2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

SapFix: Automated End-to-End Repair at Scale

A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott
Facebook Inc.

***Abstract*—We report our experience with SAPFIX: the first deployment of automated end-to-end fault fixing, from test case design through to deployed repairs in production code[1]. We have used SAPFIX at Facebook to repair 6 production systems, each consisting of tens of millions of lines of code, and which are collectively used by hundreds of millions of people worldwide.**

INTRODUCTION

Automated program repair seeks to find small changes to software systems that patch known bugs [1], [2]. One widely studied approach uses software testing to guide the repair process, as typified by the GenProg approach to search-based program repair [3].

Recently, the automated test case design system, Sapienz [4], has been deployed at scale [5], [6]. The deployment of Sapienz allows us to find hundreds of crashes per month, before they even reach our internal human testers. Our software engineers have found fixes for approximately 75% of Sapienz-reported crashes [6], indicating a high signal-to-noise ratio [5] for Sapienz bug reports. Nevertheless, developers' time and expertise could undoubtedly be better spent on more creative programming tasks if we could automate some or all of the comparatively tedious and time-consuming repair process.

The deployment of Sapienz automated test design means that automated repair can now also take advantage of automated software test design to automatically re-test candidate patches. Therefore, we have started to deploy automated repair, in a tool called SAPFIX, to tackle some of these crashes. SAPFIX automates the entire repair life cycle end-to-end with the help of Sapienz: from designing the test cases that detect the crash, through to fixing and re-testing, the process is fully automated and deployed into Facebook's continuous integration and deployment system.

The Sapienz deployment at Facebook, with which SapFix integrates, tests Facebook's apps using automated search over the space of test input sequences [7]. This paper focuses on the deployment of SapFix, which has been used to suggest fixes for six key Android apps in the Facebook App Family, for which the Sapienz test input generation infrastructure has also been deployed. These are Facebook, Messenger, Instagram, FBLite, Workplace and Workchat. These six Android apps collectively consist of tens of millions of lines of code and are used daily by hundreds of millions of users worldwide to support communication, social media and community building activities.

In order to deploy such a fully automated end-to-end detect-and-fix process we naturally needed to combine a number of different techniques. Nevertheless the SAPFIX core algorithm is a simple one. Specifically, it combines straightforward approaches to mutation testing [8], [9], search-based software testing [6], [10], [11], and fault localisation [12] as well as existing developer-designed test cases. We also needed to deploy many practical engineering techniques and develop new engineering solutions in order to ensure scalability.

SAPFIX combines a mutation-based technique, augmented by patterns inferred from previous human fixes, with a reversion-as-last resort strategy for high-firing crashes (that would otherwise block further testing, if not fixed or removed). This core fixing technology is combined with Sapienz automated test design, Infer's static analysis and the localisation infrastructure built specifically for Sapienz [6]. SAPFIX is deployed on top of the Facebook FBLearner Machine Learning infrastructure [13] into the Phabricator code review system, which supports the interactions with developers.

Because of its focus on deployment in a continuous integration environment, SAPFIX makes deliberate choices to sidestep some of the difficulties pointed out in the existing literature on automated program repair (see Related Work section). Since SAPFIX focuses on null-dereference faults revealed by Sapienz test cases as code is submitted for review it can re-use the Sapienz fault localisation step [6]. The focus on null-dereference errors also means that a limited number of *fix patterns* suffice. Moreover, these particular patterns do not require additional fix ingredients (sometimes known as *donor code*), and can be applied without expensive exploration.

We report our experience, focusing on the techniques required to deploy repair at scale into continuous integration and deployment. We also report on developers' reactions and the socio-technical issues raised by automated program repair. We believe that this experience may inform and guide future research in automated repair.

The SAPFIX project is a small, but nevertheless distinct advance, along the path to the realisation of the FiFiVerify vision [10] of fully automated and verified code improvement. The primary contributions of the present paper, which reports on this deployment of SAPFIX are:

1) The first end-to-end deployment of industrial repair;
2) The first combination of automated repair with static and dynamic analysis for crash identification, localisation and re-testing;
3) Results from repair applied to 6 multi-million line systems;
4) Results and insights from professional developers' feedback on proposed repairs.

[1] The first author, Alexandru Marginean, undertook the primary SAPFIX implementation work. The remaining authors contributed to the design, deployment and development of SAPFIX; remaining author order is alphabetical and not intended to denote any information about the relative contribution.

978-1-7281-1760-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-SEIP.2019.00039

Algorithm 1 *trigger_create_fix*, SAPFIX's trigger fix creation algorithm.

```
Input:   b_rev, the buggy revision
         b_file, the blamed file: the file that contains the crash location
         b_line, the blamed line: the line of the crash
         s_trace, the stack trace of the crash
         mid, the mid of the crash we are trying to fix
         b_author, the blamed author: the author of b_rev
         ?buggy_expressions, buggy expressions that Infer gives us. When we do not
   have Infer data, this argument is null
         high_firing_t, the high firing threshold
Output: P, a list of revisions that fix the crash under SAPFIX's testing
 1: strategy_priority := [template_fix, mutation_fix, diff_revert, partial_diff_revert]
 2: $C_P := \emptyset$                          # $C_P$ is the list of candidate fixes
 3: if is_high_firing(mid, high_firing_t) then
 4:     $C_P$ += diff_revert(b_rev)
 5:     $C_P$ += partial_diff_revert(b_rev, b_file, b_line)
 6: $C_P$ += template_fix(b_rev, b_file, b_line, buggy_expressions)
 7: $C_P$ += mutation_fix(b_rev, b_file, b_line, buggy_expressions, s_trace)
 8: $P := \emptyset$                            # P is the list of patches that fix the bug
 9: for all $p \in C_P$ do
10:     if ¬repro_crash(p, mid) ∧ pass_sapienz(p) ∧ ¬sapienz_repro_mid(p, mid) ∧
        pass_ci_tests(p) then
11:         P += p
12: for all $s \in$ strategy_priority do
13:     $P_s$ := filter(P, strategy($p \in P$) = s)
14:     if $P_s \neq \emptyset$ then
15:         $p_s$ := select_patch($P_s$)
16:         publish_and_notify($p_s$, b_author)
17:         for all $p_u \in P_s \setminus \{p_s\}$ do
18:             publish_and_comment($p_u$, $p_s$)
19:         return $P_s$
20: return null
```

THE SAPFIX SYSTEM

This section describes the SAPFIX system itself, its algorithms for repair and how it combines the components outlined in the previous section.

