Empirical Software Engineering manuscript No.
(will be inserted by the editor)

APR4Vul: An empirical study of automatic program repair techniques on real-world Java vulnerabilities

Quang-Cuong Bui ·
Ranindya Paramitha · **Duc-Ly Vu** ·
Fabio Massacci · **Riccardo Scandariato**

Received: date / Accepted: date

Abstract Security vulnerability fixes could be a promising research avenue for Automated Program Repair (APR) techniques. In recent years, APR tools have been thoroughly developed for fixing generic bugs. However, the area is still relatively unexplored when it comes to fixing security bugs or vulnerabilities.

In this paper, we evaluate nine state-of-the-art APR tools and one vulnerability-specific repair tool. In particular, we investigate their ability to generate patches for 79 real-world Java vulnerabilities in the Vul4J dataset, as well as the level of trustworthiness of these patches. We evaluate the tools with respect to their ability to generate security patches that are (i) testable, (ii) having the positive effect of closing the vulnerability, and (iii) not having side effects from a functional point of view. Our results show that the evaluated APR tools were able to generate testable patches for around 20% of the considered vulnerabilities. On average, nearly 73% of the testable patches indeed eliminate the vulnerabilities, but only 44% of them could actually fix security bugs while maintaining the functionalities.

Quang-Cuong Bui is the corresponding author.

This research was done while Duc-Ly Vu was with the University of Trento.

Quang-Cuong Bui
Hamburg University of Technology
E-mail: cuong.bui@tuhh.de

Ranindya Paramitha
University of Trento
E-mail: ranindya.paramitha@unitn.it

Duc-Ly Vu
University of Information Technology,
Vietnam National University, Ho Chi Minh City, Vietnam
E-mail: lyvd@uit.edu.vn

Fabio Massacci
University of Trento & Vrije Universiteit Amsterdam
E-mail: fabio.massacci@ieee.org

Riccardo Scandariato
Hamburg University of Technology
E-mail: riccardo.scandariato@tuhh.de

To understand the root cause of this phenomenon, we conduct a detailed comparative study of the general bug fix patterns in Defect4J and the vulnerability fix patterns in ExtraVul (which we extend from Vul4J). Our investigation shows that, although security patches are short in terms of lines of code, they contain unique characteristics in their fix patterns compared to general bugs. For example, many security fixes require adding method calls. These method calls contain specific input validation-related keywords, such as *encode*, *normalize*, and *trim*. In this regard, our study suggests that additional repair patterns should be implemented for existing APR tools to fix more types of security vulnerabilities.

Keywords Automated program repair · Java · Vulnerability · Empirical experiments

1 Introduction

Automated Program Repair (APR) techniques have been thoroughly studied in the past years [16]. These studies outline that APR tools struggle with fixes that require modifications to several parts of the software, as it becomes extremely difficult to even find the relevant lines of code to repair [60]. In this respect, security bug fixes seem to be a promising research avenue for APR. Several studies have now consistently shown that security fixes are typically very local, i.e., concentrated in a single method and typically consisting of a handful of lines [61, 13]. There might be architectural mistakes [72] which require major redesign, but they are the statistical outliers.

Automatic Vulnerability Repair (AVR) techniques could provide a much-needed solution to a pain point in the software industry, as studies have shown that developer tends to be less proactive in fixing the vulnerabilities themselves [63]. One of the reasons is that developers think it is difficult to find the root cause of security bugs [51]. Given the great potential, it is somewhat surprising that the area is relatively unexplored (see Section 3). Therefore, we started an investigation on one of the manually curated datasets of reproducible vulnerabilities [8] and used the wealth of APR tools that have been previously compared by Durieux et al. on generic bugs [16].

Figure 1 illustrates some puzzling results of our first analysis. The figure shows the patches generated by both developers and APR tools for the vulnerability CVE-2018-1324[1] of the library Apache Common Compress. The vulnerability could cause an infinite loop execution, leading to a DDoS attack due to a possible integer overflow. In our experiments, almost all APR tools (ARJA, GenProg-A, jGenProg, jKali, RSRepair-A, TBar, etc.), with the exceptions of jMutRepair and SeqTrans, could provide a patch that is considered as a plausible patch in the APR literature, i.e., it passes all the tests, including additional security tests as well as additional regression tests. However, when compared with the human patch, those APR-generated patches are actually incorrect. In the APR literature, this phenomenon is called the *overfitting* problem [35]. These observations spurred us to investigate more thoroughly the application of APR tools to security vulnerabilities. Accordingly, our first research question is:

[1] `https://nvd.nist.gov/vuln/detail/CVE-2018-1324`

```
// ORIGINAL Vulnerable Code
for (int i = 0; i < this.rcount; i++)
  for (int j = 0; j < this.hashSize; j++) {
    // loop's body
}

// ------------------------------------------
// DEVELOPER's change
- for (int i = 0; i < this.rcount; i++) {
+ for (long i = 0; i < this.rcount; i++) {
    for (int j = 0; j < this.hashSize; j++) {

// ------------------------------------------
// ARJA's generated APR PATCH
- for (int i = 0; i < this.rcount; i++) {
-   for (int j = 0; j < this.hashSize; j++) {
-     // loop's body
- }

// ------------------------------------------
// Cardumen's generated APR PATCH
- for (int i = 0; i < this.rcount; i++) {
+ for (long i = 0; this.rcount == 0; i++) {
    for (int j = 0; j < this.hashSize; j++) {

// ------------------------------------------
// Kali-A's generated APR PATCH
+ if (true) return;
  for (int i = 0; i < this.rcount; i++) {
    for (int j = 0; j < this.hashSize; j++) {

// ------------------------------------------
// TBar's generated APR PATCH
- for (int i = 0; i < this.rcount; i++) {
+ for (long i = 0; i == this.rcount; i++) {
    for (int j = 0; j < this.hashSize; j++) {
```

There is no need for security experts to see that all patches of APR tools are technically plausible in some APR evaluation studies (e.g., passing all test cases). However, when comparing with the human patch, those fixes are badly wrong.

Fig. 1 Developer's patch and APR's patches for Integer Overflow vulnerability in Apache Common Compress.

RQ1: *Can the existing APR tools generate end-to-end (E2E) tested patches in the context of real-world vulnerabilities?*

In Section 2), we present a classification of security patches, which includes a precise definition of E2E tested patches. In a nutshell, in the industrial setting, a patch must be eventually deployed without causing errors in the build process. Hence, in this research question, we accept the output of an APR tool as a feasible artifact for the *beginning of the analysis* when it is fully tested according to the project's own test suite. We remark that this paper is the first study investigating the test-based repair tools for the Java language in the context of software vulnerabilities. Our study complements the findings of Pinconschi et al. [64], which are obtained for test-based tools and the C language.

According to our definitions in Section 2, the tool-generated patches in Figure 1 are to be accepted in RQ1. The next step, therefore, is to climb in the hierarchy of what constitutes a right patch as a developer would interpret it:

RQ2: *How many of the generated patches are actually trustworthy?*

Our operative definition of 'trustworthy' is that the patches actually fix the vulnerability (positive effect), possibly without breaking the code functionality (negative side effect). To assess the trustworthiness of patches, we need to resort to *manual validation*. Our results show that purely automated criteria, as used in other vulnerability-centric studies by Pinconschi et al. [64], are insufficient. We even found some examples of 'rank inversion': the best-by-far tool in terms of automated criteria is bested by other tools when non-trustworthy patches are discounted.

We also remark that the detailed manual scrutiny of the patches in order to assess the presence of a positive security effect and the absence of other functional negative side effects is used in the field of security for the first time. To our knowledge, the work by Kechagia et al. [31] is the only APR study for security-related bugs that takes account of the correctness of the generated patches. However, in their study, the security-wise and functionality-wise correctness metrics are not assessed separately.

From the end-user perspective, the security-wise correctness of the patches is a very desirable property, as this implies that no security experience from the human developer is necessary. However, a security-fixing patch that has some functional side effects might also be desirable, as it is generally easier to tweak a fixing patch from a functional perspective rather than devising a security fix [21].

To understand the root causes of poor trustworthiness, we perform a detailed comparative study of the bug vs. vulnerability fix patterns:

RQ3: *Are the repair patterns used by humans to fix security vulnerabilities different w.r.t. fixing generic software bugs?*

This investigation gives insights into whether more or different repair patterns need to be added to the APR tools. In the (somewhat limited) scope of ExtraVul and Defects4J, we found that, while security fixing patterns are indeed short, they are not necessarily simple. Most importantly, the patterns do not necessarily resemble code lines present elsewhere in the code base. This is a major hurdle for any APR tool that bases its analysis exclusively on the code base to be fixed, thus opening up new research challenges. We also remark that this paper is the first to provide a list of empirically-identified vulnerability fix patterns for Java and a comparison of those with generic bug fix patterns.

Novelty and contribution. Platforms for automated analysis of test-based APR tools are well developed and, recently, have also been used for security vulnerabilities. For example, Durieux et al. [16] have proposed a toolchain for the automatic analysis of several APR tools in Java, Pinconschi et al. [64] have presented an evaluation of APR tools for C vulnerabilities, and Pinconschi et al. [65] have proposed a toolchain for the automatic execution of APR tools for vulnerabilities in C and Java. The key difference between traditional APR test suites

and AVR test suites is that the latter contains a special test case, called *Proof of Vulnerability (PoV)* test, which reveals the presence of a vulnerability in the program. A key issue of all the above-mentioned automated proposals is the strong assumption that the test suite is correct. Liu et al. [47] have already shown that, at a more careful semantic analysis, several APR outcomes pass all tests but are actually wrong. What we have shown in this paper is that the PoV-based method proposed by Pinchoschi et al. [64,65] and by Bui et al. [8] is not enough to guarantee semantic correctness of the fixes, even in the presence of a perfect selection of the bug location.

In summary, the paper has the following contributions:

- a methodology to capture and validate the trustworthiness of APR patches from a security standpoint.
- an empirical analysis, with extensive manual validation, of the (in)ability of test-based APR tools to generate trustworthy patches for Java security vulnerabilities. This analysis includes a discussion of the patch types that the tools are able to successfully (respectively, unsuccessfully) produce.
- a root cause analysis of the limitations found in the assessed tools, in terms of the fix patterns that are specific to security and could not be reproduced by the (generic bug-oriented) tools.

Outline. After introducing the terminology (Section 2) and a review of related works (Section 3), we present our methodology in Section 4. We further elaborate on the details of the dataset we used in Section 5, and present the evaluation results in Section 6, Section 7, and Section 8. We conclude this paper with the threats to validity (Section 9) and future work (Section 10). To support the openness in science, we have made the scripts and results of our evaluation publicly available [1].

2 Terminology

In the APR literature, a patch is considered plausible once it passes all tool internal tests in a given benchmark. Recently, more research has focused on the correctness of a patch, in this case, a patch is considered to be successful once it passes the human checks. In this study, we use a more granular classification that also considers the desired effect of the patch (i.e., fixing the vulnerability) and the absence of negative side effects (i.e., functionality being broken). Therefore, we classify the APR-generated patches as follows:

- **Generated patch**: The tool has used heuristics to produce a patch candidate and, according to the tool, the patch candidate has been vetted and has passed the tool's tests. Therefore, the tool considers the patch candidate as a successful patch and yields it at the end. Among the evaluated tools, SeqTrans does not have internal tests for itself, so we consider the predicted patches from SeqTrans's model which are *compilable* as the generated patches;
- **End-to-end (E2E) tested patch**[2]: The generated patches that actually pass all Proof of Vulnerability (PoV) and regression tests that are available for the

[2] Also known as "Plausible patch" in literature. Henceforward, we use "E2E tested patch" and "Plausible patch" interchangeably.

program. Note that the E2E tested patches do not break the build process of the projects they are applied to. In this study, we forced all the APR tools (except for SeqTrans) to use the whole test suite of the project as the validation test set for their patch candidates. Therefore, all the generated patches from these tools are E2E tested patches;
- **Security-Fixing patch**: The patch is an E2E tested patch that has been analyzed manually and in which the undesired behavior is eliminated (in our case, the vulnerability is no longer present), but some functionalities may be broken in an undetected way;
- **Correct patch**: The patch is a security-fixing patch that also preserves all the functionalities, as per the manual analysis;

In the choice of the terminology for E2E tested patches, we rely on the industrial classification of Continuous Development/Continuous Integration, e.g., by Atlassian [66]. In the test hierarchy, E2E tests are the link between tests for a subset of components (unit, functional, integration) and tests for business and deployment aspects (acceptance tests, performance) [66]. The latter cannot be available for a generic validation of APR tools as they are specific to an application. This classification describes an increasing level of patch quality, and each category is a subset of the previous one. These categories are further explained in Section 4.2.

3 Related Work

3.1 Automated Program Repair (APR)

To repair a buggy program, APR tools (particularly test-based ones) first try to localize potential faults (*fault localization*) and then apply mutations (*patch generation*) on the buggy locations until the program passes all the test cases. A patch is *plausible* if, when applied, it passes both positive and negative test cases.

According to Le Goues et al. [22], APR techniques can be grouped into three main categories, based on the way program issues are handled: Heuristic-based, Semantics-based, and Learning-based.

