Automatic repair of OWASP Top 10 security vulnerabilities: A survey

Alexander Marchand-Melsom*
alexamar@stud.ntnu.no
Norwegian University of Science and Technology
Trondheim, Norway

Duong Bao Nguyen Mai*
dbmai@stud.ntnu.no
Norwegian University of Science and Technology
Trondheim, Norway

ABSTRACT

Current work on automatic program repair has not focused on actually prevalent vulnerabilities in web applications, such as described in the OWASP Top 10 categories, leading to a scarcely explored field, which in turn leads to a gap between industry needs and research efforts. In order to assess the extent of this gap, we have surveyed and analyzed the literature on fully automatic source-code manipulating program repair of OWASP Top 10 vulnerabilities, as well as their corresponding test suites. We find that there is a significant gap in the coverage of the OWASP Top 10 vulnerabilities, and that the test suites used to test the analyzed approaches are highly inadequate. Few approaches cover multiple OWASP Top 10 vulnerabilities, and there is no combination of existing test suites that achieves a total coverage of OWASP Top 10.

CCS CONCEPTS

• Security and privacy → Web application security; **• General and reference → Surveys and overviews**.

KEYWORDS

OWASP Top 10, automatic program repair, survey

ACM Reference Format:
Alexander Marchand-Melsom and Duong Bao Nguyen Mai. 2020. Automatic repair of OWASP Top 10 security vulnerabilities: A survey. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* APR 2020, Seoul, South Korea, 8 pages. https://doi.org/10.1145/3387940.3392200

1 INTRODUCTION

For more than 20 years, researchers have attempted to create an automatic tool to repair defective programs. While the field of general bug fixing has seen a substantial rise of activity in recent years, there is yet to be a tool that properly addresses security vulnerabilities. Despite efforts at securing software, the fact remains that security vulnerabilities are a significant threat to many organisations, potentially leading to crippling economic losses. Fully automatizing the process of finding and repairing security vulnerabilities would be a significant step towards a more secure digital world.

In this survey we aim to give an overview over the state-of-the-art of automatic program repair (APR) of software vulnerabilities, in an attempt to pave the way for better APR tools. We survey and classify existing APR approaches according to which OWASP Top 10 [31] categories they cover. In order to get an indication of their usefulness, the reported performances of the surveyed approaches are analysed, and its reliability assessed on the basis of the test suites used. While testing the methods against a test suite ourselves would have been optimal to assess their performance, the lack of a runnable tool for most of them did not allow for it, hence the aforementioned approach. Adequate test suites being crucial to properly benchmark future APR tools, we further analyse the Java test suites used by the surveyed papers to test their approach and two systematic Java test suites (Juliet Test Suite [30] and Vulnerability Assessment Knowledge Base (VAKB) [32] [34]), and establish their coverage of OWASP Top 10.

The contributions of this paper are two-fold: a classification and a performance and reliability analysis of existing APR approaches, and an analysis of Java test suites for testing APR tools. We find that existing approaches have focused mainly on two OWASP Top 10 categories, namely A1 - Injection and A7 - XSS. Furthermore, the reported performance is worryingly unreliable due to the prevalence of poorly designed ad-hoc test suites, the lack of consensus on performance metrics, and the lack of reproducibility. The analysed Java test suites have an insufficient coverage of certain OWASP Top 10 vulnerabilities, but the systematic test suites provide a good framework and basis for extensions solving this coverage issue.

This paper is organized as follows. Section 2 presents its related work, while we present our approach in Section 3. The research results are presented in Section 4. We discuss these results and compare them to the related work in Section 5. Finally, we conclude and discuss future work in Section 6.

2 RELATED WORK

Liu et al. [17] systematically reviewed and classified the literature for Test-Based APR tools according to their approach. The authors discussed core issues in research of APR, which include test suite quality, fault localization accuracy, how to generate a patch, and evaluation metrics. They observed that there was a pressing need for a high quality peer-reviewed test suite for APR, but also that the way these tools were evaluated needed to be addressed, as both patch correctness and the generation of a human-readable optimal patch should be weighed in their benchmarking.