The SAPFIX *Algorithmic Workflow*

Figure 1 shows the main workflow of SAPFIX. Algorithm 1 is the main algorithm of SAPFIX that drives the automated bug fixing process. Using Phabricator, Facebook's continuous integration system, developers submit changes (called 'Diffs') to be reviewed. Sapienz, Facebook's continuous search based testing system, selects test cases to execute on each Diff submitted for review [6]. When Sapienz triages a crash to a given Diff, SAPFIX executes Algorithm 1. Line 1 in Algorithm 1 establishes the priority of fixes according to the strategy that SAPFIX used to produce them. When multiple fix strategies produce patches that pass all SAPFIX's tests, we select fixes only from the top priority strategy to report to developers. This prioritisation approach avoids polluting developers' review queues.

The *template_fix* and *mutation_fix* strategies (mentioned at Line 1 of Algorithm 1) choose between template and mutation-based fixes, favouring template-based fixes, where all else is equal, but taking account of results from Infer static analysis and also from linter reports on candidate fixes. Template fixes come from another tool, Getafix [14] that generates patches similar to the ones that human developers produced in the past; the details will be described in a subsequent publication. For the purposes of understanding the SAPFIX deployment, it can be assumed that SAPFIX has available to it, a set of template fix patterns harvested from previous successful fixes deployed by developers.

If neither template-based nor mutation-based approach produces a patch that passes all tests, SAPFIX will attempt to revert Diffs that result in high-firing crashes. Lines 3–5 in Algorithm 1 trigger the Diff revert strategies. SAPFIX triggers these strategies only for high-firing crashes that block Sapienz and other testing technologies and therefore need to be deleted from the master build we are testing as soon as possible (even if they would never ultimately leave the master build and make it to production deployment). The revert strategies revert the diffs, which thereby 'deletes the change' (made in the diff). In practice that can mean deletion, addition, or replacement of code in the current version of the system. For instance, if the offending diff added code, then it is deleted whereas, if the diff deleted code, then it is added back. If the Diff added lines of code then reversion is simply an attempt at side-effect free deletion. If the Diff removed lines of code, then reversion would add them back. Either way (and for everything in-between), conceptually speaking, reversion means to 'delete the Diff'.

Between the two available Diff reversion strategies, SAPFIX prefers (full) *diff_revert*, because *partial_diff_revert* is expected to have a higher probability of knock-on adverse effects due to dependencies between the changes in the Diff that introduced the crash. However, (full) *diff_revert* might fail because of merge conflicts with the master revision (new Diffs land every few seconds, while fix reporting can take up to 90 minutes (see Figure 3). In those cases we use *partial_diff_revert*. The changes that *partial_diff_revert* produces are smaller and thus less prone to merge conflicts.

The recognition of a crash and the distinction between different crashes requires a 'crash hash'; a function that groups together different crashes according to their likely cause. This is a non-trivial problem in its own right. Facebook uses a crash hash called a 'mid', the technical details of which are described elsewhere [6]. For this discussion, the important characteristic of a 'mid' is that it is an approximate (but reasonably accurate) way of identifying unique crashes. It can be thought of, loosely speaking, as a crash id. Improving the accuracy of such crash hashes remains an interesting and important challenge for future research [6].

Of course, we favour a fix rather than to simply attempt to delete the offending code, but when the other fix strategies fail to fix a high firing crash, SAPFIX suggests a Diff revert fix. *is_high_firing*(*mid*, *threshold*) identifies high firing crashes: it returns true if the crash with id *mid* fires more than *threshold* times, and false otherwise. Finding a way to delete the right code without affecting other subsequent Diffs is also a non-trivial problem in a large scale and rapidly changing code base, where new Diffs land every few seconds. This problem may also benefit from further attention from the research community.

Lines 6–7 in Algorithm 1 trigger the template and mutation fix strategies. Lines 9–11 look at the candidate fix patches in C_P for the ones that indeed fixed the crash, without introducing new bugs.

To identify whether a patch fixes a crash, SAPFIX uses *repro_crash* and *sapienz_repro_mid*. *repro_crash*(*rev*, *mid*) tries to reproduce *mid* in the revision *rev* using Sapienz's

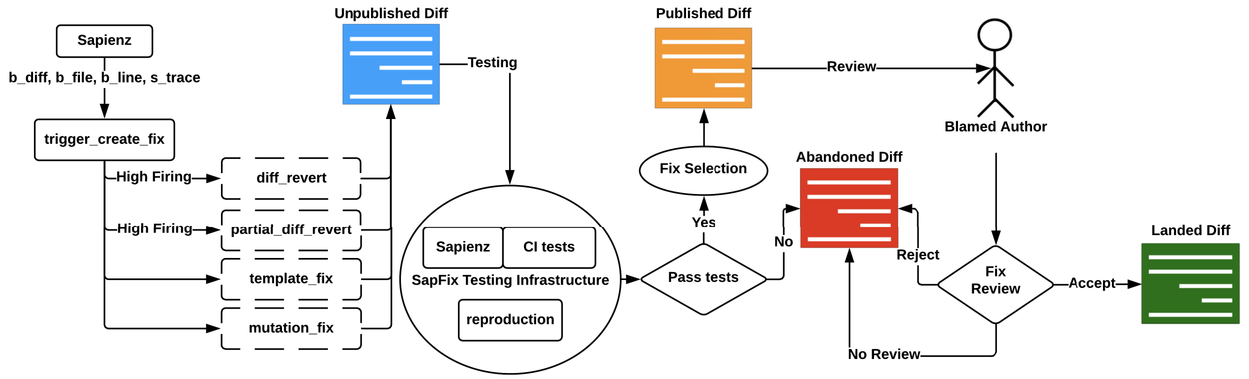

Fig. 1. SAPFIX workflow. When Sapienz triages a crash to SAPFIX, *trigger_create_fix* triggers the four fix strategies that create fix candidate (unpublished) *diffs*. Next, SAPFIX tests the candidate fixes. The fix selection stage uses heuristics to select one diff out of the ones that pass all the tests and further publishes it to notify the most relevant software engineer and add him or her as reviewer If the reviewer accepts the fix, SAPFIX lands it into the production workflow of the Phabricator Continuous Integration system. SAPFIX abandons the fix candidate if the developer rejects it or he or she fails to review it within 7 days.

reproduction workflow. We cannot always assume that we have available tests that reproduce a given crash, due to the well-known problem of test flakiness [5], [15]. Therefore, SAPFIX also uses $sapienz\_repro\_mid(rev, mid)$ to inspect the results of regular Sapienz runs over *rev* to see if any of them found *mid*. Infer is also re-executed (automatically) on the patches it has detected as a sanity check that static analysis also no longer identifies the issue that SAPFIX seeks to fix.

To identify whether a patch might also introduce new crashes or other issues, SAPFIX runs Sapienz multiple times over the candidate fix in $pass\_sapienz(rev)$. Finally, $pass\_ci\_tests(rev)$ inspects the results of (previously existing) unit, integration, and end-to-end tests in the Facebook continuous integration and deployment infrastructure. If all these tests pass, SAPFIX considers the patch to be a successful candidate to report to engineers and adds it to P, at Line 11 in Algorithm 1.