- **Heuristic-based**: this technique employs the *Generate-and-Validate* strategy, which iterates over a search space of syntactic edits to mutate the buggy program until it passes all the test cases. The repair strategy is often supervised by a genetic programming approach [38]. APR techniques that use templates for repair, such as TBar [45], can be considered heuristic-based approaches as they share the similarity [47].
- **Semantic-based**: this technique extracts semantic specifications from the test cases of the program, and attempts to synthesize valid program repairs that fulfill those specifications by leveraging the constraint-solving techniques, such as Z3 [14] and CVC4 [6].
- **Learning-based**: this technique learns the repair patterns with or without their contexts from previous bug fixes and then utilizes machine learning architectures (especially deep learning) to predict the patches. Recently, there have been many APR tools of this kind devoted to the community, such as DLFix [39], DEAR [40], CURE [28]. However, in our study, we decided not

to evaluate the tools in this category as they are generally not test-based and 'standalone' [47] APR tools.

Although current APR techniques differ in the way they generate a patch, they mainly rely on a set of *repair actions* and *repair patterns* [71] to generate patches:

- **Repair actions**: fine-grained mutations at the code component level such as assignment addition, modification, and deletion [71].
- **Repair patterns**: more abstract and semantic interpretation of a patch, such as *wraps-with* (when a patch wraps the original codes with conditional branches, try-catch, etc.), *wrong reference* (when a patch changes the variable or method reference of the program) [71].

Different APR techniques vary in how complex the repair actions and patterns they use. For instance, jKali [54] performs simple actions like removing one line of code, while Nopol [82] uses more complex patterns such as changing conditional expression.

Benchmarking. To assess the effectiveness of the above-mentioned APR techniques, researchers have created benchmarks consisting of buggy programs and test suites. Durieux et al. [16] performed a large-scale evaluation of eleven tools on 2,141 general bugs (i.e., no focus on security) coming from five different benchmarks on their proposed benchmarking platform called RepairThemAll. They found that the tools were able to fix 0.7-9.9% of the bugs and that the tools overfit Defects4J (which has the most unique patches). However, this study provided a comprehensive review of tool repairability. Several studies [47,31,53] assessed the correctness of the patches generated by the APR tools and found that not all generated patches from APR tools are correct due to the design of the bug oracle.

3.2 Automated Vulnerability Repair

Compared to general bugs, there are few repair techniques devoted to vulnerabilities, nor empirical studies to understand how advanced these techniques have become, especially for the Java programming language. Le Goues et al. [37] evaluated an APR tool called GenProg on a number of vulnerabilities, including the one that corresponds to a CVE record (CVE-2011-1148). Although this study provided an enhancement to GenProg for vulnerabilities, the number of vulnerabilities fixed is not significant, and the tool's performance on vulnerabilities has not been explained thoroughly. Instead of enhancing GenProg, which is a general bug APR tool, to fix vulnerabilities, Huang et al. [24] developed an automatic program repair tool called Senx to generate patches for memory-related vulnerabilities by using safety properties. Abadi et al. [4] developed a tool specifically for fixing injection vulnerabilities by placing sanitizers in the right place in the code.

Recently, there are several promising deep learning-based AVR tools for repairing C vulnerabilities, such as VRepair [11], VulRepair [20], SPVF [86]. However, these tools were evaluated only on vulnerabilities in several groups of CWE recorded in C vulnerability datasets. In our literature review, only a limited number of tools have been dedicated to fixing vulnerabilities for Java programs. VuRLE [50] and SEADER [85] are the techniques that infer the transformations to fix vulnerabilities *from examples.* Although the studies improved the repair rate

for specific vulnerabilities, there is limited knowledge of how the tools perform on a broader set of vulnerabilities. As far as we know, SeqTrans [12] is the first and only AVR tool that aims to fix Java vulnerabilities with a wide array of vulnerability types. This tool follows the similar idea of VRepair [11], using transfer learning to deal with the problem of shortcomings of vulnerability fix samples for training.

Benchmarking. Regarding the vulnerability benchmark used, Pinconschi et al. [64] evaluated the APR tools on DARPA's Cyber Grand Challenge (CGC) corpus [9]. However, CGC is an artificial benchmark that contains programs built with vulnerabilities planted in them. In this work, we use Vul4J [8], a benchmark consisting of *real* vulnerabilities from *real-world* projects. Our strategy is aligned with previous APR evaluations with bugs (i.e., using Defects4J [30]) and is also believed to be the ideal way to provide an evaluation that reflects the reality developers face.

Pinconschi et al. [64] followed the same paradigm as Durieux et al. [16] to implement a benchmarking platform, namely SecureThemAll, and perform a comparative study on ten APR tools for C programs. In this work, the concept of *Proof of Vulnerability (PoV)* test (also known as *negative test*) was used to measure the success rate of an APR technique. However, this study did not provide any semantic assessments of the generated patches nor perform any deep analyses on the root causes of poor repair performance of the tools, which have been carried out in our study. Also, our work focuses on a different programming language. Kechagia et al. [31] proposed a framework called APIARTy to evaluate 14 APR tools on a dataset of Java API Misuse. API misuse bugs can potentially lead to crashes and introduce vulnerabilities in software systems. However, in practice, not every API misuse is a direct software vulnerability. In our work, we evaluated and provided an explanation of the performance of APR tools on vulnerabilities in real-world Java projects. To this end, we extended a manual patch validation technique proposed by Liu et al. [47], focusing on the correctness metric. In particular, three authors manually review each patch generated from the tools to see whether it indeed eliminates the vulnerability and maintains the program's functionalities. A patch is referred to as *security-fixing patch* if it successfully eliminates the vulnerability. A *security-fixing patch* is privileged to a *correct patch* if it also does not break any functionalities.

> Current studies showed that repair patterns and actions used by existing APR tools could be leveraged for fixing vulnerabilities. However, there is very limited insight into how effective the APR tools are in fixing vulnerabilities (RQ1 and RQ2) and which repair patterns and repair actions are used the most (RQ3). This is especially true for the case of Java programming languages.

4 Methodology

This section discusses how we structured our evaluation study. We first show the selection of APR tools, our criteria for including or excluding them, and the ratio-

nale. After having a set of selected APR tools, we present the experimental setup for evaluating the selected tools, including (1) the vulnerability dataset for evaluating the tools, (2) the execution of the tools on the dataset, and (3) the manual assessment of the patch correctness. We also show separately the replication of SeqTrans, a non-test-based repair tool that we include in our study for comparison. In the last subsection, we discuss our manual assessment for analyzing the repair patterns occurring in the ExtraVul dataset (introduced in Section 5.1).

4.1 APR Tools Selection

We have considered as authoritative the list of tools in the living review of Monperrus [58]. In addition, we have analyzed the relevant studies on comparing and evaluating APR tools (e.g., Durieux et al. [16]) and reviewed APR tools that specialize in repairing security vulnerabilities (namely, we searched Google Scholar for the keywords 'program repair', 'vulnerability', and 'java').

We selected the APR tools for our evaluation based on the following criteria:
C1: Java Program Repairs. We want to focus on Java in light of the critical vulnerabilities recently detected in Java applications. For example, the recent and critical vulnerability CVE-2021-44832[3] (Log4Shell), which relates to remote code execution, has caused significant disruption on the web. To the best of our knowledge, there has not been a study evaluating the traditional APR tools on Java vulnerabilities. A study on the repair of C vulnerabilities already exists [64], albeit they did not perform the manual check that turns out to be important (cf. answer to RQ2).
C2: Source Code Available. We need to replace some of the tool's own testing features with the industrial testing pipeline and eliminate some elements of uncertainty in the vulnerability identification. Therefore, if the source code repository of a tool is not provided in its paper or we cannot find it on the internet, we will exclude the tool from our study.
C3: Executable. Some APR tools that cannot be executed due to technical issues or are not maintained anymore are excluded from our study. For example, we were not able to run ssFix [80] due to an issue connecting to its code search server.
C4: Extensible. Some APR tools require certain input formats to inject the benchmark, but there should not be a major effort to make it work with a different dataset than the tool's own recommended dataset. For example, the tool Nopol [82] was excluded because it requires a sophisticated test executor to collect runtime traces for synthesizing the patch candidates, which cannot be easily integrated with the industrial build pipeline based on Maven and Gradle build systems supported in the Vul4J benchmark.
C5: Can fix multiple kinds of bugs or vulnerabilities. The selected repair tools should be able to fix multiple types of bugs or vulnerabilities as we aim to see how well the APR tools perform on different types of Java vulnerabilities. Hence, we excluded the repair tools aiming to fix specific types of bugs or vulnerabilities. For example, we have excluded SEADER [85] because it targets only API misuses in two Java cryptographic frameworks.

[3] https://nvd.nist.gov/vuln/detail/CVE-2021-44832

C6: Test-based APR tools. We aim to check the performance of the APR tools in the presence of test cases in industrial projects (Maven/Gradle). In our evaluation, we use an extended version of RepairThemAll framework [16] which supports E2E patch testing.

Table 1 Included and excluded Java APR tools, according to criteria **C1–C6**.
* *The inclusion of SeqTrans (although not being test-based) is explained later in the text.*

	Criteria	Generic repair tools	Vulnerability-specific tools
Excluded	Not Accessible (¬ C2)	Elixir [69], Hercules [70], LoopFix [76], NS-GenProg [73], PAR [32], SOFix [48], VarFix [79], xPAR [36]	Abadi et al. [4], CDRep [49], HyperGI [57], VuRLE [50]
	Not Executable (¬ C3)	ACS [81], CapGen [77], DeepRepair [78], ssFix [80]	
	Not Extensible (¬ C4)	ConFix [33], Dynamoth [17], HDRepair [36], Jaid [10], Nopol [82], SketchFix [23], AVATAR [44], FixMiner [34], SimFix [26]	
	No Multiple Bugs/Vulnerabilities (¬ C5)	NPEFix [15]	DifFuzzAR [41], SEADER [85]
	Not Test Based (¬ C6)	CURE [28], DEAR [40], DLFix [39], KNOD [27], Recorder [87], RewardRepair [83]	
Included	Criteria Met	ARJA [84], Cardumen [55], GenProg-A [84], jGenProg [54], jKali [54], jMutRepair [54], Kali-A [84], RSRepair-A [84], TBar [45]	SeqTrans* [12]

The last row in Table 1 shows nine generic APR tools that satisfy our criteria. Since we focus on vulnerability, we consider including security-specific tools in our benchmark. This decision is also supported by the early feedback that we gathered on this paper from expert reviewers. However, there is no security-related APR tool that actually fits all the criteria, as shown in Table 1. Although there are several tools aimed at fixing vulnerabilities, as listed in the last column of Table 1, only SEADER and SeqTrans are available to us. Including SEADER would be unfair to the tool because the tool is very specific to a kind of vulnerabilities. In particular, it could only potentially fix two vulnerabilities of the same type in the Vul4J dataset. Therefore, we decided to relax the criteria C6 and included SeqTrans even though the tool is based on machine learning and not on test cases. We also remark that this tool is well-known in the APR community and regarded as being state-of-the-art and class-leading in the category of Java vulnerability repair tools. Note that we do not relax criteria C6 for generic APR tools because we are confident that we have a good representation of class-leading and state-of-the-art APR tools. For instance, these are the tools that are also used in other non-vulnerability-specific evaluations [16,31]. To make the evaluation of SeqTrans fair with respect to the generic test-based tools, we run the E2E tests on SeqTrans patches ourselves with an automated test executor which mimics the test executor of the RepairThemAll framework.

4.2 Experimental Setup

Figure 2 illustrates the overall pipeline of the evaluation in our APR study. We have applied some modifications to allow the APR tools to run on the Vul4J dataset [8], and produce vulnerability patches. We then manually assessed the patches to check whether they truly removed the vulnerabilities and did not break any functionalities of the programs (avoid breaking changes).

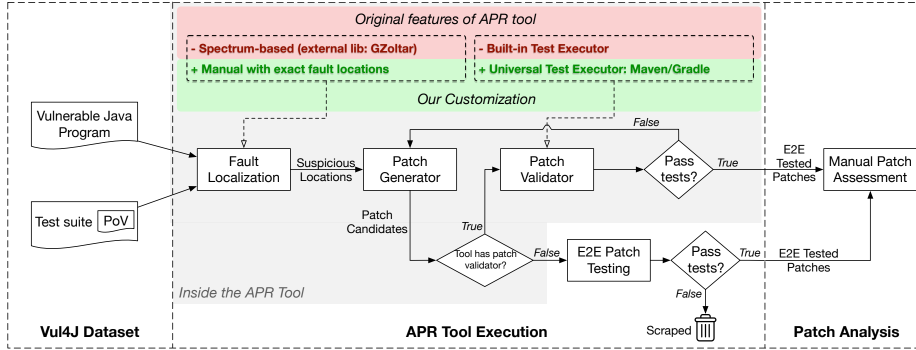


Fig. 2 The experimental setup for evaluating APR tools.

Vul4J Dataset. This dataset contains 79 Java vulnerabilities along with their corresponding PoV test cases. The PoV tests are required for the traditional APR tools as the oracle for verifying the validity of the generated patches. We discussed the detailed requirements to select the vulnerability datasets for our study in Section 5.1. We also analyzed the test execution time of the test cases in the Vul4J dataset. The results, which was reported in Section 5.3, aided us determine the proper execution time settings for running APR tools to repair the vulnerable programs in the Vul4J dataset.

