10-2024

An empirical study of automatic program repair techniques for injection vulnerabilities

Tingwei ZHU

Tongtong XU

Kui LIU

Jiayuan ZHOU

Xing HU

See next page for additional authors

Citation

ZHU, Tingwei; XU, Tongtong; LIU, Kui; ZHOU, Jiayuan; HU, Xing; XIA, Xin; ZHANG, Tian; and David LO. An empirical study of automatic program repair techniques for injection vulnerabilities. (2024). *Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME 2024): Flagstaff, AZ, USA, October 6-11*. 25-37.

Available at: https://ink.library.smu.edu.sg/sis_research/9888

An Empirical Study of Automatic Program Repair Techniques for Injection Vulnerabilities

Tingwei Zhu[†], Tongtong Xu[‡], Kui Liu[‡], Jiayuan Zhou[§], Xing Hu[¶], Xin Xia[‡], Tian Zhang[†*], David Lo[‖]
[†]State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
[‡]Huawei Technologies Co., Ltd., China
[§]Huawei Technologies Co., Ltd., Canada
[¶]Zhejiang University, Hangzhou, China
[‖]Singapore Management University, Singapore

2024 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3503-9568-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSME58944.2024.00014

***Abstract*—Injection vulnerabilities are among the most serious and dangerous security defects, as they can be exploited by attackers to inject malicious inputs and carry out cybercrimes. Timely fixing of injection vulnerabilities is crucial. However, manual repairs of injection vulnerabilities often require specialized knowledge and are prone to errors, posing a challenge and a heavy burden on developers. In recent years, Automated Program Repair (APR) techniques have shown promising momentum in automatically fixing general defects. Yet, there has been no research on how APR techniques perform in repairing injection vulnerabilities. Therefore, in this paper, we conduct an empirical study. We first construct a benchmark for injection vulnerability repair and evaluate several representative state-of-the-art APR approaches on this benchmark. The results show that existing APR tools do not adequately support the repair of injection vulnerabilities. To investigate the underlying reasons, we compare the characteristics of patches for injection vulnerabilities and general defects, and explore whether the plastic surgery hypothesis widely used in APR still holds for injection vulnerabilities. The results reveal that fixing injection vulnerabilities is more complex than fixing general defects due to significant differences in the characteristics of their patches. Additionally, the support for the plastic surgery hypothesis is much lower in the context of injection vulnerability repair. We also analyzed developers' intentions when fixing injection vulnerabilities. Finally, we summarize the implications and point out potential research directions for injection vulnerability repair.**

***Index Terms*—Injection vulnerability, automatic program repair, empirical study**

I. Introduction

Cybersecurity Ventures reported: "*Cybercrime is predicted to cost the world $9.5 trillion USD in 2024. If it were measured as a country, then cybercrime would be the world's third-largest economy after the U.S. and China*" [1]. Security defects, namely vulnerabilities, are prevalent in modern software system evolution [2]. Attackers can exploit unresolved vulnerabilities to conduct malicious cybercrimes, putting millions of users around the world at huge risk [3].

One of the most critical vulnerabilities is the injection vulnerability, which has been ranked in the top 3 most dangerous vulnerability types in the past three years [4]. Injection vulnerabilities allow an attacker to inject untrusted input into a program or database, causing it to execute unintended commands or access unauthorized data. Recently, a high-severity injection vulnerability named "Log4Shell" [5] was discovered in the popular Apache Log4j Java logging framework. The vulnerability allows attackers to execute arbitrary Java code on a server or leak sensitive information. Log4Shell affected 93% of enterprise cloud environments [6] and is arguably recognized as the most severe vulnerability ever [7]. Therefore, timely remediation of disclosed vulnerabilities constitutes a crucial aspect of secure software development, which mitigates potential security threats and safeguards system integrity.

Manually repairing vulnerabilities can be challenging, especially when it comes to complex ones such as injection vulnerabilities. Studies report that fixing vulnerabilities is a time-consuming task and it takes 52 days on average to release a patch [8], [9]. Moreover, the process of patching a vulnerability can be convoluted and iterative. In some cases, it may require several rounds of testing and revision. 11.5% of vulnerabilities are reported to be fixed multiple times because the initial patch introduces an error or is incomplete [10]. Taking Log4Shell as an example, the developers of Apache Log4j committed patches a total of four times, as the patches continually introduced new vulnerabilities. The final patch ends with the removal of related functionality (refer to § II-A). This example highlights that manually repairing injection vulnerabilities is an error-prone task and an extremely heavy burden for developers. While in industry, once an injection vulnerability is disclosed, software maintainers are required to patch the vulnerable program timely to prevent injection attacks. Therefore, it is highly necessary to automatically repair injection vulnerabilities efficiently and with high equality.

According to our literature review, currently, there are no methods specifically proposed for repairing injection vulnerabilities. Although a previous study [11] has shown that some research focusing on specific types of injection vulnerabilities, such as SQL injection and XSS injection, the overall effectiveness has not been well evaluated. Even worse, no method has been proposed to address deserialization vulnerabilities like Log4Shell. Hence, the automatic repair of injection vulnerabilities has not been fully researched and remains an unresolved research problem. In recent years, Automated Program Repair (APR) techniques have shown promising momentum

* Corresponding author.

2576-3148/24/$31.00 ©2024 IEEE
DOI 10.1109/ICSME58944.2024.00014

in liberating developers from the heavy burden of manually repairing software defects [12], [13], [14]. Considering that vulnerabilities are security defects, vulnerability repair can be seen as a special scenario of program repair. Therefore, it leads us to investigate *whether the APR methods for general defects can be directly applied to repair injection vulnerabilities. If not, what are the challenges that these methods would face?*

For this purpose, we conduct a systematic empirical study in this paper. Due to the lack of datasets for evaluation, we first create a benchmark for injection vulnerability repair from the National Vulnerability Database (NVD) [15] and Mitre CVE database [16], namely InjectionVul4J, consisting of 123 Java injection vulnerabilities and their associated developer-written patches from 65 real-world open-source projects. Then, we select three representative APR tools which demonstrate state-of-the-art bug-fixing performance from approaches that meet the task scenario conditions, and evaluate their effectiveness on InjectionVul4J. The results reveal that none of them can generate fully correct patches for any of the injection vulnerabilities. This motivates us to explore the reasons why APR tools perform well in fixing general defects but poorly in fixing injection vulnerabilities by comparing the characteristics of fixing injection vulnerabilities and general defects.

On the one hand, we define several metrics to characterize the code revisions made by the patches. Since the investigated approaches are all originally evaluated on a widely used benchmark Defects4J [17], we propose to compare the patches in InjectionVul4J and Defects4J to identify their differences. The results indicate that the patches in InjectionVul4J are more complex compared to those in Defects4J. They require more lines of code changes, are distributed across more methods, and involve revisions in multiple languages and configurations. Developers are more accustomed to creating new methods for revisions and rarely provide exploitable tests. On the other hand, we validate whether the *Plastic Surgery Hypothesis* [18], a well-known insight for APR which states that the code ingredients to fix a bug usually already exist within the same project, still holds for injection vulnerability repair. It turns out that only 13.8% of vulnerabilities in InjectionVul4J contain statements that can be identified in the project, which is far lower than the proportion for general defects.

