it is possible to provide compelling scientific evidence that a proposed technique will add value and may be likely deployable in practice. If a technique performs well according to just one of these criteria, then this might suggest that it shows promise, even should it fail on others (or on more traditional efficiency and effectiveness) criteria.

In this section, we set out evaluation criteria, with a particular focus on deployability of testing and verification techniques. We believe that techniques that perform well according to some or all of these criteria will tend to be more deployable than those that do not, irrespective of the number of faults and failures detected by the techniques concerned.

A. Sim-CIDie: Simulating Continuous Integration and Deployment (CID) Environments

Software deployment increasingly uses continuous integration and deployment, yet researchers typically do not have access to such systems. It is not essential to evaluate all research on testing and verification in the context of Continuous Integration and Deployment (CID). Nevertheless, where scientific results can be presented that support claims for CID, this will likely be a strong indicator of deployability and actionability of research findings. How then, are researchers to experiment and report on the results of their techniques in a CID setting, without working in an organisation that practices CID?

Fortunately, researchers can easily *simulate* the effects of continuous integration by taking a series of open source releases of a system, decomposing each release into smaller change components, and simulating the effect of these landing into a repository at different rates and interleavings. Large scale repositories of OSS code changes (together with their reviews) are publicly available to support this [110].

This creates a simple simulation of the typical workflow in a continuous integration environment. Researchers can also simulate the disruptive effects referred to in Section VIII, by constructing different versions of changes to the repository, each of which interfere with one another.

Indeed, the ability to control the degree of interference, the rate at which changes land into the repository, and their size and overlap, could be one of the advantages of proper experimentation; the ability to experiment with controlled parameters in the laboratory setting. It would be very encouraging to see research that tackles these challenges using laboratory–controlled experiments in terms of the parameters that affect the deployment of static and dynamic analyses in a continuous integration environment

In order to evaluate scalability, especially scalability in terms of people, researchers may have to simulate, but these simulations can be based on sensible assumptions and parameters. For example, consider the example of automated generation of test cases, for which the primary driver of scalability is likely to be the execution time (of the system under test). In order to experiment with this and report on this dimension of scalability, researchers can simply insert delays into the example systems under test on which they report.

This simple delay-insertion approach would provide a simulation that would allow researchers to report graphs indicating the impact of increases in execution time on the technique's bugs-per-minute.

Similarly, for investigating practical real-world space and time scalability on realistic examples for a static analysis technique, the driver may be code size. Once again, the researcher can easily find ways to synthetically increase the size of code in a non–trivial way, in order to report on this dimension of scalability.

B. Exploring, defining and measuring signal

Critical to all work on Testing and Verification is the 'signal' that the automated static and dynamic analysis techniques provide, both to other tools and to the software engineers they serve. The research community should surely prioitise the exploration definition and formalisation of measures of signal, since this is such a key output from all static and dynamic analysis. However, 'defining and exploring the signal' remains a surprisingly under-researched topic, given its pivotal role in actionability of research findings and deployability of the techniques about which those findings are reported.

Static and dynamic analyses can produce a variety of different kinds of signal, including (but not limited to) bug lists, proof of bug type's absence, potential fix candidates, warning of potential issues, crash reports with stack traces, test coverage achieved, and localisation information (to help locate bug causes). It is easy to become primarily concerned with the technical challenges of computing this signal in reasonable time and at scale. However, for deployment it is helpful to asses the actionability of a signal, for which we offer candidate definitions below:

Signals are provided as an input to some *client*. The client could be a developer (who potentially benefits from the signal) or a downstream tool or system (that can make use of the signal).

Definition 1 (Signal Client): The *client* for a signal, is an actor (a human, a tool, system or service) that is able to respond to the signal by performing an action.

The concept of a client is a context-sensitive one; the same human (or tool) in different contexts, would be regarded as a different client. For example, a developer receiving a signal, as he or she edits a file to make a change, is a different client to the same developer receiving the signal some time after he or she has landed the change into the repository.

Definition 2 (Actionable Signal): The signal is actionable if there is a particular action that a client can perform to improve the software system as a result of receiving the signal.

We wish to define a metric for signal effectiveness that captures the cost involved in acting on the signal. That is,

the concept of actionability implicitly has an associated cost. Suppose that the client that receives the signal is a developer. If we tell a developer that there is a crash in a system, without further information, then it is possible that they will be able, with great effort, to locate the cause of the crash and resolve it. If we flag a bug to a product developer which is due to a flaw in a library not under their control, it can be more difficult to act than for a bug whose cause is in their own code. The form and quality of error reports also affects actionability, through the cost involved in acting on the signal.

One way to measure this cost, would be through a 'signal effectiveness metric' that formulates cost (and other influencing factors) in terms of the probability of action by the client receiving the signal.

Definition 3 (Effectiveness Metric): An effectiveness metric is any function from signal, σ (and possibly additional optional qualifying parameters, $\bar{x}$ such as client c, platform p, language l, system s etc) to $[0..1]$.

The effectiveness metric, $f(\sigma, \bar{x})$, denotes the likelihood that the client, c, will act on the signal σ. When not parameterized by c, the effectiveness metric refers to an arbitrary client.

This definition is simply a generic (and abstract) definition of effectiveness; it can surely be refined. It could be broken down further, as it is influenced by the cost of actionability, the relevance of reports, timeliness, and other factors. Our goal in introducing this definition is simply to seek to stimulate the research community to refine and improve it, or to replace it, but not to ignore it. We believe that defining appropriate effectiveness metrics remains an interesting open research question for the scientific community.

Many techniques already implicitly involve an effectiveness assessment that could be characterised as a measure of the likelihood that intervention with positive intent would lead to positive outcome. For instance, the assessment of fault localisation [79], [80], [139], [140] traditionally measures the likelihood that the truly faulty statement is accurately elevated (to a top N rank) by the 'suspiciousness' metric used to localise.

Identifying the client to which the signal is sent is important in assessing effectiveness. The same signal directed to different clients can have different effectiveness. For instance, a fault localisation technique will have different effectiveness (as measured by outcome of positive-intent intervention) when sent to a human debugger and an automated repair tool [85].

In both cases the desired outcome is a fix, but the human will require the truly suspicious statement to be relatively high in the ranking, while the repair tool merely requires that the combination of localisation and repair will perform better than repair alone and may, thereby, tolerate a lower ranking of the truly faulty statement.

The Infer experience described in Section III, where diff time deployment led to a 70 percent fix rate, where previous ROFL deployment for the same analysis was near 0 percent, is an extreme example of how directing signal to different clients can have different effectiveness.