Lines 12–19 in Algorithm 1 publish the successful candidate patches. SAPFIX selects one of the published candidates and requests a reviewer for this candidate through the Phabricator code review system. The reviewer is chosen to be the software engineer who submitted the Diff that SAPFIX attempted to fix. This is the engineer who most likely has the technical context to evaluate the patch.

Other relevant engineers are also subscribed to each Diff published by SAPFIX to oversee the review process, according to heuristics implemented, as standard for all Diffs, in the Facebook code review process. Furthermore, some developers specifically ask to be subscribed to (some or all) fixes, by opting in with a so-called ‘butterfly’ subscription rule. As a result, all Diffs proposed by SAPFIX are guaranteed to have at least one (suitably qualified) human reviewer, but may have many more, through these other routes to Diff subscription.

The function $strategy(p)$ returns the strategy that SAPFIX used to produce p. Line 13 selects, in P_S, all the successful patches that the top priority strategy produced. Next, at Line 15, $select\_patch(P_s)$ selects in p_s the top priority patch from P_s. Currently, $select\_patch(P_s)$ uses the following heuristics: select the fix that Sapienz executed the most often; select the fix for which Sapienz executed the buggy statement the most often; select the smallest fix. On Line 16, SAPFIX publishes p_s and notifies the developer by calling $publish\_and\_notify(p, b\_author)$.

Algorithm 2 *mutation_fix*, SAPFIX mutation fix algorithm.

```
Input:   b_rev, the buggy revision that Sapienz blamed
         b_file, the blamed file: the file that contains the crash location
         b_line, the blamed line: the line of the crash
         s_trace the stack trace of the crash
         ?buggy_expressions candidate buggy expressions that Infer gives us. When
    we do not have Infer data, this argument is null.
Output: P, the list of bug fixing revisions
 1: $crash\_category := extract\_crash\_category(\mathrm{s\_trace})$
 2: if $crash\_category \neq$ "NPE" then
 3:     return $\emptyset$
 4: if $buggy\_expressions \neq \emptyset$ then
 5:     return $create\_rev(add\_null\_check(\mathrm{b\_file, b\_line, buggy\_expressions}))$
 6: else
 7:     $C_{buggy} := top\_of\_s\_trace(\mathrm{s\_trace, b\_file, b\_line})$ ?
        $extract\_dereferences(\mathrm{b\_file, b\_line}) : extract\_args(\mathrm{b\_file, b\_line})$
 8:     $P := \emptyset$
 9:     for all $c \in C_{buggy}$ do
10:         $P$ += $create\_rev(add\_null\_check(\mathrm{b\_file, b\_line, c}))$
11:     return $P$
```

Finally, on Lines 17–18, Algorithm 1 publishes the rest of the patches from P_s and comments with a preview of them on p_s. $publish\_and\_comment(p, p_s)$ publishes the candidate fix p and adds an inline preview of p on the selected candidate fix p_s.

Algorithm 2 is the SAPFIX mutation-based fixing algorithm. Algorithm 2 currently only supports fixing Null Pointer Exception (NPE) crashes. We are currently in the process of extending the mutation strategies to cater for other crash categories, but we have already witnessed considerable success with NPE-specific patching alone, which is encouraging. On Line 1, Algorithm 2 calls *extract_crash_category* to identify the category of crash. If the category is not “NPE”, Algorithm 2 returns the empty set. To extract the crash category, *extract_crash_category* looks at the short message on the stack trace.

Lines 4–5 in Algorithm 2 check whether a more precise cause of the NPE is known: *i.e.* which expressions in the buggy statement caused the NPE by taking the value null. This information can be obtained from Infer, in cases where both Infer and Sapienz find the same NPE. When that happens SAPFIX creates a single patch that guards *b_line* with null checks for the buggy expression. On Line 5, Algorithm 2 creates a revision for this patch. The method *add_null_check* uses eclipse JDT to parse the AST of the buggy file and to add the null check before the buggy statement.

Lines 7–11 in Algorithm 2 handle the case when we do not know which expressions are buggy. In this case, SAPFIX identifies all the expressions in the buggy statement that can potentially cause an NPE. For each such expression, Algorithm 2 produces a candidate patch. SAPFIX tries each of two simple mutations, which either return null or protect potentially null-valued expressions with a null check. The surrounding Facebook testing infrastructure will subsequently tend to reject those patches that do not actually fix the bug (such as those patches that inadvertently attempt to 'fix' the wrong expression). Therefore, failure to fully localise the buggy expression tends to affect efficiency but not effectiveness.

Infer helps to localise the likely NPE-raising expression, but dynamic analysis can also help here, where Infer signal is unavailable. Specifically, we analyse the position of the blamed line of code in the stack trace (*b_line*) at Line 7 to obtain C_{buggy}, the set of candidate buggy expressions. If *b_line* is at the top of the stack trace then *top_of_s_trace*(*s_trace*, *b_file*, *b_line*) returns true. In this case, SAPFIX need only attempt to fix expressions that are de-referenced, because the program execution does not continue after *b_line*. *extract_dereferences* extracts only the expression de-referenced at *b_line*.

If *b_line* is not at the top of the stack trace, it means that one of the arguments of a function called at *b_line* is presumed to have caused the NPE (further up in the stack trace). In this case SAPFIX need only attempt to fix the arguments of functions in *b_line*. *extract_args* extracts the function arguments at *b_line*.

Finally, Lines 9–10 in Algorithm 2 produce a patch, for each candidate buggy expression in C_{buggy}. The patch guards *b_line* with null check for the candidate buggy expression.

Sapienz: SAPFIX uses Sapienz to identify candidate crashes that require fixing and to (partially) check that fixes pass the original failing test(s), as well as generating new tests and a partial approach to detecting some categories of knock-on issues that the candidate might introduce. Sapienz uses multi-objective Search Based Software Engineering (SBSE) [16] to automatically design system level test cases for mobile apps [4], for which it finds 100s of crashes per month, approximately 75% of which are fixed by developers [6]. Like SAPFIX, Sapienz comments to developers in Phabricator, the backbone Facebook's Continuous Integration system[2], which is used for code review, handling more than 100,000 Diffs per week at Facebook [5].

Both Sapienz and SAPFIX are deployed on top of FBLearner, Facebook's Machine Learning (ML) platform through which most of Facebook's ML work is conducted [13]. There is not space here to fully explain FBLearner, but the infrastructure itself is covered in more detail elsewhere [13] and the use of FBLearner as a substrate on which to deploy search based testing is described in detail in the SSBSE 2018 keynote paper [6].