Additionally, we further analyze and categorize the intentions behind developer-written patches for injection vulnerabilities. Finally, to facilitate future research, we summarize the insights obtained from the empirical study and propose directions for future improvements. In summary, this paper makes the following contributions:

- We create a benchmark named InjectionVul4J for injection vulnerability repair[1].
- We conduct the first empirical study to evaluate the effectiveness of representative APR approaches for injection vulnerability repair.
- We compare the characteristic differences of patches for injection vulnerability repair and general defect repair.
- We are the first to investigate the degree of support for the plastic surgery hypothesis in the context of injection vulnerability repair.
- We provide guidance for the future development of injection vulnerability repair.

[1]https://anonymous.4open.science/r/InjectionVul4J

II. Motivation & Background

In this section, we first introduce the process of manually patching the Log4Shell vulnerability and analyze the challenges involved. After that, we present the concepts related to vulnerabilities that appear in this paper and describe the scope of injection vulnerability studied in this paper.

A. Bird's Eye View on Patching Log4Shell

The Log4Shell vulnerability occurred in Apache Log4j2, with the root cause being the deserialization of malicious input by the `JndiLookup` method without proper validation. To easily understand Log4Shell, we use the terms *source* and *sink* from taint analysis to describe the attack process. In this context, "source" represents the entrance where an attacker can inject malicious input, while "sink" represents the assailable point in the program, i.e., `Runtime.exec()`. In a vulnerable program, there exists a path where the sink is reachable from the source. To repair it, developers need to provide patches that block the path as a defensive implementation.

To repair the Log4Shell vulnerability, developers submitted four patches to prevent two sinks in the program. Figure 1 illustrates the main changes in each patch. The first patch (Figure 1(a)) introduced a two-stage defense. Initially, the vulnerable `JNDI.lookup` function was disabled, but this defense can be bypassed by setting `lookups` to `true`. Thus, a second defense was constructed by adding a whitelist validation using a newly added class. However, this patch was still vulnerable as attackers could inject malicious code by exploiting the missing `return` statement to bypass the validation. So the second patch (Figure 1(b)) added a `return null;` statement to enhance whitelist validation.

The first two patches resolved injection threats in Sink No.1, but Sink No.2 remained vulnerable to malicious input via `JMSManager.lookup()`. To address Sink No.2, the third patch (shown in Figure 1(c)) roughly disabled the function by default. However, an attacker could still bypass the third defense by explicitly setting the `isIsJndiEnabled()` to `true` to enable the `JMSManager.lookup` functionality. Therefore, developers provided the final patch as the fourth defense by completely removing the message lookup functionality, as shown in Figure 1(d) where the call to message lookup has been entirely deleted.

The case of patching Log4Shell vividly illustrates the difficulty of repairing injection vulnerabilities, which requires professional knowledge and substantial costs. Despite this, patches are still prone to errors. Because existing APR techniques have shown promising performance in liberating developers from the heavy burden of manually repairing general defects, we hope to investigate whether they can provide potential insights for repairing injection vulnerabilities. To this

```
    # defense 1st: disable lookup function by default
--- if (!noLookups && config != null) {
+++ if (lookups && config != null) {
    # defense 2nd: add the function of the whitelist
     validation
+++ public synchronized <T> T lookup(final String name){
+++ try{
+++ ...
+++ } catch (URISyntaxException ex) {
+++     // missed return;.
+++ }
    return (T) this.context.lookup(name);
    ...
```

(a) Excerpt 1st patch *2.15.0.rc1*

```
    # renew the defense 2nd
    try{
    ...
    } catch (URISyntaxException ex) {
---     // missed return;
+++     return null;
    }
    return (T) this.context.lookup(name);
    ...
```

(b) Excerpt 2nd patch *2.15.0.rc2*

```
    # defense 3rd: disable JndiEnabled() function by
     default
    ...
+++ if (JndiManager.isIsJndiEnabled()) {
+++ ...
+++ } else {
+++     return null;
    ...
```

(c) Excerpt 3rd patch *LOG4J2-3208*

```
    # defense 4th: disable the lookup function for ever
    ...
--- if (lookups && config != null) {
---     result = new LookupMessagePatternConverter(result
     , config);
--- }
    ...
```

(d) Excerpt 4th patch *2.16.0.rc1*

Fig. 1: The key code modifications in Log4Shell patches.

end, we propose to conduct this empirical study on using APR techniques to repair injection vulnerabilities.

B. Concepts & Scopes

The concepts used in this paper regarding software vulnerability are listed below:

CVE is short for **C**ommon **V**ulnerabilities and **E**xposures, which is a list of publicly disclosed public vulnerabilities. Each CVE entry is assigned a unique identification number. For example, the identification of the Log4Shell vulnerability is CVE-2021-44228.

CWE is short for **C**ommon **W**eakness **E**numeration, which is a list of categories representing general types of software weaknesses. For example, CVE-2021-44228 belongs to CWE-502 (i.e., Deserialization of Untrusted Data [19]).

CVSS is short for **C**ommon **V**ulnerability **S**coring **S**ystem, which is an open framework for assessing and rating the severity of vulnerabilities. For example, the CVSS of CVE-2021-44228 is 10.0, which means this vulnerability is extremely dangerous.

TABLE I: Injection vulnerabilities studied in this paper.

CWE ID	Description	Rank
79	Improper neutralization of input during the web page generation (Cross-site Scripting)	2/3
78	Improper neutralization of special elements used in an OS command (OS Command Injection)	5/3
89	Improper neutralization of special elements used in an SQL command (SQL Injection)	6/3
502	Deserialization of untrusted data	13/8
77	Improper neutralization of special elements used in a command (Command Injection)	25/3

*__x/y__ in the last column denotes the ranking of the related vulnerability in the CWE Top 25 list and the OWASP Top 10 list, respectively.

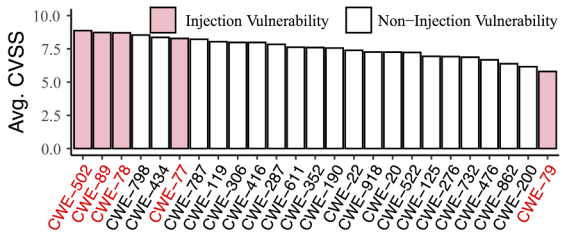


Fig. 2: The average CVSS scores of the top 25 CWEs.

POC is short for **P**roof **O**f **C**oncept, which is a realization of a certain idea, method, or principle in order to demonstrate its feasibility [20]. In the field of software security, a POC may refer to a demonstration that a vulnerability exists and can be exploited.

Exploit is a piece of code that takes advantage of a software vulnerability or security flaw. It is written either by security researchers as a POC threat or by malicious actors for use in their operations [21].