Shafiq et al. [35] systematically reviewed and classified automatic repair tools. For each solution, they individually identified its strengths

*Both authors contributed equally to this research.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392200

and weaknesses, and aggregated them according to the used algorithms. The subsequent classification of solutions is first made according to whether they are bug fixing solutions, debugging solutions, or solutions using search algorithms, and then according to six categories: Approaches, Techniques, Tools, Frameworks, Methods, and Systems.

Gazzola et al. [10] reviewed, classified, and compared techniques from 108 papers about APR. They identified two main approaches to APR tools: Generate-and-validate (G&V) and Semantics-driven.

Kong et al. [16] reviewed five APR techniques: GenProg, RSRepair, AE, Kali, and a brute-force based technique. They evaluated these techniques on a test suite composed of 17 programs of varying size, containing a total of 180 defects. The brute-force technique fared best for small sized programs, but none of the surveyed tools performed well on large test programs, with all tools suffering a significant performance drop. The authors conclude on this by stating that none of the surveyed techniques are ready to be used on large real-world programs.

Durieux et al. [7] tested 11 Java test-suite-based repair tools on five different benchmarks. They found that while the tools' performance varied greatly from benchmark to benchmark, they all were able to generate a substantially more significant amount of patches for the Defects4J benchmark [14]. The authors suggested this may be indicative of serious benchmark overfitting, and pointed out that in order to mitigate this problem, APR tools should be evaluated on diverse benchmarks.

Khalilian et al. [15] evaluated three APR techniques. In an effort to mitigate the absence of standard common benchmarks and provide a comparison framework for APR tools, the authors first compared the surveyed tools according to a set of criteria, then classified them according to a five-level maturity model they created.

Qi et al. [33] systematically surveyed recent research on APR, and evaluated their evaluation metrics. They isolated 12 main evaluation metrics, categorizing them further into two classes: repair effectiveness and repair efficiency. These findings point to the problem that there currently is no consensus on what metrics should be used to measure a tool's performance. In light of this, the authors suggest metrics that they view should be included in any evaluation.

Monperrus [25] compiled a bibliography of existing publications in APR, and categorised them according to their approach. A continuously updated version of this bibliography has also been made available by the same author [26].

3 RESEARCH IMPLEMENTATION AND DESIGN

Existing surveys on APR lack a focus on security bugs in general and OWASP Top 10 vulnerabilities in particular. Furthermore, few surveys question the reliability of the reported performance of APR approaches. This leads to a great uncertainty on what is currently the state-of-the-art and makes it more difficult to identify successful approaches which could be extended. It is also unclear how APR approaches for software security should be tested, and whether existing test suites used to test approaches are adequate, which is problematic for useful comparisons of APR approaches.

Our aim in this survey is to shed light on these issues and attempt to provide an overview over the state-of-the-art, and which test suites future APR approaches for vulnerabilities should use.

Our scope will be defined by the following inclusion criteria for the approach described by each publication:

- The approach needs to address a vulnerability found in one or several OWASP Top 10 categories;
- The approach needs to be fully automatic, i.e. it should not require any interaction with a human being beyond its launch;
- The approach needs to effectuate its repair by modifying the source code.

We use the following inclusion criteria for the selection of systematic test suites:

- The test suite needs to be systematic, i.e. it should have set procedures for inclusion or exclusion of new test cases;
- The test suite needs to contain test cases in Java;
- The test suite needs to contain test cases relating to security vulnerabilities.

We chose to limit our scope to Java for systematic test suites because of its prevalence in security-critical applications such as financial services.

3.1 Research Questions

The following research questions were formulated to address these issues:

- **RQ1:** How can the automatic repair methods reported in the literature be classified according to OWASP Top 10?
- **RQ2:** How reliable is each method based on their associated test suite and reported performance?
- **RQ3:** How well-suited are the Java test suites for testing automatic program repair tools?