Infer: SAPFIX uses Infer to assist with localisation and static analysis of fixes proposed. Infer is deployed on the majority of Facebook code and based on Separation Logic and bi-abduction [17], [18], scaled to tens of millions of lines of code, thereby allowing Infer to find thousands of bugs per year [19]. Infer is also available as open source [20] and has been used elsewhere, including AWS, Mozilla, Spotify. Like Sapienz, Infer is deployed directly into Facebook's internal continuous integration system, where the two tools collaborate to highlight to developers those bugs on which they agree [6]. At the time of writing such bugs have a 98% fix rate, largely we believe because developers have a localisation of both the likely root causing fault (from Infer) and a consequent failure (from Sapienz). Nevertheless, even for such highly 'human fixable' bugs, engineering effort and skill could be better spent on other more creative and less tedious engineering activities, thereby motivating our interest in automated fault fixing through techniques like SAPFIX.

RESULTS

Table I presents the results of applying SAPFIX over a period of three months to tackle NPEs detected by Sapienz as they were submitted for code review. Each row denotes a crash tackled by SAPFIX. Naturally, we periodically update the pool of template fixes (something that occurred once during the first three months of deployment, on the 9th of August 2018 as shown in Table I). In total, to tackle the 57 crashes reported to SAPFIX, 165 patches were constructed, of which roughly half were constructed using templates and half using mutation-based repair. Of these 165 patches, 131 correctly built and passed all tests and were thus fix candidates. Of these 131 candidates, 55 were reported to developers, covering 55 of the 57 crashes tackled by SAPFIX.

Figure 2 reports how many times the template-based and mutation-based strategies produced at least one candidate fix for each of the 57 different crashes from Table I. Although SAPFIX favours templates overall, it triggers both strategies to be able to re-test all patches produced in parallel. Triggering both the fix strategies also allows us to evaluate these two strategies in isolation. Our results suggest that having both in the pipeline leads to better overall success: In 55 of the 57 fix attempts, either the mutation-based fix strategy *or* the templates produced at least one fix candidate. Only in two cases did *both* strategies fail (one failed to build and one failed re-testing). In 13 cases, both the fix strategies produced at least one fix candidate. In isolation, the mutation-based fix strategy produced at least one fix candidate for 40 cases, while the templates did so in 28 cases. In 27 cases the mutation-based fix strategy alone was able to produce a fix candidate, while in 15 cases the templates alone produced a fix candidate.

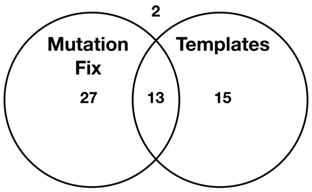

Fig. 2. The number of times mutation-based fix strategy and templates produced at least one patch to pass all tests.

Initial reactions were strongly positive: On seeing the very first SAPFIX-proposed patch, the developer reviewing the patch

[2] http://phabricator.org

commented: '*Definitely felt like a living in the future moment when it sent me the diff to review. Super cool!*'. As can be seen from Table I, about half the fixes proposed by SAPFIX were deemed, by developers, to correctly fix the failure. Of those deemed correct, about half were landed 'as is', and half were modified. Of those that were modified, about half were edited by the developers, while half were simply reviewed by the developer only after they had already fixed the bug themselves. Of the (approximately) half of all proposed fixes that were *not* ultimately landed into the code base, about half were deemed incorrect by developers (would have side effects or failed to tackle the true causes). For the remaining half that were not landed, the proposed fix was simply abandoned (after 7 days with no response from the developer).

During its first three months of deployment, SAPFIX attempted to revert 18 Diffs (14 fully and 4 partially), where these Diffs contained high firing crashes, that could not be fixed by the templates or mutation-based fixing approaches. These Diff revert recommendations were all declined by developers (and not included in Table I); it seems developers are (perhaps understandably) unwilling to simply revert their hard work.

Can SAPFIX *fix pre-existing crashes?*

The standard deployment mode, for which SAPFIX was designed (and is currently deployed), attempts to fix newly arising failures (crashes) as they are submitted in Diffs and detected as buggy by Sapienz. For this use-case, the developer has recent relevant context on the changes relating to the fix. Such relevancy has proved pivotal to the successful deployment (and human fix rates) for both Infer and Sapienz, as explained elsewhere [5]. Nevertheless, as a stretch goal we also experimented with targeting SAPFIX at pre-existing crashes that had reached production partly because Sapienz had failed to detect them (we are still working on the development of Sapienz [6], but no testing technology can be expected to stop *every* failure).

For pre-existing crashes, the developer reviewing the fixes proposed by SAPFIX has less context on the code and fix proposed. We split the results into two broad categories: long-standing (more than 3 months, the width of the Sapienz triage window [6]) and recent (first seen in the last 3 months, so potentially triagable by Sapienz, but missed by it). These long-standing crashes are also those for which the developer would be likely to have the *least* context, so it would be informative to see how many were landed by developers.

In both cases (recent and long-standing) we cannot use precise localisation, since we do not have available Sapienz triage data. SAPFIX's *template_fix* and *mutation_fix* strategies rely on a blamed line to produce candidate fixes. However, for the pre-existing crashes that we target here, we do have access to multiple stack traces. Therefore, in this mode of deployment SAPFIX identifies the longest common path across 200 sampled stack traces, starting from the top of the stacks. The bottom-most line of this common path that is inside our codebase (not library or framework code) becomes the blamed line. Since this blamed line does not correspond to an identified Diff, SAPFIX instead uses a standard default approach, used across the company, to identify the developer to whom we should report the issue detected.

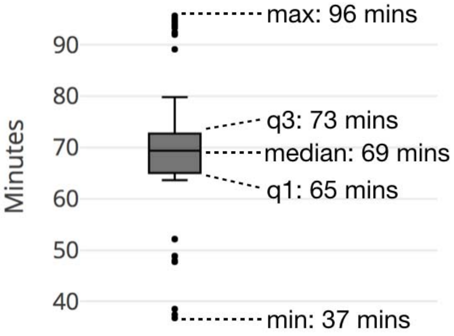

Fig. 3. SAPFIX runtime: the time that SapFix requires to publish a fix.

Table II presents the results for these fix candidates. These results are for a three day deployment window only. After three days, we switched off this experiment to avoid unnecessarily spamming our developers with multiple fix candidates from this comparatively untried-and-tested mode of deployment. In total, over the three days, SAPFIX constructed 946 candidates, of which it reported 195 to developers; 117 for recent crashes and 78 for long-standing crashes.

Had we left the experiment running longer, the proportion of landed fixes could only have increased, so we were encouraged that approximately 15% were landed within this three day period (which included a weekend; typically a quiet period for developer activity at Facebook [21]). Also, interestingly, we observed that the proportion of fixes landed was not notably different between recent crashes and longer-standing crashes. This observation gave us hope that we may ultimately be able to deploy this technology to track down and fix longer-standing crashes that developers tend to find harder to fix.

Timing issues

Figure 3 presents a time-to-fix box plot (from when SAPFIX is first notified of a need to fix, to the publication of a proposed fix to a developer). The median time from fault detection to fix publication to a developer is approximately one hour. More specifically, as shown in Figure 3, the median is 69 minutes, with a relatively tight inter-quartile range (65-73 minutes) and a worst case approximately 1.5 hours, and the fastest fix being reported to the developer 37 minutes after the crash was first detected.