Injection vulnerabilities studied in this work. Injection vulnerabilities can be further classified based on the type of interpreter being attacked and the context in which the injection occurs. Vulnerability databases define various types of injection vulnerabilities, while we mainly focus on the most common and severe types as the research object in this paper. Specifically, we refer to two authoritative ranking lists for assessing the severity of vulnerabilities: the OWASP Top 10 [4] and the CWE Top 25 [22]. As of writing this paper, OWASP has only released data up to the end of 2021. Correspondingly, we refer to the rankings in 2021 for the CWE Top 25. As shown in Table I, we find that there are five types of injection vulnerabilities that appear simultaneously in both the OWASP Top 10 and the CWE Top 25 lists, i.e., **Cross-site Scripting** (CWE-79), **OS Command Injection** (CWE-78), **SQL Injection** (CWE-89), **Deserialization of Untrusted Data** (CWE-502), and **Command Injection** (CWE-77). Therefore, we choose the aforementioned five injection vulnerabilities as the subjects of our study.

It should be noted that the CWE Top 25 list is ranked based on the overall CVSS scores, which are calculated considering both the frequency and severity of vulnerabilities. Additionally, in Figure 2, we also present the average CVSS scores for the top 25 CWEs, providing a separate measure of vulnerability severity. We find that the top 3 CWEs with the highest

average CVSS scores are all injection vulnerabilities, with CWE-502 being the most severe. This once again confirms the importance and severity of injection vulnerabilities among all vulnerabilities, highlighting the necessity of studying the techniques for repairing injection vulnerabilities.

The vulnerabilities studied in this paper focus on the Java programming language. There are two reasons for this. On the one hand, we consider that the recent severe Log4Shell injection vulnerability occurred in Java applications. While some works focus on repairing vulnerabilities, most of them target C/C++ languages [23], [24], with little attention given to vulnerabilities in Java. On the other hand, this paper investigates the performance of Automated Program Repair (APR) techniques in repairing injection vulnerabilities. Recent APR techniques have primarily focused on fixing defects in Java [13], [14] (e.g., evaluated on the well-known Java defect benchmark Defects4J [17]) and have shown better results compared to other programming languages. This could be attributed to the knowledge accumulated by practitioners [25], [26] in fixing defects in Java programs (e.g., the plastic surgery hypothesis [18] and the redundancy assumption [27]). To this end, we aim to establish first-hand knowledge of injection vulnerabilities in Java programs in this work to uncover insights into fixing injection vulnerabilities.

III. Study Design

A. Research Questions

This work aims to deepen the understanding of repairing injection vulnerabilities by exploring the following research questions:

- **RQ-1: To what extent injection vulnerabilities can be repaired by APR tools?** As mentioned earlier, there are currently no dedicated methods for repairing injection vulnerabilities. However, APR techniques have demonstrated promising performance in repairing general defects. Therefore, in this RQ, we will investigate the effectiveness of APR methods in repairing injection vulnerabilities. We start by creating a dataset, named ***InjectionVul4J***, from open-source projects for repairing injection vulnerabilities. Then, we select several representative state-of-the-art APR methods to evaluate their ability of repairs on our constructed benchmark and perform a preliminary analysis of the results.
- **RQ-2: What are the key differences between injection vulnerability patches and general defect patches?** Injection vulnerabilities are security defects involving security issues, while general defects may involve more functional issues. Therefore, their patches may also differ in characteristics. To this end, in this research question, we first define some metrics to describe patches, and then compare the characteristics of patches in InjectionVul4J and Defects4J. This can help us better analyze the results and challenges of the APR methods in repairing injection vulnerabilities.
- **RQ-3: Does injection vulnerability repair support the plastic surgery hypothesis?** The plastic surgery hypothesis [18] is a well-known insight for APR, which states that the code ingredients to fix a bug usually already exist within the same project. But for injection vulnerabilities as special security defects, this hypothesis has not yet been investigated to see whether it still holds true. In this research question, we will explore to what extent the injection vulnerability repair supports the plastic surgery hypothesis, thereby further providing implications for using APR techniques to repair injection vulnerabilities.

B. Research Methodology

1) Prerequisites for answering RQ-1: In RQ-1, we investigate to what extent injection vulnerabilities can be repaired by APR tools. Here we describe the process of constructing our injection vulnerability repair benchmark, InjectionVul4J, as well as the selection and configuration of APR methods.

Benchmark: InjectionVul4J. To make it practical, we collect vulnerable programs and their fix patches from real-world projects like Defects4J. We find that Zhou et al. [28] have constructed a dataset for the task of mining silent vulnerability fixes. Therefore, we reuse their dataset on one hand and further expand our injection vulnerability repair dataset following their data collection process to build the final InjectionVul4J benchmark. Zhou et al.'s dataset relies on two vulnerability-related data sources. One comes from the SAP KB project [29], and the other is collected and filtered by the authors from the Mitre CVE database [16]. Because the focus of this study is on the most common and severe injection vulnerabilities, we select from their dataset those vulnerabilities whose CWE ID belongs to the five types listed in Table I. 51 injection vulnerability fixes within the scope of this study are selected from their dataset of 1436 Java vulnerability fixes.

Zhou et al.'s collection from the Mitre CVE database is up until January 26, 2021. However, as we refer to statistics up until the end of 2021 when determining the research scope (refer to §II-B), we use the same approach to extend the collection to December 31, 2021. Specifically, we first collect all CVEs disclosed between January 27, 2021, and December 31, 2021, from the Mitre CVE database. Then we proceed by extracting CVEs containing a patch fixing that vulnerability, i.e., with a link toward a GitHub commit, issue, or pull request. Next, we manually collect the commit links related to all issues and pull requests. After that, we merge all the commit links as a patch list and remove duplicates. If multiple commits point to the same vulnerability, we merge them into one patch. Finally, we collect an additional 72 injection vulnerabilities and their patches. Combined with the patches from Zhou et al.'s dataset, we construct our InjectionVul4J benchmark, containing a total of 123 CVEs and their patches from 65 projects.

Collecting vulnerability exploits. To provide more information for vulnerabilities in InjectionVul4J to facilitate future research, we also attempt to collect their exploits. For each injection vulnerability, the first two authors of this paper are given a time limit of 1 hour to search for publicly available POC/Exploit information on VulDB [30], Github, and Google. The two authors then discuss and merge the information they

TABLE II: The statistics of InjectionVul4J.

CWE ID	Project	Vulnerability	Exploit
CWE-77	3	3	2
CWE-78	6	6	4
CWE-79	27	33	19
CWE-89	11	16	7
CWE-502	23	65	34
Total	65	123	66

find to identify links that can be used for reproduction. If both authors are unable to find any POC/Exploit information within the time limit, the injection vulnerability will be marked as unexploitable. In the end, 66 vulnerabilities in InjectionVul4J are marked as exploitable, and the links related to exploits will be released along with the dataset.

The statistics of InjectionVul4J are presented in Table II.