It would greatly benefit the evaluation of deployability if actionability and effectiveness metrics were to be defined and applied to report on these two important properties of the signals that emerge from a tool or technique. Even when effectiveness cannot be measured directly (e.g., if obtaining a user population is not practical) then factors that influence effectiveness such as timeliness and relevance could still be measured. All told, measures such as these would complement the more traditional, and still valuable, metrics such as false negative and positive rates.

1) Cost of obtaining signal: Bugs per unit resource: In the evaluation of testing and verification techniques, it is easy to focus exclusively on the benefits offered by the technique, and ignore the overall cost.

Indeed, as academics 'landing' in a major company, we admit that we both underestimated the significance of cost. For example, in 2015 we improved the number of (potential) bugs found by Infer's iOS analysis. However, there was a performance cost: Infer went from taking 3% of the datacenter capacity allocated to iOS CI, to taking over 20%. Infer was therefore throttled back to run less frequently, until we were able to reduce the 'perf' costs for the improved analysis. As a result, we not only started to pay even closer attention to perf, we also began to use bugs-per-minute as a measure to help us decide whether to deploy new candidate analyses.

For adoption and deployment, there is always an implicit cost-benefit analysis, and researchers can therefore improve information relevant to deployability, simply by surfacing the discussion of this trade-off. Consider presenting results that report the behaviour of the proposed approach with respect to a cost-benefit model. Bugs-per-minute and other cost-benefit measures might form part of the comparison of different analyses. As a result, there may be unexpected 'sweet spots' that render a technique useful, where it would otherwise be discounted according to 'default' cost-benefit assumptions.

There is a human cost in the signal-to-noise ratio (as discussed in Section X-B2), but there are also other resource-based costs (such as the execution time) to find that signal. Suppose a proposed new test technique, α finds twice as many (true positive) issues as the state–of–the–art technique ω. It may appear, on the surface, that α outperforms ω. The evaluation of many studies of testing verification are just so-constructed, making plausible scientific claims based on total number of faults detected by different techniques.

However, one really cannot answer the question: 'which is better α or ω?', without taking into account both the *rate* of issues reported (e.g. bugs found per minute), and the mode in which the proposed technique should best be deployed.

It is fairly obvious that the rate is more compelling scientifically (and more important practically) than some absolute amount of issues reported. The absolute amount is always an 'amount within a given budget of time' in any case, and so comparing simply the 'absolute amount of signal' might be unfair; the researcher might be able to choose a time budget for which α outperforms ω and some other time budget for which ω outperforms α. Clearly reporting merely a total number of bugs found, for example, allows the researcher to inadvertently fall into a 'Cherry-picked Time Budget'(CTB) fallacy.

Instead, if we consider the rate at which issues are reported, we also bring into focus the consideration of the proposed deployment mode. To illustrate, consider the three different

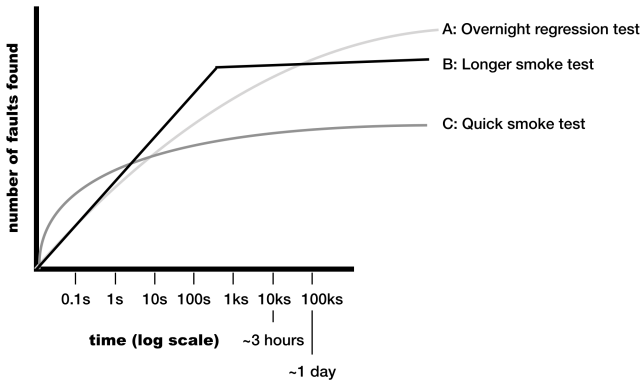

Fig. 3. Question: Which technique is best? Answer: *all* of them (each in different deployment modes)

.

program analysis techniques depicted in Figure 3. Each could be a static, dynamic or hybrid analysis; what we care about is the rate at which they find faults as a cumulative faults–reported growth profile over time.

Which of the three techniques is the top performing? Technique A finds the most faults, but not initially. If we are prepared to wait, then Technique A will yield the most faults found and be declared the 'winner'. The scenarios in which it would make sense to wait this long preclude smoke testing and regression testing of overnight builds. However, for scenarios where there is a major release every few months, for example, Technique A would be the top performer, all else being equal.

On the other hand, suppose we want to deploy the technique to perform a more shallow (but fast) analysis to find crashes quickly for a so-called 'smoke' test. For such a smoke test, perhaps Technique B or Technique C would be preferable. If we wanted a super fast smoke test, perhaps giving the developer almost instantaneous feedback on a build, then Technique C would outperform the other two. On the other hand, if the smoke build process itself takes a few minutes (or longer), then Technique B would be the best-fitted to that deployment scenario.

This discussion reveals how important it is for researchers to articulate the deployment scenario for their approach. Researchers can then demonstrate that their proposed technique is better–fitted to that scenario than a plausible alternative state–of–the–art *for that scenario*. Doing so will increase chances of adoption, and will also provide more precise scientific evidence for the evaluation of the technique's efficacy.

None of these observations is surprising; they denote little more than common sense. What is perhaps more surprising, is that few of them are taken into account in the scientific evaluation of proposed testing and verification techniques (the authors of this paper are at least as guilty of this as the rest of the community).

As a scientific community, we rightly want to accept work that demonstrates evidence for an advance on the state–of–the–art, but too often we can be seduced by simple metrics, such as 'total number of bugs reported' and thereby overlook potentially promising techniques that may have something to offer, even though they fail to outperform the state–of–the–art on, for example, total bugs reported.

By treating testing and verification evaluation as an analysis of cost-benefit, we believe that results are more likely to be useful to other scientists and to potential adopters of the proposed techniques. Therefore, we suggest that researchers should report faults or failures detected per unit of resource, and to report results that show how this measure of value (faults and failures reported) varies as the unit of resource increases.

These observations impact deployability, but also feed back into more scientific reporting considerations: suppose, for instance, we wish to perform a non-parametric inferential statistical test for the effect size of the improvement of a technique over the state–of–the–art. This is a common 'best practice" in much empirical software testing work [8], [71]. Here too, the use-case (intended deployment scenario) should also be considered, for example, to motivate the choice of transformation to be applied to the effect size test performed [105].

The specific units of resource in question will differ from study to study. They could be measured as elapsed time, CPU cycles, memory consumed, or some other domain-specific appropriate determination of the cost of testing or verification.