As shown in Figure 3, the overall range of observed values is wide (37..96 minutes). This is because the timing figures are not only influenced by the computational complexity of fixing but also by the variations in workloads on the continuous integration and deployment system. Since SAPFIX is deployed in a highly parallel, asynchronous environment, the time from detection to publication can be influenced more by the demand on the system and the availability of computing resources than by the fix problem's inherent computational cost.

TABLE I

Null Pointer Exception SapFix:“#P” is the total number of patches; “#M” is the number of mutation fix patches; “#G” is the number of template patches; “#Pass Tests” is the number of patches that passed all our tests; “#¬ Build” is the number of patches that failed to build; “#¬ Sap” is the number of patches that failed Sapienz testing; “#¬ Fix” is the number of patches that failed to fix the crash; “#¬ Pr.” is the number of patches that were not published because the rule that produced them, was subsumed by one with a higher priority; “SapFix Land” specifies whether SapFix landed the patch into production; “Developers’ Feedback” reports developers’ insights. The table’s time format is **MM.DD/HH:MM**. Note that on the $9^{th.}$ August 2018 we updated our templates with better versions that covered more types of NPE fixes. We report the summaries of our results for both time periods and overall as well.

			Fix Strategy		#Pass Tests		Failed Patches			Dev Claims Correct			Dev says:		
	App	#P	#M	#G	#M	#G	#¬ Build	#¬ Sap	#¬ Fix	SapFix Land	Correct But Late	Correct Dev Land	Wrong Fix	Unkn.	Developers’ Feedback
06.26/09:34	Facebook	3	0	3	0	3	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
07.06/09:29	Facebook	6	0	6	0	6	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
07.10/02:30	Facebook	4	0	4	0	4	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
07.11/06:04	Facebook	2	2	0	2	0	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
07.15/11:55	Facebook	2	2	0	2	0	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
07.16/03:42	Facebook	4	4	0	4	0	0	0	0	Yes	No	No	No	No	Macro: “superlike”
07.16/09:27	Instagram	1	1	0	1	0	0	0	0	No	No	No	Yes	No	Not sure about the side effects of the fix.
07.18/12:06	Facebook	4	0	4	0	4	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
07.20/02:39	Facebook	3	2	1	2	1	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
07.20/03:01	Facebook	1	1	0	1	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
07.20/03:33	Instagram	2	2	0	2	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
07.20/05:41	Instagram	1	1	0	1	0	0	0	0	No	Yes	No	No	No	Correct fix: “Oh this is cool! I didn’t notice this diff until now. I have addressed the issue in this diff Dxxxxxxx but I wish I’ve seen this earlier. :-)”
07.20/05:50	Facebook	2	2	0	1	0	1	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
07.31/04:21	Facebook	7	0	7	0	7	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
08.02/01:49	Facebook	1	1	0	1	0	0	0	0	No	No	Yes	No	No	Correct fix, landed by the developer.
08.03/01:32	Facebook	1	1	0	1	0	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
08.03/01:56	Instagram	3	3	0	3	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
08.08/00:28	Facebook	2	2	0	2	0	0	0	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
Mutation Fix Overall		24	-	-	23	-	1	0	0	4	5	1	2	1	**Fix attempts with at least 1 passing test patch: 13/18**
Templates Overall		25	-	-	-	25	0	0	0	1	0	0	2	2	**Fix attempts with at least 1 passing test patch: 6/18**
Total NPE Fixes: 18		**49**	**24**	**25**	**48**		**1**	**0**	**0**	**5**	**5**	**1**	**4**	**3**	
On 08.09.2018 we updated our templates.															
08.20/07:46	Facebook	4	0	4	0	0	4	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
08.20/07:47	Facebook	3	2	1	1	0	2	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
08.20/07:47	Facebook	2	1	1	1	1	0	0	0	No	No	Yes	No	No	Correct fix, landed by developer
08.20/08:56	Facebook	1	0	1	0	1	0	0	0	No	No	No	Yes	No	“The fix might mask a race condition”
08.20/08:56	Facebook	1	0	1	0	1	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
08.21/02:46	Facebook	4	2	2	2	1	1	0	0	No	No	No	Yes	No	Fixing the crash, but not a reasonable fix.
08.22/02:51	Facebook	2	2	0	2	0	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
08.22/02:52	Facebook	2	0	2	0	2	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.05/09:04	Messenger	1	0	1	0	1	0	0	0	No	No	No	Yes	No	fix at the wrong line (this was a bug in sapfix :( )
09.05/09:04	Messenger	1	0	1	0	1	0	0	0	No	No	No	Yes	No	Wrongly triaged.
09.05/09:15	Messenger	2	2	0	2	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.07/09:12	Facebook	4	2	2	2	2	0	0	0	No	No	No	Yes	No	Wrong fix: null guard for an expression that cannot be null.
09.07/09:12	Instagram	1	0	1	0	1	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.07/09:12	Messenger	2	1	1	1	0	1	0	0	No	No	Yes	No	No	Correct fix, landed by developer.
09.07/09:13	Messenger	3	2	1	1	1	0	0	1	No	No	No	No	Yes	Not reviewed in 7 days.
09.07/10:20	Instagram	1	1	0	1	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.08/09:29	Facebook	9	0	9	0	6	3	0	0	Yes	No	No	No	No	Macro: image with killing bugs
09.08/09:29	Messenger	1	0	1	0	0	1	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.08/09:29	WorkPlace	2	1	1	1	1	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.08/10:07	Facebook	1	1	0	1	0	0	0	0	No	No	Yes	No	No	Correct fix, landed by developer.
09.08/10:17	Facebook	1	1	0	1	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.08/10:21	Messenger	2	0	2	0	2	0	0	0	No	No	No	No	Yes	well this is cool (abandoned b/c not reviewed in time)
09.09/09:12	WorkChat	2	1	1	1	0	1	0	0	No	No	Yes	No	No	Correct fix, landed by the developer.
09.09/09:12	Facebook	6	2	4	2	3	1	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.11/01:37	Instagram	4	3	1	0	1	0	3	0	Yes	No	No	No	No	Macro: “whatatimetobealive3”
09.18/14:12	Facebook	3	3	0	2	0	1	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.18/14:13	Instagram	4	2	2	0	1	1	2	0	No	No	Yes	No	No	Correct fix, landed by the developer.
09.18/14:14	WorkPlace	3	3	0	1	0	1	1	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.18/14:14	Instagram	4	4	0	2	0	0	2	0	No	No	No	Yes	No	“Pretty sure this isn’t the cause of the crash (Which is already fixed)”
09.18/14:14	Facebook	2	1	1	1	0	1	0	0	No	No	No	Yes	No	Wrongly triaged.
09.18/14:15	Facebook	5	2	3	1	3	0	1	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
09.18/14:16	Messenger	6	1	5	1	4	1	0	0	No	No	No	No	Yes	“We do want to crash because we wanna know when it can be null.”
09.18/14:16	Facebook	2	2	0	1	0	0	1	0	No	No	No	Yes	No	“This isn’t the right fix at all, but it is really cool :)”
09.18/14:15	WorkChat	5	1	4	1	4	0	0	0	No	No	No	Yes	No	Rejected without comments.
09.18/14:17	Facebook	7	4	3	4	3	0	0	0	Yes	No	No	No	No	Fix accepted without comments.
09.18/15:16	FBLite	2	1	1	1	1	0	0	0	Yes	No	No	No	No	“lg2m :o”
09.18/15:46	Facebook	3	3	0	3	0	0	0	0	No	No	No	No	Yes	Not reviewed in 7 days.
09.19/10:42	WorkPlace	5	2	3	1	2	2	0	0	Yes	No	No	No	No	“It would be nice if the bot would also add ‘@Nullable‘” “to ImageOptions. Probably hard to do though :)”
09.19/14:00	Facebook	3	3	0	2	0	0	1	0	No	Yes	No	No	No	Correct fix: fixed by the developer before seeing the sapfix.
Mutation Fix Overall		56	-	-	40	-	4	11	1	5	1	3	6	7	**Fix attempts with at least 1 passing test patch: 27/39**
Templates Overall		60	-	-	-	43	17	0	0	4	1	2	4	6	**Fix attempts with at least 1 passing test patch: 22/39**
Total NPE Fixes: 39		**116**	**56**	**60**	**83**		**21**	**11**	**1**	**9**	**2**	**5**	**10**	**13**	
Results for our entire data set.															
Mutation Fix Overall		80	-	-	63	-	5	11	1	9	6	4	8	8	**Fix attempts with at least 1 passing test patch: 40/57**
Templates Overall		85	-	-	-	68	17	0	0	5	1	2	6	8	**Fix attempts with at least 1 passing test patch: 28/57**
Total NPE Fixes: 57		**165**	**80**	**85**	**131**		**22**	**11**	**1**	**14**	**7**	**6**	**14**	**16**	
%		**100**	**49**	**51**	**80**		**13**	**6**	**1**	**25**	**12**	**11**	**24**	**28**	
Total Correct Fixes(%)											**27/57(48%)**				