APR Tool Execution. As shown in Figure 2 (middle), the selected APR tools in our study are organized according to a pipeline of three main phases: Fault Localization, Patch Generator, and Patch Validator. The Fault Localization module runs the test suite over Java programs and uses the coverage information of passing and failing tests to identify a ranked list of suspicious code locations. To this aim, GZoltar is the spectrum-based localization library originally used by the tools. The Patch Generator is the most important module, as it implements the repair strategy of each tool to produce patch candidates. The Patch Validator applies the patch candidate to the program and executes the test suite. If all test cases pass, patch candidates are yielded (the generated patches in Section 2), otherwise, it could be used to guide the repair strategy. For instance, the number of failing and passing tests is used to compute the fitness function, which guides the repair algorithm in GenProg-A, jGenProg, and ARJA.

We have extended the RepairThemAll framework [16] to facilitate the evaluation of APR tools on the Vul4J dataset. We also integrated the selected tools in this study that have not been integrated into the framework yet. For instance, we added TBar, which is considered as the best baseline template-based APR tool,

however, was not previously available in RepairThemAll. In particular, we followed the three below steps:

First, we have *integrated Vul4J* as a benchmark by implementing all functionality requirements (such as `checkout`, `compile`, `test`, and `classpath`) and providing hooks to access the necessary information (e.g., failing test list, source and class paths).

Second, we have *replaced the Fault Localization module* (GZoltar) with a "perfect" one, i.e., the faulty lines are identified manually by us and fed to the tools. We looked at the source code locations of the changes performed by the developer in the real projects to identify the vulnerability-related lines and considered them as faulty locations. GZoltar has been replaced to remove the confounding factor of possible imprecise fault localization and better isolate the effect of the patch generation techniques. Although GZoltar is used across all repair tools, the performance of fault localization is different because the tools use different versions of GZoltar. These different settings of fault localization could introduce significant bias in the overall repair performance, as reported by Liu et al. [43]. Note that we did not modify the tools to use the same version of GZoltar because of the technical issues we encountered when we were trying to run GZoltar in its latest version (v1.7.2) on the projects contained in Vul4J. In particular, for a part of the projects in the dataset, GZoltar recognized the wrong status of the test cases, which causes the imprecise calculation of suspicious statements. For some other projects, GZoltar failed to perform the fault localization due to errors. For instance, when we were running GZoltar on Apache Struts, we observed that the exception `java.lang.ClassFormatError` with the message `Method $gzoltarInit in class...has illegal modifiers: 0x1009` was thrown and the overall process was terminated. This technical problem has been reported to the GZoltar development team and received agreement from the open-source community.[4]

Third, we have replaced the built-in test executors in the Patch Validator module of the APR tools with a *universal test executor that is more reliable*. Each APR tool in our study (except SeqTrans) now employs its own home-grown code to execute the test cases when the tool needs to evaluate a generated patch candidate. These in-house test executors are convenient but vary in implementation, and have technical issues when running the test suites on large-scale and real-world projects in Vul4J. Moreover, these built-in test executors are able to execute only the JUnit test cases, however, they cannot execute the test cases from different testing frameworks such as TestNG which exist in the Vul4J dataset. All the projects in Vul4J use Maven or Gradle as the build systems, which have been already configured by the project developers to run test cases and report test results.

To implement our universal test executor, we created commands to trigger Maven or Gradle to run the test cases (specific test cases or the whole test suite). When the repair tools want to run some test cases or the whole test suite, the corresponding commands are executed. Finally, we wrote a Python script to collect the test results from Maven and Gradle, then extract the failing tests and passing tests to report back to the APR tools. As we expected, all the patches generated by the APR tools, into which we integrated the universal test executor, are E2E tested. For the tool SeqTrans, which does not have any internal test executors,

[4] `https://github.com/GZoltar/gzoltar/pull/44`

the tests are automatically executed by the experimenters after it outputs the predicted patches (similar to what is done in the work of Kechagia et al. [31]). In Subsection 4.3, we elaborated in detail on the replication of SeqTrans and the evaluation of the generated patches from this AVR tool.

We remark that the above-mentioned modifications did not introduce any impact for what concerns the comparison of the tools' repair performance because their repair strategies were preserved. If anything, the changes reduced the bias and made the comparison conditions more even and homogeneous across the APR tools.

Patch Analysis. To determine the correctness of a patch, we manually inspect the E2E tested patches regarding both security- and functionality-wise. The manual assessment determines the number of patches generated by the APR tools that can be used by developers. By the end of the patch assessment process, a patch can be classified as a *Security-fixing Patch*, *Correct patch*, or *Incorrect patch.* In Section 2, we have already introduced the definitions of a *Security-fixing patch* and *Correct patch.* In our study, an E2E tested patch that is neither a *Security-fixing patch* nor *Correct patch* is considered as an *Incorrect patch.*

Most of the tools in our study complete their executions after generating the first E2E tested patch. However, several tools in the Arja platform (GenProg-A, ARJA, and RSRepair-A) might generate dozens of patches. In that case, we select the first generated patch (E2E tested) for the subsequent manual analysis. Note that the APR tools produce patches in an order that the tools decide internally. We argue that selecting the first patch generated by the tools is optimal because the tools tend to prioritize their highest-ranked patch candidate to generate first. After picking the E2E patches, we compare each E2E tested patch with the corresponding developer's patch. To do so, the first three authors of this paper manually examine each E2E patch to determine if it is correct or not. Then, we calculate the inter-rater reliability [56] to measure the agreement between the researchers. When having a disagreement on a patch, the researchers discuss resolving the conflicts. In our manual validation, the reviewers mostly agree with each other on the correctness of a patch (an average agreement level of 75.36% ± 2.84%).

Our next analysis aims to understand the generalizability of our findings when scaling to a larger number of vulnerabilities. To achieve this, we use the confidence interval for the *probability* that an E2E tested patch will eventually be a correct patch for the *general population* of E2E tested patches. We use the Wilson-Agresti-Coull [5] approach to compute the confidence interval. Equation 1 shows the calculation of the upper and lower bounds of the confidence interval. In this equation, the lower bound p_{lower} and upper bound p_{upper} provide the bounds for the population proportion of E2E tested patches that are actually correct. In Equation 1, p is the population proportion, $\hat{p}$ is the sampled proportion of correct E2E tested patches in the manually validated sample, $z_{95\%}$ is the two-sided 95% confidence interval, in this case, z has a value of 1.96, and n is the sample size, i.e., the total number of E2E tested patches in the sample.

$$|\hat{p} - p| = z_{95\%}\sqrt{\frac{p(1-p)}{n}} \tag{1}$$

4.3 SeqTrans Replication

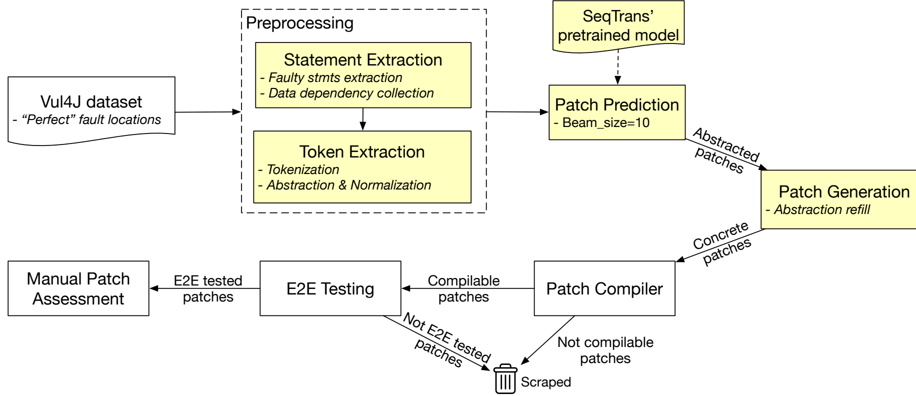


Fig. 3 SeqTrans replication and evaluation. The yellow boxes are the scripts and models of SeqTrans while the white boxes are our own components.

We evaluated SeqTrans separately from the other APR tools in the RepairThemAll platform as we explained in the sequel. To replicate SeqTrans, we obtained its source code and trained models available on GitHub.[5] SeqTrans provided several models for different evaluation purposes, e.g., comparing the results of joint training and independent training of the general bug dataset and vulnerability dataset. In our study, we chose the trained model used in the research question RQ1.3 in the SeqTrans's paper [12] because (1) this model was evaluated in the same way we evaluated other tools (i.e., for each generated patch, they will check if the patch is compilable, plausible, and correct), (2) their model was trained on a part of the 'project KB' dataset [67] which is one of the biggest Java vulnerability fix datasets in the literature. The dataset (Vul4J) used in this study was derived from the 'project KB' dataset, and there are 61 of 79 (77.2%) vulnerabilities in Vul4J also exist in the training set of SeqTrans. Therefore, the repair results of SeqTrans we reported in this study might not truly represent the repair performance of this tool in general.

Figure 3 illustrates our pipeline to replicate and evaluate SeqTrans on the Vul4J dataset. The yellow boxes indicate the scripts and the model that we downloaded from the SeqTrans' source code repository on Github, while the rest were constructed by us for the replication. After selecting the appropriate model for evaluation, we follow the guidelines in their replication package to evaluate SeqTrans on the Vul4J dataset. In particular, for each vulnerability in Vul4J that has known vulnerable locations, we extract the vulnerable statements along with their data-flow dependencies. Then, we preprocess the inputs using the abstraction and tokenization techniques. Next, we run the patch prediction of the model using a beam size of ten,[6] which returns ten patches for each vulnerable statement. Next, we perform abstract backfill and inject the patches into the vulnerable source files

[5] `https://github.com/chijianlei/SeqTrans`
[6] The default beam size used in SeqTrans

of the projects. Finally, since SeqTrans does not employ any compiler or any built-in test executor, we compile the projects with the injected patches. In particular, given a list of patches sorted by SeqTrans, we compile each patch in the list in the top-down order. If a patch does not cause any compilation errors, we proceed to the next step; otherwise, we discard that patch. We repeat the same strategy to run the E2E tests ourselves on all the compilable patches and scraped the patches that caused failed tests. If there is more than one E2E tested patch for a vulnerability, we pick the first one on the list, as we also do the same for the other tools and then manually evaluate them.

4.4 Manual Repair Patterns Collection

To collect the repair patterns, we first track the repair action list in the Defects4J dissection website,[7] which is categorized by the source code components. We then observe each vulnerability patch in ExtraVul and collect the list of changed source components and the repair actions applied to them. Some of the repair actions we found in ExtraVul have not been listed in Defects4J, e.g., *Change the keyword from `while` to `if`*. Next, we retain only the list of repair actions that existed in ExtraVul and computed the prevalence of them in ExtraVul and Defects4J. Further, we extract the common repair patterns in vulnerability patches. These patterns are formed by the aggregation of multiple atomic repair actions.

The tasks of collecting and labeling both the repair actions and the repair patterns have been carried out manually by the first author of this paper and reviewed by the second author. The two authors discussed the disagreements until a consensus was reached.

5 Dataset

A key requirement to assess the effectiveness of an APR is an appropriate benchmark, that includes test cases and vulnerability-fixed code pairs. Several of these benchmarks exist, albeit they are typically developed for evaluating static analyzers [19,75]. Examples of artificially created benchmarks include Juliet [7] and Scanstud [29], while the 'project KB' knowledge base [67] and Vul4J [8] are the manually curated datasets based on real-world free and open-source software projects.

5.1 Vulnerability Dataset Selection

To benchmark the APR tools on vulnerabilities, we selected the projects that have the following characteristics:

– **DC1: Real World Projects**. The dataset should contain real-world projects written in Java, as this is part of all the research questions in our study.

[7] https://program-repair.org/defects4j-dissection

– **DC2: Test for Proof of Vulnerability (PoV)**. Each vulnerability in the dataset should have one or more test cases that fail when the vulnerability is present and pass when it is patched. The project should also contain other functionality tests. The test cases are collectively used by APR tools for patch candidate validation, i.e., to ensure that a patch 'works' and does not introduce regression bugs.
– **DC3: Independent Human Ground Truth**. Each vulnerability in the dataset should have an associated patch written by developers (e.g., in a fixing commit), as this is necessary for the context of RQ2 and RQ3.

In literature, the Vul4J dataset [8] appears to be the only benchmark that satisfies the above requirements. Therefore, we decided to use Vul4J for the studies of the APR tools in our first two research questions: RQ1 and RQ2. Vul4J contains 79 vulnerabilities from 51 real-world open-source Java projects. The vulnerabilities are taken from the 'project KB' knowledge base [67]. The projects in Vul4J span multiple domains, including libraries, web frameworks, data-processing desktop apps, and CI/CD servers. Table 2 shows the characteristics of the top 15 projects in the Vul4J dataset that contain more than one vulnerability, and the aggregated metrics for the remaining 36 projects that have only one vulnerability. The number of test cases per project ranges from 25 to 5222.