3.2 Selection of papers and classification

Based on the inclusion criteria mentioned above, we will select papers by searching through six databases or search engines: Springer Link [28], Oria [42], IEEE Xplore [13], ACM Digital Library [2], Engineering Village [9], and Google Scholar [11]. These were chosen based on their prevalence in existing surveys and their accessibility. For each database, we will use 120 search queries crafted from the names of OWASP Top 10 vulnerabilities. For each search, we will only examine the first 100 results, as some search engines produce several thousand results, which are often not relevant to the search itself. This means that we expect the maximum amount of results to be 72,000. Due to this amount, each author will go through half of the databases. For each result, we will manually assess its relevance by reading its title and abstract. Then we will read in their entirety results found to be relevant in the previous step. Finally, results passing the previous step will be cross-examined by both authors. The classification according to OWASP Top 10 will be based on the claimed coverage of the analysed publications.

3.3 Reported performance and reliability analysis

The metrics used in the reported performance of APR tools may vary greatly from one publication to another. However, drawing from [33], we formulate the following metrics we will look for in each publication:

- Time used: the total number of seconds the method took to detect all vulnerabilities and generate all patches;
- LOC: the total number of lines of code on which the method was tested;
- Vulnerabilities detected: the number of vulnerabilities detected by the method;
- Patches generated: the number of patches that were generated by the method;
- True positives: the number of generated patches that actually fixed a vulnerability;
- False positives: the number of generated patches that did not fix a vulnerability;
- Success rate: the ratio of successful trials to total trials in repairing one vulnerability;
- NCP: the number of candidate patches that were generated prior to the valid patch.

It is challenging to directly compare different approaches, as their reported performance metrics are often very different or defined in different ways. Hence, a quantitative analysis of the approaches' reported performance becomes virtually infeasible as it is impossible to directly relate their results. Therefore, the approaches will be mostly qualitatively assessed and compared.

3.4 Test suite analysis

For each Java test suite, we will use the following metrics based on the observations made in [7]:

- LOC: the total number of Lines of Code in a test suite, which affects a tool's time performance and capacity to handle significant loads;
- Number of test cases: ideally, a test suite should contain many test cases, in order to present the most diverse testing environment;
- Size difference of test cases: test suites should have test cases of varying size;
- Type of code: if a test case consists of natural code (i.e. the test case comes from a real full-sized program), it will present a more realistic setting, but it will be more difficult to assess how many vulnerabilities are present in it, and their details (which vulnerability class, what the most appropriate patch is...). This makes assessing a tool's correctness and completeness more difficult. Conversely, if a test case consists of synthetic code (i.e. the test case was written for the express purpose of being used in a test suite), all details about the vulnerabilities present in the test case will be known. However, the setting will be less realistic, and such test cases also tend to be smaller. An ideal test suite would contain a combination of both types of code;
- Availability: the test suite should be easily and openly available for researchers, in order to promote usage and peer-review;
- Vulnerability classes: the test suite should cover most possible classes of vulnerabilities. In this analysis we will look at which OWASP Top 10 classes each test suite covers;

The degree of availability will be measured on a scale defined as follows:

- Not available: the test suite has not been found despite our best efforts;
- Low: the test suite was found through great efforts, for instance through contacting the test suite's authors or through the use of the Wayback Machine [5];
- Medium: the test suite was found, but is not found easily by using a search engine;
- High: the test suite can be found in the first few pages of a search engine's results.

We will also, where it is possible, assess what percentage of each OWASP Top 10 category each test suite covers, based on the CWEs they cover and using CWE's mapping [23] between OWASP Top 10 and CWE.

In order to get the information needed to analyze the test suites, we will download each test suite and use SLOCCount [43] to find the number of LOC. Vulnerability classes will be elicited primarily based on descriptions found inside the test suites (i.e. annotations in test cases or documentation) and, should this prove insufficient, on any other external trustworthy documentation. If the correct version for a test case is not found, we will not analyze other versions we find, as vulnerabilities present in one version might not be present in another. We will find the systematic test suites through searching in the same databases we use for the papers.

4 RESEARCH RESULTS

4.1 RQ1

The results of our search are presented in Table 1, which shows the number of results for each database or search engine. The total number of results amounts to about half of the expected number of results.