TABLE II
SAPFIX EXPERIMENT RESULTS ON PRE-EXISTING CRASHES WHERE WE LACK SAPIENZ TRIAGE DATA. "#C" IS THE NUMBER OF CRASHES. THE OTHER COLUMNS ARE THE SAME AS THOSE IN TABLE I.

		Strategy			Failed Patches				Developer says:		
Crash Type	**#P**	**#M**	**#G**	**#Pass Tests**	**#¬ Build**	**#¬ Sap**	**#¬ Pr.**	**#C**	**SAPFIX Land**	**Wrong**	**Unkn.**
Recent Crashes	**547**	288	259	213	166	65	103	**117**	16	35	66
%	**100**	53	47	39	30	12	19	**100**	14	29	56
Longstanding Crashes	**399**	230	169	139	132	39	89	**78**	13	24	41
%	**100**	58	42	35	33	10	22	**100**	17	30	53
Total	**946**	518	428	352	298	104	192	**195**	29	59	107
%	**100**	55	45	37	31	11	21	**100**	15	29	56

Lessons Learned and Future Work

Our philosophy in deploying automated repair was to focus on industrial deployment, rather than further research. This philosophy has strongly influenced all of the decisions we took. For example, it has been known for some time that random search over a suitably-constrained (fault localised) search space, can be surprisingly effective at finding candidate repairs [1]. Indeed, several fixes reported in early work on repair were found in the very first generation [22].

Our earlier work on Sapienz had fortunately led to scalable and sufficiently precise fault localisation, which contributed to the 75% fix rate for human developers, reported on elsewhere [6]. The existing deployment of Sapienz, together with these results from the literature gave us confidence that we could deploy a relatively simple end-to-end repair approach as a starting point. We also sought to re-use developer-defined patch templates as a starting point, knowing that the scientific literature demonstrated that this can work [23], but also in the firm belief that this would lead to more human-acceptable patches.

Finally, we were also motivated by more recent work on automated repair that has highlighted issues concerning weak oracles [24]. To ameliorate this problem, we use a combination of static and dynamic analysis to check, re-check, localise and identify the code that needs to change. We also use a combination of regeneration of search based tests with Sapienz, and human-written end-to-end tests, to provide a testing environment in which to check the repairs constructed by SAPFIX.

Humans still play the role of final gatekeeper with SAPFIX: no repair is landed into production without human oversight, so the repair system, although fully automated is, nevertheless, at this point merely a recommender system. This final human gatekeeper phase also provides us with insights from real-world developers' reactions when presented with automated repair candidates, on which we report.

We target Null Pointer Exceptions (NPEs) in the first instance because NPEs are such an important category of fault [25]. NPEs are also a highly prevalent fault category: Coelho *et al.* [26] analyze a set of 6000 Android stack traces that they extracted from more than 600 Android apps. They observe that more than 50% of the crashes are NPEs. This lower bound of 50% has been replicated for the top 1,000 android apps, using Sapienz automated testing [27] and we also found, at Facebook, that NPEs constitute at least 50% of the crashes triaged by Sapienz to developers. All of this empirical evidence pointed to NPEs denoting a natural high impact class of faults on which to direct our initial focus. NPEs also have the advantage, for automated repair, that fixes tend to be localised and small. As such, we anticipated a higher probability that mutation operators, combined with identification of fix patterns may lead to successful deployment

Much remains to be done, but we believe our initial deployment has allowed us to garner some experience, insights and initial results that may be useful to other researchers and practitioners, which we summarise in the remainder of this section.

End-to-end automated repair can work at scale in industrial practice: We have existential proof that developers do accept some automated patches; approximately one quarter of our patches landed into production code and a further quarter were deemed correct but not landed, either because the developer tweaked the fix or because they had already fixed the crash themselves when they first saw the proposed fix. This is encouraging. Clearly much more research and development work is needed and we certainly do not underestimate the challenges that lie ahead. Nevertheless, our results suggest that the hitherto open question as to whether end-to-end automated repair *could be deployed in industrial practice* is now answered, allowing the community to devote its full energy to tackling the many (exciting and impactful) open problems.

Developers are a useful final oracle: Automated oracles [28], and testing and verification will hopefully advance in the years to come, thereby widening the remit of automated repair. However, the developers' role as final gate keeper is likely to remain important for repair deployment while we await such further advances. Work on automated oracles can best support this aspect of the repair agenda by seeking to reduce (rather than replace) developer effort.

