Evaluating Automatic Program Repair Capabilities to Repair API Misuses

Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman

Abstract—API misuses are well-known causes of software crashes and security vulnerabilities. However, their detection and repair is challenging given that the correct usages of (third-party) APIs might be obscure to the developers of client programs.
This paper presents the first empirical study to assess the ability of existing automated bug repair tools to repair API misuses, which is a class of bugs previously unexplored. Our study examines and compares 14 Java test-suite-based repair tools (11 proposed before 2018, and three afterwards) on a manually curated benchmark (APIREPBENCH) consisting of 101 API misuses. We develop an extensible execution framework (APIARTY) to automatically execute multiple repair tools.
Our results show that the repair tools are able to generate patches for 28% of the API misuses considered. While the 11 less recent tools are generally fast (the median execution time of the repair attempts is 3.87 minutes and the mean execution time is 30.79 minutes), the three most recent are less efficient (i.e., 98% slower) than their predecessors. The tools generate patches for API misuses that mostly belong to the categories of missing `null` check, missing value, missing exception, and missing call. Most of the patches generated by all tools are plausible (65%), but only few of these patches are semantically correct to human patches (25%).
Our findings suggest that the design of future repair tools should support the localisation of complex bugs, including different categories of API misuses, handling of timeout issues, and ability to configure large software projects. Both APIREPBENCH and APIARTY have been made publicly available for other researchers to evaluate the capabilities of repair tools on detecting and fixing API misuses.

Index Terms—Automatic Program Repair, Application Programming Interfaces, API Misuses, Bug Benchmarks.

1 INTRODUCTION

ALMOST all modern software systems (e.g., software platforms, mobile, and web applications) depend heavily on Application Programming Interfaces (APIs).[1] As new initiatives and trends emerge, APIs quickly follow, thereby supporting reuse, component-driven development and architectural encapsulation. For example, consider the recently appeared blockchain applications.

While APIs accelerate software development and reduce software production costs [1], violations of APIs' explicit or implicit usage constraints (i.e., their contracts) can have detrimental effects (software crashes and security vulnerabilities) on the user experience of client programs [2], [3], [4]. These incorrect usages of APIs by client applications represent API *misuses* [2], [5]. An API misuse occurs, for instance, when a client program calls an API method that expects a non-`null` constrained formal parameter (based on its specification) without validating (i.e., via `null` checks or error handling) the inputs passed to its arguments. Additionally, developers should be always aware of API-versioning issues. As APIs are part of the software, APIs themselves evolve, and the client programs that call them need to be timely updated to new API versions to avoid further API misuses [3].

Automatic program repair tools (hereafter, referred to as *repair tools*) can be promising tools to address potential issues related to opaque APIs. These techniques automatically detect bugs in the source code and generate corresponding repair patches using *oracles* [6] (e.g., test cases, logic rules, specification, historical data). Related work shows that repair tools can reveal real bugs [7] and have been already successfully used in industry [8].

However, the performance (effectiveness and efficiency) of repair tools in automatically detecting and fixing API misuses remains unknown. To the best of our knowledge, this happens for two main reasons. First, there are no available repair tools specifically designed to automatically detect and fix API misuses that can be directly evaluated by researchers. Second, even though API misuses may exist in the benchmarks used for the evaluation of state-of-the-art repair tools, previous research does not focus on whether repair tools can detect and fix API misuses.

Therefore, in this paper, we conduct the first empirical study that evaluates the capabilities of state-of-the-art repair tools for automatically detecting and fixing API misuses. Moreover, based on the results of our study, we derive suggestions for aiding repair tools to effectively and efficiently target API misuses. We believe that our study is vital for researchers to understand and tackle the challenges for designing API-repair tools in the future.

To this end, we design and carry out the first large-scale empirical study on the performance of repair tools for API-misuse repair, following the most recent best practices in the field [2], [6], [9]. Specifically, our study compares the ability of 14 Java test-suite-based repair tools to automatically detect and fix API misuses in client Java programs. According to Liu et al. [10], we categorise these tools into *less recent* (introduced before 2018) and *more recent* (introduced after 2018) tools. In our study, we compare 11 earlier tools (first

• *Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman are affiliated with University College London, Department of Computer Science. Mark Harman is also affiliated with Facebook London.*
E-mail: m.kechagia@ucl.ac.uk, s.mechtaev@ucl.ac.uk, f.sarro@ucl.ac.uk, mark.harman@ucl.ac.uk

1. With the term APIs we also refer to software libraries.

11 tools in Table 4) and three more recent tools (last three tools in Table 4).

As there is no available bug benchmark that can be directly used for evaluating API-repair tools, we have manually curated a benchmark of API misuses, named APIREPBENCH. We derived this benchmark from three large and diverse bug benchmarks to improve the generalisability of our findings [9]. Two of the three benchmarks (namely, BEARS [11], BUGS.JAR [12]) are for program repair, and one benchmark (namely, MUBENCH [2]) is for static API-misuse detection. APIREPBENCH contains 101 revisions of buggy software projects with API misuses, along with test cases (oracle) that can expose the API misuses.

Additionally, for a fair evaluation of the repair tools, we consider the particular classes of bugs (i.e., targeted defect classes) the tools are designed to work for, as recommended by Monperrus [6]. We also take into account the categories of API misuses as proposed by Amann et al. [2].

To carry out our experiments, we devise an execution framework called APIARTY, which allows us to automatically run multiple repair tools, and make it publicly available.[2] APIARTY can be extended with additional buggy project revisions with API misuses and repair tools, and it is machine independent.

Our results show that all repair tools can generate patches for 28% API misuses in our benchmark. The median time of the repair attempts of the 11 less recent tools, executed on the full APIREPBENCH, is 3.87 minutes and the mean time 30.79 minutes. The three recent tools, examined on a subset of APIREPBENCH, take significantly more time (more than two hours median and mean time) to run. The tools generate patches for API misuses that mostly belong to the categories of missing `null` check, missing value, missing exception, and missing call. Even though for all tools most of the generated patches are plausible (65%), only few of the generated patches are semantically correct to human patches (25%). Overall, recent tools are more effective for the bugs examined, with AVATAR and TBAR generating more than 60% plausible and semantically correct patches.

The contributions of this paper are the following:

- An empirical evaluation of 14 state-of-the-art Java test-suite-based repair tools and insights on advancing existing repair tools concerning their capabilities on API-misuse repair.
- A benchmark of 101 API misuses, APIREPBENCH, which can be used as a robust foundation for evaluating the capabilities of repair tools on detecting and fixing API misuses. APIREPBENCH is used for the present study but it can be also used for future work. Thus, we make the benchmark publicly available.
- A software infrastructure, APIARTY, that facilitates the systematic and reproducible evaluation of repair tools targeting API misuses.

In the rest of the paper, we first outline the key definitions and background of our study (Section 2). We then describe the used data sets and methods (Section 3). In Section 4, we present and discuss the results on the effectiveness and efficiency of the examined repair tools. We finally mention our threats to validity in Section 5 and list related work in Section 6. We conclude with an overview of our study and future work in Section 7.

2. Our benchmark, platform, and results are publicly available at: https://github.com/SOLAR-group/APIARTy, https://solar.cs.ucl.ac.uk/os/apiarty.html.

TABLE 1
Java Repair Tools Available in Literature

Tool	Type	Benchmark
Heuristic-based		
ARJA [13]	Generic	REPAIRTHEMALL
ARJA-GENPROG [7]	Generic	REPAIRTHEMALL
ARJA-KALIA [14]	Functionality removal	REPAIRTHEMALL
ARJA-RSREPAIR [15]	Statement change	REPAIRTHEMALL
ASTOR-CARDUMEN [16]	Statement change	REPAIRTHEMALL
ASTOR-GENPROG [17]	Generic	REPAIRTHEMALL
ASTOR-KALI [18]	Functionality removal	REPAIRTHEMALL
ASTOR-MUTREPAIR [18]	Generic	REPAIRTHEMALL
AVATAR [19]	Static violations	DEFECTS4J
CAPGEN [20]	Generic	DEFECTS4J
IFIXR [21]	Template-based	DEFECTS4J
SIMFIX [22]	Similarity-based	DEFECTS4J
SKETCHFIX [23]	Generic	DEFECTS4J
TBAR [24]	Template-based	DEFECTS4J
Constraint-based		
ACS [25]	`if` conditions	DEFECTS4J
CLOTHO [26]	Invalid `String` inputs	Custom
DYNAMOTH [27]	Method calls	REPAIRTHEMALL
Exception repair [28]	Incorrect assignments	Custom
FOOTPATCH [29]	Memory leaks	Custom
JAID [30]	Generic	DEFECTS4J
JAVACC [31]	Syntax errors	Custom
JFIX [32]	Generic	DEFECTS4J
JIST [33]	Branching-time logics	Custom
LOOPFIX [34]	Loop conditions	DEFECTS4J
NOPOL [35]	Conditional checks	REPAIRTHEMALL
NPEFIX [36]	Null pointer checks	REPAIRTHEMALL
Timestamps [37]	Timestamp overflows	Custom
VFIX [38]	Null pointer checks	Custom
Learning-aided-based		
ASTOR-DEEPREPAIR [18]	Generic	DEFECTS4J
CCA [39]	AST changes	DEFECTS4J
CONFIX [40]	AST changes	DEFECTS4J
ELIXIR [41]	Method invocations	Industrial
FIXMINER [42]	Generic	DEFECTS4J
GENPAT [43]	Generic	DEFECTS4J
GETAFIX [43]	Static violations	Industrial
HDREPAIR [44]	Generic	Custom
LASE [45]	Systematic edits	Custom
LSREPAIR [46]	Similarity-based	DEFECTS4J
PHOENIX [47]	Static violations	Custom
SEQUENCER [48]	Sequence-to-sequence	DEFECTS4J
SOFIX [49]	Template-based	DEFECTS4J
SSFIX [50]	Generic	DEFECTS4J
VURLE [51]	Security vulnerabilities	Custom

2 BACKGROUND AND DEFINITIONS

In this section, we describe the main concepts and definitions regarding automatic program repair, API misuses, and bug benchmarks broadly used in the rest of the study.

2.1 Automatic Program Repair

The following paragraphs explain the main concepts and evaluation limitations of automatic program repair, and review the existing Java repair tools available in literature.

Automatic program repair refers to the process of automatically fixing software bugs without human intervention. Program repair consists of two main phases. Initially, a

repair tool applies a fault localisation approach to identify a bug, and, then, it generates patches (using one or more of several approaches e.g., search-based software engineering [13], logic rules [29] etc.) that can possibly fix that bug. Afterwards, the (usually manual) evaluation of the correctness of the patches follows. An overview on the fundamental concepts of automatic program repair can be found in the surveys of Gazzola et al. [52], Monperrus [53], and Le Goues et al. [54].

Fault localisation: Automated program repair techniques receive as an input a buggy program and a correctness criterion (e.g., the program's test suite for the evaluation of the generated patches). Most repair tools start by using fault localisation to identify those locations in the source code that should be repaired. Such fault localisation procedures typically rank locations in the source code that are buggy. Popular fault localisation approaches for Java program repair include GZOLTAR and OCHIAI [55].

Java repair tools: There is a plethora of available repair tools. We conducted a survey on these tools that comprises two passes. First, we systematically searched[3] in the software engineering literature for repair tools. This search returned 332 results that we reviewed considering the title and abstract of the research papers of the tools. From these results, we found 115 papers that present novel repair tools. Out of these, 61 refer to Java repair tools (excluding Android). We have excluded working papers (e.g., tools that are published only on ARXIV), duplicates (i.e., different references for the same tool), and irrelevant papers to program repair (e.g., debugging tools). We thereby arrived at a final set that includes 43 repair tools (see Table 1).

We classified the 43 tools found into three categories, (namely, *heuristic repair*, *constraint-based repair*, and *learning-aided repair*), based on the repair approaches used. According to Le Goues et al. [54], the techniques for constructing repair patches can be divided into these three categories by considering two criteria: what types of patches are constructed and how the search is conducted.

Specifically, "Heuristic-based" tools use a generate-and-test methodology. First, the tools construct patches by iterating over a search space of syntactic program modifications. Second, the tools validate the generated patches by calculating the number of tests that pass when a suggested patch is applied to the program under examination. We note that in this category we also include tools that use template patterns since according to Liu et al. [10] technically these tools work similarly to "Heuristic-based". Additionally, "Constraint-based" tools proceed by constructing a repair constraint that the patched code should satisfy. These tools use symbolic execution and other constraint-solving techniques to extract properties for the functions to be synthesised. Finally, "Learning-aid-based" tools leverage the availability of previously generated patches and bug fixes to generate patches. These tools rely on machine learning to learn patterns for patch generation. Learning-aid-based techniques can be used by "Heuristic-based" and "Constraint-based" repair tools to improve the performance of these tools.

3. In particular, we used the following query in Google Scholar: https://scholar.google.com/scholar?q="patch generation"+"program repair"+"synthesis"+"java". Accessed on the 3rd of March 2020.

We also list the Java bug benchmarks that have been used for the evaluation of the tools, in the related work, as presented in the third column of Table 1. Note that "REPAIRTHEMALL" refers to all the bug benchmarks used by Durieux et al. [9], "Custom" means that a tool has been evaluated on a non-standard bug benchmark (e.g., on the ten most popular Java projects mined from GITHUB), and "Industrial" means that a tool has been evaluated on a bug benchmark from an industrial setting.

We further extracted repair tools that have *comparable* characteristics (e.g., similar design, inputs, outputs) to each other for studying the tools' capabilities on repairing API misuses (see Section 3.2). Additionally, there are several approaches (e.g., [56], [57], [58], [59]) that attempt to synthesise client code and examples for a given API. However, these approaches are evaluated on small programs. Here, we are interested in repair tools that can be broadly applied to client programs, and fix already written code with API misuses.

Evaluation limitations: Recent studies on automatic program repair highlight the following limitations, in the evaluation, of state-of-the-art repair tools. First, these tools can repair particular types of bugs (e.g., missing `null` check) and should be evaluated based on the types of bugs that the tools are designed to repair [6]. Second, these tools may suffer from imprecision. The tools produce repair patches that developers need to (manually) validate since may the patches do not fix any bugs or should be updated. This can happen for instance when a repair tool cannot identify a bug to repair afterwards [55]. Finally, most tools are evaluated based on bug benchmarks that do not necessarily reflect the complexity of real systems' bugs [9].