Database/search engine	Number of results
Oria	7,721
IEEE Xplore	354
ACM Digital Library	11,777
Engineering Village	881
Springer Link	7,295
Google Scholar	10,067
Total	38,095

Table 1: Results per database

From these 38,095 results, 27 were admitted to the cross-examination phase. During this phase, we found one duplicate, one publication that relied on user interaction to repair its target, and five papers that implemented so-called runtime approaches which do not modify the source code. All these publications were removed, which leaves 20 publications that were inside the scope we had defined. While we unfortunately did not take note of which papers were rejected before the cross-examination, they were largely similar both in nature and in proportion as the papers rejected in the cross-examination phase. These are summarised and classified according to the OWASP Top 10 categories they cover and their target language in Table 2. What is immediately noticeable is that there are only two target languages, namely Java and PHP, despite the omnipresence of JavaScript in web development. The OWASP Top 10 coverage is summarised in Figure 1, which underscores the strikingly sparse coverage of OWASP Top 10 categories. The predominant categories are very clearly A1 - Injection and A7 - XSS, with 14 and 10 approaches targeting them respectively, followed by A3 - Sensitive Data Exposure with 6 approaches. The other OWASP Top 10 categories are scarcely evaluated, and half of them are not evaluated at all. This may be in part due to the fact that most of the targeted vulnerabilities can be solved with proper input sanitization, which is a repair that is heavily standardised and can often be implemented by inserting a single line of code, and is hence easier to automatize.

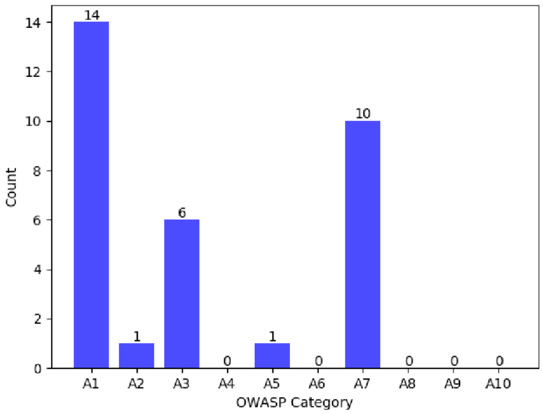

Figure 1: OWASP coverage histogram

4.2 RQ2

The reported performances of the surveyed papers are summarised in Table 3. P8 presented two distinct approaches, which were labeled P8-1 and P8-2. Some papers (such as P11) tested their approach on distinct test suites with different metrics, hence their reported performance is summarised individually for each test suite.

We may immediately see that two papers (P19 and P20) did not test their approach, making any analysis of their effectiveness very difficult. Also difficult to analyse are P17 and P14, due to their lack of reported performance metrics, or vagueness in their report (*"All"* patches were true positives in the case of P17, which, without the amount of generated patches, does not provide much information).

P2 suffers from a very small test suite of only 250 estimated LOC, made solely for the purpose of testing this approach, which increases the likelihood of test suite overfitting. It is difficult to determine its aptitude to repair vulnerabilities outside this test suite, a weakness recognised by the authors. The same problem is present in the case of P8-1, with a test suite of only 488 LOC.

Although its test suite is better than P8-1, and its number of generated patches is good, P8-2 does not report whether any of these patches were correct, which weakens its reliability. While having a test suite much larger than P8-1, P1 still suffers from the fact that its test suite is composed of a single test case, which is not enough to emulate the varied environment that is real-world vulnerability repair, and increases the probability of test suite overfitting. P9 presents the same problem, with the added issue of not testing whether the patches break any functionality, although its performance results remain impressive. P18 uses the same general technique (i.e. generating PreparedStatments) as P9, but is tested on a larger test suite composed of 9 test cases, which would indicate a greater reliability. Its biggest weakness is its omission of the time metric. Also generating PreparedStatements, P16 has been tested on an even larger test suite, but containing only 4 test cases. Interestingly, while the test suite is larger, the amount of vulnerabilities detected and patched is much lower in P16 than in P18 or P19, suggesting either that P16 is not as proficient, or that the test suites used by the two aforementioned papers were exceptionally vulnerable.