In RQ3, we have relaxed our selection criteria to create a larger dataset of vulnerability fixes which allowed our study to be more comprehensive. Indeed, the criteria **DC2** has been omitted as PoV test cases are not necessarily needed for the fix pattern analyses of the patches. To this end, we followed all the steps described in the Vul4J paper [8] except for *Vulnerabilities reproduction* and *Missing PoV test cases creation* to collect more vulnerability fixes from the 'project KB' knowledge base. We then retained only the patches with less than or equal to ten changed lines as we wanted to exclude the patches that included unrelated changes to the vulnerability fixes, such as refactoring and aesthetic code changes. The patches that had already existed in Vul4J were obviously excluded. As a result, we have collected 119 additional valid patches for 119 unique vulnerabilities. We then combined them with the 79 vulnerability patches from Vul4J and created a new dataset of 198 vulnerability fixes in total (called ExtraVul from here on), which was used for the study in our RQ3.

5.2 Reference Dataset Selection

To understand the difference between the repair patterns used by humans for fixing security vulnerabilities and generic software bugs (RQ3), we can compare the repair patterns adopted by developers and compare the case of security vulnerabilities versus other software defects. To do this, we considered several defect datasets that are often used in APR research papers: Defects4J [30], IntroClass [18], QuixBugs [42], Bugs.jar [68], Bears [52], SARD [2]. We selected Defects4J [30] as it fits our below requirements:

– The dataset should contain only Java real-world projects (as this is the focus of our work) and openly accessible;
– The dataset should contain a variety of fixing patterns, e.g., one should not contain just statement deletions or skips;

Table 2 Top projects included in the Vul4J dataset [8].
On average, the projects are relatively large (169 kLoC) compared to, for instance, 85 kLOC average in Defects4J. They also have a large number of test cases.

Project	#Vuls	kLOC	#Tests
apache/struts	10	359	1697
apache/commons-compress	4	48	927
jenkinsci/jenkins	3	275	518
spring-projects/spring-framework	3	684	2189
spring-projects/spring-security	3	198	513
apache/camel	2	925	5222
apache/commons-fileupload	2	6	70
apache/commons-imaging	2	42	562
apache/cxf	2	737	89
apache/pdfbox	2	145	359
apache/sling	2	507	25
cloudfoundry/uaa	2	182	2669
FasterXML/jackson-dataformat-xml	2	9	140
inversoft/prime-jwt	2	2	33
OpenRefine/OpenRefine	2	144	516
36 other projects (mean)	1	121	567
all projects (mean)	1.5	169	705

- The dataset should contain the correct patches written by developers, which should not be just defined as the ones passing all the tests (i.e., there should be some additional quality guarantee for the patches).

Based on the above-mentioned criteria, we excluded SARD [2]. Although SARD contains buildable Java projects, it lacks a variety of repairs, such as adding a method invocation. On the other hand, Defects4J is relevant when compared to ExtraVul since (1) it is one of the largest benchmarks, (2) widely used in evaluating APR techniques with good repair results reported [47]. In addition, the work on Defects4J dissection [71] provides a concrete list of repair actions and patterns that serve as a baseline in our comparison.

5.3 Discussion on test cases in Vul4J

This part discusses Vul4J, the main dataset used in our paper, and our chosen settings for experimenting with it. A quarter of the vulnerabilities in the Vul4J dataset (25.32%, 20 out of 79) are associated with projects containing more than 1000 test cases at the vulnerable revisions. So, it will take time to execute all the test cases for these vulnerabilities. As a consequence, this may impact the overall performance of the repair tools if we set a similar time budget for each repair attempt in the previous work (e.g., Durieux et al. [16] used a two-hour budget in their study).

In this work, we executed the PoVs and the whole test suites of the projects on their vulnerable revisions (corresponding to the vulnerabilities in Vul4J) and measured the running time for their executions. Table 3 summarizes the results of test execution time in the Vul4J dataset by the metrics. We observed that, for a vulnerability in Vul4J, it took, on average less than 16 seconds to run its PoV tests and less than six minutes to run the whole test suite of the project. While the PoVs take at most less than five minutes to run, some projects take

Table 3 Descriptive statistics of the test execution in Vul4J dataset.

**Outlier: VUL4J-58*

Variable	min	Q25%	median	Q75%	max	mean	st.dev
#Test cases	1	139.5	620	1656	5222	1068.41	1236.54
PoV tests execution time (s)	2.27	5.85	8.76	13.72	292.63	15.96	34.06
All tests execution time (s)	2.22	9.17	27.04	74.50	8690.39*	337.83	1223.84

hours to run their whole test suites. For example, it took 2.41 hours to run all the test cases for `jenkinsci/subversion-plugin` (VUL4J-58) and 1.39 hours for `jenkinsci/junit-plugin` (VUL4J-56). Interestingly, these two projects do not belong to the projects containing more than 1,000 test cases, they have only 234 and 429, respectively. Meanwhile, `apache/camel`, which is the project containing the largest number of test cases (5222 test cases) in the Vul4J dataset, took approximately only 35 minutes to run all the tests.

Based on the results of the test execution time of the dataset, we decided to set two hours as the timeout for our universal test executors, which we injected into the repair tools. We also set the timeout for each repair process to four hours. These decisions are made based on our observations about the consumed time of test execution in the dataset and the time limit settings of the evaluated APR tools [46]. In addition, the evaluation conducted by Vu et al. [74] indicated that there is no point in giving more time as the number of patches does not increase with time. We observed that there is only one case (VUL4J-58) requiring more than two hours for its test suite execution that we cannot cover with our configured timeout. For this special case, we set a bigger time budget (72 hours) and let the tools generate at least ten patch candidates before we terminate the repair attempts due to timeout exceeding.

6 RQ1: Performance of APR tools on vulnerabilities

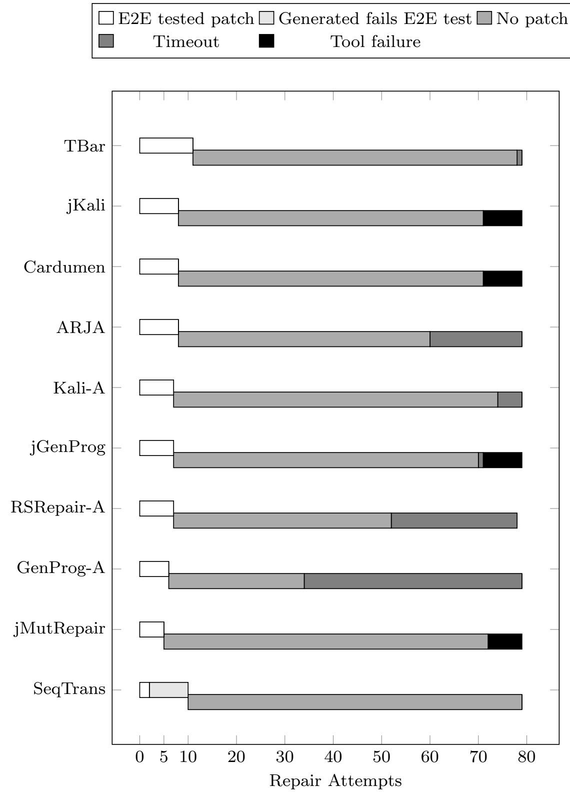
6.1 Concrete Experimental Setup

Our experiments were conducted on an Ubuntu 20.04 Docker container, which was deployed on a macOS machine with a 2GHz Quad-Core Intel Core i5 Processor, and 16 GB of RAM. We allocated up to 4 CPUs and 8 GB of RAM for the Docker container via Docker Desktop.

6.2 Experimental Findings

To answer this research question, we analyzed the repair capability of the ten selected APR tools on the 79 vulnerabilities in Vul4J. Figure 4 shows the number of generated patches on the upper bar and the number of failed attempts on the lower bar. For each tool, the repair attempts generated at least one patch, while the failed attempts could not generate any patch.

Only one-fifth of the vulnerabilities in Vul4J are patched by the tools. We divide the number of generated patches (the upper bar) in Figure 4 into E2E tested patches and Generated patches that failed the automatic sanity check of re-running



TBar had the highest number of E2E tested patches (white bar). SeqTrans also generated some patches but those failed to successfully pass the E2E tests. Otherwise, most of the repair attempts concluded in either the tool terminating but not reporting any generated patch (first shifted slanted bar) or in a timeout or other tool failures.

Fig. 4 RQ1: Repair capability of APR tools on Vul4J.

all the tests. The total number of Generated vs. E2E tested patches for each tool is also reported in Table 4 (first and second column, respectively). Considering the lower bar (failed attempts) in Figure 4, we divide the data into internal technical failures, timeout, or other reasons, where the tool terminates but does not report any patch.

Collectively, the APR tools generated patches for 23 unique vulnerabilities that account for 29.11% of the total number of vulnerabilities in the dataset. Regarding the repair attempts, 78 out of 790 attempts (9.87%) could generate at least one patch for a vulnerability. Of those 78 generated patches, 89.74% graduated as E2E tested patches, while the rest 10.26% which failed are all produced by SeqTrans. Note that tools in the Arja platform (ARJA, GenProg-A, RSRepair-A) can generate *dozens* of 'equivalent' patches for a single repair attempt. We tested all the 'equivalent' ones and, for the patches that were successfully E2E tested, we selected the first one as the E2E tested patch that would be manually analyzed in the next research question. As a result, we are left with 70 E2E tested patches for 16 unique vulnerabilities.

As shown in Figure 4, TBar, and SeqTrans generated patches for the highest number of vulnerabilities (11 and 10, respectively). Following them, RSRepair-A, jKali, Cardumen, and ARJA each generated patches for eight vulnerabilities, while Kali-A and jGenProg generated patches for seven vulnerabilities. GenProg-A and jMutRepair are less effective in producing patches with only six and five vulnerabilities, respectively.

Even the security-specific tools cannot improve the repair performance. When considering E2E tested patches only, SeqTrans's performance drops dramatically, i.e., among ten compilable patches that we obtained from this tool, only two of them are E2E tested. This implies that SeqTrans can generate E2E tested patches for only 2.53% of the vulnerabilities in the Vul4J dataset. Meanwhile, the number of considered patches for other repair tools remains. The reason behind this fall is that all the tools except for SeqTrans employ the universal test executor to validate their patch candidates before yielding them, which are equivalent to the E2E testing.

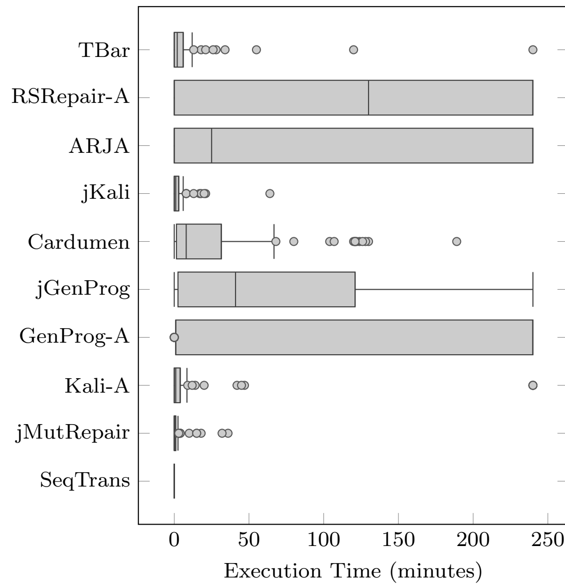
Most of the repair attempts do not succeed. A total of 712 repair attempts (90.12%) could not generate patches, of which 4.35% is due to some failures of the tools, 13.62% is due to timeout, and 82.03% is due to the limitation of the repair strategy of the tools. The APR tools that use genetic programming or random search approaches (ARJA, GenProg-A, and RSRepair-A) mostly have the highest number of timeout attempts (92.78%). It should be noted that some repair attempts by ARJA, GenProg-A, and RSRepair-A exceeded the time limit, but only after the tool had already produced a patch previously. In such cases, we do not count these attempts in the timeout category.

For what concerns tool failures, we observed that occasionally tools from the Astor platform (jGenProg, jKali, jMutRepair, and Cardumen) abruptly terminated the repair attempt due to some internal error, while other tools did not encounter the same problem. For instance, when we ran the repair tools to fix the Cross-site scripting (XSS) vulnerability CVE-2013-4378 in the JavaMelody project, all the Astor-based tools raised the same *java.lang.ClassFormatError* error and terminated the repair processes immediately. Other repair tools were able to generate patches for this vulnerability. This failure could be due to either configuration or implementation issues of the Astor platform.

Most of the tools run in a reasonable amount of time, especially SeqTrans, jKali, Kali-A, and jMutRepair. Figure 5 shows a boxplot of the time (in minutes) spent by each tool to repair the vulnerabilities. On average, the tools take around 50 minutes to complete the repair attempts. SeqTrans, jKali, Kali-A, and jMutRepair are the fastest repair tools, with a median time of less than or equal to one minute. TBar and Cardumen run a bit slower, but they still have speedy repair attempts with a median time of two minutes and eight minutes, respectively. However, there are exceptions like GenProg-A, RSRepair-A, and jGenProg, ARJA with a median of 240, 130, 41, and 25 minutes, respectively.

This can be explained by two reasons: (1) these tools employ genetic programming and random search as the repair strategies, which usually come with a very large search space, and (2) technical implementations of the tools, as Arja-based tools sometimes did not stop their repair attempts even though they produced the patches before the time limit.

The technical implementations of ARJA and GenProg-A cause overheads for their execution time. We observed that there is a significant difference between the



Some of the differences are explained by differences in implementation (e.g., GenProg-A vs. jGenProg). GenProg-A and ARJA always run both PoV tests and regression tests for every patch candidate, which causes overhead for projects with large test suites. Among the tools that generate the most correct patches (See Figure 4), TBar and jKali have the best time/quality trade-off.