Selection of APR Tools. Existing APR approaches can be summarized into three categories: heuristic-based [31], [32], [33], [12], template-based [34], [35], [36], [13], and learning-based [37], [38], [39], [14]. In order to ensure that the selected approaches are representative, we choose one state-of-the-art tool from each category that meets the following criteria: (1) the artifact of the APR approach is available and can be successfully executed; (2) the APR approach does not rely on other external information as input; (3) the APR approach does not utilize feedback information provided by tests for iterative modification. Some recent APR works propose to enhance through approaches such as retrieval [40], but they require additional data collection and cannot be fairly compared with other works. Therefore, we set criterion-2 to filter out these approaches. Moreover, since generate-and-validate APR works typically require running tests to determine if a patch is plausible, some works use information from test outputs to guide patch generation or modification [41]. However, in our injection vulnerability repair scenario, there is a lack of such test cases. Therefore, we set criterion-3 to exclude such approaches. Finally, we select three APR tools, i.e., the heuristic-based tool **SimFix** [12], the template-based tool **TBar** [13], and the learning-based tool **Recoder** [14]. These approaches have demonstrated state-of-the-art bug-fixing performance by generating correct patches for 34, 43, and 53 defects in Defects4J, respectively.

Configuration of APR Tools. Regarding the fault localization phase in APR, we follow a common practice in APR [13], [42], [43] using the perfect fault localization. It assumes that the exact buggy code lines can be accurately localized by fault localization tools, which enables fair assessment of APR techniques independently of the fault localization strategy used. When generating the patches, we maintain the default settings provided in its replication package for each tool. Following Liu et al. [13], we set the time limitation for generation to 3 hours.

Evaluation Metric. Considering the lack of executable test cases, we use Exact Match (EM) to determine the correctness of generated patches when compared with developer-written patches. It is a common practice when evaluating the performance of repair tools on benchmarks without test cases [44], [45], [24], [46]. Additionally, considering the difficulty of repairing injection vulnerabilities, we define the metric Partial Match (PM), which indicates that the generated patch partially matches the developer-written patch at the statement granularity.

2) Prerequisites for answering RQ-2: In RQ-2, we propose to compare the patches in InjectionVul4J and Defects4J to identify their differences. To do this, we use the following metrics to characterize the code revisions made by the patches:

- **ΔLines:** the number of revised statements.
- **ΔMethods:** the number of revised methods.
- **ΔLanguages:** the number of program languages related to the patch. If ΔLanguages is larger than one, the patch requires a multiple-language revision.
- **ΔConfigures:** the number of configure files revised within the patch. If ΔConfigures is larger than zero, the patch requires revising configure files rather than the source code.
- **#Created-Methods:** the number of methods newly created within the patch. If #Created-Methods is larger than zero, the patch requires newly created methods.
- **#Exploitable Tests:** the number of test cases revised in the patch which can diagnose the existence of a defect, i.e., test cases fail on the program of the buggy version while pass on that of the fixed version.

3) Prerequisites for answering RQ-3: In RQ-3, we investigate whether injection vulnerability repair supports the plastic surgery hypothesis in the field of APR. Following Barr et al. [18] when validating this hypothesis in a study, we measure the fix code ingredients at statement-level. Considering the challenges of fixing injection vulnerabilities, we also look for fix code ingredients at a finer granularity, i.e., expression-level. Specifically, we define the following two concepts:

- **Identified Expressions:** the expressions performed as fix ingredients in developer-written patches that already exist in the codebase at the time of the revision.
- **Identified Statements:** the statements performed as fix ingredients in developer-written patches that already exist in the codebase at the time of the revision.

It is worth noting that an Identified Statement always includes Identified Expressions, but not vice versa. We will tally the number of vulnerabilities in InjectionVul4J whose patches contain Identified Expressions/Statements and compare them with patches for general defects.

IV. Experimental Results

A. RQ-1: Effectiveness of APR tools on InjectionVul4J

To investigate to what extent injection vulnerabilities can be repaired by APR tools, we execute selected APR tools on InjectionVul4J and count the number of EM and PM generated patches. The results are presented in Table III. Our experimental results reveal a predominantly negative trend, as none of the three APR tools are able to generate any correct patches for the injection vulnerabilities in the benchmark. The occurrence of such results may be attributed

TABLE III: The effectiveness of three APR tools in repairing injection vulnerabilities from InjectionVul4J.

CWE ID	#Vul.	SimFix		TBar		Recoder	
		#EM	#PM	#EM	#PM	#EM	#PM
CWE-77	3	0	0	0	0	0	0
CWE-78	6	0	0	0	0	0	0
CWE-79	33	0	0	0	0	0	**3**
CWE-89	16	0	0	0	0	0	0
CWE-502	65	0	0	0	0	0	**2**
Total	123	0	0	0	0	0	**5**

*__#EM__ and __#PM__ denote the number of patches that exactly match and partially match the developer-written patches, respectively.

to the complexity of repairing vulnerabilities, which often requires domain-specific expertise. The state-of-the-art APR tools primarily focus on repairing simple defects through relatively straightforward code changes [47].

Among all generated patches, only five patches generated by Recoder are partially matched, that is, they share some common statements with the developer-written patch. In contrast, the heuristic-based approach SimFix and the template-based approach TBar even fail to generate any partially matched patches. This indicates that for injection vulnerabilities, learning-based approaches may have more potential. Since the repair principles for injection vulnerabilities are relatively complex, traditional APR approaches with manually defined heuristic rules or templates lack flexibility and have significant limitations. While learning-based approaches automatically learn the mapping between buggy programs and patches from a large amount of data, which may lead to better results in repairing complex injection vulnerabilities. Given the recent emergence of more and more powerful deep learning models, learning-based approaches may further improve their ability to repair injection vulnerabilities.

Figure 3 illustrates an example of partially matched patches. For the injection vulnerability CVE-2017-14735 in the project `antisamy`, Figure 3(a) depicts three patches generated by Recoder, while Figure 3(b) depicts the corresponding developer-written patch. We find that Recoder successfully modifies "`compile(REGEXP_BEGIN + value + REGEXP_END)`" in three methods to "`compile(value)`" in stages, which is partially identical to the patch written by developers. However, Recoder fails to provide a fix for the code that needs to be modified in "`global.html`". This example illustrates two challenges in repairing injection vulnerabilities: multiple hunks and multiple languages. Patches for repairing injection vulnerabilities may span across multiple places in the project and involve multiple programming languages (as seen in the example in Figure 3 with four hunks and two languages Java & HTML).

> *Existing APR tools exhibit poor overall performance in repairing injection vulnerabilities. Learning-based approaches can generate a small number of partially correct patch fragments, showing more potential compared to other approaches.*

```
# 1st patch generated by Recoder
@@ -588,7 +585,7 @@ ... getAllowedRegexps
--- ... compile(REGEXP_BEGIN + value + REGEXP_END));
+++ ... compile(value));
# 2nd patch generated by Recoder
@@ -617,7 +614,7 @@ ... getAllowedRegexps2
--- ... compile(REGEXP_BEGIN + value + REGEXP_END));
+++ ... compile(value));
# 3rd patch generated by Recoder
@@ -634,16 +631,14 @@ ... getAllowedRegexps3
--- ... compile(REGEXP_BEGIN + value + REGEXP_END));
+++ ... compile(value));
```