Rather than inhibiting scientific work by providing yet another burdensome set of evaluation criteria, this cost-benefit approach may free scientists to find new avenues of deployment, by considering interesting sweet spots for their particular technique that maximally optimise the cost-benefit outcome for some chosen deployment mode.

2) Cost of exploiting signal: Signal to Noise Ratio and a generalisation of false positives: Research on software testing and verification often pays particular attention to false positive rates. However, developers may be tolerant of a modest level of false positives, provided that the technique provides overall value. The false positive rate is a specific instance of the more general problem of 'noisy reporting'. We believe that a more nuanced evaluation, in terms of this more general noisy reporting problem, would provide more reliable scientific findings regarding the effectiveness of testing and verification techniques.

For example, consider the situation, illustrated in Figure 4, where Technique A has a false positive rate of 20%, and each false positive has to be investigated by a developer, costing the developer, on average, five minutes of time to establish that 'this is a false positive'. Compare this to Technique B, which has a false positive rate of 40%, all of which need to be investigated, but which require developers to spend only 20 seconds each, on average, to determine that the false positive is, indeed, not worth considering further. Suppose both techniques find approximately the same number of faults with similar severities, and differ only in their false positive rates. Clearly the technique with the *higher* false positive rate (Technique B) will be preferable.

The take home message is that noise to the developer (the overall cost of these false positives) is not merely a one-dimensional false positive count. It is (at least) a two-dimensional measurement, according to the twin costs of number of false positives and time to dismiss them. More generally,

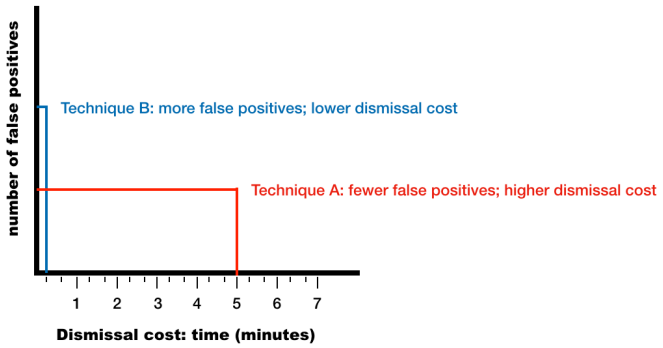

Fig. 4. Noise has *at least* two dimensions of cost: number of false positives and the cost of dismissing them from further consideration. However, most scientific evaluations of static and dynamic analysis techniques consider only a *single* dimension: number of False Positives. This can lead to the abandonment of otherwise promising research ideas and prototypes.

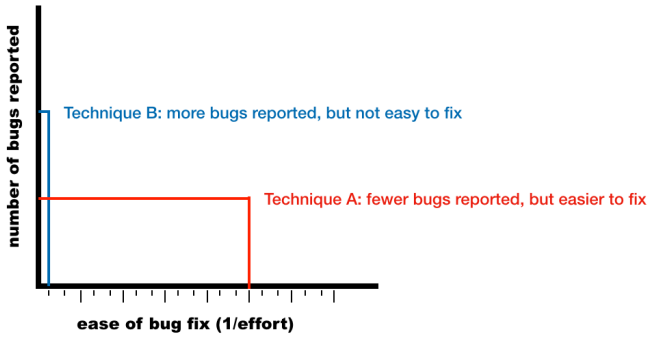

Fig. 5. Signal has *at least* two dimensions of value: number of issues reported (e.g. bugs reported), shown here as the vertical axis and the ease with which they can be fixed, due to the 'debug payload' of the reporting technique (the horizontal axis here). The overall signal is the area under the graph. Typically, scientific evaluations simply report along a single dimension, such as number of bugs found, but a more nuanced evaluation involving other dimensions might also highlight promising scientific advances that might otherwise go unnoticed.

the overall *noise* (to the downstream client application or user) is the property that we should seek to measure, evaluate and reduce in the scientific understanding of static and dynamic analysis. This 'noise' level is likely to be multidimensional, since there are different types of cost along different axes, and each may also have a different weight in its contribution to the overall cost. Taking this into account provides for a far richer space of candidate solutions to analysis problems and, thereby opens the door to many more potentially promising research ideas.

If noise is (at least) two-dimensional, what about signal? This too, we argue, is multidimensional. Consider two techniques with identical false positive rates and dismissal costs, but where there is a difference in the time required to fix the bugs reported as true positives, based on the information provided by each technique. This situation is depicted in Figure 5. Even when a technique reports a true positive, there is still the cognitive load on the developer who has to fix the bug, or, in the case of automated repair, has to check and satisfy him or herself that the patch is correct. This cognitive load is also a cost; the cost of action. The debugging payload provided to the developer may contain irrelevant information, and this has the same detrimental effects as the false positives studied so much more widely by the community.

Overall, therefore, researchers need to provide estimates of the relevance of the information provided in order to fix true positives relative to the total information provided (which we term the 'Relevance Ratio'). Researchers also need to estimate the amount of time required to dismiss false positives, (which we term the 'False-Positive Dismissal Cost').

Techniques with higher relevance and lower false positive dismissal cost will tend to be preferable, even if they have higher rates of false positives and find fewer faults. Although providing these estimates is challenging without involving human subjects, we believe that surfacing these issues in scientific publications will help to ensure that they are discussed and accorded the full scientific attention they merit.

Researchers can and should report on such metrics in their research submissions, articles and grant proposals. We surface them here with the aim of facilitating and motivating the value of this reporting for wider uptake and deployment of research prototypes.

Reporting on the signal–to–noise ratio, rather than focusing solely on false positives will support a more nuanced scientific debate of merits and de-merits of a proposed approach. Taking account of these nuances may help to ensure that promising techniques are not overlooked because of superficially poor performance (in terms, for example, of their total numbers of false positives). We therefore suggest, merely that researchers should report an assessment, even if very approximate, of the Relevance Ratio and the False-Positive Dismissal Cost.

XI. Deployability

Whether or not research is undertaken with ultimate deployment in mind, it may be helpful to understand, and report on some of the factors that affect deployability of research techniques. A lot has been said in the literature on the topic of the scale of industrial software systems and the need for techniques to scale up. This is undoubtedly important and we touch on it briefly later in this section. However, less widely studied and potentially far more important, is the issue of friction.

That is, it may be possible to find useful engineering compromises that trade precision for scale, making scale a relatively surmountable challenge to ultimate deployment. However, a technique that exhibits unnecessary friction is not only less likely to be deployed, but it may be harder to overcome this potent barrier to deployment than it is to scale the apparently unscalable.