Fig. 5 RQ1: Execution time of the repair tools.

execution time of GenProg-A (median = 240) and jGenProg (median = 41), which are the different Java implementations of the same repair tool for C, GenProg [38]. This is due to the difference between their implementation for patch candidate evaluation strategy. In particular, GenProg-A (and ARJA) always run both PoV tests and regression tests for every patch candidate, which could cause a significant time overhead for projects with large test suites. On the other hand, jGenProg, by default configuration, does not run the regression tests on the patch candidate unless the patch has already passed the PoV tests.

Human-competitiveness. As mentioned by Monperrus et al. [59], there are two criteria for the tools to be human-competitive: (1) the tool generates the patch faster than the human developer, and (2) the human developer accepts the generated patch. The second criterion is out of scope for this paper (it has been studied by our prior work [62]). For the first criterion, most of the tools in our study (TBar, Cardumen, Kali-A, jKali, jMutRepair) halt immediately after the first E2E tested patch is found. This is, on average, under one hour, which appears competitive with the human developer. SeqTrans generates all the patches in a few seconds, hence, it would be (if the patches were correct) always competitive wrt a human developer. Three tools from the Arja framework (ARJA, GenProg-A, RSRepair-A) generate dozens of patches before the repair process terminates. However, on average, these tools complete their executions after two hours, which would be acceptable for fixing vulnerabilities.

Main findings for RQ1: APR tools perform rather poorly as even *collectively* can only generate patches that turn out to be successfully E2E tested for 20.25% of the total number of vulnerabilities in the Vul4J dataset. This happens *in spite of* all tools being tipped on the exact location of the vulnerability.

To gain insight into the failed repair attempts, we further examined the human fixes for the vulnerabilities in Vul4J for which no repair tools in our study could produce E2E tested patches. We found several kinds of patches interesting to discuss in this context, which we elaborate on below.

```
// VUL4J-5
- String targetDirPath = targetDirectory.getCanonicalPath();
+ String targetDirPath =
    targetDirectory.getCanonicalPath() + File.separatorChar;
...
File f = new File(targetDirectory, nextEntry.getName());
if (!f.getCanonicalPath().startsWith(targetDirPath)) { ...

// VUL4J-71
- private static final Random RND = new Random();
+ private static final Random RND = new SecureRandom();

// VUL4J-66
switch (str.charAt(i)) {
  case '/':
  case '\\':
  ...
+  case '\n':
  return false;
```

Fig. 6 Developer's patches for VUL4J-5, VUL4J-71, and VUL4J-66. None of the tools in our study can fix these vulnerabilities despite the minimal changes.

Patches may require minimal changes, however, they need a vocabulary of security-related code. Figure 6 shows a couple of human fixes that require minimal changes that the APR tools failed to generate successful patches. To fix the vulnerabilities, the repair tools should be equipped with the appropriate vocabulary of security-relevant code. For example, APR tools should have knowledge about the path-related fix ingredients (e.g., `File.separatorChar`) to prevent the path traversal vulnerability in VUL4J-5. Other patch examples in Figure 6 also require security understanding to generate secure random numbers in Java programs (VUL4J-71), and the special characters (e.g.,'`!`','`@`','`\n`', etc.) should be validated for web input handling (VUL4J-66).

Patches that require an understanding of API usage for security. The XML External Entity (XXE) vulnerability in Figure 7 is caused by the insecure configuration when creating a new `SAXParser` instance. The developer had to disable the insecure feature '*load-external-dtd*' of the parser to prevent this vulnerability. We observed in our study that traditional APR tools such as jKali tend to fix bugs by

```
// VULNERABLE code
saxFactory = SAXParserFactory.newInstance();

try {
  saxFactory.setFeature("...external-general-entities", false);
  saxFactory.setFeature("...external-parameter-entities", false);
} catch (...) { /* Exceptions handling */ }

// ------------------------------------------
// DEVELOPER's change
saxFactory.setFeature("...external-parameter-entities", false);
+  saxFactory.setFeature("...load-external-dtd", false);

// jKali's patch
+ if (true) return null;
saxFactory = SAXParserFactory.newInstance();

// SeqTrans' patch
- saxFactory = SAXParserFactory.newInstance();
+ saxFactory=SAXParserFactory.newInstance();
```

Fig. 7 Developer's patch and compilable patches of APR tools for VUL4J-2. jKali's patch tried to skip the XML parsing and returned a null Document, which makes the program compilable but does not pass the E2E test. SeqTrans produced a nonsense patch with no semantic changes.

removing code, directly or indirectly. As shown in Figure 7, although this kind of pattern, can make the program compilable, the generated patch cannot pass the E2E test. To fix the vulnerability in Figure 7, APR tools should be able to learn about the usage of a given security API, and how to change the usage to remove the vulnerability.

Complicated patches. Several security patches written by developers require complex changes. For example, in Figure 8, to prevent attackers from exploiting a NULL byte injection vulnerability, the developer had to validate the input file name carefully (i.e., (1) check if `repository` is not null, (2) make sure `repository` is indeed a directory, and (3) check if there exist any NULL bytes in `repository`). Another example of a complicated patch is shown in Figure 9. The vulnerability allows attackers to leverage Spring SpEL to trigger remote code execution. To fix the vulnerability, the developer had to make changes in multiple locations to create the randomness for synthesizing the final string '`result`'. To generate such complicated patches, The evaluated APR tools should learn to understand the context of the vulnerability and compose a patch based on the context.

7 RQ2: Trustworthiness of generated patches

Among the APR tools, RSRepair-A achieves the highest ratio of generating correct patches. Table 4 (right) shows the results of our manual validation, including the 95% Agresti-Coull confidence interval (Section 4.2). The correctness percentage of a tool is the ratio of Correct over E2E tested patches. For the generic tools, although TBar has the highest number of E2E tested patches, the correctness percentage of this tool is only 45.5%. On the other hand, the tools with the

```
+ if (repository != null) {
+  if (repository.isDirectory()) {
+   if (repository.getPath().contains("\0")) {
+    throw new IOException("Null character");
+   }
+  } else throw new IOException("Not a directory");
+ }
...
File tempDir = repository;
String tempFileName = format("...%s_%s.tmp", UID, getUniqueId());
tempFile = new File(tempDir, tempFileName);
dfos = new DeferredFileOutputStream(sizeThreshold, outputFile);
...
dfos.write(cachedContent);
```

Fig. 8 Complicated patch of VUL4J-10 that prevents attackers from injecting NULL byte in the input file name to write to arbitrary files on the system.

```
+ String prefix = new RandomValueStringGenerator().generate() + "";
+ String maskedTemplate = template.replace("${$", prefix);
...
- this.helper = new PropertyPlaceholderHelper("${$", "}");
+ this.helper = new PropertyPlaceholderHelper(prefix, "}");
...
- String result = helper.replacePlaceholders(template, resolver);
+ String result = helper.replacePlaceholders(maskedTemplate, resolver);
+ result = result.replace(prefix, "${");

response.setContentType(getContentType());
response.getWriter().append(result);
```

Fig. 9 Complicated patch of VUL4J-75, a vulnerability that allows leveraging Spring SpEL to trigger remote code execution.

highest correctness percentages are RSRepair-A and ARJA (71.43% and 62.5%, respectively), whose number of E2E tested patches is lower than TBar. Indeed, we found that genetic programming-based tools (ARJA, GenProg-A) and random search-based tools (RSRepair-A) can eliminate the vulnerabilities that require the addition of method calls while other tools cannot. For example, these APR tools added a method called `t.emitTagPending()` to fix a Cross-site scripting vulnerability as shown in Figure 10. However, we observed that the method call must reside somewhere in the codebase, so the tools can reuse it. In case the fix requires using a new method call, that is not in the codebase, most of the APR tools will fail. Generating the patches when their ingredients are not present in the code base is a challenging task, not only for traditional APR tools but also for vulnerability repair tools.

The correct patches generated by SeqTrans (security-related) actually exist in the tool's train dataset. When evaluating the two E2E tested patches generated by SeqTrans, we found that both of them are correct, which means that SeqTrans has (by chance) the correctness of 100%. These patches are even identical to the human patches. However, our further investigation reveals that those generated

```
// VULNERABLE code
case eof:
  t.eofError(this);
  t.transition(Data);
  break;

// -------------------------------------------
// DEVELOPER's change, ARJA's and RSRepair's patch
t.eofError(this);
+ t.emitTagPending();

// GenProg-A's patch, Security-fixing (Partially Correct)
- t.eofError(this);
+ t.emitTagPending();
```

Fig. 10 Developer's patch for VUL4J-59. Genetic programming- and random search-based tools have good performance for this vulnerability.

Table 4 RQ1 and RQ2: Assessment of the security patches generated by the APR tools. "%Correct" is the ratio of Correct patches over End-to-End tested patches. The confidence interval for correctness is also reported.

Tool	RQ1	RQ2				
	#E2E tested patches	**#Security-Fixing patches**	**#Correct patches**	**%Correct**	**C.I. Lower**	**C.I. Upper**
ARJA	8	8	5	62.5%	30.6%	86.3%
Cardumen	8	3	2	25.0%	7.1%	59.1%
GenProg-A	6	6	2	33.3%	9.7%	70.0%
jGenProg	7	4	2	28.6%	8.2%	64.1%
jKali	8	5	3	37.5%	13.7%	69.4%
jMutRepair	5	2	1	20.0%	3.6%	62.4%
Kali-A	7	6	3	42.9%	15.8%	75.0%
RSRepair-A	7	7	5	71.43%	35.9%	91.8%
SeqTrans	2	2	2	100.0%	34.2%	100.0%
TBar	11	7	5	45.5%	21.3%	72.0%

correct patches by SeqTrans actually existed in their training dataset. This may lead to the problem of overfitting when the model learns too much about the training set.

More than half of the patches eliminate the vulnerabilities, however, hinder the maintainability. On average, only 46.66% of the patches (correct patches) can be used directly by developers, as they can eliminate the vulnerability while maintaining the program's functionality. On the other hand, the average fixing percentage (the ratio of Security-fixing patches over E2E tested patches) is 74.65%, which gives us a 28% difference from the correctness percentage. Although these patches can be used by developers to eliminate the vulnerability in their program, additional modifications are needed for the patches to prevent breaking functionalities.

The vulnerabilities the APR tools perform the best often use the patches that remove or skip code. Figure 11 shows the developer's patches for VUL4J-39 and VUL4J-50, which most tools in our study were able to generate Correct and Security-fixing patches, respectively. VUL4J-39's fix requires the deletion of a code block to remove the vulnerabilities. Most of the tools support the deletion repair action, especially Kali-A, and jKali. In our study, nine of the ten repair tools (except for SeqTrans) can fix this vulnerability successfully. Regarding the vulner-

ability VUL4J-50, the parameter `remoteAddr` should be carefully sanitized before passing to the method `write`, which prints the value to the web, to avoid a potential XSS vulnerability. Many tools generated patches that try to skip or remove this statement. TBar was close to fixing correctly the vulnerabilities by changing the statement to `javascriptEncode(remoteAddr);`. SeqTrans generated an identical patch to the human patch. All patches from the tools pass the E2E test. In most cases, `remoteAddr` is not printed to the web anymore, therefore, there were no XSS vulnerabilities. However, the deletion of the vulnerable code (like jKali) will break the functionality, that is, we do not have the information about the `remoteAddr` on the web. The reason is that there were no regression test cases checking if `remoteAddr` is printed correctly and safely, but only checking for the existence of a vulnerability. This indicates the necessity of a manual evaluation. In our manual assessment, we classified the patches as Security-fixing, which means that it can successfully remove the vulnerabilities, however, it introduces problems for the system's functionalities.

```
// VUL4J-39
- if (sessionId != null) {
-   ...
-   sb.append("sessionId=").append(sessionId);
- }

// VUL4J-50
- write(remoteAddr);
+ write(htmlEncodeButNotSpace(remoteAddr));
```

Fig. 11 Developer's patches for VUL4J-39 and VUL4J-50.

Figure 12 visualizes the trend of the patch count in each category. The trend is going down for all the ten evaluated tools, from the number of Generated to Correct patches, which correlates with the correctness percentage we have discussed before. Since we embedded the universal test executor into the tools (except for SeqTrans), all generated patches generated by the tools are equivalent to the E2E tested patches. In the case of SeqTrans, which does not have any test executor inside and we ran the E2E tests ourselves, only 20% of the generated compilable patches actually passed the tests.

Looking at the *E2E-Tested-to-Security-Fixing* trend in Figure 12, we observed that there are only four straight lines (three generic tools: ARJA, RSRepair-A, and GenProg-A; and SeqTrans). Therefore, only four out of nine (44.4%) tools actually maintain the reliability of their E2E tested patches in terms of eliminating the vulnerability. The last trend is the one and only straight line from Security-fixing to Correct, which is achieved by SeqTrans. This trend is straight because of the presence of the fixing patches in the training dataset.