(a) Generated by Recoder

```
# patch of CVE-2017-14735 (antisamy)
@@ -588,7 +585,7 @@ ... getAllowedRegexps
--- ... compile(REGEXP_BEGIN + value + REGEXP_END));
+++ ... compile(value));
@@ -617,7 +614,7 @@ ... getAllowedRegexps2
--- ... compile(REGEXP_BEGIN + value + REGEXP_END));
+++ ... compile(value));
@@ -634,16 +631,14 @@ ... getAllowedRegexps3
--- ... compile(REGEXP_BEGIN + value + REGEXP_END));
+++ ... compile(value));
@@ -54,7 +54,8 @@ ... global.html
--- <regexp name="onsiteURL" value="([\p{L}\p{N}\\\.\#@\
     $%\+&amp;;\-_~,\?=/!]+|\#(\w)+)"/>
+++ <regexp name="onsiteURL" value="^(?![\p{L}\p{N
     }\\\.\#@\$%\+&amp;;\-_~,\?=/!]*(&amp;colon))[\p{L}\p
     {N}\\\.\#@\$%\+&amp;;\-_~,\?=/!]*"/>
+++ ... ...
+++ ... ...
```

(b) Written by developers

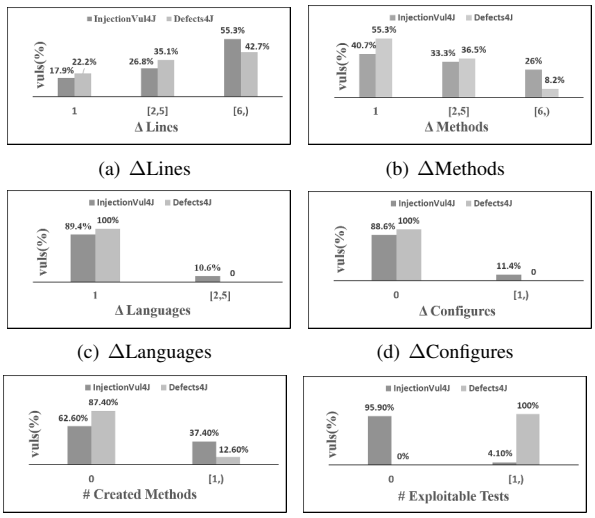
Fig. 3: The APR-generated and developers-written patches for CVE-2017-14735.

B. RQ-2: Key Differences between Patches of InjectionVul4J and Defects4J

The results from RQ1 indicate that existing APR tools struggle to repair any injection vulnerabilities in InjectionVul4J. However, they have been shown to perform well in fixing general defects in Defects4J. With the same tool configurations, the primary factor contributing to this performance difference lies in the datasets used for evaluation. Therefore, for further analysis of the results from RQ1 and to advance future research on injection vulnerability repair, in RQ2, we compare the differences between patches in InjectionVul4J and Defects4J. We use the metrics defined in § III-B2 to characterize the developer-written patches in two benchmarks, the results are presented in Figure 4.

Figure 4(a) illustrates the number of code lines modified in patches across the two benchmarks. We observe that 55.3% of the vulnerabilities in InjectionVul4J require modifications of 6 lines or more, and multi-line revision is one of the acknowledged challenges for APR [48]. In contrast, this proportion is 42.7% in Defects4J, which is 12.6% lower than InjectionVul4J. The percentage of one-line revisions in InjectionVul4J is 17.9%, which is 4.3% lower than Defects4J's 22.2%. This indicates that repairing injection vulnerabilities in InjectionVul4J is more complex and involves more extensive modifications compared to general defects in Defects4J.

Figure 4(b) shows the number of methods involved in patches. In Defects4J, 55.3% of patches are completed within one method, while the proportion of one-method revision in InjectionVul4J is significantly lower, at only 40.7%. Further-



(a) ΔLines (b) ΔMethods

(c) ΔLanguages (d) ΔConfigures

(e) #Created-Methods (f) #Exploitable Tests

Fig. 4: Characteristics of developer-written patches.

more, 26% of patches in InjectionVul4J are dispersed across 6 or more methods. The results in Figure 4(b) indicate that repairing injection vulnerabilities not only requires more lines of modification but also typically involves more dispersed multi-hunks, posing greater challenges for APR capabilities.

In Defects4J, patches are only located within product-related source code, i.e., they involve only Java functional code. However, in InjectionVul4J, we notice that 10.6% of patches require modifications across multiple languages (as shown in Figure 4(c), i.e., modifications involving code in addition to Java, such as HTML, CSS, JavaScript); furthermore, 11.4% of patches require modifications not only in functional code but also in configuration files (as shown in Figure 4(d)). Both types of modifications exceed the repair capabilities of the investigated APR tools, which again reflects the complexity and difficulty of repairing injection vulnerabilities.

Figure 4(e) reflects the repair mode of the patches. Interestingly, we find that in InjectionVul4J, more than one-third (37.4%) of the patches create new methods for modification, while in Defects4J, the vast majority (87.4%) of the patches are applied within previously defined methods. This suggests that the coupling between repairing injection vulnerabilities and the existing functional logic may not be as strong as in the case of general defects. For example, creating a method for whitelist validation and calling this method before the execution of the main functionality to prevent injection. APR tools seem to struggle to provide correct patches for such relatively independent repairs, as we find that not only in InjectionVul4J but even in Defects4J, existing APR tools are unable to generate correct patches for vulnerabilities requiring created-methods revision.

In Figure 4(a)-4(e), we focus on revision in production code, while in Figure 4(f), we focus on revisions in test code along with the patches. In Defects4J, having an exploitable test is a requirement imposed by its authors for including a defect, that

```
    # patch of CVE-2019-15566
@@@ HistorySearchProvider.java
+++ queryBuilder.setStrict(true);
    # the same statement within the project
@@@ SyncContentProvider.java
    ... ...
    queryBuilder.setStrict(true);
    ... ...
```

(a) Identified Statement

```
    # patch of CVE-2020-26945
@@@ AbstractSerialStateHolder.java
--- try { ObjectInputStream in = new
     LookAheadObjectInputStream();
+++ try { ObjectInputStream in = new ObjectInputStream()
     ;
    # the same expression within the project
@@@ SerializableProxyTest.java
    ... ...
    ... ... = new ObjectInputStream();
    ... ...
```

(b) Identified Expression

Fig. 5: The examples of fix code ingredients in patches.

is, the buggy program fails on the exploitable test but passes on the fixed version. Therefore, the proportion of patches with exploitable tests in Defects4J is 100%. In contrast, in InjectionVul4J, this proportion is only 4.1%. This limitation hampers the ability of APR tools, which typically require generated patches to be validated. It indicates that for injection vulnerabilities, developers do not tend to release exploitable tests with patches, or constructing such test cases itself is a challenging task. Therefore, to promote injection vulnerability repair, we take the first step by collecting POC/Exploit information for InjectionVul4J, as described in § III-B.

> *The patches in InjectionVul4J are more complex compared to those in Defects4J. They require more lines of code changes, are distributed across more methods, and involve revisions in multiple languages and configurations. Developers are more accustomed to creating new methods for revisions and rarely provide exploitable tests.*

C. RQ-3: The Plastic Surgery Hypothesis for Repairing Injection Vulnerabilities

The plastic surgery hypothesis posits that the code ingredients necessary to fix a bug typically already exist within the same project. Many APR tools heavily rely on this hypothesis, employing various methods to mine repair templates from the project or devise heuristic rules. However, no work has yet validated the plastic surgery hypothesis for repairing injection vulnerabilities. If this hypothesis still holds true for repairing injection vulnerabilities, it suggests that future efforts can still be directed toward searching within the current project. Conversely, if it does not hold true, it implies that relying solely on the current project for repairing injection vulnerabilities may not further improve the success rate of repairs, necessitating the consideration of more external information.