2.2 API misuses

An API can be considered as a bundle of interfaces, classes, and methods that client programs call or implement. According to Fazzini et al. [3], an API usage is any call of one or more methods of either old or new API versions, i.e., $API = [m1, ..., mk]$. An API misuse occurs when the clients of an API violate an API specification (i.e., the implicit or explicit usage constraints of an API).

In a recent work, Amann et al. [2] provide a taxonomy of API misuses and evaluate static API-misuse detectors. Table 2 lists representative categories of API misuses from this survey [2]. API violations can have detrimental effects, such as software crashes and security vulnerabilities, on software and users [5]. Therefore, it is essential for developers to fix API misuses in client programs as early as possible before releasing the software. Here, we use this taxonomy and benchmark of API misuses (MUBENCH) to evaluate the capabilities of repair tools on fixing API misuses. We focus on the "Missing API-usage element" category referring to the first seven rows of Table 2 as these elements come from real projects (not synthetic), count a higher number of projects ("Distribution"), and mostly lead to crashes.

To build software that complies with the usage constraints of called APIs in client programs, developers need to read, and comprehend, APIs' reference documentation. The API reference documentation should list all the appropriate usage constraints of an API, so that developers can correctly use that API. Many empirical studies [60], [61], [62] have,

TABLE 2
API-misuse Categories and Representative Examples Extracted From MUBENCH [2]

API-misuse Category	Category Description	Representative API Misuse	API-Misuse Description	Distribution (#)
Missing API-usage element				
Method call (MC)	Missing method call sequence	jackrabbit–5	Missing `java.io.InputStream.close` in `QValueFactoryImpl.BinaryQValue`	30
Null check (MNC)	Missing `if` condition for `null` check	aclang–1	Missing `null` check for `StringBuilder.getNullText` called in `StringBuilder.appendFixedWidthPadLeft`	25
Value or state (MV)	Missing check for input value or state	minecraft–launcher–1	Passed unsafe input (`PBEWithMD5AndDES`) in `javax.crypto.Cipher` called in class `BaseAuthenticationService`	21
Synchronisation (MS)	Missing synchronisation element	testng–17	Unsynchronized `java.util.List.m_configIssues` collection in `JUnitXMLReporter.generateReport`	1
Context (MX)	Component called from the wrong context	synthetic_directives–callondte	UI component (`javax.swing.JFrame`) called within a wrong context (`CallOnDTE.main`)	1
Iteration (MI)	Missing condition for next iteration in loop	synthetic_directives–wait–loop	Missing `java.lang.Object.wait` in `WaitWithoutLoop.misuse`	1
Exception handling (ME)	Missing `try–catch` construct	itext–1	Uncaught might-thrown `InvalidKeyException` by `javax.crypto.Cipher.init` called in `PdfPublicKeySecurityHandler.-computeRecipientInfo`	10
Redundant API-usage element				
Method call (RC)	Redundant method call (possibly replaced by another version of the called overloaded method)	adempiere–1	Unspecified encoding when a `String` is converted to bytes in `Cipher.doFinal`; `java.lang.String.getBytes()` is replaced by `java.lang.String.getBytes("UTF8")`	13
Null check (RNC)	Redundant `null` check for a non-`null` `Object`	closure–1	Redundant `null` check for `com.google.javascript.rhino.jstype.-UnionTypeBuilder.build()` (non-`null` `Object`) called in `UnionType.meet`	3
Value or state (RV)	Restrictive condition for value or state for iterations	synthetic_directives–toorestrictive	Restrictive iteration condition that skips the last element of a `java.util.List` in `Iterate.pattern`	2
Synchronisation (RS)	Redundant use of synchronisation	synthetic_directives–deadlock	The same `Object` is needlessly synchronised twice within (synchronised) block causing a deadlock	1
Context (RX)	Dispatched component to a context where it is already called	synthetic_directives–alreadyondte	Dispatched work of an UI element (`javax.swing.JButton`) to a context (DTE), while the current execution context is already the right one (DTE)	1
Iteration (RI)	Incorrect initialisation whithin iteration	synthetic_jca–loop–init	Reinitialised `javax.crypto.Cipher` with each iteration in `ReuseCipher.misuse`	1
Exception handling (RE)	Too generic caught `Exception` types	jigsaw–mudetect–12	Too generic caught `Exception` type (unrelated to lock conflicts) in `ToolsListerFrame.handle`	1

however, shown that API reference documentation is often inadequate, outdated, or complicated. Thus, programmers prefer instead to use informal references, such as STACKOVERFLOW, which can neither be accurate [4], [63], [64]. Therefore, the development of automatic tools that assists developers to detect and repair API misuses is vital.

Listing 2 presents an illustrative API misuse, in the `itext` Java project, of the `Javax Cipher` API displayed in Listing 1. In particular, when an invalid key is given for initialising a cipher, the `init` method of Listing 1 (line 14) will throw an `InvalidKeyException`. However, in Listing 2 (line 7) there is not any check or error handling for a possibly malformed passing value to the called `init` method. To correct this issue, the developers of `itext` used, in the revision 5091 of the project, a `try–catch` construct to catch this might-thrown exception.[4]

Listing 1. `Cipher` API `itext`

```
1  public class Cipher {
2      ...
3      /**
4       * Initializes this cipher with a key.
5       * ...
6       * @exception InvalidKeyException if the given key is
             inappropriate for
7       * initializing this cipher, or requires
8       * algorithm parameters that cannot be
9       * determined from the given key, or if the given key
             has a keysize that
10      * exceeds the maximum allowable keysize (as
            determined from the
11      * configured jurisdiction policy files).
12      * ...
13      */
14     public final void init(int opmode, Key key) throws
           InvalidKeyException {
15        init(opmode, key, JceSecurity.RANDOM);
16     }
17     ...
18 }
```

2.3 Bug Benchmarks

Software engineers build benchmarks of real-world and synthetic bugs to assess the performance of novel methods and tools. A bug in such a benchmark for evaluating repair tools typically includes: 1) a *buggy project revision* (e.g., the last commit (SHA) in the project's repository, where the bug exists), 2) a *fixed project revision*, where the bug is fixed (e.g., the commit (SHA) in the project's repository that fixes the bug), 3) a *bug-related test case* that triggers the bug (this test case is usually provided in the fixed project revision).

We use the buggy project revisions from bug benchmarks (BUGS.JAR [12] and BEARS [11]) for the evaluation of test-suite-based repair tools, along with buggy project revisions from a bug benchmark (MUBENCH [2]) originally designed for the evaluation of static API-misuse detectors. Each buggy project revision can contain zero or many API misuses; an API misuse can belong to more than one API-misuse categories (Table 2). Namely, $R = [a1, ..., ak]$, where R represents a buggy project revision, and a an API misuse.

4. https://github.com/stg-tud/MUBench/blob/master/data/itext/misuses/1/misuse.yml. All links have been accessed on the 16th of January 2021.

Listing 2. `Cipher.init` misused in `itext`

```
1  public class PdfPublicKeySecurityHandler {
2      ...
3      private KeyTransRecipientInfo computeRecipientInfo(
           X509Certificate x509certificate, byte[] abyte0)
4          throws GeneralSecurityException, IOException
5      {
6          ...
7          Cipher cipher = Cipher.getInstance(
               algorithmidentifier.getAlgorithm().getId());
8          cipher.init(1, x509certificate);
9          ...
10     }
11 }
```

Following the guidelines of Amann et al. [2] for MUBENCH, we consider an API misuse as a particular *instance* of a buggy project revision. Therefore, we can have different instances for the same buggy project revision. Hereafter, we call these instances as API *misuses*.

As MUBENCH is not constructed for evaluating repair tools, there are missing test cases for some buggy project revisions. To address this issue, we add amendments to its revisions (e.g., by constructing test cases), where it is needed. Section 3.3 presents our approach in detail.

3 EXPERIMENTAL SETUP

To evaluate repair tools, while repairing API misuses, we form three research questions that investigate the effectiveness and efficiency of the examined tools, and the API-misuse categories for which the tools can generate patches. We gather, from the related work, existing state-of-the-art Java test-suite-based repair tools and bug benchmarks that include API misuses. For our experiments, we develop a software infrastructure for the automatic execution of the tools. In the following, we present our methods in detail.

3.1 Research Questions

While related work shows that repair tools fix real-world bugs [9], [65], it remains unclear how these tools perform for API misuses. We aim to shed some light on this unexplored topic. We evaluate 14 state-of-the-art Java test-suite-based repair tools to investigate their repairability on API misuses, to understand any challenges and opportunities regarding API repair, and to make relevant suggestions for future repair tools. Our research questions (RQs) follow.

RQ1a): What percentage of API misuses can be automatically repaired by repair tools? With this RQ, we investigate the *effectiveness* of the repair tools on generating patches able to fix API misuses found in the bug benchmarks used for our study. Addressing this RQ helps us to understand the strengths and weaknesses of program repair on fixing API misuses.

RQ1b): What are the causes of non-patch generation in the context of API misuses? Following the guidelines of the study of Durieux et al. [9], which examines the reasons why tools may fail to generate patches, we systematically identify the causes of non-patch generation in the context of API misuses. This can help the repair community to evaluate and improve practical limitations of existing repair tools.

RQ2: How efficient is a repair tool when it automatically repairs an API misuse?

TABLE 3
Excluded Java Test-suite-based Repair Tools

Criterion Violated	Repair Tools
Availability	LOOPFIX, PHOENIX, VFIX, ELIXIR, SOFIX, GETAFIX, LASE, VURLE
Executability	CAPGEN, JFIX, LSREPAIR, ACS FOOTPATCH, PHOENIX, SEQUENCER
Configurability	DEEPREPAIR, SKETCHFIX, JAID, IFIXIR, FIXMINER, CONFIX, GENPAT, HDREPAIR, SSFIX

TABLE 4
Examined Java Test-suite-based Repair Tools

Tool	Framework	GITHUB Repository	Checked-out SHA	Evaluation Data Set
ARJA, GENPROG, RSREPAIR, KALI	ARJA	yyxhdy/arja	3e01305	APIREPBENCH
JGENPROG, JKALI, JMUTREPAIR, CARDUMEN	ASTOR	SpoonLabs/astor	da8a267	APIREPBENCH
NOPOL, DYNAMOTH	NOPOL	SpoonLabs/nopol	bf4a92f	APIREPBENCH
NPEFIX	–	SpoonLabs/npefix	80cfc38	APIREPBENCH
AVATAR	–	SerVal-DTF/AVATAR	68a1386	APIREPBENCH-D4J
TBAR	–	SerVal-DTF/TBar	d1b1555	APIREPBENCH-D4J
SIMFIX	–	xgdsmileboy/SimFix	c2a5319	APIREPBENCH-D4J

With this RQ, we examine the *efficiency*, i.e., the execution time and the number of patch candidates (NPC), of the repair tools while attempting to detect and fix API misuses. Addressing RQ2 shows us the factors (if any) that may delay the repair of certain API misuses. We assume that these factors may refer to technical characteristics of the tools.

RQ3: For which classes of API misuses do repair tools perform best? With this RQ, we reveal for which *classes* of API misuses (Table 2) repair tools can most effectively and efficiently generate patches able to fix API misuses. Answering this RQ helps us to understand whether particular characteristics of API-misuse classes can affect the performance of the tools. For instance, it may be more challenging for a repair tool to generate correct patches for misuses of missing API method calls rather than of incorrect values in `if` conditions. Additionally, answering this RQ help us to form suggestions for future repair tools.

3.2 Tool Selection

We systematically search the related work to find all the available state-of-the-art Java repair tools (Section 2.1). To be selected for inclusion in our study, a repair tool, from these listed in Table 1, must use the test suite of a project under examination for validation and meet the following inclusion criteria according to the guidelines of the recent related work [9], [10]. We exclude from the beginning CLOTHO [26], Exception repair [28], JAVACC [31], JIST [33], and Timestamps [37] since they are not typical test-suite-based repair tools.

- **Availability**: The repair tool should be publicly available. This is needed for supporting the reproducibility of our study since our study evolves the automatic downloading and execution of repair tools. Therefore, if the paper where the tool was presented

does not provide a link for the repository of this tool or we are not able to locate a repository of the tool in the internet (because for instance the tool has confidentiality issues as in the case of ELIXIR [41]), we exclude the tool from our study.

- **Executability**: The repair tool needs to be well-documented and easy-to-use. This is essential because one needs to understand how the tools work, to execute them, and to adapt them, so that they can run on our execution platform (Section 3.4). Even though some tools are publicly available, such as CAPGEN [20], it has been shown that these tools could not run due to technical issues [9]. Therefore, such tools are excluded from our study. Furthermore, the repair tool should be functional and able to run on a basic execution environment as ours (see Section 3.5). If a tool cannot run because of technical issues, we exclude it. For instance, ACS [25] does not allow programmed queries because of a change in the GITHUB interface it uses.[5]
- **Configurability**: The evaluated tools are test-suite-based, and, thus, these tools should use test cases and source code (input) as an oracle for generating repair patches [53]. However, there are tools such as SKETCHFIX [23] that need as input extra information regarding the faulty methods or classes that one cannot automatically retrieve. At this stage of our study, if a tool requires such extra inputs, we exclude it. We opt for this for the fair comparison of the tools, so that all tools receive similar inputs as presented in Section 3.4. Therefore, our criteria exclude all "Learning-aided-based" tools as they are not standalone [10], but require extra data (e.g., from GITHUB repositories for learning patterns).

Table 3 presents the repair tools excluded from our study along with the inclusion criteria they violate (i.e., reasons for their exclusion). Table 4 lists the tools and frameworks finally selected. Except for NPEFIX, AVATAR, TBAR, and SIMFIX, which are stand-alone repair tools, the remaining tools belong to three frameworks (ARJA, ASTOR, and NOPOL) as indicated in the second column of Table 4. For instance, the ARJA framework includes ARJA, GENPROG, KALI, and RSREPAIR.

In summary, we select all the tools that are publicly available, functional, and can run on the API-misuse benchmark we manually curated (i.e., APIREPBENCH) as described in Section 3.3. Additionally, we include three recently proposed repair tools in our study, namely, AVATAR, TBAR, and SIMFIX. These tools use novel repair approaches and have shown promising results [10] on DEFECTS4J bugs. Thus, it is also important to investigate their effectiveness and efficiency on API misuses. Since the current implementation of these tools does not allow the tools to directly run on non-DEFECTS4J bugs, we could assess these tools only on the DEFECTS4J API misuses contained in APIREPBENCH, and we name this subset of API misues APIREPBENCH-D4J.