Sometimes deletion (reverting) is useful: high firing crashes in a master build of the system, even if never ultimately deployed to customers, will block further testing, so deleting them can be useful. This is an important use-case where the previously observed apparent predilection of automated repair to simply delete code (or to mask a failure, rather than tackling the root cause) is a behaviour that we seek; it can re-enable testing in the presence of a high-firing crash. Therefore, although deletion should be an anti-pattern for automated repair more generally [29], it is deployable in this specific use-case. However, more research is needed on the problem of finding the right code to delete without affecting subsequently-landed code modifications. Program slicing techniques [30], [31] might find in this repair-orientated problem, a new application domain. We also found that developers are resistant to Diff reversion (perhaps understandably). More work is therefore required on partial deletion and crash-masking, so that the effects of a crash can be suppressed while minimally affecting onward computation, not only nor even necessarily for end-user release, but also to support further testing.

Sociology: Developers may prefer to clone-and-own proposed fixes, rather than to simply land them (approximately one quarter of fixes deemed correct by engineers were, nevertheless, edited by them prior to landing). More work needs to be done on the sociology of automated repair; the interfaces between human and machine in repair.

Automated Explanations: Developers often showed a readiness and interest in communicating with the SAPFIX bot (even though they *knew* it to be a bot), as indicated by their comments and feedback. There is a significant, and as-yet untapped, potential for *dialog* between the automated repair tool and engineer. More work is needed on techniques for repair (and more generally program improvement [2], [32]) that *interact* with the developer to 'discuss' proposed changes. Sapienz uses an automated experimental framework [6] that seeks to scale up best practice empirical software engineering experimentation. This could be a starting point for automated experimentation.

Combine static and dynamic analysis: We have found that both static and dynamic analysis are complementary and mutually re-enforcing more generally [5], but also here in the specific case of automated repair. More work is needed to find blended analyses [33] that target repair.

Root cause analysis: SAPFIX might simply remove the symptom rather than addressing the root cause; masking the failure rather than fully fixing it. Although failure masking remains useful to unblock testing, techniques for identifying root causes of failures and appropriate (automatable) remediation remains very pressing problem.

Side Effects: Without fully automated oracles, our ultimate defence against side effects remains, as it does with human fixes, the developers and reviewers of Diffs. Much more work is still needed on automated test design and automation of (strong) oracles so that we can have greater confidence that passing all tests makes it unlikely that some knock-on effect is caused by a patch. Ultimately one would prefer to verify the absence of side effects; the FiFiVerify vision [10].

RELATED WORK

SAPFIX is grounded in the approach to software engineering known as Search Based Software Engineering (SBSE) [34], [35]; the space of potential fixes to a software systems is considered to be a search space constructed from small modifications to an existing system under test. In the deployment of SAPFIX we do not claim any strong novelty in terms of the core repair algorithm. In fact, SAPFIX does not use any of the repair approaches from the literature on repair and SBSE, since their sophistication might have inhibited scalability. Instead we favoured using more simple approach in which a patch is simply a higher order mutant [9], [36] and we perform a single generation search over this space; essentially little more than random search, with some smart selection. As such, our results may best be thought of as a base line [37], against which to measure the advances we hope to see produced by future research and development.

Automated fault finding

There are a several approaches that generate crashing test inputs for Android apps. SAPFIX is a general automated bug fixing technique that can be used in conjunction with any of these approaches: given a test case that one of these approaches produces, SAPFIX tries to generate patches for the bug that the failing test case reveals.

Currently, SAPFIX uses Sapienz [4], [6] and Infer [38], but could use other test generation techniques, such as AndroidRipper [39], an automated test case generation technique based on GUI ripping [40], ACTEVE [41] or Collider [42], concolic testing systems for Android [41], A^3E [43], a static data flow test tool, and/or other static analyses.

SAPFIX could also use Dynodroid [44], a feedback directed random test tool or Android Monkey, which are other popular Android testing tools. Although Sapienz has previously been shown to outperform both [4], these earlier results also indicated that the three techniques are, nevertheless, complementary.

There are also model based test tools such as FSMdroid [45] and fuzzers such as Fuzzdroid [46] that could be used to complement our results. It is likely that *any* or all of the techniques listed above (and many more that we could not list for brevity) might find additional faults not found by Sapienz and could thereby complement our initial deployment of SAPFIX. More generally, any static or dynamic analysis tool that scales to tens of millions of lines of code and 100k+ commits per week could be used as either a replacement or a complement to our use of Sapienz and/or Infer.

Automated fault fixing (repair)

SAPFIX is a simple realisation of Automated Program Repair, a topic that has been the subject of much research interest for over a decade [3], [47], [48], [49], [50], partly building on research on SBSE that has been a topic a research for more than two decades [34], [35]. In the present paper we do not seek to make significant novel contributions to the underlying science of automated repair, but rather to demonstrate, explain and bear witness to the real world applicability these research agendas on Automated Repair, Automated Test Case Design and SBSE.

Much related work exists on the topic of repair alone, so we cannot hope to do justice to all of it here. In the remainder of this section, we briefly review recent related work on Automated Program Repair and its differences and similarities to our deployment of SAPFIX. A more detailed survey can be found in the work of Monperrus [51]. According to the classification of Monperrus, SAPFIX is an offline behavioral repair approach.

Behavioral repair implies changing the source code of the buggy program. It requires an oracle to identify whether or not the bug is successfully fixed. Monperrus identifies three types of such oracle: test suites (the most closely related to SAPFIX), pre- and post- conditions, and abstract behavioral models [52].

Unlike the (many) other behavioral repair approaches in the existing literature [3], [48], [53], [54], [55], SAPFIX uses three different oracles to asses the quality and correctness of a fix: test cases from Facebook's CI, crash triggering sequences of UI events (similar to the work of Tan *et al.* [56]), and human reviewers. One novelty of our work derives directly from our industrial deployment; we are able to rely on expert engineers to act as the final arbiter of correctness in each case of a deployed repair.

This is the first time that professional engineers have played this role in the repair literature. It is also, simultaneously, an empirical evidence to support the claim that, at least there do exist automated program repairs that are, *ipso facto*, acceptable to expert professional software engineers.

Our SAPFIX approach targets Android NPEs and draws on templates automatically learned from human testers. Previously, Tan *et al.* [56] studied a set of Android crashes from Github to identify a set of 8 mutation operators that are often used by Android developers to fix bugs. One of their mutation operators is 'Missing Null Check', which is similar to our NPE mutation operators in the mutation fix strategy. Cornu *et al.*. have also targeted NPEs for repair [23]. NPEFix uses a predefined template-based approach similar with SAPFIX's templates.

SemFix [57] uses symbolic execution and code synthesis. Angelix [53] is an extension of SemFix that improves scalability and applicability by enhancing the symbolic execution stage. PAR [55] also uses repair templates to fix common types of bugs in Java programs. One of their templates is also a "Null Pointer Checker" that is parameterized by the variable name. PAR randomly applies the templates and validates the fix.