Although scale can usually be recast as a trade-off, friction, on the other hand, often arises because of the very design and fundamental static and dynamic analysis characteristics, philosophy or approach adopted. Where such decisions become 'baked in' early in the design of a technique they may doom the research agenda to ultimate failure. It is therefore prudent to consider friction at the outset and throughout the development of a research technique, even if there is no immediate plan or desire for deployment.

A. Friction

In discussing friction, we do not mean to focus on the generic barriers and problems that inhibit the adoption and on-going use of a technique proposed by the research community and considered for adoption in industry. These barriers to adoption have been widely discussed elsewhere [38], [114], [121]. Excessive focus on them can lead to the research community becoming demotivated and disengaged from the challenges of seeking real-world industrial impact.

Rather, we wish to view friction from the specific point of view of the would-be adopters (implementors and users) of a proposed technology; how much does the technique itself require effort from them in order to adopt and continue to use the technique? Thus friction is not a generic barrier that inhibits any and all deployment attempts, but a technique-specific resistance to adoption that can be identified, tackled and reduced to maximise adoption likelihood; minimising friction will maximise deployability.

There are two types of friction that can impede deployment of static and dynamic analysis tools: inertial friction, and ongoing drag.

We define inertial friction for an analysis technique to be the time taken from the decision to adopt to the first developer receiving the first signal from the system.

Definition 4 (Inertial Friction, $\mathcal{IF}$): The inertial friction $\mathcal{IF}(t, c)$ of a tool, t for a client c is the minimum possible effort that c will expend between the decision to adopt the tool and the first signal returned by the tool.

We distinguish $\mathcal{IF}$, the inertial friction, from the 'Drag of On-going use' (or 'on-going drag'), $\mathcal{DO}$, which is any effort required by a client, per signal, in order to get further signals, once the first signal has been obtained from the technology.

Definition 5 (Drag On-going, $\mathcal{DO}$): The Drag (On-going), $\mathcal{DO}(v, c)$ for deployment version v, and a client c, is a measure of the average effort required from the client c, per signal produced by v between v_{t_0} and $v_{t'}$, where v_{t_0} is the time at which the first signal from version v is returned to the client c, and $v_{t'}$ is the time at which version v is decommissioned.

By contrast with inertial friction, on-going drag is defined per version, because one important goal of deployment should be to continually release new versions of the technology, with each successive version reducing the ongoing drag. Many techniques have been adopted, yet subsequently discarded due to ongoing drag; the effort required to continue to receive useful signal from the technology.

While it will undoubtedly be difficult for developers of research prototypes to envisage all the new scenarios that might lead to such ongoing drag, any instances that can be factored out and considered to be germane should naturally be the subject of considerable attention; reducing such inherent on-going drag will be critical to winning champions within an organisation, once the inertial friction has been overcome.

Inertial friction is something that potential adopter developers might assess before adopting a technique, and reducing it is critical for increasing initial deployment likelihood. It is worthwhile considering multiple modes of deployment, in order to reduce inertial friction, even if this means sacrificing some of the signal that can subsequently be obtained.

The on-going drag must also be continually checked (and perhaps reduced) to ensure continual deployment. This gives us the following simple '**if**...**do**' pseudo code for $\mathcal{IF} \ldots \mathcal{DO}$ deployment, the goal is to repeatedly deploy friction reduction such that this pseudo code fails to terminate:

```
begin
if reduce(IF) ≤ sufficiently_low
  then do
       reduce DO
       if DO > tolerable_drag_threshold
         then exit fi
       od
  else no_deployment
fi cease_deployment
```

We parameterise $\mathcal{IF}$ and $\mathcal{DO}$ by the client, c because there are at least two distinct types of client affected by inertial friction when deploying a new tool:

1) the clients that will use the signal from the new tool to improve software products;
2) clients (likely engineers assisted by existing tooling) that will build and maintain the new tool so that it can provide this signal efficiently and effectively.

Here are two examples of $\mathcal{IF}$:

Inertial Friction (for tool users) for Infer Static Analysis: The ROFL-myth example for Infer, mentioned in Section III, in which we initially (and unsuccessfully) expected developers to switch context to solve bug reports in batch mode was one example of high inertial friction for tool users. We attribute our initial failure to successfully deploy entirely to this inertial friction. Once we moved beyond the ROFL myth to diff-time testing, as explained in Section III, we went from zero (fix rate) to hero ($\sim$70% fix rate) pretty fast.

Inertial Friction (for tool users) for Sapienz Dynamic Analysis: In order to automatically design tests that target the revelation of problems we need to know what constitutes a 'problem'. In order to automate test design this means that the test tool needs oracle information [14]. One way of making this available to the tool would be for the developer to add assertions to their code that, *inter alia*, communicate the definition of a good (respectively bad) state to the test tool. Having this information could dramatically improve the performance of the tool, but it introduces friction on the developer who has to define and insert the assertions.

If that friction is necessarily encountered before the tool can be deployed then it would be inertial friction. Of course, it could be that the tool would initially use an implicit oracle [14] (such as 'applications should not crash') and, only some time after deployment, seek additional assertions from authors, thereby deferring the friction from inertial friction to on-going drag.

Of course, the potential adopter community are not the only engineers who might be concerned with the inertial friction of the software testing for verification technology. We have witnessed many otherwise promising research techniques, including those with low inertial friction to potential adopters,

that fail to be adopted in industry due to the inertial friction to the tool developers.

For example, a technique with a highly fragile language-sensitive instrumentation technology that needs to be carefully tuned and adapted before it can be applied, or a technique that requires detailed mathematical models to be constructed for large parts of the system to be verified, would each have high developer inertial fiction.

As a result, neither is likely to find adoption due to inertial friction, not on the engineers who might use the signal from these tools, but due to the friction on those who have to develop technology to deliver that signal. Successful deployment therefore also requires an initial version of the technique for which tool-developer inertial friction is also minimised.

Here we give two concrete examples of inertial friction on the tool developers: one static (drawing on experience with Infer) and one dynamic (drawing on experience with Sapienz):

Inertial Friction (for tool developers) for Infer Static Analysis:

Static analysis tools are often thought of as push-button, but researchers and practitioners in the field know that there can be considerable human startup time. For example, the tool developer might need to design a model for intricate (and changing) programming language details, such as library calls for which source is unavailable. One of the precursors of Infer developed an adaptive program analysis, the aim of which was to make it easier to run the tool and get more meaningful results than previously obtainable. Dino Distefano, one of the founders of Monoidics and an engineer on the Infer team at Facebook, used to talk about minimizing the 'time spent before pressing the button', aiming to shrink it from months to days or hours. He conceived of technical advances related to discovering data structure descriptions and discovering preconditions precisely in order to minimise this time [16], [32]. Measuring and minimising "Push The Button" (PTB) time is critical to determining whether a technique can be successfully deployed.