The first repair tools had been developed for repairing C programs [7], [66], [67] and new versions of them have been realised for Java program repair in the ARJA and ASTOR tools. These works rest on the Plastic Surgery Hypothesis [68], that is, patches for a given software can be assembled out of fragments of code found in the existing code. In particular, GENPROG by Le Goues et al. [7] is a program repair approach based on genetic programming. It works at statement level and its repair operations include insertion, removal, and replacement of statements. KALI by Qi et al. [14] has operators that include removal of statements, modification of `if` conditions, and insertion of `return` statements. RSREPAIR by Qi et al. [15] is a variation of GENPROG that uses random search; not genetic programming as GENPROG.

ARJA by Yuan and Banzhaf [13] is a framework for Java program repair that comprises of four approaches, ARJA, and Java versions of GENPROG, KALI, and RSREPAIR. ARJA is a genetic programming approach that optimises the exploration of the search space by using several techniques including multi-objective search optimisation.

ASTOR by Martinez and Monperrus [18] is a Java library for program repair that provides implementations of C program repair tools (GENPROG, KALI, MUTREPAIR) in Java. It also includes CARDUMEN that refers to a repair approach based on mined templates [16].

DYNAMOTH by Durieux and Monperrus [27] is a sub-module of NOPOL. It targets buggy and missing `if` conditions but it also fixes method calls. Contrary to NOPOL, it uses the Java Debug Interface to collect information about the runtime context, variables, and method calls.

NOPOL by Xuan et al. [35] is a semantics-based repair tool that repairs `if` conditions. It is based on dynamic analysis and code synthesis with Satisfiability Modulo Theory (SMT). NOPOL uses angelic values (i.e., arbitrary values that make the failing test cases from a program under repair to pass) to determine the expected behavior of suspicious statements. Then, it collects runtime data and compares the expected with the actual values. Finally, using SMT, it finds feasible solutions and translates them into patches.

NPEFIX by Durieux et al. [36] automatically generates patches for fixing the `NullPointerException`. NPEFIX explores the search space of possible patches by using metaprogramming (i.e., automated code transformation).

AVATAR by Liu et al. [19] leverages fine-grained patterns of static analysis violations mined from software projects. Then, it uses these patterns as ingredients for automated patch generation to fix semantic bugs exposed by the test suites of the projects under examination.

TBAR by Liu et al. [24] considers a suspicious statement's AST, and tries to match this AST with the context AST of the right fix pattern that can be found in the search space. TBAR applies changes including `Insert`, `Update`, `Delete`, and `Move` until it identifies the right fix patterns.

SIMFIX by Jiang et al. [22] uses both existing patches and similar code to identify repair patches. Then, it validates the patches against the test suites of the projects examined.

3.3 Benchmark Selection and Filtering (APIREPBENCH)

We search the related work to find available bug benchmarks that can be used in the evaluation of repair tools targeting API misuses. Since existing bug benchmarks do not satisfy the criteria listed in Section 3.3.1, we derived,

5. https://github.com/Adobee/ACS

TABLE 5
Selected Bug Benchmarks

Sources	Total Projects	Total Revisions	Checked Revisions	Final Revisions	Final Projects
BEARS	72	251	251	19	10
BUGS.JAR	8	1,159	506	40	7
MUBENCH	68	280	244	42	12
Total	148	1,690	1,001	101	29

TABLE 6
Categories of API Misuses in APIREPBENCH

API-misuse Category	Misuses (#)
Missing Method call (MC)	31
Missing Value or state (MV)	28
Missing Exception Handling (ME)	25
Missing Null Check (MNC)	19
Missing Context (MX)	4
Missing Iteration (MI)	3
Missing Synchronisation (MS)	0
Total	110

from these, a benchmark specific to the task of evaluating API-repair tools. We followed an approach similar to the one adopted by Amann et al. [2], who built MUBENCH (a benchmark for static API-misuse detectors) by refining and filtering data from publicly available bug benchmarks. Specifically, to build our API-misuse benchmark, we manually extract and curate API-specific bugs from three publicly available bug benchmarks, namely BEARS [11], BUGS.JAR [12], and MUBENCH [2]. We opt for a benchmark having highly diverse software projects since this helps in strengthening the generability of the results of repair studies [9]. In the following, we describe the process we followed to construct our benchmark, named as APIREPBENCH.

3.3.1 Selection

To identify the bug benchmarks that we include in our study, we define the following inclusion criteria.

- **Java related**: The bug benchmarks must contain API-related bugs in the Java programming language. In particular, these bugs should involve misuses of APIs that can lead to software crashes [2]. We currently do not consider Android projects since the tools of our study cannot repair bugs for Android.
- **Documented**: The bug benchmarks should provide adequate information for one to easily find human patches (bug fixes) to the API misuses.
- **Test-suite availability**: The bug benchmarks must include test cases [55] (or bug fixes that can be used for easily writing relevant test cases [21]) for each API-related bug. Namely, for each API misuse, we should have at least one failing test case.
- **Public**: The bug benchmarks should be publicly available and presented in a research paper in the software engineering literature. This is required for supporting the reliability of our study.
- **Buildable**: The bug benchmarks should contain projects that can be built using the `ant`, `mvn`, and `gradle` building tools. We opt for these tools since they facilitate the automatic building of software projects, which is critical for our large-scale study.
- API **misuses**: The bug benchmarks should contain projects with API misuses referring to misused APIs that are called inside the main code of the projects under examination; we do not currently consider misuses located inside test code.[6]
- **Under active development**: The bug benchmarks should contain real and active software projects. This means that we exclude synthetic bug data sets. We opt for real projects since we are interested in investigating the capabilities of the tools on live projects and support the reproducibility of our study.

In the next paragraphs, we present the benchmarks selected and their particular characteristics.

MUBENCH[7] has been developed by Aman et al. [2], [69] and refers to an extensible Java bug benchmark. It contains 280 buggy project revisions all with API misuses stemming from 68 open-source Java and Android software projects. MUBENCH is made of previously available bug benchmarks (including BUGCLASSIFY [70], DEFECTS4J [71], QACRASHFIX [72]), newly constructed bug data sets for evaluating static API-misuse detectors, and synthetic bug data sets from developer studies. MUBENCH has been used for assessing static API-misuse detectors.

BEARS[8] has been introduced by Madeiral et al. [11] and presents an extensible Java bug benchmark. It contains 251 buggy project revisions from 72 GITHUB projects. This makes it the most well-known bug data set with bugs from the highest number of different software projects. The benchmark has been developed for the evaluation of Java program repair.

BUGS.JAR[9] has been published by Ripon et al. [12] and represents a large-scale Java bug benchmark. It contains 1,158 buggy project revisions from eight large Apache projects forming the largest known Java bug benchmark. The benchmark has been developed for the evaluation of Java repair tools.

3.3.2 Filtering and Refinement

MUBENCH contains 68 software projects in total. From those projects, we initially kept only 40 that can be automatically built using `ant`, `mvn`, and `gradle` commands. Accordingly, from MUBENCH, we kept only the buggy project revisions (that contain API misuses) of the projects selected (i.e., 244 out of 280 from all the projects initially found in MUBENCH). We further excluded Android projects and projects with bugs only in test cases. Additionally, we kept project revisions that contain tests that can trigger the API misuses. Given that there were not available test cases for all the buggy revisions, we created test cases that were missing (for the WEIBO, CONFUCIUS, and ASTERISK projects of Table 7). This is essential for benchmarks used in the evaluation of test-suite-based repair tools [55].

BEARS and BUGS.JAR are two data sets for the evaluation of repair tools. Thus, they already contain tests for relevant

6. For instance, cases such as https://github.com/stg-tud/MUBench/blob/master/data/jodatime/misuses/280/misuse.yml.
7. https://github.com/stg-tud/MUBench/tree/master/data
8. https://github.com/bears-bugs/bears-benchmark
9. https://github.com/bugs-dot-jar/bugs-dot-jar

TABLE 7
Projects in APIREPBENCH

Project Name	APIREPBENCH ID	Revision (SHA)	LOC	Java Files (#)	Misuses (#)
Activiti-activiti-cloud-app-service	ACTIVITI	d66c4337	952	22	1
apache-accumulo	ACCUMULO	17344890	348,681	1,514	7
apache-camel	CAMEL	62b2042b	1,381,397	17,072	5
apache-commons-lang	LANG	1fe5439ba	143,989	488	2
apache-commons-math	MATH	41f29780	477,725	2,572	7
apache-dubbo	DUDDO	c4dfc3ac	157,444	2,058	1
apache-flink	FLINK	494212b3	340,036	3,313	4
apache-jackrabbit	JACKRABBIT	56c328e9	278,741	2,851	1
apache-jackrabbit-oak	OAK	56accddf	350,780	2,698	8
apache-logging-log4j2	LOG4J2	86d8944f	107,541	1,294	4
apache-lucene-solr	LUCENE-SOLR	0cb96adf	172,230	1,368	1
apache-wicket	WICKET	61122bab	195,507	2,971	7
asterisk-asterisk	ASTERISK	41461b41	247,687	1,186	1
FasterXML-jackson-databind	DATABIND	93f7e14d	99,729	797	3
google-closure-compiler	CLOSURE	d1cfe679	292,252	1,436	3
hoverruan-weiboclient4	WEIBO	18b596ad	45,273	470	7
HubSpot-Baragon	BARAGON	a4a9387a	14,851	202	1
INRIA-spoon	SPOON	0e478718	80,660	1,189	1
IvanTrendafilov-confucius	CONFUCIUS	2fefd5eb	3,887	32	9
jfree-jfreechart	JFREE	2266	585,967	2,105	9
JodaOrg-joda-time	JODA	76fa4373	85,511	329	3
jriecken-gae-java-mini-profiler	PROFILER	30be3177	3,100	16	1
raphw-byte-buddy	BUDDY	f41aa877	149845	1089	1
spring-cloud-spring-cloud-gcp	SPRING-GCP	e35a0987	21,710	457	2
spring-projects-spring-data-commons	SPRING-DATA	cdbd0720	90,294	1,084	1
SzFMV2018-Tavasz-AutomatedCar	CAR	6a656ea5	607	23	1
Thomas-S-B-visualee	VISUALEE	58fbf0b8	10,309	90	3
traccar-traccar	TRACCAR	3f5122cb	43,683	763	7
Total	-	-	5,730,388	49,489	101

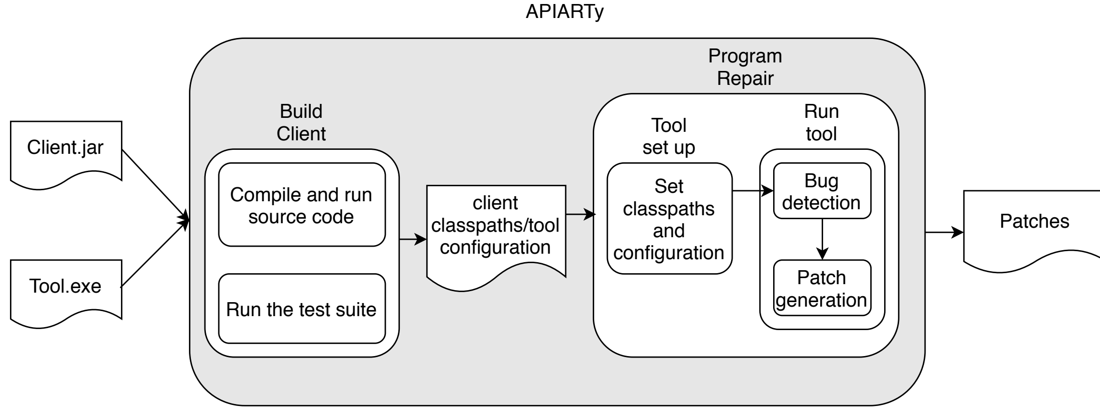


Fig. 1. Approach overview.

code changes in project revisions and human commits. We chose these bug benchmarks since they could be appropriately built and are ready to be used for program repair studies. However, contrary to MUBENCH, not all of the bugs in BEARS and BUGS.JAR represent API misuses. For this reason, we thoroughly manually examined the bugs from these two bug benchmarks and distinguished API-related bugs from general ones following the process below:

1) We observed and comprehended the taxonomy of API misuses of MUBENCH [2].
2) We extracted one representative example per API-misuse category as shown in Table 2.
3) The first two authors manually checked the buggy project revisions that belong to the BEARS and BUGS.JAR bug benchmarks to properly identify API-related bugs.
4) In order to further validate whether a bug is an API misuse, we used the SOOT framework [73] to automatically check whether this bug includes a call to an external API.[10]
5) The first two authors manually categorised the new bugs from BEARS and BUGS.JAR having as a guiding principle the representative examples of Table 2.
6) The first two authors discussed and reached on an

10. We used SOOT as it can analyse the byte code of Java projects and reveal the full namespace (e.g., `java.util.*`) of the calls of each statement in the source code. Thus, by revealing the namespace of these calls, we can automatically check whether there is a call to an external API (e.g., Java API, Apache libraries).

agreement regarding the category of each identified API misuse in BEARS and BUGS.JAR.

Given a bug present in an existing benchmark, we determine whether it is related to an API misuse by checking whether it satisfies all the criteria listed in the following:

1) The source code in the buggy project revision under examination contains a call to an external API (e.g., Java API, Apache libraries), and according to the test case associated with the buggy project revision (as described in Section 2.3), this call can lead to a crash. We note that a buggy project revision can consist of many lines of code. However, we are only interested in those that contain API calls.
2) The API called in the source code of the buggy project revision under examination is *misused* according to the API's reference documentation (e.g., there is a missing `null` check, which can lead to a `NullPointerException`).
3) The API-misuse candidate can be classified according to the categories of Table 2 from MUBENCH [2].

We note that since BUGS.JAR contains a large number of bugs, 1,159 program revisions, it was not feasible for us to manually check all these program revisions to distinguish API misuses from *general* bugs, and categorise them into the classes of Table 2. To address this issue, we applied twice stratified sampling (each time balanced with the number of bugs from BEARS), and extracted 506 revisions in total from BUGS.JAR. The total number of the examined buggy project revisions (i.e., "Revisions") appears in Table 5.

Table 6 shows the distribution of the API misuses (of the project revisions) extracted per API-misuse category. Consider that, in APIREPBENCH, we have 101 *unique* API misuses. However, in Table 6, we have in *total* 110 API misuses, because an API misuse can belong to more than one API-misuse categories (see "Misuse Category" in Table 8).