We are the first to investigate the plastic surgery hypothesis for repairing injection vulnerabilities. In § III-B2, we have defined two key concepts, namely Identified Expressions

TABLE IV: Statistics of code fix ingredients in InjectionVul4J.

CWE ID	#Vul.	#IS (ratio)	#IE (ratio)
CWE-77	3	0 (0%)	0 (0%)
CWE-78	6	0 (0%)	3 (50%)
CWE-79	33	4 (12.1%)	9 (27.3%)
CWE-89	16	7 (41.2%)	7 (41.2%)
CWE-502	65	6 (9.2%)	9 (13.8%)
Total	123	17 (13.8%)	28 (22.8%)

*#IS and #IE denote the number of vulnerabilities whose patches contain Identified Statements and Identified Expressions, respectively.

and Identified Statements, which represent fix ingredients that can be found in the current codebase. Figure 5(a) and Figure 5(b) illustrate the statement-level granularity and expression-level granularity of fix code ingredients, respectively. Figure 5(a) shows a partial code snippet of the patch for CVE-2019-15566. It can be observed that the statement `queryBuilder.setStrict(true);` in the patch, located in the `HistorySearchProvider.java` file, can be found with a completely identical statement in the `SyncContentProvider.java` file. Figure 5(b) depicts a partial code snippet of the patch for CVE-2020-26945. Although an exact match of the statement `ObjectInputStream in = new ObjectInputStream();` in the patch cannot be found in the code repository, the same expression `new ObjectInputStream()` can be found in other files.

Table IV lists the number of vulnerabilities in InjectionVul4J whose patches contain Identified Statements/Expressions. From the table, it can be observed that for vulnerabilities belonging to CWE-77 and CWE-78, we cannot find any Identified Statements in the current repository. Among the 65 vulnerabilities classified under CWE-502, only 6 (9.2%) have Identified Statements, and 9 (13.8%) have Identified Expressions. However, for CWE-89, which represents SQL Injection, 41.2% of vulnerabilities' patches contain Identified Statements. This indicates that compared to other types of injection vulnerabilities, CWE-89 is more likely to be correctly repaired by obtaining useful information from the code repository or finding repair templates.

Overall, only 13.8% of vulnerabilities in InjectionVul4J contain Identified Statements. For Identified Expressions, this proportion is slightly higher (22.8%), but still relatively low. According to Barr et al.'s study [18] on fixing Java general defects, 84% of commits contain Identified Statements in the current project. Furthermore, 11% of commits are 100% graftable, which means that every statement in the patch could be reconstructed from the current project. Compared to fixing general defects, the degree to which the plastic surgery hypothesis holds is much lower for fixing injection vulnerabilities, suggesting that the repair of injection vulnerabilities may not be as closely related to the project in which they reside. This implies the difficulty in automatically repairing injection vulnerabilities, as it may require resorting to external expertise rather than relying solely on information within the project.

> *Compared to general defect repair, the support for the plastic surgery hypothesis is much lower in the context of injection vulnerability repair.*

V. Discussion

A. Intention Analysis

Analyzing developers' intentions during patching is crucial for advancing the development of automatic injection vulnerability repair techniques. By analyzing developers' intentions, we can gain a deeper understanding of the mechanisms and processes involved in repairing injection vulnerabilities.

We conduct an open card sorting [49] to categorize patch intentions in InjectionVul4J. The first two authors and a security expert collaborate to sort the cards. We create one card for each patch, and each participant is asked to analyze its intention and describe it briefly. Given the challenges of repairing injection vulnerabilities, classifying intentions as "Unknown" is allowed. We provide each participant with 10 patches for each iteration. After each round of individual classification, the three participants engage in discussions to resolve discrepancies, and merge, or split categories. Once a consensus is reached, we proceed to the next iteration. Finally, eight kinds of intentions are identified: Security API Invoking, Whitelist filtering, Blacklist filtering, String filtering, Parameterization, Disabling vulnerable functions permanently, Disabling vulnerable functions by default, and Unknown.

Table V shows the intention classification results of each CWE in InjectionVul4J. We find that for Cross-site Scripting (XSS, CWE-79), 21 out of 22 discernible patches invoke security APIs to validate the untrusted input. This indicates that for XSS vulnerabilities, there may already exist some standardized repair strategies in the industry, which can effectively prevent untrusted input from being used for XSS attacks. Additionally, developers and security experts may be more inclined to adopt proven effective security measures to repair vulnerabilities, rather than developing custom solutions.

For SQL injection (CWE-89), 6 patches also invoke existing security APIs. Four patches employ string filtering, which detects and removes malicious SQL statements or special characters. By detecting malicious characters, removing or escaping special characters, and validating the legality of input, these patches effectively prevent the unauthorized execution of malicious SQL statements. Three patches fix vulnerabilities by whitelist filtering, allowing only predefined character sets to pass input validation. Parameterization, a specific repair intention for SQL injection, is observed in 2 patches in InjectionVul4J. Parameterized queries involve passing user-provided input values as parameters to database queries instead of directly embedding them into SQL query statements. This is typically achieved through prepared statements [50], [51] and precompiled SQL statements.

For deserialization vulnerabilities (CWE-502), the majority (84.6%) of patches are implemented using blacklist or whitelist filtering strategies. Blacklist filtering patches are more prevalent, which prohibit specific malicious strings in

TABLE V: Intention analysis for repairing injection vulnerabilities in InjectionVul4J.

CWE ID	Intention	#Vul.
CWE-77	Security API Invoking	1
	Whitelist filtering	1
	Disabling vulnerable functions by default	1
CWE-78	Security API Invoking	2
	Blacklist filtering	1
	Unknown	3
CWE-79	Security API Invoking	21
	Whitelist filtering	1
	Unknown	11
CWE-89	Security API Invoking	6
	String filtering	4
	Whitelist filtering	3
	Parameterization	2
	Unknown	1
CWE-502	Blacklist filtering	39
	Whitelist filtering	16
	Security API Invoking	4
	Disabling vulnerable functions permanently	3
	Disabling vulnerable functions by default	3

the input. Blacklist filtering is less strict than whitelist filtering because it only blocks known malicious inputs. This suggests that developers typically prevent deserialization attacks based on known attack patterns and exploit cases, and accurately identifying benign serialized object types can be challenging. Only 4 patches for CWE-502 invoke security APIs, much fewer than for CWE-79. This indicates that compared to fixing XSS vulnerabilities, there are fewer standardized repair strategies in the industry for addressing deserialization vulnerabilities, requiring developers to customize solutions more often. Additionally, some patches implement the strategy of disabling vulnerable functions, either permanently (by directly removing them) or by default (conditionally disabling them). This underscores the difficulty of fixing deserialization vulnerabilities, sometimes leading developers to remove functionality to prevent attacks when no suitable solution is found.