Inertial Friction (for tool developers) for Sapienz Dynamic Analysis: The design of Sapienz was explicit in its attempt to minimize intertial friction for tool developers, because the original deployment scenario was that it would be run on arbitrary APK files from Android applications, for which there was no source code available. As a start-up, it was essential that the tool would run 'out of the box' on an unseen application and give some signal (find some crashes) with *zero* set up cost. Its ability to do this meant that we assumed absolutely no white box coverage. Doing so traded fitness function guidance for (considerable) reduction in inertial friction and was critical to the early adoption of the technology at Facebook.

The definitions we have introduced in the section are merely suggestions. One important research question for the scientific community is to define suitable metrics that can be used to assess the inertial friction and ongoing drag for testing and verification techniques, as well as other software engineering techniques. Such metrics would be useful in addressing questions of research actionability, by assessing the research-controllable parameters that influence deployability.

B. Serendipitous Deployment Pivots

The first few versions of the proposed research technique to be deployed might benefit from a 'Lean Startup' model [116]. The goal of deployment is to address the practical static or dynamic analysis problem in hand. However, a 'lean startup model' would also aim to gain information, concerning use-cases and, in the case of friction, the ongoing drag that each use case incurs. This information can be used to optimise subsequent versions of the system to reduce the ongoing drag of the most frequent use cases.

It can often happen that the mode of deployment envisaged by the research team is not that found most beneficial to engineers who use the tools. Such unanticipated use-cases need not invalidate the tool; they may well *enhance* its impact. Unexpected avenues of impact can offer new opportunities to exploit the research in ways not originally foreseen by the researchers.

Many researchers have experienced and commented, often anecdotally, or off-the-record, on the unexpected use-case phenomenon. We have witnessed it too and would like to document it here with an example. In 2002, one of the authors and his colleagues deployed a static analysis tool at DaimlerChrysler in Berlin, called Vada; a simple variable dependence analysis tool [63], based on research work on program slicing. The aim of Vada was to assist Search Based Testing at DaimlerChrysler. The use-case initially envisaged was to deploy dependence analysis to reduce search space size [64], [98].

However, the developers also found Vada useful to understand the relationships between input parameters to C functions and their influence on defined variables. The engineers found Vada's analysis to be particularly useful for globals, as a means of program comprehension, which was unplanned and unexpected. This additional, unanticipated use-case occasioned a great deal of subsequent development to augment with may- and must- analysis and techniques to better expose the previously internal dependence analysis to developers.

Before long we were starting to field requests from our users for new analyses to support the unplanned deployment and we started to develop visualisations to further support this use-case. This visualisation effort fed back in to research development, with our subsequent work reporting on the visualisations themselves [18], and the prevalence of dependence clusters we were finding [20].

These dependence clusters were an apparent artifact of a (very simple) visualisation; the Monotonic Slice size Graph (MSG), which simply plots the size (vertical axis) of all slices in increasing order of size (on the horizontal axis). Other researchers subsequently also reported finding widespread prevalence of dependence clusters in other languages such as Java [125] and Cobol [57] and in both open and closed source [1].

More recently we were able to demonstrate the potentially pernicious effects of dependence clusters [135], thereby motivating this initial interest. As a result of chasing an unexpected use-case for a deployed research prototype, we were thus rewarded with a rich seam of novel research questions and intellectually-stimulating scientific investigation.

Ironically, our research team had started to move away from its initial vision of slicing and dependence analysis as a comprehension support [21], [62], because of the large size of static slices [19]. Instead, we had become more focused on dependence analysis as a support for downstream applications [65]. However, through our engagement with industrial deployment of our prototype tooling, we discovered that real-world developers were, indeed, finding slices useful when debugging, *just as Weiser had initially envisaged* and reported in his seminal PhD work from the 1970s that introduced program slicing [130].

C. Scale

Scale means different things to different people. Different techniques will scale (or fail to scale) in different dimensions, but scalability needs to be addressed in order to assess the degree of deployability for a technique.

However, lack of scalability should *not* be a reason to reject a proposed scientific project or research paper; many *apparently* unscalable techniques have subsequently been found to scale, due to subsequent innovations and discoveries. It is important to report on apparently unscalable, yet otherwise promising technologies in the hope that others may subsequently find techniques to tackle their scalability.

Nevertheless, for those researchers who additionally want to demonstrate deployability and thereby to maximise chances of more immediate impact, it will be important to report on scalability and, to identify the relevant dimension and drivers of scalability. For example, one might imagine that the primary driver of computational time scalability for a static analysis technique will be the code size to which the technique is applied. By contrast, for a dynamic analysis, the execution time of the system under test may have more influence on the overall compute-time scalability.

In order to address scale, researchers should identify the primary drivers of scalability and present results that report on the impact of these drivers on the overall speed (see Section X-B1).

In scientific studies of software analyses, 'scale' is typically used to refer to space and time scalability and is, thereby, directly related to well-understood algorithmic space and time complexity analyses. However, there are other dimensions of scalability that may prove equally and sometimes more important for a technique to be deployed.

In industry, the term 'scale' is often used to refer to scaling in terms of people, *as well as* engineering artefacts, such as machines or lines of code. Is it much harder for a technology to be deployed to 1,000 people than to 10? This notion of 'scalability' draws us into the question of the inertial friction and ongoing drag (on tool deployers and their users). These aspects of a proposed software analysis technology can also be addressed, assessed and evaluated and they are, themselves, important when judging deployability, and to which we now turn.

XII. The Fi Fi/Gi Verify Challenge: Combining Testing and verification for Analysis and Manipulation

No keynote paper on a topic as broad as static and dynamic analysis for testing and verification would be complete without a grand challenge. The supply of grand challenges to software engineering and programming languages researchers shows little sign of drying up. Indeed, the supply of such challenges considerably out-paces the deployment of techniques that meet them.

Therefore, we do not propose to introduce, here, yet another completely new challenge. Rather, we would like to develop and better articulate the existing challenge of FiFiVerify (Find, Fix and Verify systems automatically), generalising it to the wider challenge of FiGiVerify: Find, (Genetically) Improve and Verify, automatically.