Table 7 lists 28 unique projects in APIREPBENCH, and summarises the characteristics of the projects including the project size (lines of code, LOC, and the number of the Java files). Note that in Table 5, we have 29 projects as MATH belongs both to BUGS.JAR and MUBENCH. For calculating these project metrics, we consider the latest reversion (SHA) of each project in APIREPBENCH.

3.4 Execution Framework (APIARTY)

For the evaluation of state-of-the-art Java test-suite-based repair tools on benchmarks of API misuses, we introduce the execution framework, APIARTY. The design of APIARTY is based on the fact that these repair tools share similar interfaces that typically receive the following list of parameters from the project revision that a bug belongs to:

- the path to the root directory of the source code;
- the path to the root directory of the test code;
- the path to the root directory of all the compiled classes of the source code;
- the path to the root directory of all the compiled classes of the test code;
- the paths to all the dependencies (`.jar` files).

The top level architecture of APIARTY is illustrated in Figure 1. APIARTY provides users with two interfaces. Users can run one (or more) repair tool(s) on the whole benchmark of API misuses (APIREPBENCH) or they can run one (or more) repair tool(s) on a project revision that is associated with a specific API misuse. For each project revision, APIARTY automatically builds the client program and specifies the right `classpath` to be used in the next phase by a repair tool. During the program repair phase, the tool under examination should first consider any configuration issue, and start the fault localisation. Then, the tool generates patches that should be (manually) evaluated afterwards. APIARTY stores the generated patches and any execution report produced (execution time and errors) for further investigation. APIARTY differs from REPAIRTHEMALL by Durieux et al. [9] that also evaluates repair tools as it works at project-revision level and not at benchmark level. This is essential as a bug benchmark may contain bugs that are not API misuses, and, then, that benchmark should be curated before being used by APIARTY.

APIARTY receives as an input a set of buggy project revisions that contain API misuses. Each buggy revision refers to a specific API misuse and is represented by a `.json` file that includes information such as: the project's name, the project's repository, the project's revision (SHA), where an API misuse is fixed, the test case that triggers the API misuse in that revision. APIARTY has already loaded and processed 101 buggy project revisions (API misuses). However, easily one can add more API misuses by providing appropriate `.json` files. Furthermore, APIARTY currently includes 14 Java test-suite-based repair tools. Users can also configure the tools by providing particular attributes (e.g., a timeout limit). APIARTY offers the following advantages:

- It is provided as a software image, and can run on different machines, for replicability and reuse in future work.
- Users can easily add more tools and benchmarks based on the documentation of APIARTY.
- It stores the generated patches and execution reports (execution time and errors) of the repair tools.
- It focuses on the evaluation of API-repair tools as it can process benchmarks containing API misuses.

3.5 Experiments and Settings

Given the 101 buggy project revisions, from the 28 open-source software projects presented in Table 7, we removed eight duplicated project revisions with repeated bugs and four project revisions with check-out issues and broken dependencies. We run the 11 less recent repair tools on the whole APIREPBENCH (89 *unique* buggy project revisions). We run the most recent tools (AVATAR, TBAR, and SIMFIX) on 12 out of 89 buggy project revisions that also exist in DEFECTS4J (see Section 3.2). Namely, we consider the following projects from Table 7: LANG, MATH, CLOSURE, and, JFREE. Therefore, we performed 1,015 repair attempts in total from all 14 repair tools. It took 36 days for all the repair tools to complete all experiments. We used the default tool settings of all tools. Also, we set the timeout to two hours for the 11 less recent tools, following the guidelines of the study of Durieux et al. [9]. For the most recent

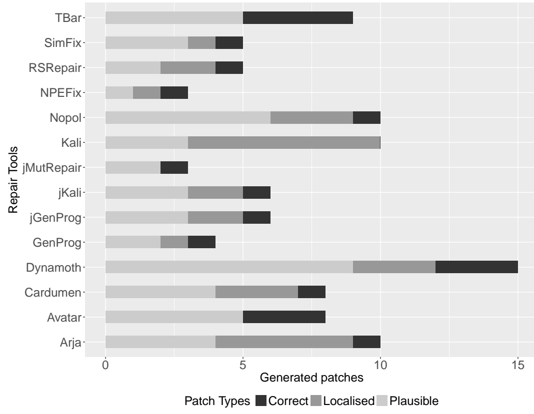


Fig. 2. **RQ1a):** Repairability of the repair tools based on the number of the patches generated by the tools.

tools (AVATAR, TBAR, and SIMFIX), we set the timeout to four hours, because we observed that these tools perform more operations than the 11 less recent tools to evaluate the generated patches against the whole test suite [19], [22], [24], and we wish to run all tools under a fair execution. All experiments run on an UBUNTU 16.04 LINUX docker image of APIARTY deployed on a 2-core PC with 8 GB RAM and 3,1 GHz Intel Core i5 processor. We used Java 1.8 (`amd64`) of the OPENJDK, allocating up to 4 GB for the JVM.

4 RESULTS

4.1 RQ1a): What percentage of API misuses can be automatically repaired by repair tools?

To answer RQ1a) we count (*per repair tool*) the number of the following elements:

1) **Generated patches** that each repair tool yields by the end of its execution.
2) **Plausible generated patches** that pass the initially failing API-related tests and all tests that were initially passing. To find those *test-suite-adequate* patches, given a patch generated by a repair tool, we manually inject the patch in the buggy project revision under examination and rerun the project's test suite. If the initially failing API-related test does not happen again, and there are not any new failing tests, we consider the generated patch as plausible. The ARJA-based tools can generate more than 100 patches while repairing a software bug. For these tools, we manually examine the first patch generated for plausibility and, then, correctness.
3) **Semantically correct generated patches** that are plausible and semantically similar to the human patches (*correctness criterion*). Since we have at our disposal the commits (human patches) that correct the buggy project revisions, we manually compare the generated patches with the human patches. For this comparison, we manually examine whether the generated patches and the developer-provided patches are *identical* (i.e., exactly same patches), *semantically-similar* (i.e., not identical patches but with the same effect on the program behaviour), or *incorrect* (i.e., irrelevant patches without having the same effect on the program behaviour) [10]. In particular, to define semantically-similar patches as correct, we considered the examples of semantic similarity rules used by Liu et al. [10]. To strengthen the validity of our results, the first two authors have cross-checked the semantic correctness of the generated patches. For the purposes of our study, we consider the *identical* and *semantically-similar* patches as *semantically correct* patches, given that both (*identical* and *semantically-similar*) have the same effect on the program behaviour. The explanation of the patches categorised as semantically correct and incorrect can be found in our public repository.[11]

Table 8 presents our results for the projects that the repair tools generate at least one patch. The results for the first 11 repair tools refer to the whole APIREPBENCH examined, whereas for the last three repair tools the results refer to the examination of a subset of projects that belong both to APIREPBENCH and DEFECTS4J, as mentioned in Section 3.2 and Section 3.5.

For all tools, we found that the repair tools can generate patches for 25 out of 89 unique API misuses (28%). In terms of repair attempts, 80 attempts out of 1,015 (8%) have generated patches. From these 80 generated patches, 52 patches are plausible (65%). The remaining generated patches were classified as *implausible* via separate validation because they either did not compile or did not pass the provided test suite. Additionally, from the generated patches, 20 patches are semantically correct (25%) to human patches. Even though the tools generate few semantically correct patches, 30 of the generated patches are semantically *incorrect* patches (38%) yet generated for the right bug locations in the source code.

Table 9 shows the overlap between each pair of repair tools in number of bugs. We use the same semantics used in previous work [9], [10]: The main diagonal, where the column name and the line name are the same, shows the number of unique bugs patched by a single tool, while the other cells show the overlap between every pair of tools. Specifically, bugs have been uniquely patched only by DYNAMOTH (3), KALI (1), and TBAR (1). We observe that there is a significant overlapping among those repair tools that are based on a same framework (see Table 4). Consider that ARJA has the highest overlap of > 40% with the ARJA-based repair tools KALI, GENPROG, and RSREPAIR.

Furthermore, between NOPOL-based tools (NOPOL, DYNAMOTH) there is a high overlap of > 60%. Among ASTOR-based tools there is also a high overlap ranging from 40% to 100%. By contrast, NPEFIX notes the lowest overlapping with the remaining tools, except for AVATAR, TBAR, and SIMFIX (> 20%). AVATAR, TBAR, and SIMFIX have also overlapping with ASTOR-based and NOPOL-based tools. Finally, ASTOR-based tools have high overlapping with ARJA-based tools since they use similar repair approaches. This finding comes in accordance with the observations of Durieux et al. [9] for the repairability of the tools on "general bugs".

11. Our benchmark, platform, and results are publicly available at: https://github.com/SOLAR-group/APIARTy.

TABLE 8

RQ1a): Summary of API Misuses From APIREPBENCH that Tools Generate Patches. ✓ means that there is a generated patch, ✗ means that there is no generated patch, ❍ means that the patch is plausible, ● means that the patch is semantically correct, ◆ means that the patch is semantically incorrect yet localised, and — means that the tool has not run on that bug.

Misuse ID	Misuse Category	Repair Tools ARJA	KALI	GENPROG	RSREPAIR	JGENPROG	JKALI	JMUTREPAIR	CARDUMEN	NOPOL	DYNAMOTH	NPEFIX	AVATAR	TBAR	SIMFIX
Bears															
SPRING-84	MC	✗	✓◆	✗	✗	✗	✗	✗	✗	✗	✗	✗	—	—	—
TRACCAR-101	MV	✓❍◆	✓❍◆	✗	✗	✓◆	✓◆	✗	✓❍◆	✗	✗	✗	—	—	—
TRACCAR-104	MV	✓❍◆	✓❍◆	✗	✗	✗	✗	✗	✗	✗	✗	✗	—	—	—
BUDDY-178	MI, ME	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	—	—	—
CAR-188	MC	✓❍●	✓◆	✓❍●	✓❍●	✓❍●	✓❍●	✗	✓❍◆	✗	✗	✗	—	—	—
Bugs.jar															
ACCUMULO-844	ME	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	—	—	—
MATH-891	MV	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	—	—	—
FLINK-1761	MV	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓❍	✗	—	—	—
FLINK-2800	MC, ME	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	—	—	—
OAK-2465	MV, MI	✗	✗	✗	✗	✗	✗	✗	✗	✓❍◆	✓❍◆	✗	—	—	—
LOG4J2-834	MC, ME	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	—	—	—
WICKET-5359	MV	✓❍	✓❍	✓❍	✓❍	✓	✓	✓	✓	✓❍	✓❍	✗	—	—	—
MUBench															
CLOSURE-3	MNC	✗	✗	✗	✗	✗	✗	✗	✗	✓❍◆	✓❍◆	✗	✗	✗	✗
LANG-1	MNC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓❍●	✗
MATH-2	MNC	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓❍●	✗
VISUALEE-29	MV, MC	✗	✗	✗	✗	✗	✗	✗	✗	✓❍●	✓❍●	✗	—	—	—
JFREE-3a	MNC	✗	✗	✗	✗	✓❍	✓❍	✓❍	✗	✓❍◆	✓❍◆	✗	✓❍	✓❍	✓❍
JFREE-6	MNC	✗	✗	✗	✗	✓❍◆	✓❍◆	✓❍●	✓❍◆	✗	✓❍●	✗	✓❍●	✗	✓❍●
JFREE-1	MC	✗	✗	✗	✗	✗	✗	✗	✓❍●	✗	✗	✗	✓❍●	✓❍●	✗
JFREE-2	MNC	✗	✗	✗	✗	✗	✗	✗	✗	✓❍	✓❍	✗	✓❍	✗	✗
JFREE-5	MNC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓◆	✓❍●	✗	✗
JFREE-7a	MNC	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓❍●	✗	✓❍●	✓❍◆
CONFUCIUS-95	ME	✓◆	✓◆	✗	✓◆	✗	✗	✗	✗	✗	✗	✗	—	—	—
CONFUCIUS-98	ME	✓◆	✓◆	✓◆	✗	✗	✗	✗	✗	✗	✓❍●	✗	—	—	—
CONFUCIUS-101	ME	✓◆	✓◆	✗	✓◆	✗	✗	✗	✗	✗	✗	✗	—	—	—

TABLE 9

RQ1a): Number of Overlapped Patched Bugs Per Repair Tool. Each row presents the percentage of bugs patched by a tool selected that were also patched by the remaining tools. For instance, 43% of the bugs patched by ARJA (first row) are also patched by GENPROG (third column). By contrast, 100% of the bugs patched by GENPROG (third row) are also patched by ARJA (first column). The percentages of bugs patched by AVATAR, TBAR, and SIMFIX refer to the DEFECTS4J bugs examined in our study. For instance, 100% of the (DEFECTS4J) bugs patched by JGENPROG (fifth row) are also patched by AVATAR (12th column).