P5 uses a decently-sized test suite with 5 test cases of varying size, however the approach is non-deterministic and its success rate is not reported. Additionally, the time metric is not reported despite the use of a Genetic Algorithm, a method that may potentially take a substantial amount of time. It is uncertain then whether the reported performance would hold in a different setting. The same test suite as P5 is used by P6, which presents a deterministic method with a slightly better performance, although with a high number of detected, but not repaired, vulnerabilities. P7 uses a similar test suite as P6, but it is difficult to assess its performance due to the use of number of vulnerable files instead of number of vulnerabilities, which makes the report more imprecise.

A large test suite with 10 test cases was used by P3, however it suffers from a relatively high number of false positives, and uses an inordinate amount of time for a test suite of this size.

P15 uses a very well thought out approach to test its tool, validating patches both manually and through the help of the developers of the vulnerable applications. Its test suite is also extremely large, with 8,640 test cases. However, it is not reported how many vulnerabilities were found in each application, nor how large the apps were, which makes it harder to compare with other methods.

The test suite used by P4 is one of the largest in this survey, and contains 6 different test cases of varying size. The paper reports an important amount of detected vulnerabilities, however only less than half of these are repaired, albeit with no false positives.

P10, P11, P12, and P13 all relate to the same tool, WAP, with each paper introducing improvements on the previous one. The tool is thoroughly tested with several huge test suites containing both synthetic and natural code, and a large amount of test cases of varying size. In terms of reliability, the only aspect that may be commented on, is the suspiciously low amount of false positives considering the amount of false positives in the real world, which is a trend we observe throughout the surveyed papers.

It is interesting to note that most test suites in this survey are ad-hoc and with natural code: about 70% of test cases are made of open source web applications, and only two papers use formal test suites. Barely 8% of test cases used synthetic code, and most in the form of purposefully vulnerable web applications.

ID	Name	Target language	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	
P1	Automated Detecting and Repair of Cross-Site Scripting Vulnerabilities through Unit Testing	Java							x				[24]
P2	Using Automated Fix Generation to Secure SQL Statements	Java	x										[38]
P3	Fix Me Up: Repairing Access-Control Bugs in Web Applications	PHP					x						[37]
P4	Auto-locating and fix-propagating for HTML validation errors to PHP server-side code	PHP			x								[29]
P5	An Approach for Cross-Site Scripting Detection and Removal Based on Genetic Algorithms, Removing Cross-Site Scripting Vulnerabilities from Web Applications using the OWASP ESAPI Security Guidelines	Java							x				[12]
P6	Automated removal of cross site scripting vulnerabilities in web applications	Java							x				[36]
P7	Monitoring Web Applications for Vulnerability Discovery and Removal Under Attack	PHP	x						x				[4]
P8	Automatic Detection and Repair of Input Validation and Sanitization Bugs	PHP	x						x				[3]
P9	Automated Fix Generator for SQL Injection Attacks	PHP	x										[8]
P10	Securing Energy Metering Software with Automatic Source Code Correction	PHP	x		x				x				[19]
P11	Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives	PHP	x		x				x				[20]
P12	Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining	PHP	x		x				x				[21]
P13	Equipping WAP with WEAPONS to Detect Vulnerabilities	PHP	x	x	x				x				[22]
P14	Preventing SQL Injection through Automatic Query Sanitization with ASSIST	Java	x										[27]
P15	CDRep: Automatic Repair of Cryptographic Misuses in Android Applications	Java			x								[18]
P16	On automated prepared statement generation to remove SQL injection vulnerabilities	PHP	x										[39]
P17	Automatic Detection and Correction of Vulnerabilities using Machine Learning	PHP	x						x				[40]
P18	TAPS: automatically preparing safe SQL queries	PHP	x										[6]
P19	Prevention of attack on Islamic websites by fixing SQL injection vulnerabilities using co-evolutionary search approach	Java	x										[41]
P20	Code-motion for API migration: fixing SQL injection vulnerabilities in Java	Java	x										[1]

Table 2: Papers found in the survey, classified according to OWASP Top 10. A cross (x) indicates that the approach covers a vulnerability contained within the OWASP Top 10 category. An ID is given to each paper for easier referencing.