Fixing bugs is merely one way in which systems can be improved. Increasingly, however, it is becoming realised that functional correctness is merely one form of overall fitness for purpose. We therefore generalise the FiFiVerify vision to cater for all forms of fitness for purpose, not merely the more narrowly construed functional correctness of the software system.

This challenge rests on what we believe to be one of the most exciting possibilities for research on testing and verification: its combination with automated program improvement, through techniques such as Genetic Improvement [69], [82], [112] Code Synthesis [56], [88], Code transplantation [13], [113], [122], [126], Tuning [74], [131] and Automated Repair [12], [85], [106].

The original formulation of the FiFiVerify Challenge considered analysis, to find and fix bugs and verify the fixes; code manipulation was limited to bug–fixing patches. What if we could further manipulate the program code to achieve performance improvements? After all, in many practical deployment scenarios, particularly mobile

performance is the new correctness

We believe that there is an open challenge to find effective and efficient ways to combine testing, *improvement* and verification; testing to find issues (both functionality failure and performance problems), improvement to address the issues, and verification to ensure that the improvements are correct (e.g., side effect free). In this section, we briefly recap the FiFiVerify challenge and generalise it to the FiGiVerify challenge.

We have previously argued for both the importance and near realisability of the FiFiVerify challenge [68]; the community is close to realising the ability to create "tools that will find, fix and verify the systems to which they are applied": to "take a program that may contain bugs (from some identified bug class, such as memory faults) and return an improved program" that "has all bugs for the specified class fixed and is verified as being free from this class of faults" [68].

Sadly this FiFiVerify 'vision' remains just that; a vision, with a set of associated open problems. However, it would still appear that the primary challenge is to find scalable ways in which to compose various existing solutions to test case

design [27], [48], [91] decomposible verification [30], [32], [53], [86] and automated repair [85], [106]. There is no known fundamental (technical or scientific) reason why we cannot find such a composition and thereby achieve the 'FiFiVerify' vision.

A. FiGiVerify: Find, Genetically Improve and Verify

We can generalise FiFiVerify to GI (Genetic Improvement): FiGiVerify would consist of finding performance issues, such as regressions through which a system has become insufficiently performant, and fixing these using genetic improvement. The verification obligation requires a demonstration that there has been no functional regression. It may turn out that this version of the vision is, in part, *easier* to achieve than FiFiVerify, because the verification constraint is simply to demonstrate that the improved version of the system is *at least as correct* as the previous version, rather than requiring a demonstration of the absence of an entire class fault.

There has been recent research on such regression verification problems, in which the verification obligation is, not to prove the system fully correct, but merely to demonstrate the absence of regression with respect to a reference implementation [53]. Such regression verification approaches have also recently found application in automated bug fixing [99], and so we can be optimistic that they may find application, both for FiFiVerify and in FiGiVerify.

It is, indeed, exciting to imagine a combination of techniques that allows us to identify scenarios where performance can be improved, perhaps even opportunistically during development as an additional 'program improvement collaborator'; a FiGiVierfy bot that confers/interacts with human engineers through the continuous integration backbone that runs through the modern code review systems at the heart of most CI deployment.

Such a program improvement collaborator could suggest small modifications to the system that improve performance, according to multiple non-functional properties [69], and to automatically improve these using test-guided genetic improvement, but with the additional certainty that comes from a guarantee of the absence of any functional regressions.

One might imagine that human developers would continue to add new functionality, but without needing to pay too much attention to the daunting task of balancing perhaps five different performance criteria, such as footprint size, bandwidth consumption, battery consumption, execution time, throughput and so on.

Instead of the human engineers concerning themselves with this impossible multi-criteria balancing act, a '21st-century compiler' or IDE, imbued with testing and verification and genetic improvement capabilities would optimise the multiple competing non-functional constraints and properties, potentially producing multiple versions of the system for different use cases, platforms and domains.

This challenge was formulated in 2012 as the GISMOE (Genetic Improvement of Software for Multiple Objectives) challenge [69]. Since 2012, there has been considerable progress, and several breakthroughs, on Genetic Improvement techniques [112]. The GISMOE challenge did not include any notion of verification of the improved code, so FiGiVerify is essentially the meet point of the FiFiVerfiy Challenge and the GISMOE Challenge:

$$\text{FiGiVerify} = \text{GISMOE} + \text{FiFiVerify}$$

XIII. Acknowledgements

Thanks to the Infer and Sapienz teams at Facebook for helping the authors learn about static and dynamic analysis at scale, and to the Developer Infrastructure (DevInfra) organisation and leadership at Facebook for their steadfast support for this work. We are also grateful to Kathy Harman for proof reading an earlier draft of this paper.

References

[1] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, 33^{rd} *International Conference on Software Engineering (ICSE 2011)*, pages 746–755, Waikiki, Honolulu, HI, USA, May 2011. ACM.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. Mutation testing using genetic algorithms: A co-evolution approach. In *Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103*, pages 1338–1349, Seattle, Washington, USA, June 2004. Springer.

[3] M. Allamanis, E. T. Barr, R. Just, and C. A. Sutton. Tailored mutants fit bugs better. *CoRR*, abs/1611.02516, 2016.

[4] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In 26^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3 – 12, Lawrence, Kansas, USA, 6th - 10th November 2011.

[5] S. Anand, A. Bertolino, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, J. Li, P. McMinn, and H. Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.

[6] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 367–381, 2008.

[7] K. Androutsopoulos, D. Clark, H. Dan, M. Harman, and R. Hierons. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In 36^{th} *International Conference on Software Engineering (ICSE 2014)*, pages 573–583, Hyderabad, India, June 2014.

[8] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In 33^{rd} *International Conference on Software Engineering (ICSE'11)*, pages 1–10, New York, NY, USA, 2011. ACM.

[9] A. Arcuri, D. R. White, J. A. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In X. Li, M. Kirley, M. Zhang, D. G. Green, V. Ciesielski, H. A. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K. C. Tan, J. Branke, and Y. Shi, editors, 7^{th} *International Conference on Simulated Evolution and Learning (SEAL 2008)*, volume 5361 of *Lecture Notes in Computer Science*, pages 61–70, Melbourne, Australia, December 2008. Springer.

[10] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Foundations of software engineering (FSE '98)*, pages 46–55, 1998.

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, pages 73–85, 2006.

[12] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In 22^{nd} *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, pages 306–317, Hong Kong, China, November 2014.

[13] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 257–269, 2015.

[14] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[15] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[16] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *CAV: Computer Aided Verification, 19th International Conference*, pages 178–192, 2007.