B. Implications & Road Ahead

Constructing exploitable test cases. When constructing our benchmark, we encounter challenges in collecting exploitable test cases for injection vulnerabilities. Our findings in RQ-2 also suggest that developers rarely provide exploitable test cases with patches. This may be due to the complexity of reproducing a vulnerability, which requires strict preconditions. Furthermore, unlike general bugs that utilize assertions as oracles, determining the triggering or fixing states for injection vulnerabilities is more challenging (e.g., unauthorized database operations or server process invocations). However, for automated repair tools, the use of exploitable test cases to determine the plausibility of patches is crucial. Additionally, the execution results and output content of test cases can provide feedback guidance for repairs. In this work, we have taken the first step in collecting vulnerability exploit information. However, we have not been able to construct exploitable test cases for all vulnerabilities in our benchmark. To further advance the development of injection vulnerability repair, both industry and academia need to collaborate in constructing exploitable test cases for vulnerability patches.

Take more languages and types of files into consideration. When comparing patches between InjectionVul4J and Defects4J in RQ-2, we note that for repairing injection vulnerabilities, some revisions span files of different languages and types within the project. However, existing APR tools are limited to fixing code in a single language within the business logic. Given that the causes and fixes for injection vulnerabilities are more intricate than general defects, the presence of multi-languages and configuration file revisions in patches exceeds the capabilities of current APR tools. Researchers should design more advanced automated repair tools that can consider a broader range of languages and file types to enhance the ability to repair injection vulnerabilities.

It is not enough to excavate fix code ingredients only from within the project. From RQ-3, we observe that injection vulnerability repair is not highly supportive for the plastic surgery hypothesis. This hypothesis is a cornerstone of many APR techniques, especially heuristic-based and template-based ones, which generate patches by mining fix code ingredients from the current project. However, for injection vulnerabilities, relying solely on information from within the project is insufficient. To address this limitation, we propose two improvement strategies. Firstly, consider the retrieval of valuable templates or code ingredients from other projects when fixing vulnerabilities in the current project. Secondly, further enhancing learning-based approaches. Although current learning-based methods may not produce correct patches, they show promise by generating partially correct patches. Learning-based techniques train models using data from various open-source projects, thus incorporating knowledge beyond the current project. Future advancements in training with more powerful models and targeted data may further enhance the repair capability for injection vulnerabilities.

Intention-aware patch generation. The analysis of developers' intentions behind patches in § V-A suggests that Intention-aware patch generation could be a promising direction for repairing injection vulnerabilities. We find that developers have certain intention preferences for specific CWEs. For example, they tend to invoke existing security APIs for XSS vulnerabilities, while favoring blacklist/whitelist filtering for deserialization vulnerabilities. If we can first determine the possible repair intentions of developers based on the type of vulnerability and other factors during the repair process, patch generation can be more targeted. Referring to other patches with similar intentions may lead to better repair results for the current vulnerability.

We still have a long way to go. Results from RQ-1 indicate that existing APR tools designed for general defects perform poorly in repairing injection vulnerabilities, mainly due to the significant differences in characteristics and repair mechanisms between injection vulnerabilities and general defects. Currently, there are no dedicated tools or datasets specifically tailored for repairing injection vulnerabilities. In this work,

we have taken the first step towards automatically repairing injection vulnerabilities by constructing InjectionVul4J and empirically investigating APR performance. However, designing effective methods for automatic injection vulnerability repair remains an unresolved issue. Therefore, there is still a long way to go to automatically repair injection vulnerabilities.

C. Threats to Validity

External validity. A major threat to external validity is that we restrict the research scope within Java. We have illustrated our consideration in Section II. To mitigate this threat, in the future, we plan to conduct larger-scale experiments among different program languages.

Internal validity. In our experiments, a major threat to internal validity is the possible faults in the implementation of the benchmark and the configuration of the subject APR tools. To address the threat, we review our code and experimental scripts to ensure their correctness before conducting the experiments.

Construct validity. When evaluating the correctness of patches generated by APR tools, we use the exact match as the evaluation metric. This may underestimate the performance as generated patches may differ but still achieve the intended effect. Unfortunately, following Liu et al.'s suggestion [47] to evaluate using metrics such as semantically correct repairs and plausible patches requires executable test cases, which are lacking in InjectionVul4J. In the future, we plan to construct a benchmark with exploitable tests to address this limitation.

VI. Related Work

A. Automatic Program Repair

Mainstream APR approaches include heuristic-based, template-based, semantic-based, and learning-based ones. Heuristic-based approaches [31], [52], [53] iterate over a search space of syntactic edits to mutate the buggy program, supervised by search algorithms like genetic algorithms or multi-object search. The main challenge these approaches face is balancing the vast search space with limited computational resources. Template-based approaches [54], [47] use predefined templates to generate repair solutions. These templates are typically designed by human developers and contain common repair patterns or strategies. When an error is detected in the program, the APR tool attempts to match these errors with predefined templates and then applies the matching template to generate patches. Semantic-based approaches [55], [56], [57] extract semantic specifications from the program's test cases and attempt to synthesize effective program repairs that satisfy these specifications using constraint solving techniques such as Z3 [58]. We do not consider such approaches in our experiments in this paper because the absence of exploitable test cases prevents us from extracting semantic specifications. Learning-based approaches [42], [59], [14], [39], [48] employ deep learning models to learn repair patterns from existing defect repairs and generate patches accordingly. With the great boost in the field of deep learning, learning-based approaches have gone through rapid progress in recent years and become the mainstream approaches in the current state of the art.

B. Specific Type Vulnerability Repair

This study focuses on repairing five specific injection vulnerabilities (CWE-77, 78, 79, 89, and 502). MarchandMelsom et al. [60] investigated repair for CWE-79 and CWE-89, while no dedicated tools currently address other injection vulnerabilities. There are also some research efforts that address the repair of other specific types of vulnerabilities. To address Buffer Out-of-bounds Write/Read vulnerabilities (CWE-787, CWE-125), Sidiroglou et al. [61] proposed a template-based approach that leverages implicit information flow analysis. Shaw et al. [62] proposed a static analysis-based approach to replace unsafe API or data structures with safe ones. Gao et al. [63] proposed a static warning validation approach to eliminate the problem of false positive warnings. This approach first employs the static analyzer to get a list of static warnings, and then performs symbolic execution to validate the reachability of specific statements. For Integer Overflow or Wraparound (CWE-190), Long et al. [64] leveraged an on-demand static analysis to filter unsafe program inputs. Wang et al. [65] exploited dynamic taint analysis to detect some integer overflow and then generate one patch. Muntean et al. [66] employed a template-based approach to generate patches for integer overflow. There are also some works [67], [68], [69], [70] focusing on vulnerability repair in other specific contexts, such as Android applications and smart contracts.