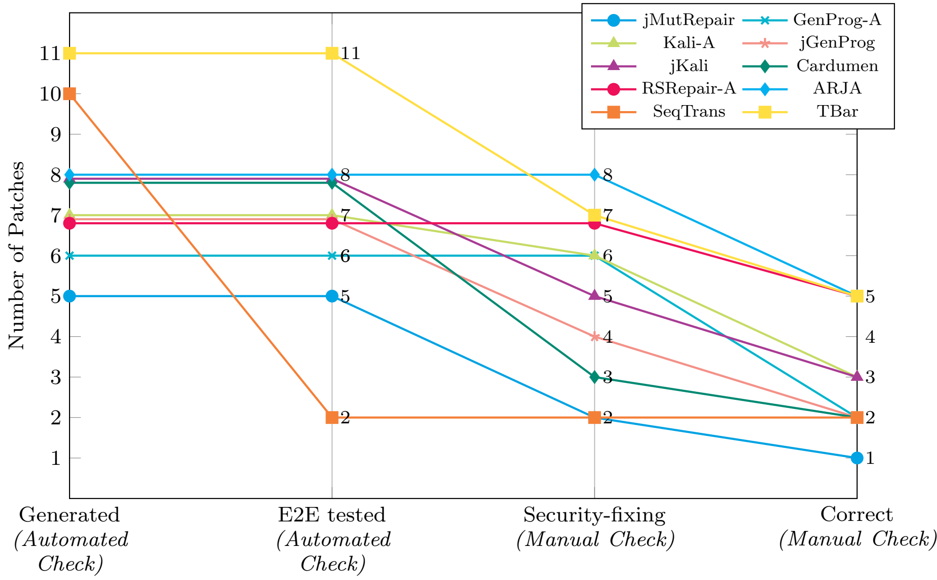


Fig. 12 RQ2. Trustworthiness of patches generated by APR tools on Vul4J.

Table 5 RQ2: Proportion of correct to generated patches in Defects4J. The data is derived from Liu et al. [47].

	From literature		Computed in this paper		
Tool	**#E2E Tested**	**#Correct**	**%Correct**	**C.I. Lower**	**C.I. Upper**
ARJA	58	6	10.3%	4.8%	20.8%
Cardumen	12	3	25.0%	8.9%	53.2%
GenProg-A	30	2	6.7%	1.8%	21.3%
jGenProg	20	5	25.0%	11.2%	46.9%
jKali	25	6	24.0%	11.5%	43.4%
jMutRepair	22	5	22.7%	10.1%	43.4%
Kali-A	65	3	4.6%	1.6%	12.7%
RSRepair-A	41	4	9.8%	3.9%	22.5%
TBar	72	24	33.3%	23.5%	44.8%

7.1 Generic Tools Result in Comparison to Defect4J

To compare the performance of the APR tools on fixing vulnerabilities vis-à-vis fixing general bugs, we extracted the patch correctness percentage in the Defects4J dataset [30] from [47]. Table 5 shows that TBar tends to generate many patches, as its number of E2E Tested patches is the highest among its peers in both benchmarks. Although TBar's correctness is the best for general bugs (33.3%), it ranks the third-lowest correctness for vulnerability only. On the other hand, ARJA and GenProg-A have low performance in general bugs (10.3% and 6.7%), but are top of the rank in vulnerability repair with a 50% correctness percentage.

APR tools from different platforms perform differently on general and security bugs. The APR tools in the Arja platform (ARJA, GenProg-A, Kali-A, and RSRepair-A) have a medium-low correctness percentage for general bugs, but for

vulnerabilities, they have a high-medium performance. On the other hand, the APR tools in the Astor platform (Cardumen, jGenProg, jKali, and jMutRepair) obtain higher ranks in general bugs (all in the top five) but have varying performance for vulnerability. In the same platform, we observed that Cardumen and jMutRepair have lower correctness percentages compared with the other two.

The trustworthiness of APR tools could be different when it comes to repairing vulnerabilities In the Defects4J benchmark, for some tools (namely, ARJA, GenProg-A, Kali-A, RSRepair-A), the E2E-tested to Correct trends are steeper than in the case of vulnerabilities. This means the trustworthiness of APR tools could be different when it comes to repairing vulnerabilities rather than general bugs. The next section explains this phenomenon in more detail.

Main findings for RQ2: If developers managed to obtain *ten* End-to-End tested patches from APR tools, *three* of them would be useless, *three* of them would remove the vulnerability with additional manual modifications, and *only four* out of them could be used as-is. This phenomenon also exists for generic bugs, however, is different for each tool. Independent validation of a sample of results is therefore mandatory to assess the real effectiveness of APR techniques.

8 RQ3: Root Causes, Developers and APR

In this research question, we conducted a comprehensive and systematic study to understand the reasons why current test-based APR tools failed to repair Java vulnerabilities by analyzing the fix patterns in vulnerability patches and comparing them to generic bug patches. We also gave some insights that could help improve current APR tools or develop new AVR techniques to fix vulnerabilities.

To identify the characteristics of fixing patterns used by developers in the ExtraVul and Defects4J datasets, we consider two granularity levels: (1) atomic repair actions, which are the smallest actions that could be applied to source code components (e.g., addition or removal, or modification of an assignment); and (2) repair patterns, which combine multiple repair actions.

Table 6 RQ3: Repair capability of tools by CWE categories.

The "E2E tested" column denotes the number of End-to-End tested patches across the tools, while the numbers on each of the tool columns denote the correct patches by the tool. The symbol "✓" means that the tool generated some correct patches (the number of correct patches is marked by the followed number). The symbol "❍" means that the tool generated E2E tested patches, but all of them are incorrect. The symbol "✗" indicates that there is no E2E tested patch generated by a tool for a CWE category at all. Included CWE categories are the only categories that cover at least one result from one of the tools.

CWE	Keyword	E2E tested	ARJA	Cardumen	GenProg-A	jGenProg	jKali	jMutRepair	Kali-A	RSRepair-A	SeqTrans	TBar
22	Path Traversal	5	✗	❍	✗	❍	❍	❍	✗	✗	✗	❍
79	XSS	11	✓(1)	✓(1)	❍	✗	✗	✗	❍	✓(1)	✓(1)	✓(1)
200	Info. Exposure	9	✓(1)	❍	✓(1)	❍	✓(1)	✓(1)	✓(1)	✓(1)	✗	✓(1)
310	Crypto. Issues	2	✓(1)	✗	✗	✗	✗	✗	✗	✓(1)	✗	✗
502	Data Deserializ.	8	❍	❍	❍	❍	❍	❍	❍	✗	✗	❍
522	Vuln. Credential	1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓(1)
835	Infinite Loop	24	✓(1)	✓(1)	✓(1)	✓(1)	✓(1)	❍	✓(1)	✓(1)	✗	✓(1)
-	Not Classified	9	✓(1)	❍	✗	✓(1)	✓(1)	✗	✓(1)	✓(1)	✓(1)	✓(1)

The starting point for analyzing the root causes of such a lackluster performance is an interesting observation by Iannone et al. [25]: the most frequent net

Table 7 RQ3: Distribution of the repair actions in the Defects4J and ExtraVul datasets. *The emphasized values in ExtraVul column indicate that ExtraVul contains a bigger portion of the corresponding repair action when compared to Defects4J.*

Category	Abbr.	Repair Action	Defects4J	ExtraVul
Conditional Block	CB1	Addition/Removal of conditional branch	56.2%	36.36%
	CB2	Change of conditional expression	21.3%	10.6%
	CB3	Change of keyword for conditional stmt.	0.0%	**0.51%**
Exception Handler	EH1	Addition of `throw` stmt.	9.6%	**15.5%**
	EH2	Addition/Removal of `try-catch` block	1.5%	**6.1%**
Method Call	MC1	Addition/Removal of method call	65.3%	**73.7%**
	MC2	Change of arguments of method call	14.4%	**18.2%**
	MC3	Change of name of method call	12.8%	4.6%
Return Statement	RS1	Addition/Removal of `return` stmt.	34.9%	18.2%
	RS2	Change of return value	20.3%	11.1%
Loop	LP1	Addition/Removal of loop	11.1%	4%
	LP2	Addition of `break`/`continue` stmt.	0.0%	**2%**
	LP3	Change of iteration variable	0.3%	**0.5%**
Object Instantiation	OI1	Addition/Removal of object instantiation	3.3%	**23.2%**
	OI2	Change of arguments of constructor	1.8%	**2%**
	OI3	Change of constructor type	1.8%	**2.5%**
Method Definition	MD	Addition/Removal of method definition	6.8%	**13.6%**
Type	CT	Change of type extension	0.5%	**1.5%**
Assignment	AS1	Addition/Removal of assignment stmt.	39.0%	27.3%
	AS2	Change of assignment expression	14.9%	9.6%
Variable	VR1	Addition/Removal of variable	30.1%	17.7%
	VR2	Change of variable type	2.5%	**3.5%**

effect of a Correct patch is to change or add new lines. For example, a simple solution to sanitize an external input is to add missing checks. Hence, the APR tools that generate patches by adding code might perform better than APR tools removing code for those types of vulnerabilities. The preliminary insights from the failed attempts of APR tools, which we present in RQ1 and RQ2, also provide several hints for analyzing the characteristics of vulnerability fixes that the current APR tools lack. For example, patches for vulnerabilities might be simple, however, they require knowledge of the vocabulary specific to security-related ingredients.

Therefore, starting with the Vul4J dataset, we analyzed the distribution of E2E tested and Correct patches across the different CWE categories. Unfortunately, the tools could not generate any E2E tested patches for 18 CWE categories (72% of the total number of CWE categories in Vul4J). Table 6 shows the performance among the CWE categories for which APR tools can generate an E2E tested patch. The APR tools mostly generate patches that can be successfully E2E tested for Infinite Loop (CWE-835), Information Exposure (CWE-200), Data Deserialization (CWE-502), and XSS (CWE-79). Instead, Cryptographic Issues (CWE-310) and Vulnerable Credential (CWE-522) have the least number of E2E tested patches generated. This distribution in repair results shows that there might be some bugs and fix patterns that the APR tools have not assessed.

In line with this, we compared in detail the characteristics of the *repair patterns used by developers* in real-world projects from ExtraVul against the repair patterns in Defect4J. The goal is to understand what the APR tools would have missed in order to repair vulnerabilities. To do so, we extracted the repair patterns in Defects4J from the work by Sobreira et al. [71].

ExtraVul and Defects4J share a similar distribution in terms of repair actions. Table 7 shows the distribution of atomic repair actions in both datasets. Overall, ExtraVul and Defects4J have equivalent distributions for repair actions. In par-

ticular, CB1, MC1, and AS1 are the top 3 repair actions used by security patches and general bug patches. Some repair actions are only present in ExtraVul such as *LP2 – Addition of `break/continue` statement*. However, we observed these repair actions are uncommon.

More than 75% of fix patterns in ExtraVul do not exist in Defects4J or carry security-specific traits. Table 8 and Table 9 show the prevalence of the fix patterns in ExtraVul. These patterns are categorized into six groups: Infinite Loop Handling, Object Instantiation, User's Permission Management, External Input Validating and Handling, Configuration, and the Others. In contrast to the repair actions, we noticed that there are significant differences between security patches and general bug patches regarding the repair patterns. In particular, many repair patterns contain combinations of basic actions that are entirely new in ExtraVul and whose purposes are not present in Defect4J. Furthermore, many repair patterns in ExtraVul have their own traits in terms of source code components when compared to general bugs. For instance, the repair pattern *Addition of method call to sanitize external input*, which is based on the repair action MC1 (Method Call Addition/Removal), often contains unique keywords related to input validation, such as `encode`, `normalize`, `trim`, etc.

Main findings for RQ3: While security patches in ExtraVul are indeed short in terms of lines of code, they contain instantiations of fixing patterns that are not present in Defects4J. This suggests that the repair strategies in the studied APR tools might be insufficient. Also, the fixing code contains keywords that are not used in the rest of the program.

8.1 Discussion on the repair patterns for vulnerabilities

In this subsection, we describe in detail the vulnerability fix patterns that we have compiled from the ExtraVul dataset. We also distill some insights into these patterns and provided suggestions for the repair community on designing APR tools for fixing vulnerabilities specifically.

Infinite loop handling. The repair patterns in this category fix the CVEs in the CWE-835 (Infinite Loop). Generated patches are either changing the loop condition (LP3); or adding `break/continue/return` statements to the exit loop (LP2, EH1). These repair patterns utilize *Loop* repair actions, which apparently are rarer for vulnerabilities compared to general bugs (6.5% in ExtraVul and 11.4% in Defects4J, see Table 7). However, we observed that the pattern of adding `break/continue/return` statements to exit the loop is not present in Defects4J, while it is actually effective in removing the *Infinite Loop* vulnerabilities in the ExtraVul dataset (even applying the patch breaks the functionalities, the vulnerability is solved). Researchers, therefore, should consider including this repair pattern when building an automatic vulnerability repair tool.

Object Instantiation. The repair patterns in this category are unique to vulnerabilities. These patterns use a secure class (VR1, OI2), add a method call to avoid deserialization (MC1), or instantiate a secure parser for `YAML` file (OI1) to fix the CVEs in CWE-352 (Cross-Site Request Forgery), CWE-918 (Server-Side

Table 8 RQ3: Prevalent Fixing Patterns in the ExtraVul dataset.