	ARJA	KALI	GENPROG	RSREPAIR	JGENPROG	JKALI	JMUTREPAIR	CARDUMEN	NOPOL	DYNAMOTH	NPEFIX	AVATAR	TBAR	SIMFIX
ARJA	**0% (0)**	100% (7)	43% (3)	57% (4)	43% (3)	43% (3)	14% (1)	43% (3)	14% (1)	29% (2)	0% (0)	—	—	—
KALI	88% (7)	**13% (1)**	38% (3)	50% (4)	38% (3)	38% (3)	13% (1)	38% (3)	13% (1)	25% (2)	0% (0)	—	—	—
GENPROG	100% (3)	100% (3)	**0% (0)**	67% (2)	67% (2)	67% (2)	33% (1)	67% (2)	33% (1)	67% (2)	0% (0)	—	—	—
RSREPAIR	100% (4)	100% (4)	50% (2)	**0% (0)**	50% (2)	50% (2)	25% (1)	50% (2)	25% (1)	25% (1)	0% (0)	—	—	—
JGENPROG	60% (3)	60% (3)	40% (2)	40% (2)	**0% (0)**	100% (5)	60% (3)	80% (4)	40% (2)	60% (3)	0% (0)	100% (2)	50% (1)	100% (2)
JKALI	60% (3)	60% (3)	40% (2)	40% (2)	100% (5)	**0% (0)**	60% (3)	80% (4)	40% (2)	60% (3)	0% (0)	100% (2)	50% (1)	100% (2)
JMUTREPAIR	33% (1)	33% (1)	33% (1)	33% (1)	100% (3)	100% (3)	**0% (0)**	67% (2)	67% (2)	100% (3)	0% (0)	100% (2)	50% (1)	100% (2)
CARDUMEN	60% (3)	60% (3)	40% (2)	40% (2)	80% (4)	80% (4)	40% (2)	**0% (0)**	20% (1)	40% (2)	0% (0)	100% (2)	50% (1)	50% (1)
NOPOL	10% (1)	10% (1)	10% (1)	10% (1)	20% (2)	20% (2)	20% (2)	10% (1)	**0% (0)**	100% (10)	0% (0)	50% (2)	50% (2)	25% (1)
DYNAMOTH	13% (2)	13% (2)	13% (2)	7% (1)	20% (3)	20% (3)	20% (3)	13% (2)	67% (10)	**20% (3)**	0% (0)	60% (3)	40% (2)	40% (2)
NPEFIX	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	0% (0)	**0% (0)**	50% (1)	50% (1)	50% (1)
AVATAR	—	—	—	—	40% (2)	40% (2)	40% (2)	40% (2)	40% (2)	60% (3)	20% (1)	**0% (0)**	40% (2)	40% (2)
TBAR	—	—	—	—	20% (1)	20% (1)	20% (1)	20% (1)	40% (2)	40% (2)	20% (1)	40% (2)	**20% (1)**	40% (2)
SIMFIX	—	—	—	—	67% (2)	67% (2)	67% (2)	33% (1)	33% (1)	67% (2)	33% (1)	67% (2)	67% (2)	**0% (0)**

Figure 2 illustrates the capabilities of the repair tools regarding their repairability on API misuses. For the repair tools that run on APIREPBENCH, we observe that DYNAMOTH along with NOPOL, KALI, and ARJA generate the highest number of patches. By contrast, NPEFIX produces the lowest number of patches. Furthermore, all tools, except for KALI, generate more than half of the patches as plausible ("Plausible"). Specifically, CARDUMEN, GENPROG and JMUTREPAIR generate the highest percentages (more than 60%) of plausible patches, whereas KALI generates the lowest percentage (38%) of plausible patches.

Also, for all tools, less than half of the generated patches are semantically correct ("Correct") when compared to the human patches. For instance, even though DYNAMOTH, NOPOL, KALI, and ARJA generate most of the patches, less than 20% of the patches of DYNAMOTH, NOPOL, and ARJA are semantically correct, and none of the patches of KALI.

The tools generate few semantically correct patches, but correctly detect the locations of the API misuses in the source code for almost half of the generated patches, given that both the semantically correct and the semantically incorrect patches are generated for the right bug locations ("Correct" and "Localised"). Finally, all the semantically correct patches are also plausible. Namely, "Correct" $\leqslant$ "Plausible".

Additionally, Figure 2 shows that AVATAR and TBAR generate the highest number of patches when they run on the DEFECTS4J subset of APIREPBENCH. The generated patches of AVATAR, TBAR, and SIMFIX are all plausible and most of the generated patches are statically correct (62%). Observing these results, one could say that possible extensions of these tools, which can make them able to run on diverse bug benchmarks, could increase the overall number of fixes by the tools. Furthermore, new API-repair tools should leverage techniques (i.e., patterns of static-analysis violations, fix-pattern templates, and code-similarity scores) used by AVATAR, TBAR, and SIMFIX to increase their effectiveness.

Overall, it seems promising that tools not originally designed for repairing API misuses can generate patches for almost one quarter (28%) of the API misuses examined. In particular, NPEFIX, RSREPAIR, GENPROG, DYNAMOTH, and the ASTOR-based tools (except for CARDUMEN)

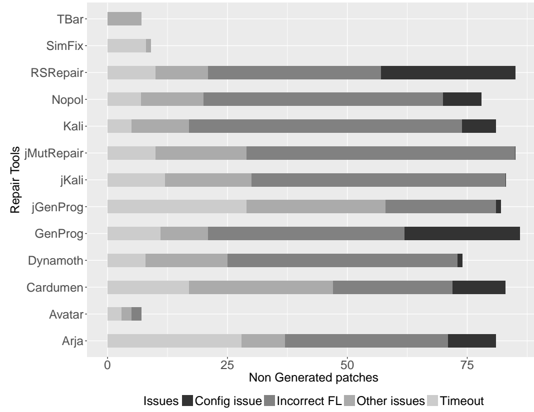


Fig. 3. **RQ1b):** Non-patch generation cause categories.

generate a significant number of plausible patches that are correct (more than 30%). By contrast, ARJA, CARDUMEN, and NOPOL generate fewer plausible patches that are correct (less than 30%). Finally, AVATAR, TBAR, and SIMFIX generate few patches but a high number of plausible patches that are also correct (62%).

> **Answer to RQ1a).** The repair tools generate patches for API misuses (28%). Most of the generated patches are plausible (65%). Few of the generated patches are semantically correct (25%). 38% of the generated patches are semantically *incorrect* yet generated for the right API-misuse locations. NOPOL-based tools, KALI, and ARJA generate the highest number of patches. However, less than half of their plausible patches are semantically correct. AVATAR and TBAR have the highest number of plausible and correct patches (⩾ 60%). There is a significant overlap in the API misuses identified by repair tools that are built based on a same framework.

4.2 RQ1b): What are the causes of non-patch generation in the context of API misuses?

To answer RQ1b), we consider the categories of causes of non-patch generation listed in the study of Durieux et al. [9]. Specifically, we manually examine 935 reports of failure to generate patches produced by the 14 repair tools, label them, and classify each into a non-patch generation category. For the labelling, we examined the error messages of the non-patch generation reports produced by the repair tools. In the following, we explain each cause category.

- **Cannot repair**: Repair tools do not generate patches for fixing bugs outside the search space they are designed for. Specifically, NPEFIX is not designed to generate patches for bugs that do not lead to a `NullPointerException`. There are 70 such bugs in our study. One of these, namely TRACCAR-104[12] represents a missing value API misuse, and NPEFIX failed to fix it. We also found that for 12 bugs the tools are not able to generate patches that should import new APIs into the source code, i.e., new `import` statements, such as in the case of CLOSURE-3.[13]
  Future repair tools should be designed for repairing different bug categories such as those presented in Table 2. This implies that novel repair tools should be also able to repair bugs related to missing APIs required in the source code.
- **Incorrect fault localisation (FL)**: The fault localisation of repair tools does not successfully detect the location of a bug, and the repair tools are not able to generate any patch for this bug. This can arise because of limitations of the fault localisation approaches used by the repair tools, or to the suspiciousness threshold used. For instance, the tools could not repair PROFILER-39.[14] This can happen because the tools do not consider the specification of the API used. In total, 435 repair attempts belong to this category.
  To overcome such issues, future repair tools could combine API-misuse detectors [2] with the fault localisation approaches used in current repair tools to precisely identify and fix API misuses. New studies could also evaluate the quality of test cases for detecting API misuses, and equip novel repair tools with more precise test-driven fault localisation.
- **Multiple faulty locations**: Repair tools are not able to generate patches that can fix multi-location bugs (i.e., bugs that require changes in more than one location in the source code to be fixed). In the context of API-misuses, we found for instance that for fixing DATABIND-8,[15] developers made an API change and a change in the client code. Repair tools could suggest a change for the client code but not for the API itself. We found four such bugs in our study.
  Therefore, future repair tools need to generate multi-location patches to fix a single bug, and also be able to suggest changes both for APIs and client code.
- **Timeout**: Repair tools are not able to generate patches for fixing a bug within a given time budget. In our study, 148 repair attempts failed by timeout.
  Future studies need to evaluate in depth timeout issues, and suggest solutions that can overcome bottlenecks occurred by existing repair tools. For instance, new repair tools can be faster if they use more precise fault localisation for API-misuse detection [74]. Additionally, future studies need to devise and investigate extra evaluation measures for the efficiency of repair tools as investigated by Liu et al. [10].
- **Configuration issues**: Repair tools fail to generate patches and fix bugs because of a failure to correctly set up a project under examination and compilation errors. In our study, 38 repair attempts failed by missing dependencies and 52 repair attempts by compilation errors during patch evaluation. Dependency issues are mostly related to the structure of the projects used, which can have many dependencies,

12. https://github.com/traccar/traccar/commit/c53feac38ec149fed84121704c620503bf1f7820

13. https://github.com/google/closure-compiler/commit/d1cfe67977d8f3aaa85ec20c262171da394d5977

14. https://github.com/emopers/gae-java-mini-profiler/commit/30be31776655f73487a59d443b30c7f7408f251b

15. https://github.com/FasterXML/jackson-databind/commit/64967c410514ec8a94bb11bb26d6a37fafafc14b

and some of them could not be properly retrieved. Configuration issues happen in large-scale empirical studies, and, thus, new configuration tools constantly aim at resolving such issues [9].

- **Other issues**: Repair tools cannot execute for some projects under examination for technical reasons. This could happen because of the length of command lines or because a dependency (e.g., the fault localisation tool) of a repair tool cannot be found. For instance, we found that ARJA failed to fix ACCUMULO-844[16] by the `error=7, Argument list too long` and all ARJA-based tool failed to fix SPOON-80[17] by a `NullPointerException` related to GZOLTAR. Furthermore, in this category, we have added repair attempts that failed by an `OutOfMemoryError`, a killed process, or hanging (e.g., the analysis never starts after having set up a project). In total, we have 175 repair attempts in this category.

Once we have manually categorised each of the non-patch generation reports, we can count how many belong to each category. Figure 3 illustrates our findings. Note that we exclude NPEFIX since it can only repair bugs that lead to the `NullPointerException`, and all bugs that it cannot fix belong to the "Cannot repair" category. Furthermore, we separately count the cases that belong to the "Cannot repair" and "Multiple faulty location" (as previously mentioned) since these categories refer to the characteristics of the bugs themselves and not to the characteristics of the tools as the remaining categories presented in Figure 3.

Answer to RQ1b). Half (51%) of the reports from all tools are associated with "Incorrect FL". A significant number of reports is related to "Other issues" (21%) and "Timeout" (17%), whereas only few reports refer to "Configuration issues" (11%). Specific characteristics of API misuses can contribute to the repairability of the tools since 86 repair attempts failed by causes belonging to the "Cannot repair" and "Multiple faulty locations" categories.

4.3 RQ2: How efficient is a repair tool when it automatically repairs an API misuse?

To answer RQ2, we follow two strategies. First, as in most empirical repair studies [9], [75], [76], we calculate the *execution time per repair attempt* for each repair tool. Second, we use as an efficiency measure the *number of patch candidates* (NPC) as recently introduced by Liu et al. [10]. According to Liu et al., NPC shows the efficiency of a repair tool based on the number of patch candidates generated by the tool until the first *valid* (i.e., plausible) patch is found [10]. In particular, Liu et al. [10] argue that NPC can be considered as an accurate efficiency metric since the measurement of execution time alone can be biased depending on the capabilities of the execution environment, where repair tools run on. In the following, we present our results in detail.

TABLE 10
RQ2: Execution Time of Repair Attempts

Repair Tool	Median (Minutes)	Mean (Minutes)
JGENPROG	66.18	61.61
ARJA	47.87	55.41
JKALI	9.17	29.56
CARDUMEN	6.00	36.03
JMUTREPAIR	5.75	27.83
GENPROG	3.87	26.52
RSREPAIR	3.37	27.05
KALI	3.02	16.44
NOPOL	2.82	23.26
NPEFIX	2.28	13.42
DYNAMOTH	2.08	21.61
	3.87	**30.79**
AVATAR	104.5	124.17
TBAR	240	170.92
SIMFIX	240	203.5
	240	**166.19**

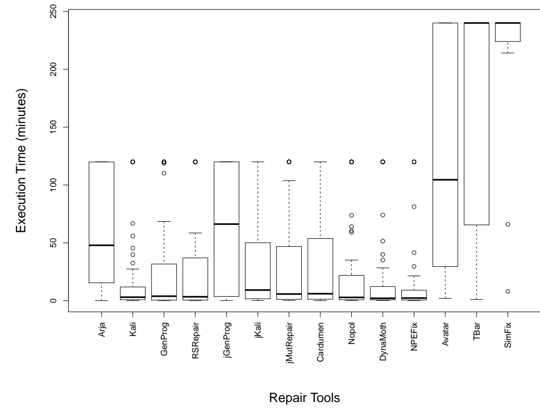


Fig. 4. **RQ2:** Execution time per repair tool. The timeout for AVATAR, TBAR, and SIMFIX is four hours, whereas for the rest two hours.

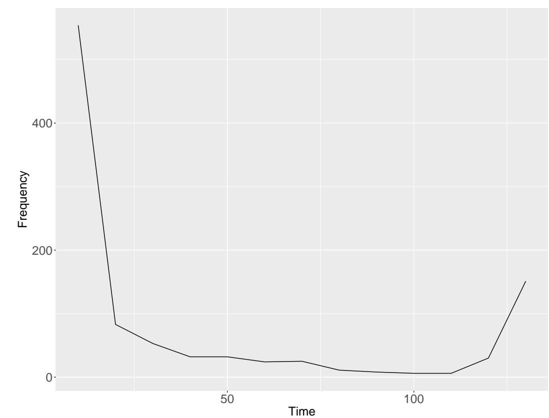


Fig. 5. **RQ2:** Overall trend of repair attempts from all tools.

16. https://github.com/apache/accumulo/commit/692efde2c24b2ec100b04ada0656079ef8f60fbf
17. https://github.com/INRIA/spoon/commit/0e4787187d8d6192c43144de0fc91e5047fa867b

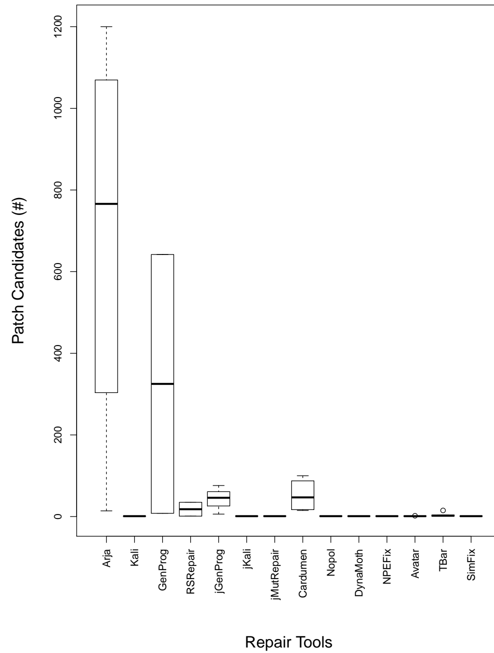


Fig. 6. **RQ2:** Distribution of the NPC scores.