4.3 RQ3

None of the 20 test cases contained within the 5 Java test suites used in the surveyed papers were found, most links having expired and surviving versions of programs being different from the ones used. P15's 8,640 apps were not found either. Hence, we only analyse the two systematic test suites.

Juliet and VAKB are strikingly different (see Table 4): VAKB dwarfs Juliet in terms of total number of LOC, but contains much less test cases, with 164 of them even being unavailable. The size difference between the smallest and largest test case is also very different: Juliet's largest test case is about 84 times larger than its smallest, while VABK's largest test case is more than 5,800 times larger than its smallest. This is explained by the fact that Juliet is entirely composed of synthetic code, while VAKB is entirely composed of natural code. Yet, they complete each-other in their coverage of OWASP Top 10 categories, and being of different types of code, are intended for different types of testing loads. It should be noted that we could not find the repository or fix commit for 164 test cases in VAKB.

Test suite	Juliet	VAKB
Type of code	Synthetic	Natural
Total number of LOC	2,670,821	344,886,942
Number of test cases	44,105	1,282 (1,118 usable)
Largest LOC	335	1,819,277
Smallest LOC	4	311
Vulnerability classes	A1, A2, A3, A5, A6, A7, A10	A1, A2, A3, A4, A5, A6, A7, A8
Degree of availability	High	High

Table 4: Overview of the Juliet and VAKB test suites

Paper	Time used (s)	LOC	Vulnerabilities detected	Patches generated	True positives	False positives	Success rate	NCP
P1	493	112,000	24	24	24	0	N/A	49
P2	Unreported	250	5	5	0	0	N/A	Unreported
P3	8,890	201,508	37	37	30	7	N/A	Unreported
P4	20.3	427,000	2,547	982	982	0	N/A	Unreported
P5	Unreported	109,835	201	201	201	0	Unreported	Unreported
P6	Unreported	109,835	753	231	197	34	N/A	Unreported
P7	Unreported	104,503	174	23 files	23 files	0	N/A	Unreported
P8-1	3.05	488	13	13	13	0	N/A	Unreported
P8-2	633.15	26,440	278	278	Unreported	Unreported	N/A	Unreported
P9	0.4225	37,000	328	328	328	0	N/A	Unreported
P10	Unreported	7,843	23	23	20	3	N/A	Unreported
P11(1)	473	470,496	294	Unreported	Unreported	Unreported	N/A	Unreported
P11(2)	Unreported	157,724	83	83	83	0	N/A	Unreported
P12(1)	557	1,381,943	388	Unreported	Unreported	Unreported	N/A	Unreported
P12(2)	Unreported	153,034	83	83	83	0	N/A	Unreported
P13(1)	123	1,196,702	413	309	291	18	N/A	Unreported
P13(2)	Unreported	Unreported	169	166	164	2	N/A	Unreported
P14	5,880	57,261	Unreported	Unreported	All	0	N/A	Unreported
P15(1)	166,752	Unreported (8,640 apps)	8,582 apps	8,582 apps	Unreported	Unreported	N/A	Unreported
P15(2)	24,356.6	Unreported (1,262 apps)	1,262 apps	1,232 apps	1,193 apps	99	N/A	Unreported
P16	Unreported	75,345	68	52	52	0	N/A	Unreported
P17	Unreported	Unreported (50 pages)	15	Unreported	Unreported	Unreported	N/A	Unreported
P18	Unreported	46,900	1,030	1,002	1,002	0	N/A	Unreported
P19	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
P20	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 3: Summary of all reported performance. "Unreported" indicates that a metric has not been reported by the paper, and "N/A" means that the paper has not tested its approach. The "False positives", when not reported, were calculated by subtracting the number of true positives from the number of generated patches.