[17] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In 22^{nd} *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, pages 109–120, Hong Kong, China, November 2014.

[18] D. Binkley and M. Harman. An empirical study of predicate dependence levels and trends. In 25^{th} *IEEE International Conference and Software Engineering (ICSE 2003)*, pages 330–339, Los Alamitos, California, USA, May 2003. IEEE Computer Society Press.

[19] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance*, pages 44–53, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[20] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[21] D. Binkley, M. Harman, L. R. Raszewski, and C. Smith. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension*, pages 161–170, Los Alamitos, California, USA, June 2000. IEEE Computer Society Press.

[22] S. Blackshear and P. O'Hearn. Finding inter-procedural bugs at scale with Infer static analyzer. code.facebook.com blog post, 6 Sept 2017.

[23] S. Blackshear and P. O'Hearn. Open-sourcing RacerD: Fast static race detection at scale. code.facebook.com blog post, 19 Oct 2017.

[24] M. Büchler, J. Oudinet, and A. Pretschner. Security mutants for property-based testing. In M. Gogolla and B. Wolff, editors, *5th International Conference on Tests and Proofs (TAP 2011), Zurich, Switzerland, June 30 - July 1*, volume 6706 of *Lecture Notes in Computer Science*, pages 69–77. Springer, 2011.

[25] C. Cadar. Targeted program transformations for symbolic execution. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 906–909, 2015.

[26] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering (ICSE 2011)*, 2011.

[27] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.

[28] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium*, pages 3–11, 2015.

[29] C. Calcagno, D. Distefano, and P. O'Hearn. Open-sourcing Facebook Infer: Identify bugs before you ship. code.facebook.com blog post, 11 June 2015.

[30] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, 36^{th} *Symposium on Principles of Programming Languages (POPL 2009)*, pages 289–300, Savannah, GA, USA, 2009. ACM.

[31] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011. Preliminary version appeared in POPL'09.

[32] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.

[33] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, pages 55–66, 2014.

[34] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608, 2017.

[35] A. Chou. Static analysis in industry. POPL'14 invited talk. http://popl.mpi-sws.org/2014/andy.pdf.

[36] D. Clark and R. M. Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8–9):335 – 340, 2012.

[37] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[38] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *International Workshop on Program Comprehension (IWPC'03)*, pages 196–206, 2003.

[39] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[40] P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proceedings of SSGRR*, 2001.

[41] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 128–148, 2013.

[42] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[43] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.

[44] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information & Software Technology*, 52(1):14–30, 2010.

[45] D. Erb, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, D. Stewart, and T. Tei. Deploying search based software engineering with Sapienz at Facebook (keynote paper). In 10^{th} *International Symposium on Search Based Software Engineering (SSBSE 2018)*, page 10, Montpellier, France, September 8th-10th 2018. To Appear.

[46] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[47] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems Software*, 38:235–253, 1997.

[48] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 416–419. ACM, September 5th - 9th 2011.

[49] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, 5^{th} *International Conference on Software Testing, Verification and Validation (ICTS 2012)*, pages 121–130, Montreal, QC, Canada, April 2012. IEEE.

[50] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, 37^{th} *International Conference on Software Engineering (ICSE 2015)*, pages 55–65, Florence, Italy, May 16-24 2015. IEEE Computer Society.

[51] S. L. Gerhart. Correctness-preserving program transformations. In *2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '75)*, pages 54–66, 1975.

[52] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56, 2010.

[53] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[54] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In 20^{th} *International Conference on Software Engineering (ICSE '98)*, pages 188–197. IEEE Computer Society Press, Apr. 1998.

[55] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *International Symposium on Software Testing and Analysis (ISSTA 2012)*, pages 67–77, 2012.

[56] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, Aug. 2012.

[57] Á. Hajnal and I. Forgács. A demand-driven approach to slicing legacy COBOL systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011. Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.533.

[58] M. Harman. Sword fight at midnight, a computer game, Sunshine Publications Ltd., 1983.

[59] M. Harman. Sifting through the wreckage. *EXE*, page 5, Mar. 1999. Editorial.

[60] M. Harman. Making the case for MORTO: Multi objective regression test optimization (invited position paper). In 1^{st} *International Workshop on Regression Testing (Regression 2011)*, Berlin, Germany, 2011.

[61] M. Harman. We need a formal semantics for testability transformation. In 16^{th} *International Conference on Software Engineering and Formal Methods (SEFM 2018)*, Toulouse, France, 2018.

[62] M. Harman, S. Danicic, and Y. Sivagurunathan. Program comprehension assisted by slicing and transformation. In M. Munro, editor, 1^{st} *UK workshop on program comprehension*, Durham University, UK, July 1995.

[63] M. Harman, C. Fox, R. M. Hierons, L. Hu, S. Danicic, and J. Wegener. Vada: A transformation-based system for variable dependence analysis. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[64] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *ACM Symposium on the Foundations of Software Engineering (FSE '07)*, pages 155–164, Dubrovnik, Croatia, September 2007. Association for Computer Machinery.

[65] M. Harman, L. Hu, R. M. Hierons, C. Fox, S. Danicic, A. Baresel, H. Sthamer, and J. Wegener. Evolutionary testing supported by slicing and transformation. In *IEEE International Conference on Software Maintenance*, page 285, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[66] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[67] M. Harman, Y. Jia, and W. B. Langdon. A manifesto for higher order mutation testing. In 5^{th} *International Workshop on Mutation Analysis (Mutation 2010)*, Paris, France, April 2010.

[68] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing (keynote paper). In 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, Austria, April 2015.

[69] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs (keynote paper). In 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 1–14, Essen, Germany, September 2012.

[70] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, November 2012.

[71] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In B. Meyer and M. Nordio, editors, *Empirical software engineering and verification: LASER 2009-2010*, pages 1–59. Springer, 2012. LNCS 7007.

[72] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *24th International Symposium on High-Performance Computer Architecture (HPCA 2018), February 24-28, Vienna, Austria*, 2018.

[73] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[74] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 199–212, Mar. 2011.

[75] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[76] S. T. Iqbal and E. Horvitz. Disruption and recovery of computing tasks: field study, analysis, and directions. In M. B. Rosson and D. J. Gilmore, editors, *Human Factors in Computing Systems (CHI 2007)*, pages 677–686, San Jose, California, USA, 2007. ACM.

[77] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 661–667, 2009.

[78] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, September–October 2011.

[79] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *24th International Conference on Software Engineering (ICSE '02)*, pages 467–477, New York, NY, USA, 2002. ACM.