Execution time. Table 10 lists the median and mean execution time for the repair attempts per repair tool. Figure 4 illustrates the differences in the execution time among the different repair tools by depicting the minimum, maximum, sample median, and first and third quarterlies values.

Figure 5 graphically shows the overall trend of the repair attempts. The graph considers the total time of each repair attempt. In particular, 55% of the repair attempts take less than ten minutes to execute. By contrast, 15% of the repair attempts from all repair tools examined take $\geqslant$ 120 minutes to execute. The uptick at the end of the graph includes all these repair attempts that would have taken more than two hours if we had not set a timeout, which in our case is set to 120 and 240 minutes depending on the tool (see Section 3.5).

Overall, one could say that the earlier 11 tools are fast since their median execution time for a repair attempt is 3.87 minutes (see Table 10), which is significantly lower than the timeout (two hours) set for these tools. By contrast, the three more recent tools, AVATAR, TBAR, and SIMFIX, are the slowest since they mostly take more than two hours to run.

In particular, JGENPROG and ARJA perform much slower than the remaining repair tools evaluated on the whole APIREPBENCH. This may happen due to the technical characteristics of the tools. Specifically, JGENPROG is quite slow having 66.18 minutes median repair time. This comes in accordance with a related study by Martinez et al. [75], and possibly happens because the search space of JGENPROG is extremely large. Additionally, ARJA uses a multi-objective approach [13], and executes the whole test suite of each analysed project revision that could increase the execution overhead of the tool. According to the study of Yuan and Banzhaf [13], which evaluates ARJA-based tools on DEFECTS4J, a successful repair attempt for ARJA lasts maximum one hour. We find that the median repair time for ARJA is 47.87 minutes. However, we run ARJA on different types of bugs and projects. Also, we consider the time of all ARJA repair attempts. Finally, DYNAMOTH is the fastest tool possibly because its analysis is focused on specific test case(s) given by the user and based on dynamic exploration [27]. As it has been shown from the related work, when trimming the search space using focused testing, the analysis becomes significantly faster [54], [74]. Finally, AVATAR, TBAR, and SIMFIX are slower than the remaining tools. One possible reason for that is that these tools run the whole test suite against each patch candidate.

Number of patch candidates. Figure 6 shows the distribution of the NPC scores among the repair tools under examination. ARJA generates the highest number of patches until it generates the first valid (plausible) patch. GENPROG, RSREPAIR, JGENPROG, CARDUMEN AVATAR, and TBAR also generate more than one patch, whereas the remaining tools generate mostly one patch each time, and are faster.

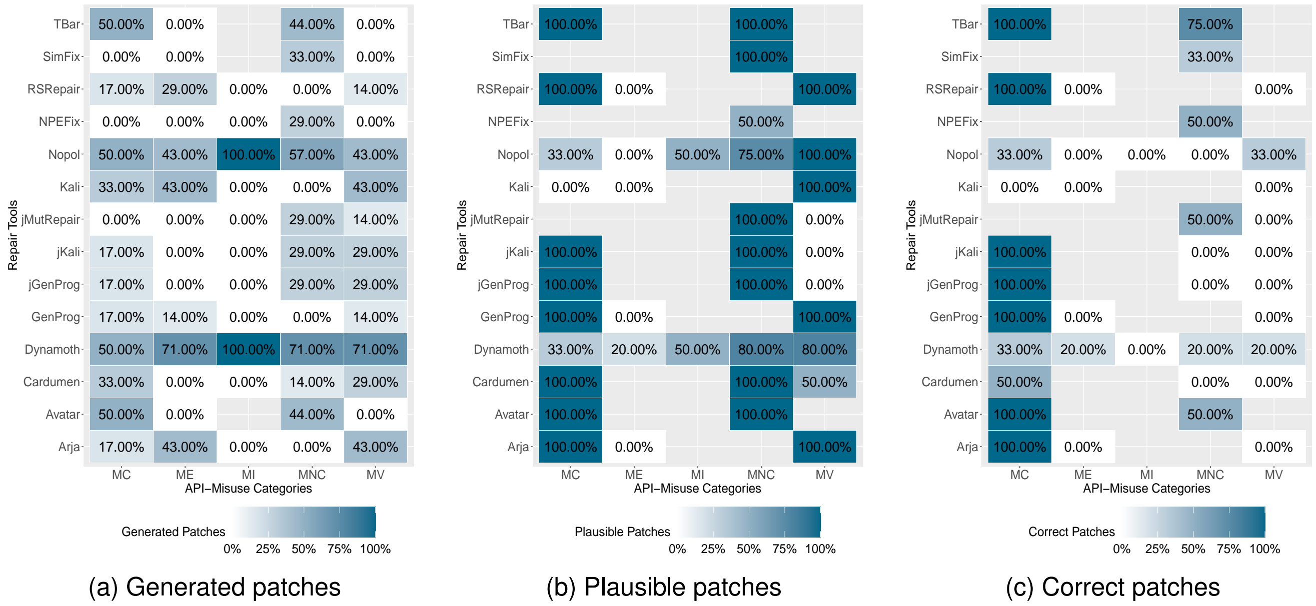
Both Figure 4 (execution time) and Figure 6 (NPC) indeed agree that ARJA, GENPROG, RSREPAIR, JGENPROG, and CARDUMEN are the slowest tools, whereas KALI, NOPOL, DYNAMOTH, and NPEFIX are the fastest tools. However, in Figure 4, JKALI and JMUTREPAIR seem to perform slower, whereas in Figure 6 perform faster. There are such divergences between the two figures since in Figure 4 we consider all repair attempts, whereas in Figure 6 only the repair attempts of valid patches. Furthermore, AVATAR, TBAR, and SIMFIX are among the slowest tools in Figure 4 and among the fastest tools in Figure 6. This difference possibly occurs because we run these tools on a small data set. Additional experiments on API misuses could further show the efficiency of AVATAR, TBAR, and SIMFIX using NPC.

Answer to RQ2. The 11 less recent repair tools have a median execution time of 3.87 for a repair attempt targeting an API misuse and a mean execution time of 30.79 minutes. The three more recent tools are the slowest overall, exhibiting a much higher median execution time (98% slower) than the earlier ones. Among the earlier tools, JGENPROG, ARJA, and CARDUMEN are the slowest tools based on execution time and NPC. Whereas NOPOL, NPEFIX, and DYNAMOTH are the fastest ones.

4.4 RQ3: For which classes of API misuses do repair tools perform best?

To answer RQ3, we examine the performance (effectiveness and efficiency) of the repair tools per API-misuse category. To measure the effectiveness, we calculate (per API-misuse category) the number of: 1) the generated patches, 2) the generated patches that are plausible, and 3) the generated patches that are semantically correct to the human patches. To measure the efficiency, we calculate the NPC and the execution time each repair attempt takes per API-misuse category. Figure 7 summarises, in heat maps, our results on the effectiveness of the tools. Figure 8, Figure 9, and Figure 10 give further information about the efficiency of the tools.

Figure 7a shows that the repair tools mostly generate patches for the missing call (MC), missing `null` check



(a) Generated patches (b) Plausible patches (c) Correct patches

Fig. 7. **RQ3:** Repair tools, API-misuse categories, and patches. For instance, DYNAMOTH has generated 71% patches that belong to the ME category. From these patches, 20% are plausible and 20% are semantically correct. The results for AVATAR, TBAR, and SIMFIX refer to the categories of the DEFECTS4J projects examined (see Section 3.5).

(MNC), missing value (MV), and missing exception (ME) API-misuse categories; no tool has generated any patches for the missing context (MX) API-misuse category. This comes in accordance with our initial expectations as the tools that could repair these categories of API misuses are designed to fix the aforementioned types of bugs. Furthermore, NOPOL-based tools generate patches for all the considered API-misuse categories, except for MX. ARJA-based tools generate patches for the MC, ME, and MV API-misuse categories. ASTOR-based tools generate patches for the MC, MNC, and MV API-misuse categories. NPEFIX generates patches only for the MNC category, as expected, since NPEFIX is designed to only repair such bugs. Finally, AVATAR, TBAR, and SIMFIX generate patches only for MC and MNC.

Figure 7b shows that most of the patches generated by the repair tools are plausible. We observe that for the ME category, only DYNAMOTH generates plausible patches, whereas, for the MC category, only KALI does not generate any plausible patch. Additionally, for the MV category, all tools generate plausible patches except for JGENPROG, JKALI, and JMUTREPAIR. Overall, only DYNAMOTH generates plausible patches for all categories. Furthermore, CARDUMEN generates all patches as plausible for the MC and MNC categories, and half patches as plausible for the MV category. NPEFIX also generates half of the generated patches for the MNC category as plausible. Finally, AVATAR, TBAR, and SIMFIX generate only plausible patches.

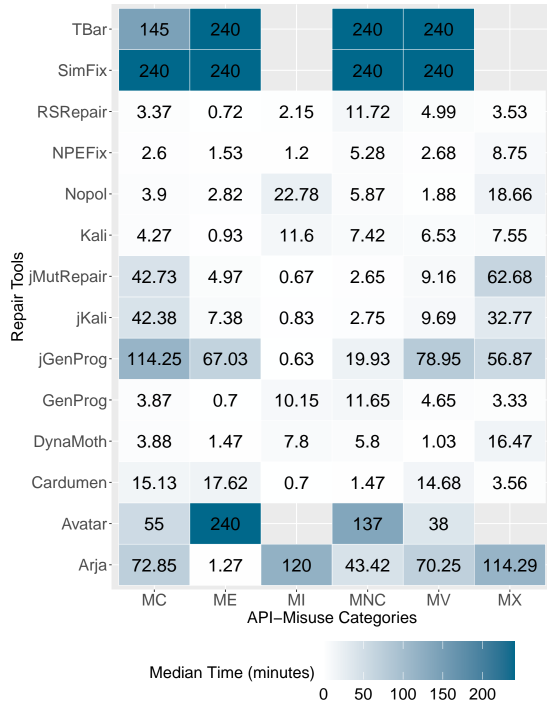
Figure 7c reveals that most of the generated patches are *not* semantically correct. This, with previous studies on overfitting, argues that only a few generated patches by program repair are correct [71], [77]. Regarding the categories where the tools generate most of the patches (i.e., MC, MNC, MV, and ME), the tools generate the highest number of semantically correct patches for the MC category. Furthermore, only DYNAMOTH generates semantically correct patches for all categories except for MI. Additionally, AVATAR, TBAR, and SIMFIX mostly generate patches (for MC and MNC) that are plausible and correct ($\geqslant$ 30%).

Since the repair tools examined are not originally designed to fix API misuses, we expected that it would be more challenging for a repair tool to generate correct patches to fix misuses of missing API method calls rather than, for instance, misuses due to incorrect values in `if` conditions. However, we observe that the MC category exhibits the highest number of semantically correct patches. This may happen for the following reasons. First, repair tools are designed to repair bugs that generally need a small change (for example done via mutation analysis in just one statement, one token, or one line) to be fixed, and such bugs could belong to the MC category. Therefore, the tools would be able to fix those MC bugs that require a single-line change. Consider CAR-188[18] and JFREE-1[19] that more than one repair tool is able correctly fix them. Second, one should consider that the percentages of semantically correct patches in Figure 7c depend on the number of generated patches in Figure 7a. For instance, even though RSREPAIR, GENPROG, ARJA can repair a few API misuses of the MC category (17%) presented in Figure 7a, all generated patches are semantically correct. Therefore, these tools achieve a high correctness score (100%) in Figure 7c.

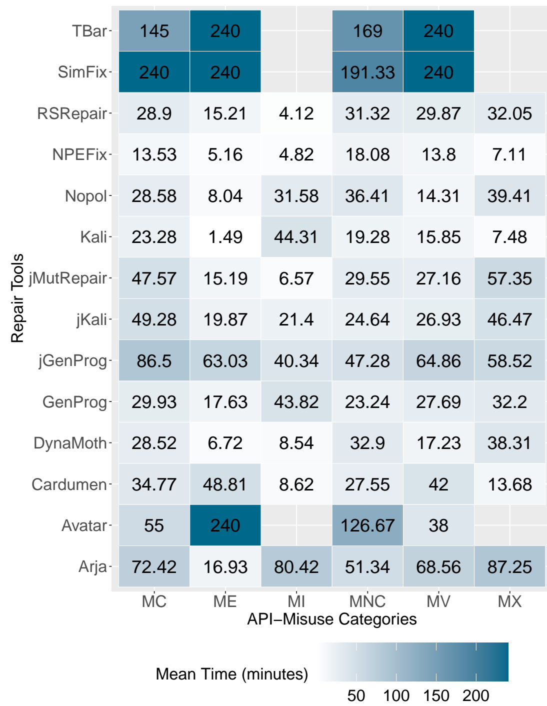
Figure 8 illustrates the median and mean time (in minutes) that the tools spent to repair API misuses belonging to the API-misuse categories. For the categories where the tools generate most of the patches (i.e., MC, MNC, MV, and ME), the repair tools perform as follows (see Figure 8a). ARJA and JGENPROG spend most time to repair API misuses

18. https://github.com/SzFMV2018-Tavasz/AutomatedCar/commit/6a656ea5686fa563e85754282f059264f3664471

19. https://sourceforge.net/p/jfreechart/code/1025/tree/trunk/source/org/jfree/chart/util/ShapeUtilities.java?diff=50b53b485fcbc92b6542a639:1024



(a) Median time



(b) Mean time

Fig. 8. **RQ3:** Repair tools, API-misuse categories, and execution time. For instance, ARJA needed 72.85 median time and 72.42 mean time (in minutes) to repair API-misuses of the MC category. The results for AVATAR, TBAR, and SIMFIX refer to the categories of the DEFECTS4J projects examined (see Section 3.5). For these three tools the timeout is four hours and for the rest two hours.