The "New" column indicates that the repair pattern exists only in ExtraVul but not in Defects4J. The "#" column indicates the total number of vulnerability patches we found in ExtraVul which used the repair pattern. "Stmt." stands for "statement", "MC" stands for "Method Call".

Repair Pattern	Observation/Explanation	CWE Mapping	Rep. Vulnerability	New	#
Infinite Loop Handling					
Add `break/continue/throw` stmt. to exit loop (LP2, EH1)	Avoid an infinite loop i.e., DDoS attack	CWE-835	CVE-2019-12402	✓	5
Change loop header/termination condition (LP3)	Avoid integer overflow causing infinite loop	CWE-835	CVE-2018-1324	✓	2
Object Instantiation					
Use secure class for object instantiation (VR1 + OI2)	E.g., Replace `Random` by `SecureRandom`	CWE-352, CWE-918	CVE-2018-1272	✓	4
Add MC to avoid deserialization of untrusted data (MC1)	E.g., Avoid instantiating object of `Void.class`	CWE-502	CVE-2017-1000355	✓	6
Change object instantiation to secure parser (OI1)	`SnakeYAML` Parser allow executing arbitrary code from input `YAML` file with default config	CWE-502	CVE-2017-1000207	✓	2
User's Permission Management					
Add MC to change user's permission (MC1)	E.g., Restrict user's permission to read-only	-	PDFBOX-3341	✓	5
Add MC to check user's authorization (MC1)	Check permission of the user executing current code	CWE-287, CWE-532	CVE-2018-1000089	✓	12
Configuration					
Add MC to configure DOM/SAX Parser (with `try-catch` block) (MC1 (+ EH2)))	Insecure/secure features of DOM/SAX Parser should be disabled/enabled to avoid XXE vuls	CWE-74, CWE-611, CWE-918	CVE-2016-7051	✓	20

Table 9 RQ3: Prevalent Fixing Patterns in the Vul4J dataset cntd.

The "New" column indicates that the repair pattern exists only in Vul4J but not in Defects4J. The "#" column indicates the total number of vulnerability patches we found in Vul4J which used the repair pattern. "Stmt." stands for "statement", "MC" stands for "Method Call".

Repair Pattern	**Observation/Explanation**	**CWE Mapping**	**Rep. Vulnerability**	**New**	**#**
External Input Validating and Handling					
Add conditional branch with exception throwing (CB1 + EH1)	The conditional expression is new from the code base, e.g., check null-byte '`\0`'	CWE-20, CWE-22, CWE-264, CWE-269, CWE-770, CWE-835	CVE-2013-2186		20
Add conditional branch with return stmt. (CB1 + RS1)	The return value is `null`, `false`, `-1`, etc.	CWE-20, CWE-287, CWE-310, CWE-522	CVE-2019-3775		16
Change Regular Expression (AS2)	Improve the rule for validating input	CWE-20	CVE-2016-4465	✓	5
Add MC to sanitize external input (MC1)	The MC name contains one of these keywords: `encode, normalize, replace, trim`	CWE-20, CWE-22, CWE-74, CWE-79, CWE-284	CVE-2013-4378	✓	24
Add/Remove '/' to the system path or URI path (AS2)	E.g., Add '/' at the end of the directory path to avoid path traversal	CWE-22, CWE-601	APACHE-COMMONS-001	✓	3
Others					
Remove code	Avoid exposure of data in Web UI or API	CWE-200, CWE-863	CVE-2018-1192		14
Move code	-	CWE-79, CWE-332	CVE-2019-3795		4

Request Forgery), and CWE-502 (Deserialization of Untrusted Data). Out of 142 vulnerabilities which patch we can map to Table 8 and 9, this group covers twelve vulnerabilities (8.45%). While these patterns are similar to *Wrong reference* and *Copy/Paste* patterns from [71], they are more specific for vulnerabilities. The uniqueness of vulnerabilities and their fixing patterns are the main reasons the APR tools we evaluated failed to fix them.

User's Permission Management. The repair patterns in this category account for 17 vulnerabilities (11.97%) of 142 that we could map. These repair patterns are grouped into two more specific patterns: adding method call (MC1) to (1) change the user's permission and (2) check the user's authorization. We observed that these repair patterns only exist for vulnerabilities, even though they use the repair action MC1, which is also used in a bunch of patches in Defects4J (65.3%). Although these patterns seem 'simple', they are once again specific to the system's access control features, making it harder for APR tools to generate them.

Configuration. This repair pattern was used to fix 20 vulnerabilities out of 142 (14.08%). This pattern also uses repair action MC1, with an additional try-catch block (EH2) in some cases. However, adding a method call in this pattern is specifically configuring a DOM/SAX parser. This pattern of changing configuration only appears to fix a vulnerability, as the functionality stays the same. The pattern is mostly specific to the DOM/SAX parser they configure, and the added method call is (a "new" method call) not present in the codebase before. This makes the process of producing the right patch harder because the tool cannot just uses the *Copy/Paste* pattern. APR tools, therefore, should learn different kinds of parsers and their secure configuration combination to, in the end, be able to prevent the parsers from being prone to vulnerabilities.

External Input Validating and Handling. This category of patterns covers the most vulnerabilities in our dataset (68 out of 142 from ExtraVul, 47.89%). The first two patterns are part of *Conditional Block* pattern from [71]: adding conditional branch with (1) exception throwing and (2) return statement. These patterns utilize CB1, which is used in many patches in the Defects4J dataset. The following patterns are changing Regular Expressions and adding/removing '/' to the system/ URI path (AS2). Although these patterns are related to *Expression Fix (RegEx)* for general bugs, they are specifically for RegEx and paths, which can be exploited as a vulnerability. For these patterns, the APR tools are stuck in understanding the right RegEx to add/remove a '/' in the variable to make it not vulnerable. The last pattern is similar to other groups that utilize MC1, but this pattern is specifically used to sanitize external input. Implementing this can be challenging for APR tools as they must understand which external input can be malicious and devise the right method call (and its parameter) to sanitize it.

Others. This group covers 18 out of 142 vulnerabilities we mapped to the Table 8 and 9 (12.68%). In this group, the pattern *Move code* was also used for fixing general bugs, so it makes sense that traditional APR tools can generate this kind of patch. The other pattern, *Remove code*, was used to remove the data exposure method (CWE-200), and has actually been used extensively by the APR tools to remove other vulnerabilities, e.g., to remove an infinite loop (CWE-835). However, researchers should be careful in using this pattern because it might break the functionality of the program while removing a vulnerability.

Unlisted. We can only map 142 out of 198 vulnerabilities that we manually analyzed in Table 8 and 9. The reason is that the patches for the remaining 56

vulnerabilities (28.28%) are too specific to the vulnerabilities that we could not generalize them to the repair patterns. This specificity is also why most APR tools failed to fix them.

9 Threats to Validity

Our pipeline may influence the search strategy of some tools. ARJA, GenProg-A, Kali-A, and RSRepair-A are designed to automatically use a subset of the available tests instead of the whole test suite. Hence, using fewer test cases will reduce complexity and might allow the tool to expand the search space. However, we decided to force those tools to use the whole test suite instead. Even though it might not be natural for them, using the whole test suite will completely ensure that a patch will not break the program.

We only considered ten repair tools in our study. Including other APR tools in our pipeline requires their source code to be open and extensible. Our developed pipeline is flexible, in which only minimal configuration is needed to integrate more tools. Among the selected tools, we acknowledge that several of them can have varying results on different executions, e.g., GenProg-A and jGenProg which use genetic programming. However, we doubt that this will change the overall evaluation results.

We only analyzed three datasets for Java. Our study is based on Vul4J, ExtraVul and Defect4J. Therefore, the findings might not generalize to other datasets. In the case of vulnerabilities, Vul4J is the only suitable dataset at the moment, to the best of our knowledge. ExtraVul is similar to Vul4J with more vulnerabilities. Defects4J is a well-known dataset that is extensively used by the research community.

Our evaluation setup may influence the result. The results may be influenced by the execution environment, e.g., executing processors, and virtual memory availability. In our experiments, we set the time budget of four hours for a repair execution, which is based on what has been done in other studies [46]. However, most repair tools (except GenProg-A and RSRepair-A) in our evaluation actually take less than 30 minutes on average to generate a patch, which is significantly lower than the allocated time budget.

The Vul4J benchmark used in our study may not cover all types of vulnerabilities that affect Java projects. The vulnerabilities are curated from the public Snyk vulnerability database and the NVD database. Six out of eight CWE categories in our study are listed in the OWASP Top 10 [3]. Hence, we believe that the vulnerabilities used in this study are sufficiently considerable and reflect the real severe vulnerabilities in the wild.

10 Conclusion and Future Work

In this paper, we have presented the first thorough evaluation of nine test-based repair tools and one security-specific repair tool for Java and their potential to repair real-world security vulnerabilities. Although our results seem to paint a somewhat gloomy picture, we believe that we have provided interesting insights and actionable suggestions (especially in RQ3) for the program repair research

community. The study is an incentive to perform further research in order to optimize APR tools for vulnerabilities specifically. From a methodological perspective, we have also outlined the necessity of more research in order to support the (possibly automated) validation of the patches for what concerns their correctness. Today, this still represents a major obstacle in the rigorous evaluation of the tools' performance, as manual validation is a very time-consuming effort. It is left for our future work to further extend our benchmarking platform, both in terms of tools and additional vulnerabilities.

A conclusion of general interest from this paper, consistent with the previous findings by Durieux et al. [16] on general bug repair, is that performing a manual validation of a sample of results obtained by automatic means is mandatory for assessing the confidence of the quality of an APR technique. We also make the artifact of our evaluation available in a publicly-accessible repository [1] to support the Open Science movement.

Declarations

Data Availability

To support the openness in science, we have made the scripts and results of our evaluation publicly available at:
`https://github.com/tuhh-softsec/APR4Vul`.

Funding and/or Conflicts of interests/Competing interests.

Funding. This work is partly funded by EU grants No. 952647 (AssureMOSS) and No. 101120393 (Sec4AI4Sec).

Financial/Non-financial Interest. All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Conflict of Interests. The authors declare that they have no conflict of interest.

Employment

Quang-Cuong Bui is employed by Hamburg University of Technology, Hamburg, Germany. Ranindya Paramitha is employed by the University of Trento, Trento, Italy. Duc-Ly Vu is employed by University of Information Technology, Vietnam National University, Ho Chi Minh City, Vietnam. Fabio Massacci is employed by the University of Trento, Trento, Italy and Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. Riccardo Scandariato is employed by Hamburg University of Technology, Hamburg, Germany.

Acknowledgments

This work is partly funded by EU grants No. 952647 (AssureMOSS) and No. 101120393 (Sec4AI4Sec).

CRediT statements. *Conceptualization:* QCB, DLV, FM, RS; *Methodology:* QCB, DLV; *Software:* QCB; *Validation:* QCB, RP, DLV; *Formal analysis:* RP, FM; *Investigation:* QCB, RP, DLV; *Data Curation:* QCB, RP, DLV; *Writing - Original Draft:* QCB, RP, DLV, FM, RS; *Writing - Review & Editing:* QCB, RP, DLV, FM, RS; *Visualization:* QCB, RP; *Supervision:* FM, RS; *Project administration:* FM, RS; *Funding acquisition:* FM, RS;

References

1. Apr4vul: An empirical study of automatic program repair techniques on real-world java vulnerabilities. https://github.com/tuhh-softsec/APR4Vul. (The artifact of our evaluation study)
2. The nist software assurance reference dataset project. https://samate.nist.gov/SARD/. Accessed: 2022-01-25
3. Owasp top 10 - 2021. https://owasp.org/Top10/. Accessed: 2022-03-06
4. Abadi, A., Ettinger, R., Feldman, Y.A., Shomrat, M.: Automatically fixing security vulnerabilities in java code. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pp. 3–4 (2011). DOI https://doi.org/10.1145/2048147.2048149
5. Agresti, A., Franklin, C., Klingenberg, B.: Statistics: The Art and Science of Learning from Data. Pearson Education (2016). URL https://books.google.it/books?id=Vql5CwAAQBAJ
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23, pp. 171–177. Springer (2011)
7. Black, P.E., Black, P.E.: Juliet 1.3 test suite: Changes from 1.2. US Department of Commerce, National Institute of Standards and Technology (2018)
8. Bui, Q.C., Scandariato, R., Ferreyra, N.E.D.: Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In: International Conference on Mining Software Repositories (MSR) (2022)
9. Caswell, B.: Cyber grand challenge corpus. http://www.lungetech.com/cgc-corpus/. Accessed: 2023-01-27
10. Chen, L., Pei, Y., Furia, C.A.: Contract-based program repair without the contracts. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 637–647. IEEE (2017). DOI https://doi.org/10.1109/ASE.2017.8115674
11. Chen, Z., Kommrusch, S., Monperrus, M.: Neural transfer learning for repairing security vulnerabilities in c code. IEEE Transactions on Software Engineering **49**(1), 147–165 (2022)
12. Chi, J., Qu, Y., Liu, T., Zheng, Q., Yin, H.: Seqtrans: Automatic vulnerability fix via sequence to sequence learning. IEEE Transactions on Software Engineering (2022). DOI https://doi.org/10.1109/TSE.2022.3156637
13. Dashevskyi, S., Brucker, A.D., Massacci, F.: A screening test for disclosed vulnerabilities in foss components. IEEE Transactions on Software Engineering **45**(10), 945–966 (2018). DOI https://doi.org/10.1109/TSE.2018.2816033
14. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
15. Durieux, T., Cornu, B., Seinturier, L., Monperrus, M.: Dynamic patch generation for null pointer exceptions using metaprogramming. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 349–358. IEEE (2017). DOI https://doi.org/10.1109/SANER.2017.7884635