The number of LOC per OWASP category for each test suite (Figure 2) emphasises the massive difference in size between the test suites, but also indicates that they have a very different focus, with Juliet being more focused on A1, and VAKB on A5. It should be noted that A9 does not have any corresponding CWEs, and is hence omitted.

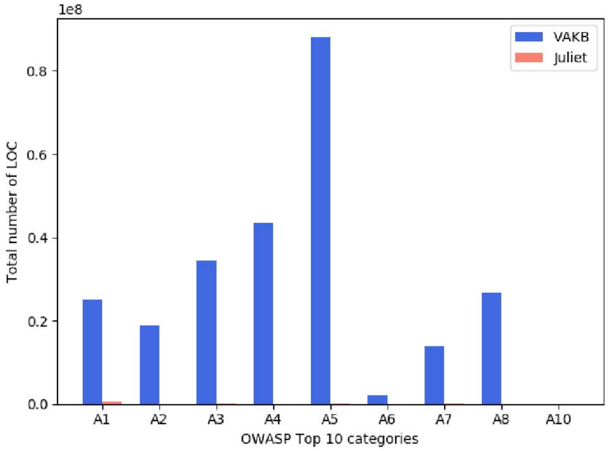

Figure 2: Number of LOC per OWASP category in Juliet and VAKB

This is further reflected in the number of test cases per OWASP category (Figure 3), where Juliet also seems to emphasize A7. However, it also shows that Juliet heavily emphasises A1 and A7, and seems to lack test cases for the other OWASP Top 10 categories. The number of test cases for A10 in particular is lacking. When contrasted with Figure 2, it underscores how Juliet and VAKB have very different approaches: Juliet has many very small test cases, while VAKB has fewer but much larger test cases.

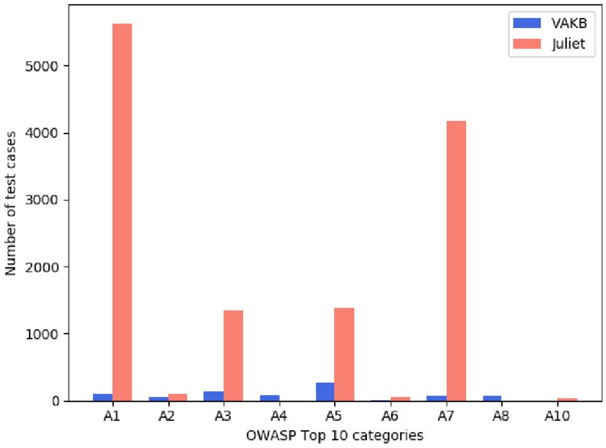

Figure 3: Number of test cases per OWASP category in Juliet and VAKB

Finally, looking at what percentage of OWASP's CWEs each test suite has in each category (Figure 4), we notice that VAKB seems to also have a better coverage of each OWASP category individually in addition of covering more OWASP categories. It should be noted that while VAKB covers 100% of A4, A7, and A8, these categories are also the smallest, containing only one or two CWEs. Conversely, A6 contains many very precise and small CWEs, which somewhat skews the coverage results.

However, what is certain is that both test suites are still very far from completely covering every OWASP Top 10 category.

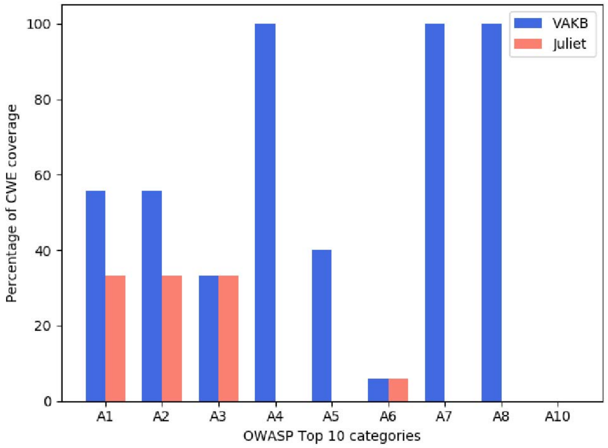

Figure 4: Coverage of OWASP Top 10 of Juliet and VAKB

5 DISCUSSION

5.1 Comparison with related work

Shafiq et al. effectuated a survey like us, although we restricted our scope to APR of vulnerabilities, and to source code modification. This increased focus allowed us to identify 19 approaches that Shafiq et al. had not classified, despite 18 of them being published before their literature review, and being inside their acceptance criteria. Thus, all but one approach we surveyed had not been analyzed or categorised by Shafiq et al. We also performed a deeper analysis of each approach, as we looked at their performance in the context of their test suite, which we also analyzed.