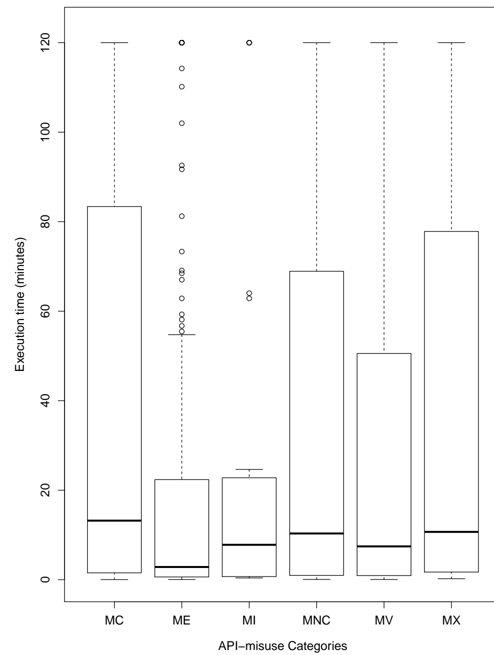


Fig. 9. **RQ3:** Execution time for all tools per API-misuse category.

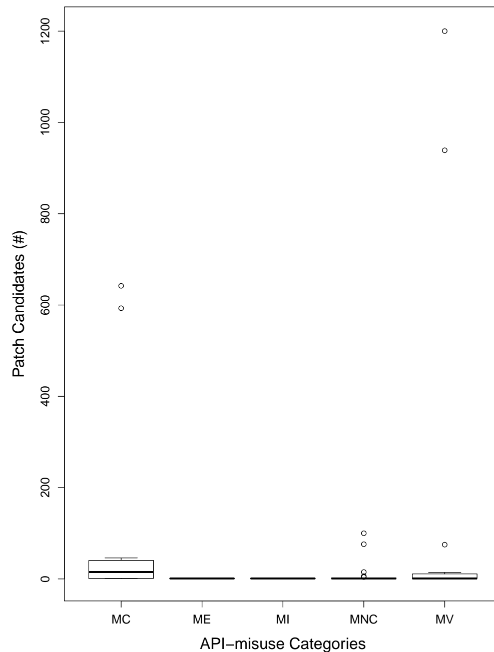


Fig. 10. **RQ3:** NPC scores for all tools per API-misuse category.

of all categories. CARDUMEN spend a significant amount of time to repair API misuses of the ME, MC, and MV categories. Additionally, JMUTREPAIR and JKALI spend a lot of time for the MC and MX categories, whereas RSREPAIR and GENPROG for the MNC category. All tools, except for ASTOR-based tools spend the least time on repairing API misuses of ME. AVATAR, TBAR, and SIMFIX spend the most time for all categories examined.

Figure 9 presents the execution time of all repair attempts grouped by API-misuse category. To be able to unify our results from AVATAR, TBAR, SIMFIX, and the remaining tools, we group all repair attempts that take more than 120 minutes into a single category, where we assume that these attempts spend 120 minutes. For the MC, MNC, and MX categories, the repair tools spend most of the time (more than ten minutes median time), whereas repair attempts take the least time for the ME category (2.84 minutes median time). The reason why MNC has high median repair time is that AVATAR, TBAR, and SIMFIX (the slowest tools according to Section 4.3) mostly repair API misuses that belong to the MNC category, and can affect the results. Furthermore, tools that are not originally designed to repair API misuses that belong to the MC and MX categories could spend more time to repair these bugs. The tools spend the least time to repair bugs belonging to the ME category, because possibly these misuses throw specific exceptions that fault localisation can identify fast. Note that even if most repair tools are designed to repair bugs that belong to the MV and MNC categories, the tools spend a significant amount of time (more than seven minutes median time) to repair these categories.

Figure 10 shows the NPC scores per API-misuse category. We observe that the highest number of patches generated until the generation of a valid patch happens for the MC and MV categories. However, this finding should be further investigated in future work since NPC could be affected by the high number of patches generated by ARJA, GENPROG, JGENPROG, and CARDUMEN. Finally, both Figure 9 and Figure 10 agree that tools take the most time to generate patches for the MC category.

Answer to RQ3. The repair tools mostly generate patches for the MNC, MV, ME, and MC API-misuse categories. For all categories, the majority of the patches are plausible but not semantically correct. Most semantically correct patches exist for the MC and MNC categories. Overall, the repair tools spend more time on repairing API misuses that belong to the MC, MNC and MX categories. The highest NPC refers to the MC and MV categories.

4.5 Discussion

From our investigation, we show that state-of-the-art Java test-suite-based repair tools need to be improved in order to be able to fix API misuses. In the following paragraphs, we discuss the main challenges and opportunities we found from our study for API-repair tools.

Focused design. Our study shows that only for 28% API misuses the repair tools can generate patches. This may happen for the following reasons. First, this limitation may depend on the *design* of the tools. The tools are designed to repair only specific types of bugs, including missing `null` check (MNC) and missing value (MV). For instance, tools that are not designed to repair missing exception (ME) issues could not generate relevant patches. Namely, such a patch does not exist in the search space [55]. Second, this limitation could be related to *space exploration* issues. This means that, although the solution does reside in the search space, it remains unfound [78]. Third, this limitation should derive from the *configuration* of the tools. As it is, also, argued by Derieux et al. [9], possibly the tools cannot generate a patch during a predefined timeout. Finally, according to Le Goues et al. [54], *project characteristics* can affect the effectiveness of the repair tools. Complex software projects with large size, many sub projects and dependencies require more time to be analysed. Thus, repair tools can more easily reach the timeout without having generated any patch. Addressing these open questions, the program-repair community can make repair tools more precise and practical for researchers and practitioners.

Synergy. Our study reveals that most of the generated patches are plausible (65%). However, the majority of these patches are *not* correct (only 25% semantically correct patches found). This refers to the so-called *overfitting* issue of existing repair tools [77]. To address this issue, program repair could combine static analysis and machine learning techniques (e.g., based on historical data) [54]. Furthermore, more precise fault-detection approaches [55], [78] (e.g., focused search spaces), and further filtering of the results can lead to less false positives and human effort for the cross-checking of the generated patches.

Scalability. In our study, we observed that, overall, it takes time (Figure 5), for repair tools to analyse software projects, detect bugs, and generate repair patches. According to Table 10, for 11 tools the median time is almost four minutes and the mean time 31 minutes, per repair attempt. The remaining three tools require much more time (more than two hours median time).

There are several reasons for that. Initially, the efficiency of program repair possibly varies depending on the tools' *design*. For instance, ARJA-based tools take into account the whole test suite of a project to trigger specific bugs, whereas NOPOL-based tools use a more focused test-driven detection approach, as the interesting test(s) are given by the user as inputs. Therefore, it is expected for ARJA-tools to be slower than NOPOL-based tools (Figure 4). Additionally, the time the tools spend to repair API misuses can vary based on *targeted classes of bugs*, i.e., here, the categories of the API misuses (Figure 9). This possibly happens as each repair tool is designed to repair different bug categories. In particular, the repair attempts of NPEFIX, which targets MNC bugs, take less time when detecting and fixing API misuses of the MNC category than tools that target generic bugs (e.g., GENPROG). Other reasons we observed that can possibly affect the efficiency of repair tools include: the given *configuration* (e.g., timeout) of the tools and the capabilities of the *execution environment* (e.g., available Virtual Memory) [9], as well as the *project characteristics* of the analysed software [54]. As Marginean et al. [8] mention in their industrial study, with the SAPFIX repair tool, practitioners can be reluctant to use repair tools available in literature since the sophistication of these tools might have inhibited scalability. Therefore, solving performance issues, such as

the aforementioned, can further support the adoption of repair tools by practitioners.

Adequate documentation. From our study on repair tools, we observed that some repair tools lack publicly available documentation. Specifically, 25 out of 43 (58%) of the examined repair tools listed in Table 1 provide documentation. Since we were able to learn how to set up and use only repair tools with documentation, we set as one of the criteria for the selection of the 14 repair tools of our study the existence of publicly available documentation (Section 3.2). As previous studies have also focused on the importance of the documentation of software tools [9], we hope that future research on program repair will continue supporting the documentation of repair tools.

Uniform outputs. In our study, we observed that each repair framework, ASTOR, ARJA, and NOPOL (Table 4), gives outputs of distinct formats. The remaining tools, also, give outputs of their particular format. Therefore, we needed to put some effort on understanding the results from the tools of our study. One reason for developing APIARTY was, indeed, to automatically handle these different formats and provide summarised reports from all tools. Future research may investigate, as Brittany et al. [79] have shown for static analysis tools, whether guidelines on the presentation of repair tools' results can further support the use of repair tools by researchers and practitioners.

5 THREATS TO VALIDITY

This section presents the threats to validity of our study. We present the implementation issues that possibly exist in the design of our study (*internal validity*), issues regarding the generability of our results (*external validity*), and we argue about the reproducibility of our experiments (*reliability*).

Internal validity. We acknowledge that APIARTY is an artefact that cannot be free of bugs. This could affect the results of our study presented in Section 4. For instance, even though our scripts have been thoroughly tested, there could be cases where the tools could not properly run because of missing dependencies in the examined projects. This is, however, a common issue that our community needs to solve for studies that include large software projects [9].

Furthermore, since we have manually selected the API misuses of our benchmark, APIREPBENCH, the reader should be aware of any human error; although the first two authors have cross-checked the API misuses (Section 3.3.2). In future, the identification of API misuses in program repair can be automated by extending existing API-misuse detection tools such as MUDETECT [2], [69]. Adapting such an API-misuse detector for program repair, API-repair tools could achieve more precise fault localisation, and improve their effectiveness and efficiency by pruning the search space. Future studies could also examine whether different fault localisation approaches can be biased while benchmarking diverse repair tools [58].

Also, since the performance of the repair tools significantly depends on the execution environment (e.g., available Virtual Memory, used JDK)[20] the experiments run on, the results and execution time of our experiments may differ from platform to platform.

Additionally, due to the nature and design of tools such as GENPROG (which is based on genetic programming) and RSREPAIR (which is based on random search) the results of these tools can vary among different executions.

Finally, the reader should consider that the number of the found patches as semantically correct depends on the authors best understanding of the bugs and the relevant fixes. However, this is a common threat in overfitting studies that manually check the correctness of the generated patches [75], [80]. This threat also applies to the manual classification of the reports produced by the tools into non-patch generation cause categories.

External validity. Regarding the generability of our findings (Section 4), we acknowledge that our results regard the specific API misuses of our benchmark, APIREPBENCH. However, we argue, that APIREPBENCH is made of other three benchmarks that include well-diversified and well-known software projects as suggested by Durieux et al. [9]. Thus, we expect that our conclusions can also apply to other sets of API misuses.

Reliability. For the reproducibility of our study, we have developed an extensible execution framework, APIARTY, and we have made its source code, as well as our benchmark, APIREPBENCH, publicly available. We also provide the metadata of the bugs used in our experiments (*input*) and the found patches (*output*).

6 RELATED WORK

In this section, we present related work to our study. We discuss existing program repair approaches, research on detecting API misuses, and available bug benchmarks for evaluating repair tools.

6.1 Test-driven Program Repair Evaluation

Among the first research works on program repair, Le Goues et al. [7] introduced and evaluated GENPROG on 105 defects from eight open source programs. Additionally, Le Goues et al. [76] evaluated three repair tools for the C programming language using the MANYBUGS and INTROCLASS benchmarks. Both studies refer to the repairability (patch generation effectiveness) of repair tools. Qi et al. [14] first evaluated repair tools, regarding the test-suite adequacy and correctness of generated patches, showing that repair tools produce only a small number of correct patches that can fix software bugs. Smith et al. [77] introduced the notion of *overfitting* and compared two C repair tools using the INTROCLASS benchmark [76]. They found that even good test suites can lead to patch overfitting. Qi et al. [15] developed a repair tool (RSREPAIR) for C programs based on a random search approach, and they revealed that random search repair approaches can be effective and efficient over program repair that relies on genetic programming.

On the Java front, Martinez et al. empirically compared three repair tools using the DEFECTS4J benchmark [71] to evaluate the patch correctness and efficiency of the tools [75]. Ye et al. [80] empirically compared the patch correctness of five generate-and-validate repair tools, using the QUIXBUGS benchmark [81]. Both benchmarks have

20. https://github.com/program-repair/RepairThemAll_experiment/issues/4

TABLE 11
Empirical Evaluations Among Test-suite-based Repair Tools

Work	Programming Language	Tools (#)	Benchmarks (#)	Patch Generation	Patch Plausibility	Patch Correctness	Benchmark Overffiting	Bug Class Overfitting	Repair Efficiency
Goues et al. [76]	C	3	2	✓	✗	✗	✗	✗	✓
Qi et al. [15]	C	2	1	✓	✗	✗	✗	✗	✓
Smith et al. [77]	C	2	1	✓	✗	✓	✗	✗	✗
Qi et al. [14]	C	4	1	✓	✓	✓	✗	✗	✓
Motwani et al. [65]	C, Java	9	2	✓	✗	✗	✗	✗	✗
Martinez et al. [75]	Java	3	1	✓	✓	✓	✗	✗	✓
Xiong [25]	Java	5	1	✓	✗	✓	✗	✗	✗
Ye et al. [80]	Java	5	1	✓	✓	✓	✗	✗	✓
Hua et al. [23]	Java	5	1	✓	✓	✓	✗	✗	✓
Yuan, Banzhaf [13]	Java	7	1	✓	✗	✓	✗	✗	✓
Liu et al. [55]	Java	14	1	✓	✓	✓	✗	✗	✗
Liu et al. [10]	Java	16	1	✓	✓	✓	✗	✗	✓
Durieux et al. [9]	Java	11	5	✓	✗	✗	✓	✗	✓
This study	Java	14	3	✓	✓	✓	✗	✓	✓

highlighted the main challenges on the assessment of the correctness of generated patches. Furthermore, Lui et al. [55] compared 14 repair tools using the DEFECTS4J bug benchmark, and revealed that imprecision in the fault-localisation step of program repair can lead to overfitting issues.

Additionally, several papers that introduce new repair approaches compare novel techniques to the state-of-the-art [13], [23], [25]. Recently, Durieux et al. conducted the first large-scale empirical study on evaluating whether the repairability of repair tools can be generalised over different bug benchmarks [9]. Furthermore, Motwani et al. compared nine automated repair techniques using the MANYBUGS and DEFECTS4J benchmarks to show whether such tools can repair hard and important bugs [65]. Finally, Lui et al. first presented an empirical study that evaluates the efficiency of 16 state-of-the-art repair tools using the DEFECTS4J bug benchmark, and defined specific correctness criteria and a new efficiency measure (i.e., NPC), which we have adopted in this study [10].