Nopol [54] is an automated bug fixing tool for two types of bugs: buggy `if` conditions and missing preconditions. In the case of missing preconditions, Nopol adds a guard (`if` statement), similar to our mutation fix operator for NPEs. Nopol synthesizes the fix, using oracles (input-output pairs) to guide component based patch synthesis [58].

Although SAPFIX represents the first industrial deployment of end-to-end repair (from automated test design through to fix deployment), there have been previous deployments of other forms of automated code change, both in industry, and to open source development communities. For example, Google's Tricorder system is reported to recommend fixes [59]. However, Tricorder fixes are typically manually specified along with the analysis check, whereas Sapfixes are detected and checked by automatically-constructed tests as with SAPFIX. Automated refactoring has also been widely studied [60] and has found deployment at scale in industry [61]. However, refactoring seeks to apply known-to-be semantically safe changes; essentially altering syntax without disrupting semantics. Therefore, although undoubtedly important, refactoring is less challenging than repair, which necessarily affects semantics as well as syntax.

The Repairnator system was first deployed in 2017 [62] to suggest repairs to Github Java projects. Repairnator uses existing test suites in these open source projects to identify crashes so, unlike SAPFIX, it does not offer end-to-end test generation to repair. However, like SAPFIX, Repairnator does provide for the continuous deployment of repair techniques at considerable scale.

CONCLUSIONS AND FUTURE WORK

The SAPFIX system is now running in continuous integration and deployment. This is the first time that automated end-to-end repair has been deployed into production on industrial software systems in continuous integration and deployment. Much remains to be done. Our repairs aim to tackle the most prevalent (yet arguably also the most simple) bugs, fixable by small patches, comparatively easily checked by the final human gatekeeper. As such, our paper tackles few, if any, of the many interesting and exciting open research problems for automated repair. SapFix patches tend to ameliorate rather than fix root causes, which remains an open problem. Nevertheless, even masking a newly-landed failure can be useful to unblock automated testing of other recent code changes. We share the lessons we learned from the deployment of SAPFIX in this paper seeking to provide additional input to the development of this challenging but important research field from a practical industrial perspective. We hope thsi contributes to on-going and further research and development on Automated Program Repair, Automated Software Testing and Search Based Software Engineering.

REFERENCES

[1] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *SQJ*, vol. 21, no. 3, pp. 421–443, 2013.

[2] 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.

[3] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[4] 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)*, 2016, pp. 94–105.

[5] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis (keynote paper)," in 18th *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*, Madrid, Spain, September 23rd-24th 2018, pp. 1–23.

[6] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with Sapienz at Facebook (keynote paper)," in *SSBSE 2018*, 2018, pp. 3–45.

[7] ——, "Deploying search based software engineering with Sapienz at Facebook (keynote paper)," in 10th *International Symposium on Search Based Software Engineering (SSBSE 2018)*, Montpellier, France, September 8th-10th 2018, pp. 3–45, springer LNCS 11036.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649 – 678, September–October 2011.

[9] ——, "Higher order mutation testing," *Journal of Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

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

[11] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[12] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011.

[13] 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 *HPCA*, 2018.

[14] J. Bader, S. Chandra, E. Lippert, and A. Scott, "Getafix: How Facebook tools learn to fix bugs automatically." [Online]. Available: https://code.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/

[15] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[16] M. Harman, "The current state and future of search based software engineering (invited paper)," in *29th International Conference on Software Engineering (ICSE 2007), Future of Software Engineering (FoSE)*, Minneapolis, USA, 2007.

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

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

[19] 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*, 2008, pp. 385–398. [Online]. Available: https://doi.org/10.1007/978-3-540-70545-1_36

[20] 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.

[21] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at Facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[22] W. Weimer, "Automatically finding patches using genetic programming (talk slides)," $1^{st.}$ CREST Open Workshop, 24th - 25th November 2009, CREST Centre, London, UK. [Online]. Available: http://crest.cs.ucl.ac.uk/cow/1/

[23] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus, "NPEFix: Automatic runtime repair of null pointer exceptions in java," 2015, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01251960

[24] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*, 2015.

[25] C. A. R. Hoare, "Null references: The billion dollar mistake," in *QCON conference*, London, England, 2009. [Online]. Available: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[26] R. Coelho, L. Almeida, G. Gousios, A. Van Deursen, and C. Treude, "Exception handling bug hazards in android," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1264–1304, 2017.

[27] K. Mao, "Multi-objective search-based mobile testing," Ph.D. dissertation, University College London, Department of Computer Science, CREST centre, 2017.

[28] 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*, vol. 41, no. 5, pp. 507–525, May 2015.

[29] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016.

[30] M. Harman and R. M. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.

[31] M. Weiser, "Program slicing," in 5^{th} *International Conference on Software Engineering*, San Diego, CA, Mar. 1981, pp. 439–449.

[32] 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)*, Essen, Germany, September 2012, pp. 1–14.

[33] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *ISSTA*, 2007.

[34] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[35] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 11:1–11:61, November 2012.

[36] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*. Beijing, China: IEEE Computer Society, 2008, pp. 249–258.

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

[38] 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*, 2015, pp. 3–11.

[39] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[40] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *null*. IEEE, 2003, p. 260.

[41] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 59.

[42] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *ISSTA*, 2013.

[43] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[44] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[45] T. Su, "FSMdroid: guided GUI testing of android apps," in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*. IEEE, 2016, pp. 689–691.

[46] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 300–311.

[47] A. Arcuri and X. Yao, "A Novel Co-evolutionary Approach to Automatic Software Bug Fixing," in *IEEE Congress on Evolutionary Computation (CEC'08)*. Hongkong, China: IEEE Computer Society, 1-6 June 2008, pp. 162–168.

[48] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard, "Automatically patching errors in deployed software," in *SOSP'09*, 2009, pp. 87–102.

[49] F. Long and M. Rinard, "Automatic patch generation by learning correct code."

[50] ——, "Staged program repair with condition synthesis," in *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 2015, pp. 166–178.

[51] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[52] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 181–190.

[53] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016.

[54] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[55] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE'13*, 2013, pp. 802–811.

[56] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in 40^{th} *International Conference on Software Engineering (ICSE 2018)*, 2018.

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

[58] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 215–224.

[59] 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*, 2015, pp. 598–608. [Online]. Available: https://doi.org/10.1109/ICSE.2015.76

[60] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, 2004.

[61] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, "Large-scale automated refactoring using clangmr," in *International Conference on Software Maintenance (ICSM 2013)*, 2013, pp. 548–551.

[62] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot? insights from the repairnator project," in *40th International Conference on Software Engineering, Software Engineering in Practice track (ICSE 2018 SEIP track)*, May 27 2018, pp. 1–10.