C. Vulnerability Dataset

Researchers have constructed several datasets related to vulnerabilities. Some works focus on the vulnerability detection task, where datasets like VulinOSS [71] and those by Cao et al. [72] are mined from NVD and GitHub. The Code Gadget Database [73] specifically targets buffer errors (CWE-119) and resource management errors (CWE-399). For the vulnerability repair task, Fan et al. [74] built Big-Vul from open-source GitHub projects, containing 91 different vulnerability types. Bhandari et al. [75] created CVEfixes from the NVD, which supports various types of data-driven software security research. However, these datasets primarily focus on C/C++ programs. For Java, Ponta et al. [76] created a manually curated dataset of Java vulnerability fixes, comprising 624 vulnerabilities. Zhou et al. [28] further expanded the number of vulnerabilities to 1,436 by mining direct or indirect GitHub commits from Ponta et al.'s dataset. While these works collect vulnerabilities and fixes across all types, our InjectionVul4J in this work specifically targets injection vulnerabilities. Additionally, we extend Zhou et al.'s dataset using their collection method. Currently, InjectionVul4J is the largest known dataset for Java injection vulnerability fixes.

VII. Conclusion

In this paper, we systematically conduct an empirical study of automatic program repair techniques for injection vulnerability repair. We first construct a benchmark named InjectionVul4J, consisting of 123 injection vulnerabilities with associated developer-written patches. Subsequently, we select

several representative APR methods and investigate their performance in repairing injection vulnerabilities. The results indicate that existing APR tools do not adequately support the repair of injection vulnerabilities. To investigate the underlying reasons, we compare the characteristics of patches for injection vulnerabilities and general defects, and explore whether the hypothesis widely used in APR still holds for injection vulnerabilities. Furthermore, we analyze developers' intentions when fixing injection vulnerabilities. Finally, we summarize the implications and outline the road ahead for repairing injection vulnerabilities.

ACKNOWLEDGEMENTS

This research is supported by the National Natural Science Foundation of China (No. 62232014).

REFERENCES

[1] O. C. R. 2023, "Cybercrime to cost the world $9.5 trillion usd annually in 2024," https://cybersecurityventures.com/cybercrime-to-cost-the-world-9-trillion-annually-in-2024, 2024.

[2] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "Large scale characterization of software vulnerability life cycles," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 4, pp. 730–744, 2019.

[3] M. Dowd, J. McDonald, and J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[4] "Owasp top ten," https://owasp.org/Top10.

[5] "Log4shell," https://en.wikipedia.org/wiki/Log4Shell.

[6] A. Luttwak and A. Schindel, "Log4shell 10 days later: Enterprises halfway through patching," https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell, 2024.

[7] D. Goodin, "As log4shell wreaks havoc, payroll service reports ransomware attack," https://arstechnica.com/information-technology/2021/12/as-log4shell-wreaks-havoc-payroll-service-reports-ransomware-attack, 2024.

[8] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 618–635.

[9] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 539–554.

[10] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2201–2215.

[11] A. Marchand-Melsom and D. B. Nguyen Mai, "Automatic repair of owasp top 10 security vulnerabilities: A survey," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 23–30.

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

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

[14] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[15] "National vulnerability database," https://nvd.nist.gov/vuln.

[16] "Mitre cve database," https://cve.mitre.org/.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[18] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 306–317.

[19] "Cwe-502: Deserialization of untrusted data," https://cwe.mitre.org/data/definitions/502.html.

[20] "Proof of concept," https://en.wikipedia.org/wiki/Proof_of_concept.

[21] "exploit," https://www.trendmicro.com/vinfo/us/security/definition/exploit.

[22] "2021 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[23] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.

[24] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "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*, 2022, pp. 935–947.

[25] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015, pp. 913–923.

[26] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 275–286.

[27] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 492–495.

[28] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *The 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.

[29] "Sap kb project," https://sap.github.io/project-kb/.

[30] "Vuldb," https://vuldb.com/.

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

[32] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 648–659.

[33] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond genprog," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019.

[34] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 802–811.

[35] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "FixMiner: mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[36] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 456–467.

[37] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[38] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[39] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[40] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 146–158.

[41] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1506–1518.

[42] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[43] W. Zhong, H. Ge, H. Ai, C. Li, K. Liu, J. Ge, and B. Luo, "Standup4npr: Standardizing setup for empirically comparing neural program repair systems," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[44] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1162–1174.

[45] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[46] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[47] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021.

[48] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 511–523.

[49] G. Rugg and P. McGeorge, "The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts," *Expert Systems*, vol. 14, no. 2, pp. 80–93, 1997.

[50] "Using prepared statements," https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html.

[51] "Prepared statements and stored procedures," https://www.php.net/manual/en/pdo.prepared-statements.php.

[52] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.

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

[54] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020, pp. 625–627.

[55] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 772–781.

[56] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 691–701.

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

[58] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[59] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 602–614.

[60] A. Marchand-Melsom and D. B. N. Mai, "Automatic repair of OWASP top 10 security vulnerabilities: A survey," in *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 23–30. [Online]. Available: https://doi.org/10.1145/3387940.3392200

[61] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 43–54. [Online]. Available: https://doi.org/10.1145/2737924.2737988

[62] A. Shaw, D. Doggett, and M. Hafiz, "Automatically fixing C buffer overflows using program transformations," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 124–135. [Online]. Available: https://doi.org/10.1109/DSN.2014.25

[63] F. Gao, Y. Wang, L. Wang, Z. Yang, and X. Li, "Automatic buffer overflow warning validation," *J. Comput. Sci. Technol.*, vol. 35, no. 6, pp. 1406–1427, 2020. [Online]. Available: https://doi.org/10.1007/s11390-020-0525-z

[64] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. C. Rinard, "Sound input filter generation for integer overflow errors," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 439–452. [Online]. Available: https://doi.org/10.1145/2535838.2535888

[65] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings*, ser. Lecture Notes in Computer Science, S. Dietrich, Ed., vol. 8550. Springer, 2014, pp. 255–275. [Online]. Available: https://doi.org/10.1007/978-3-319-08509-8_14

[66] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert, "Intrepair: Informed repairing of integer overflows," *IEEE Trans. Software Eng.*, vol. 47, no. 10, pp. 2225–2241, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2946148

[67] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, X. Chen, X. Wang, and X. Huang, Eds. ACM, 2016, pp. 711–722. [Online]. Available: https://doi.org/10.1145/2897845.2897896

[68] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 792–801. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606625

[69] T. D. Nguyen, L. H. Pham, and J. Sun, "SGUARD: towards fixing vulnerable smart contracts automatically," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1215–1229. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00057

[70] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 539–554. [Online]. Available: https://doi.org/10.1109/SP.2019.00071

[71] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "Vulinoss: a dataset of security vulnerabilities in open-source systems," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 18–21. [Online]. Available: https://doi.org/10.1145/3196398.3196454

[72] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "*BGNN4VD*: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, p. 106576, 2021. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106576

[73] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society,

2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[74] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, S. Kim, G. Gousios, S. Nadi, and J. Hejderup, Eds. ACM, 2020, pp. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

[75] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.

[76] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, M. D. Storey, B. Adams, and S. Haiduc, Eds. IEEE / ACM, 2019, pp. 383–387. [Online]. Available: https://doi.org/10.1109/MSR.2019.00064