Many empirical evaluations have been conducted on the effectiveness and efficiency of repair tools using bug benchmarks. Table 11 classifies these studies based on particular evaluated parameters. Our study differs from the previous studies in two primary ways: 1) it considers a previously unexplored yet important class of bugs: API misuses, and 2) it draws conclusions on multiple (three) and large benchmarks for evaluating the performance (patch plausibility, correctness, and repair efficiency) of repair tools.

6.2 API-Misuse Detectors

There are several available API-misuse detectors, static, dynamic, and hybrid. Amann et al. recently published the first systematic evaluation of static API-misuse detectors [2]. The main limitations of static detectors include the generation of a high rate of false positives and the need for manual assessment and confirmation of the discovered misuses. To overcome the issues of static detectors, recent API-misuse detection approaches combine both static and dynamic analysis to increase their precision and performance, and eliminate developers' workload. Specifically, Wen et al. developed MUTAPI that uses mutation analysis to expose API misuses [82] and Kechagia et al. introduced a novel approach, CATCHER, that combines static exception propagation and search-based test case generation to effectively and efficiently identify API misuses [74]. Additionally, there are only a few research-oriented tools that can automatically repair specific types of API misuses. For instance, consider those that repair incorrect error handling [26], [83], [84].

Additionally, consider program synthesis approaches that assist developers to avoid errors in sequences of API method calls. Characteristically, Yang et al. introduced EDSYNTH that synthesises API sequences with conditionals and loops [58]. Feng et al. presented an approach for component-based synthesis for complex APIs [57]. However, these approaches have been currently evaluated only on small programs.

6.3 Bug Benchmarks

Researchers have created benchmarks of real-world bugs to evaluate the performance of novel methods and tools. Just et al. introduced a database (DEFECTS4J) of programs, bugs, and test suites extracted from well-known Java projects [71]. Recently, the BEARS [11] and BUG.JAR [12] Java bug benchmarks were produced for the evaluation of repair tools. Le Goues et al. developed benchmarks of C bugs (MANYBUGS and INTROCLASS) for the evaluation of repair tools [76]. Lin et al. published the QUIXBUGS benchmark suite for Python and Java [81]. Recently, Amann et al. introduced a benchmark of Java API misuses, MUBENCH [69].

7 CONCLUSION

Several studies have compared repair tools, using bug benchmarks, and argued that these tools suffer from imprecision and can generate patches for only a few different types of bugs. To advance program repair further, recent work shows that repair tools should be evaluated based on different bug benchmarks and on the types of bugs that the tools are designed to work for. Our work presents the first large-scale empirical study of repair tools concerning their ability to repair an unexplored class of bugs, API misuses.

Our study includes 14 state-of-the-art Java test-suite-based repair tools and a bug benchmark for program repair (APIREPBENCH) that comprises 101 API misuses stemming from three bug benchmarks (BEARS, BUGS.JAR, and

MUBENCH). For our experiments, we develop an execution framework (APIARTY) that helps in the automatic execution of multiple repair tools.

Our results show that all repair tools can generate patches for 28% API misuses in our benchmark. The repair attempts of the 11 earlier tools take on average 3.87 minutes (median execution time) and 30.79 minutes (mean execution time). The three recent tools take significantly more time (more than two hours on average) to run. The tools generate patches for API misuses that mostly belong to the categories of missing `null` check, missing value, missing exception, and missing call. Most of the patches generated by all tools are plausible (65%), but only few of the generated patches are semantically correct to human patches (25%). Most correct patches belong to the missing call, missing `null` check, and missing value categories. We also observe that recent tools are more effective than their predecessors in repairing the APIREPBENCH-D4J misuses, with AVATAR and TBAR to have more than 60% plausible and semantically correct patches generated.

Future research can explore building repair tools able to generate patches for API misuses by focusing the search space and targeting specific classes of API misuses. Another aspect that could be tackled in future studies is the prevention of overfitting while repairing API misuses. Finally, the development of additional bug benchmarks can help us to further study and address the challenges of API repair.

ACKNOWLEDGMENTS

This work is supported by the ERC Advanced fellowship grant no. 741278 (EPIC). The authors would like to thank Dr. Matias Martinez for responding promptly to our questions about using ASTOR.

REFERENCES

[1] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019.

[3] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 204–215.

[4] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIs?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 935–946.

[5] A. Reinking and R. Piskac, "A type-directed approach to program repair," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 511–517.

[6] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 234–242.

[7] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.

[8] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 269–278.

[9] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," ser. FSE, 2019.

[10] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the efficiency of test suite based program repair," in *Proceedings of ICSE*, 2020.

[11] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019, pp. 468–478. [Online]. Available: https://arxiv.org/abs/1901.06024

[12] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.Jar: A large-scale, diverse dataset of real-world Java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. ACM, 2018, pp. 10–13.

[13] Y. Yuan and W. Banzhaf, "ARJA: Automated repair of Java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[14] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 24–36.

[15] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 254–265.

[16] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 65–86.

[17] ——, "Astor: Exploring the design space of generate-and-validate program repair beyond GenProg," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019.

[18] ——, "ASTOR: A program repair library for Java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 441–444.

[19] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: Fixing semantic bugs with fix patterns of static analysis violations," *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1–12, 2018.

[20] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, pp. 1–11.

[21] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "IFixR: Bug report driven program repair," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019.

[22] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. ACM, 2018, pp. 298–309.

[23] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, pp. 12–23.

[24] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. ACM, 2019, pp. 31—-42.

[25] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE. IEEE Press, 2017, pp. 416–426.

[26] A. Dhar, R. Purandare, M. Dhawan, and S. Rangaswamy, "CLOTHO: Saving programs from malformed strings and incorrect string-handling," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 555–566.

[27] T. Durieux and M. Monperrus, “DynaMoth: Dynamic code synthesis for automatic program repair,” in *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*, 2016, pp. 85–91.

[28] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, “Fault localization and repair for Java runtime exceptions,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA. ACM, 2009, pp. 153—-164.

[29] R. van Tonder and C. L. Goues, “Static automated program repair for heap properties,” in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 151–162.

[30] L. Chen, Y. Pei, and C. A. Furia, “Contract-based program repair without the contracts,” in *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 637–647.

[31] P. van der Spek, N. Plat, and C. Pronk, “Syntax error repair for a Java-based parser generator,” *SIGPLAN Not.*, vol. 40, no. 4, pp. 47–50, 2005.

[32] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, “JFIX: Semantics-based repair of Java programs via symbolic pathfinder,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 376—-379.

[33] R. Alur, P. Černý, P. Madhusudan, and W. Nam, “Synthesis of interface specifications for Java classes,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’05. ACM, 2005, p. 98–109.

[34] W. Wang, Z. Meng, Z. Wang, S. Liu, and J. Hao, “LoopFix: an approach to automatic repair of buggy loops,” *Journal of Systems and Software*, vol. 156, pp. 100–112, 2019.

[35] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. L. Berre, and M., “Nopol: Automatic repair of conditional statement bugs in Java programs,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[36] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, “Dynamic patch generation for null pointer exceptions using metaprogramming,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 349–358.

[37] G. Liva, M. T. Khan, M. Pinzger, F. Spegni, and L. Spalazzi, “Automatic repair of timestamp comparisons,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[38] X. Xu, Y. Sui, H. Yan, and J. Xue, “VFix: Value-flow-guided precise program repair for Null pointer dereferences,” in *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 512–523.

[39] J. Kim, J. Kim, E. Lee, and S. Kim, “The effectiveness of context-based change application on automatic program repair,” *Empirical Software Engineering*, vol. 25, no. 1, pp. 719–754, 2020.

[40] J. Kim and S. Kim, “Automatic patch generation with context-based change application,” *Empirical Software Engineering*, vol. 24, no. 6, pp. 4071–4106, 2019.

[41] R. K. Saha, H. Yoshida, M. R. Prasad, S. Tokumoto, K. Takayama, and I. Nanba, “Elixir: An automated repair tool for java programs,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE. ACM, 2018, pp. 77–80.

[42] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, “FixMiner: Mining relevant fix patterns for automated program repair,” *Empirical Software Engineering*, pp. 1–45, 2020.

[43] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, “Inferring program transformations from singular examples via big code,” in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 255–266.

[44] X. B. D. Le, D. Lo, and C. L. Goues, “History driven program repair,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 213–224.

[45] N. Meng, M. Kim, and K. S. McKinley, “Lase: Locating and applying systematic edits by learning from examples,” in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 502–511.

[46] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, “LSRepair: Live search of fix ingredients for automated program repair,” in *25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018, pp. 658–662.

[47] R. Bavishi, H. Yoshida, and M. R. Prasad, “Phoenix: Automated data-driven synthesis of repairs for static analysis violations,” in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, p. 613–624.

[48] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, “SEQUENCER: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[49] X. Liu and H. Zhong, “Mining StackOverflow for program repair,” in *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 118–129.

[50] Q. Xin and S. P. Reiss, “Leveraging syntax-related code for automated program repair,” in *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 660–670.

[51] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, “VuRLE: Automatic vulnerability detection and repair by learning from examples,” in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 229–246.

[52] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[53] M. Monperrus, “Automatic software repair: A bibliography,” *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, 2018.

[54] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Commun. ACM*, vol. 62, no. 12, pp. 56—-65, 2019.

[55] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, “You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems,” in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 102–113.

[56] R. P. L. Buse and W. Weimer, “Synthesizing API usage examples,” in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 782–792.

[57] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, “Component-based synthesis for complex APIs,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. ACM, 2017, pp. 599–612.

[58] Z. Yang, J. Hua, K. Wang, and S. Khurshid, “EdSynth: Synthesizing API sequences with conditionals and loops,” in *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 161–171.

[59] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, “T2API: Synthesizing API code usage templates from english texts with statistical translation,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2016, p. 1013–1017.

[60] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374.

[61] M. P. Robillard and R. Deline, “A field study of API learning obstacles,” *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[62] M. Kechagia, D. Mitropoulos, and D. Spinellis, “Charting the API minefield using software telemetry data,” *Empirical Software Engineering*, vol. 20, no. 6, pp. 1785–1830, 2015.

[63] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, “You get where you’re looking for: The impact of information sources on code security,” in *IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289–305.

[64] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, “Are code examples on an online Q&A forum reliable?: A study of API misuse on Stack Overflow,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[65] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, “Do automated program repair techniques repair hard and important bugs?” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2901–2947, 2018.

[66] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *ICSE*. IEEE, 2009, pp. 364–374.

[67] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, “A genetic programming approach to automated software repair,” in *Genetic and Evolutionary Computation Conference*, F. Rothlauf, Ed. ACM, 2009, pp. 947–954.

[68] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, “The plastic surgery hypothesis,” in *Proceedings of the 22nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, (FSE). ACM, 2014, pp. 306–317.

[69] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, “MUBench: A benchmark for API-misuse detectors,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 464–467.

[70] K. Herzig, S. Just, and A. Zeller, “It’s not a bug, it’s a feature: How misclassification impacts bug prediction,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 392–401.

[71] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, 2014, pp. 437–440.

[72] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, “Fixing recurring crash bugs via analyzing Q&A sites (T),” in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 307–318.

[73] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot: A Java bytecode optimization framework,” in *CASCON First Decade High Impact Papers*. Riverton, NJ, USA: IBM Corp., 2010, pp. 214–224.

[74] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, “Effective and efficient API misuse detection via exception propagation and search-based testing,” ser. ISSTA ’19, 2019.

[75] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. , “Automatic repair of real bugs in Java: a large-scale experiment on the Defects4J dataset,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[76] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The ManyBugs and IntroClass benchmarks for automated repair of C programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[77] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? overfitting in automated program repair,” in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE. ACM, 2015, pp. 532—-543.

[78] F. Long and M. Rinard, “An analysis of the search spaces for generate and validate patch generation systems,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE. ACM, 2016, pp. 702—-713.

[79] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681.

[80] H. Ye, M. Martinez, T. Durieux, and M. , “A comprehensive study of automatic program repair on the QuixBugs benchmark,” in *IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, 2019, pp. 1–10.

[81] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. ACM, 2017, pp. 55–56.

[82] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, “Exposing library API misuses via mutation analysis,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 866–877.

[83] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, “Repairing crashes in Android apps,” in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 187–198.

[84] Y. Tian and B. Ray, “Automatically diagnosing and repairing error handling bugs in C,” in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 752–762.

Maria Kechagia is a Research Fellow at University College London. Previously, she was a postdoctoral researcher at the Delft University of Technology. She obtained a PhD degree from the Athens University of Economics and Business and a MSc degree from Imperial College London. Her research interests include program analysis, software testing, automated program repair, and software analytics. She has been a programme committee member of ASE, MSR, and ICSME, and a reviewer for TSE, EMSE, JSS. Web page: http://www0.cs.ucl.ac.uk/staff/M.Kechagia/

Sergey Mechtaev is a Lecturer at University College London. Previously, he obtained a PhD degree from the National University of Singapore. His research interests include automated program repair, program synthesis and symbolic execution. He has received ACM SIGSOFT Outstanding Dissertation Award for his PhD thesis on semantic program repair. He has been a programme committee member of ASE, and a reviewer of TSE, TOSEM and EMSE. Web page: http://mechtaev.com

Federica Sarro is a Professor of Software Engineering at University College London. Her research covers Predictive Analytics for Software Engineering (SE), Empirical SE and Search-Based SE. On these topics she has co-authored over 80 papers and has also received several international awards including the FSE’19 ACM Distinguished Paper Award. She is an active member of the SE community: She has served on several steering, organisation, programme committees, and has been awarded several service awards including the ACM Distinguished Reviewer Award at ICSE’18 and ICSE’20. Web page: http://www0.cs.ucl.ac.uk/staff/F.Sarro/

Mark Harman works full time at Facebook London as a Research Scientist in a team focussing in AI for scalable software engineering. He also holds a part-time professorship at UCL. Previously, Mark was the manager of the Facebook team that deployed Sapienz to test mobile apps, which grew out of Majicke, a start up co-founded by Mark and acquired by Facebook in 2017. In his more purely scientific work, Mark co-founded the field Search Based Software Engineering (SBSE), and is also known for scientific research on source code analysis, software testing, app store analysis and empirical software engineering. He received the IEEE Harlan Mills Award and the ACM Outstanding Research Award in 2019 for this work. In addition to Facebook itself, Mark’s scientific work is also supported by the European Research Council (ERC), with an advanced fellowship grant, and has also been regularly and generously supported by the UK Engineering and Physical Sciences Research Council (EPSRC), with regular grants, a platform and a programme grant. Web page: http://www0.cs.ucl.ac.uk/staff/M.Harman/