16. Durieux, T., Madeiral, F., Martinez, M., Abreu, R.: Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 302–313 (2019). DOI https://doi.org/10.1145/3338906.3338911
17. Durieux, T., Monperrus, M.: Dynamoth: dynamic code synthesis for automatic program repair. In: Proceedings of the 11th International Workshop on Automation of Software Test, pp. 85–91 (2016). DOI https://doi.org/10.1145/2896921.2896931
18. Durieux, T., Monperrus, M.: Introclassjava: A benchmark of 297 small and buggy java programs (2016)
19. Flynn, L., Snavely, W., Kurtz, Z.: Test suites as a source of training data for static analysis alert classifiers. In: 2021 IEEE/ACM International Conference on Automation of Software Test (AST), pp. 100–108. IEEE (2021). DOI https://doi.org/10.1109/AST52587.2021.00019
20. Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V., Phung, D.: Vulrepair: a t5-based automated software vulnerability repair. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 935–947 (2022)
21. Gasiba, T.E., Lechner, U., Pinto-Albuquerque, M., Mendez, D.: Is secure coding education in the industry needed? an investigation through a large scale survey. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pp. 241–252. IEEE (2021)
22. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM **62**(12), 56–65 (2019)
23. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th international conference on software engineering, pp. 12–23 (2018). DOI https://doi.org/10.1145/3180155.3180245
24. Huang, Z., Lie, D., Tan, G., Jaeger, T.: Using safety properties to generate vulnerability patches. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 539–554. IEEE (2019)
25. Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., Palomba, F.: The secret life of software vulnerabilities: A large-scale empirical study. IEEE Transactions on Software Engineering (2022). DOI https://doi.org/10.1109/TSE.2022.3140868
26. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp. 298–309 (2018)
27. Jiang, N., Lutellier, T., Lou, Y., Tan, L., Goldwasser, D., Zhang, X.: Knod: Domain knowledge distilled tree decoder for automated program repair. arXiv preprint arXiv:2302.01857 (2023)
28. Jiang, N., Lutellier, T., Tan, L.: Cure: Code-aware neural machine translation for automatic program repair. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1161–1173. IEEE (2021)
29. Johns, M., Jodeit, M.: Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 523–530. IEEE (2011). DOI https://doi.org/10.1109/ICSTW.2011.32
30. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In: ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440. San Jose, CA, USA (2014). DOI https://doi.org/10.1145/2610384.2628055. Tool demo
31. Kechagia, M., Mechtaev, S., Sarro, F., Harman, M.: Evaluating automatic program repair capabilities to repair api misuses. IEEE Transactions on Software Engineering (2021). DOI https://doi.org/10.1109/TSE.2021.3067156
32. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 802–811. IEEE (2013). DOI https://doi.org/10.1109/ICSE.2013.6606626
33. Kim, J., Kim, S.: Automatic patch generation with context-based change application. Empirical Software Engineering **24**(6), 4071–4106 (2019). DOI https://doi.org/10.1007/s10664-019-09742-5
34. Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Klein, J., Monperrus, M., Le Traon, Y.: Fixminer: Mining relevant fix patterns for automated program repair. Empirical Software Engineering **25**, 1980–2024 (2020)

35. LE, D.X.B.: Overfitting in automated program repair: Challenges and solutions (2018)
36. Le, X.B.D., Lo, D., Le Goues, C.: History driven program repair. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol. 1, pp. 213–224. IEEE (2016). DOI https://doi.org/10.1109/SANER.2016.76
37. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 3–13. IEEE (2012). DOI https://doi.org/10.1109/ICSE.2012.6227211
38. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. Ieee transactions on software engineering **38**(1), 54–72 (2011). DOI https://doi.org/10.1109/TSE.2011.104
39. Li, Y., Wang, S., Nguyen, T.N.: Dlfix: Context-based code transformation learning for automated program repair. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 602–614 (2020)
40. Li, Y., Wang, S., Nguyen, T.N.: Dear: A novel deep learning-based approach for automated program repair. In: Proceedings of the 44th International Conference on Software Engineering, pp. 511–523 (2022)
41. Lima, R., Ferreira, J.F., Mendes, A.: Automatic repair of java code with timing side-channel vulnerabilities. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 1–8. IEEE (2021)
42. Lin, D., Koppel, J., Chen, A., Solar-Lezama, A.: 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, pp. 55–56 (2017). DOI https://doi.org/10.1145/3135932.3135941
43. Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Le Traon, Y.: You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE conference on software testing, validation and verification (ICST), pp. 102–113. IEEE (2019). DOI https://doi.org/10.1109/ICST.2019.00020
44. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1–12. IEEE (2019)
45. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: Tbar: Revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 31–42 (2019). DOI https://doi.org/10.5281/zenodo.3237378
46. Liu, K., Li, L., Koyuncu, A., Kim, D., Liu, Z., Klein, J., Bissyandé, T.F.: A critical review on the evaluation of automated program repair systems. Journal of Systems and Software **171**, 110817 (2021)
47. Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T.F., Kim, D., Wu, P., Klein, J., Mao, X., Traon, Y.L.: On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, p. 615–627. Association for Computing Machinery, New York, NY, USA (2020). DOI https://doi.org/10.1145/3377811.3380338. URL `https://doi.org/10.1145/3377811.3380338`
48. Liu, X., Zhong, H.: Mining stackoverflow for program repair. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER), pp. 118–129. IEEE (2018). DOI https://doi.org/10.1109/SANER.2018.8330202
49. Ma, S., Lo, D., Li, T., Deng, R.H.: Cdrep: Automatic repair of cryptographic misuses in android applications. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 711–722 (2016)
50. Ma, S., Thung, F., Lo, D., Sun, C., Deng, R.H.: Vurle: Automatic vulnerability detection and repair by learning from examples. In: European Symposium on Research in Computer Security, pp. 229–246. Springer (2017). DOI https://doi.org/10.1007/978-3-319-66399-9_13
51. Ma, W., Chen, L., Zhang, X., Zhou, Y., Xu, B.: How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 381–392. IEEE (2017). DOI https://doi.org/10.1109/ICSE.2017.42

52. Madeiral, F., Urli, S., Maia, M., Monperrus, M.: Bears: An extensible java bug benchmark for automatic program repair studies. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 468–478. IEEE (2019). DOI https://doi.org/10.1109/SANER.2019.8667991
53. Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M.: Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empirical Software Engineering **22**(4), 1936–1964 (2017). DOI https://doi.org/10.1007/s10664-016-9470-4
54. Martinez, M., Monperrus, M.: Astor: A program repair library for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 441–444 (2016). DOI https://doi.org/10.1145/2931037.2948705
55. Martinez, M., Monperrus, M.: Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: International Symposium on Search Based Software Engineering, pp. 65–86. Springer (2018). DOI https://doi.org/10.1007/978-3-319-99241-9_3
56. McHugh, M.L.: Interrater reliability: the kappa statistic. Biochemia medica **22**(3), 276–282 (2012)
57. Mesecan, I., Blackwell, D., Clark, D., Cohen, M.B., Petke, J.: Hypergi: automated detection and repair of information flow leakage. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1358–1362. IEEE (2021)
58. Monperrus, M.: The living review on automated program repair. Tech. Rep. hal-01956501, HAL/archives-ouvertes.fr (2018)
59. Monperrus, M., Urli, S., Durieux, T., Martinez, M., Baudry, B., Seinturier, L.: Repairnator patches programs automatically. Ubiquity **2019**(July) (2019). DOI 10.1145/3349589. URL `https://doi.org/10.1145/3349589`
60. Neto, E.C., Da Costa, D.A., Kulesza, U.: The impact of refactoring changes on the szz algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 380–390. IEEE (2018). DOI https://doi.org/10.1109/SANER.2018.8330225
61. Nguyen, V.H., Dashevskyi, S., Massacci, F.: An automatic method for assessing the versions affected by a vulnerability. Empirical Software Engineering **21**(6), 2268–2297 (2016). DOI https://doi.org/10.1007/s10664-015-9408-2
62. Papotti, A., Paramitha, R., Massacci, F.: On the acceptance by code reviewers of candidate security patches suggested by automated program repair tools. arXiv preprint arXiv:2209.07211 (2022)
63. Pashchenko, I., Vu, D.L., Massacci, F.: A qualitative study of dependency management and its security implications. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1513–1531 (2020). DOI https://doi.org/10.1145/3372297.3417232
64. Pinconschi, E., Abreu, R., Adão, P.: A comparative study of automatic program repair techniques for security vulnerabilities. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), pp. 196–207. IEEE (2021). DOI https://doi.org/10.1109/ISSRE52982.2021.00031
65. Pinconschi, E., Bui, Q.C., Abreu, R., Adão, P., Scandariato, R.: Maestro: A platform for benchmarking automatic program repair tools on software vulnerabilities. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 789–792 (2022)
66. Pittet, S.: The different types of software testing. `https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing`. Accessed: 2022-03-12
67. Ponta, S.E., Plate, H., Sabetta, A., Bezzi, M., Dangremont, C.: A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 383–387. IEEE (2019). DOI https://doi.org/10.1109/MSR.2019.00064
68. Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., Prasad, M.R.: Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 15th international conference on mining software repositories, pp. 10–13 (2018). DOI https://doi.org/10.1145/3196398.3196473
69. Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R.: Elixir: Effective object-oriented program repair. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 648–659. IEEE (2017). DOI https://doi.org/10.1109/ASE.2017.8115675

70. Saha, S., et al.: Harnessing evolution for multi-hunk program repair. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 13–24. IEEE (2019). DOI https://doi.org/10.1109/ICSE.2019.00020
71. Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., de Almeida Maia, M.: Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 130–140. IEEE (2018). DOI https://doi.org/10.1109/SANER.2018.8330203
72. Vanciu, R., Abi-Antoun, M.: Finding architectural flaws using constraints. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 334–344. IEEE (2013). DOI https://doi.org/10.1109/ASE.2013.6693092
73. Villanueva, O.M., Trujillo, L., Hernandez, D.E.: Novelty search for automatic bug repair. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp. 1021–1028 (2020). DOI https://doi.org/10.1145/3377930.3389845
74. Vu, D.L., Pashchenko, I., Massacci, F.: Please hold on: more time= more patches? automated program repair as anytime algorithms. In: 2021 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 9–10. IEEE (2021). DOI https://doi.org/10.1109/APR52552.2021.00009
75. Wagner, A., Sametinger, J.: Using the juliet test suite to compare static security scanners. In: 2014 11th International Conference on Security and Cryptography (SECRYPT), pp. 1–9. IEEE (2014)
76. Wang, W., Meng, Z., Wang, Z., Liu, S., Hao, J.: Loopfix: an approach to automatic repair of buggy loops. Journal of Systems and Software **156**, 100–112 (2019). DOI https://doi.org/10.1016/j.jss.2019.06.076
77. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.C.: Context-aware patch generation for better automated program repair. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 1–11. IEEE (2018). DOI https://doi.org/10.1145/3180155.3180233
78. White, M., Tufano, M., Martinez, M., Monperrus, M., Poshyvanyk, D.: Sorting and transforming program repair ingredients via deep learning code similarities. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 479–490. IEEE (2019). DOI https://doi.org/10.1109/SANER.2019.8668043
79. Wong, C.P., Santiesteban, P., Kästner, C., Le Goues, C.: Varfix: balancing edit expressiveness and search effectiveness in automated program repair. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 354–366 (2021). DOI https://doi.org/10.1145/3468264.3468600
80. Xin, Q., Reiss, S.P.: Leveraging syntax-related code for automated program repair. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 660–670. IEEE (2017). DOI https://doi.org/10.1109/ASE.2017.8115676
81. Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 416–426. IEEE (2017). DOI https://doi.org/10.1109/ICSE.2017.45
82. Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M.: Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering **43**(1), 34–55 (2016). DOI https://doi.org/10.1109/TSE.2016.2560811
83. Ye, H., Martinez, M., Monperrus, M.: Neural program repair with execution-based backpropagation. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1506–1518 (2022)
84. Yuan, Y., Banzhaf, W.: Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Transactions on software engineering **46**(10), 1040–1067 (2018). DOI https://doi.org/10.1109/TSE.2018.2874648
85. Zhang, Y., Kabir, M., Xiao, Y., Meng, N., et al.: Data-driven vulnerability detection and repair in java code. arXiv preprint arXiv:2102.06994 (2021). DOI https://doi.org/10.48550/arXiv.2102.06994
86. Zhou, Z., Bo, L., Wu, X., Sun, X., Zhang, T., Li, B., Zhang, J., Cao, S.: Spvf: security property assisted vulnerability fixing via attention-based models. Empirical Software Engineering **27**(7), 171 (2022)
87. Zhu, Q., Sun, Z., Xiao, Y.a., Zhang, W., Yuan, K., Xiong, Y., Zhang, L.: A syntax-guided edit decoder for neural program repair. In: Proceedings of the 29th ACM Joint Meeting

on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 341–353 (2021)