[80] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

[81] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: opportunities, applications, and challenges. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 201–204, 2010.

[82] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEVC)*, 19(1):118–135, Feb 2015.

[83] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. D. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[84] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[85] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[86] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–146, 2012.

[87] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In 22^{nd} *International Symposium on Foundations of Software Engineering (FSE 2014), address =*.

[88] Z. Manna and R. J. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6(2):175–208, 1975.

[89] K. Mao. *Multi-objective Search-based Mobile Testing*. PhD thesis, University College London, Department of Computer Science, CREST centre, 2017.

[90] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017.

[91] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 94–105, 2016.

[92] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 16–26, 2017.

[93] K. Mao, Y. Yang, M. Li, and M. Harman. Pricing crowdsourcing-based software development tasks. In 35^{th} *ACM/IEEE International Conference on Software Engineering (ICSE 2013 — NIER track)*, San Francisco, USA, 2013.

[94] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9), 2017.

[95] K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 123–136, 2006.

[96] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[97] P. McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In F. Rothlauf, editor, *Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 1689–1696, Montreal, Québec, Canada, 2009. ACM.

[98] P. McMinn, M. Harman, Y. Hassoun, K. Lakhotia, and J. Wegener. Input domain reduction through irrelevant variable removal and its

effect on local, global and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering*, 38(2):453 – 477, March&April 2012.

[99] S. Mechtaev, M. D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In 40^{th} *ACM/IEEE International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, May.

[100] A. M. Memon and M. B. Cohen. Automated testing of GUI applications: models, tools, and controlling flakiness. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, 35^{th} *International Conference on Software Engineering (ICSE 2013)*, pages 1479–1480, San Francisco, CA, USA, May 18-26 2013. IEEE Computer Society.

[101] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In 39^{th} *International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242, Buenos Aires, Argentina, May 20-28 2017. IEEE.

[102] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976. Two Volumes.

[103] G. J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.

[104] Nashat Mansour, Rami Bahsoon and G. Baradhi. Empirical comparison of regression test selection algorithms. *Systems and Software*, 57(1):79–90, 2001.

[105] G. Neumann, M. Harman, and S. M. Poulding. Transformed Vargha-Delaney effect size. In 7^{th} *International Symposium on Search-Based Software Engineering (SSBSE)*, pages 318–324, Bergamo, Italy, September 2015.

[106] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: program repair via semantic analysis. In B. H. C. Cheng and K. Pohl, editors, *35th International Conference on Software Engineering (ICSE 2013)*, pages 772–781, San Francisco, USA, May 18-26 2013. IEEE.

[107] P. O'Hearn. Continuous reasoning: Scaling up the impact of formal methods research (invited paper). In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, page 13, Oxford, July 2018.

[108] P. O'Hearn. Separation logic. *Commun. ACM*, 2018. to appear. (will give web link in time).

[109] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, 2001.

[110] M. Paixao, J. Krinke, D. Han, and M. Harman. CROP: Linking code reviews to source code changes. In *International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[111] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flakey tests? In *International conference on software maintenance and evolution (ICSME 2017)*, pages 1–12. IEEE Computer Society, 2017.

[112] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2018. To appear.

[113] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, pages 132–143, Granada, Spain, April 2014.

[114] S. T. Redwine, Jr. and W. E. Riddle. Software technology maturation. In 8^{th} *International Conference on Software Engineering (ICSE '85)*, pages 189–200, London, UK, 28–30 Aug. 1985. IEEE.

[115] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM SIGACT and SIGPLAN, ACM Press, 1995.

[116] E. Ries. *The Lean Startup: How Constant Innovation Creates Radically Successful Businesses Paperback*. Penguin, 2011.

[117] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608, 2015.

[118] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In 23^{rd} *Automated Software Engineering (ASE '08)*, pages 218–227, L'Aquila, Italy, 2008. IEEE.

[119] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 451–471, 2013.

[120] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[121] K. Sherif and A. S. Vinze. Barriers to adoption of software reuse: A qualitative study. *Information & Management*, 41(2):159–175, 2003.

[122] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, pages 43–54, 2015.

[123] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In 34^{th} *International Conference on Software Engineering (ICSE 2012)*, pages 870–880, 2012.

[124] L. L. Strauss. National association of science writers founders' day dinner, 1954.

[125] A. Szegedi, T. Gergely, Á. Beszédes, T. Gyimóthy, and G. Tóth. Verifying the concept of union slices on Java programs. In 11^{th} *European Conference on Software Maintenance and Reengineering (CSMR '07)*, pages 233 – 242, 2007.

[126] J. Temperton. Code 'transplant' could revolutionise programming. *Wired.co.uk*, 30 July 2015. Online.

[127] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

[128] J. M. Voas. COTS software: The economical choice? *IEEE Software*, 15(2):16–19, 1998.

[129] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[130] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[131] F. Wu, M. Harman, Y. Jia, J. Krinke, and W. Weimer. Deep parameter optimisation. In *Genetic and evolutionary computation conference (GECCO 2015)*, pages 1375–1382, Madrid, Spain, July 2015.

[132] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke. Memory mutation testing. *Information & Software Technology*, 81:97–111, 2017.

[133] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *20th CAV*, pages 385–398, 2008.

[134] L. Yang, Z. Dang, T. R. Fischer, M. S. Kim, and L. Tan. Entropy and software systems: towards an information-theoretic foundation of software testing. In *2010 FSE/SDP Workshop on the Future of Software Engineering Research*, pages 427–432, Nov. 2010.

[135] Y. Yang, M. Harman, J. Krinke, S. S. Islam, D. Binkley, Y. Zhou, and B. Xu. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 296–307, 2016.

[136] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 140 – 150, London, United Kingdom, July 2007. Association for Computer Machinery.

[137] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[138] S. Yoo and M. Harman. Test data regeneration: Generating new test data from existing test data. *Journal of Software Testing, Verification and Reliability*, 22(3):171–201, May 2012.

[139] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information theoretic and coverage based approaches. *ACM Transactions on Software Engineering and Methodology*, 22(3 (Article 19)), July 2013.

[140] S. Yoo, X. Xie, F. Kuo, T. Y. Chen, and M. Harman. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 26(1):4:1–4:30, 2017.

[141] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In C. S. Pasareanu and D. Marinov, editors, *International Symposium on Software Testing and Analysis (ISSTA 2014)*, pages 385–396, San Jose, CA, USA, July 21 - 26 2014. ACM.