The approach of King et al. is fundamentally different from our approach, as they focused on benchmarking the tools they surveyed on a common test suite, while we compared tools by analyzing their reported performance and test suite. Our approach allowed for more tools to be analysed, with the additional analysis of test suites giving greater insights into how future APR tools should be tested.

The benchmarking efforts of Durieux et al. gave interesting insight into the potential widespread test suite overfitting in APR, and possible mitigation strategies. Our results seem to indicate a similar tendency in APR of vulnerabilities, seeing that almost all of the test suites used to assess the performance of approaches we analyzed were of low quality and arbitrarily chosen, which might be an indication of test suite overfitting.

The work on performance metrics for evaluating APR done by Qi et al. was valuable to decide on which metrics we should use when comparing tools, and some of their recommended metrics were used in this survey. We augmented these with metrics that were valuable to describe tools aiming at repairing security vulnerabilities. Our research also showed that in the case of tools repairing security vulnerabilities, many of the metrics that Qi et al. identified as being very common were not often reported by our papers (such as NCP, Simplicity, Robustness, Patch maintainability, and number of fitness evaluations).

Despite Monperrus' extensive bibliography of APR publications, our focus on approaches targeting OWASP Top 10 vulnerabilities still allowed us to discover 18 publications that were not included in Monperrus' living review.

5.2 Limitations (threats to validity)

During the systematic search, we manually reviewed the results obtained from each database and included them based on our perception of whether they were within our scope, which may have lead to some research bias. This bias however is mitigated by our use of cross-examination during the later stages of the publication selection.

The use of MITRE's mapping between OWASP Top 10 and CWEs may also have influenced some of the findings in RQ3, as we found some irregularities with the mapping during the survey (for instance, CWE-80 Basic XSS is not considered as being part of A7 - XSS).

The papers presented in the survey were not implemented nor tested by us, thus the performance metrics presented by the authors are taken at face value. We did attempt to identify unreliable results, however this remains theoretical.

6 CONCLUSION AND FUTURE WORK

This survey reveals a very uneven coverage of OWASP Top 10 in APR, with categories such as A1 and A7 being the subject of multiple papers, while other categories are not even mentioned. There is room for improvement in this respect, and future work should probably focus on other categories than A1 or A7. It is also interesting to note that many papers rejected from this survey implemented runtime APR, i.e. they did not modify source code. Our work shows that few researchers adequately test their approaches, which corroborates previous research that pointed to a lack of standardised performance assessment. A standard performance measure approach and a well-designed standardised test suite would greatly facilitate comparisons between tools. A good basis of test suites is present in the form of Juliet and VAKB, which could be extended to cover the missing OWASP Top 10 categories. Regardless of the test suite used, it should be easily and durably available for other researchers to analyse and use, as most test suites found in this survey were no longer available.

REFERENCES

[1] Aharon Abadi, Yishai A. Feldman, and Mati Shomrat. 2011. Code-motion for API Migration: Fixing SQL Injection Vulnerabilities in Java. In *Proceedings of the 4th Workshop on Refactoring Tools* (Waikiki, Honolulu, HI, USA) *(WRT '11)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/1984732.1984734

[2] ACM. [n.d.]. ACM Digital Library. https://dl.acm.org. [Online; accessed 16.12.2019].

[3] Muath Alkhalaf. 2014. Automatic Detection and Repair of Input Validation and Sanitization Bugs.

[4] Paulo Antunes. 2018. Monitoring web applications for vulnerability discovery and removal under attack.

[5] Internet Archive. [n.d.]. Wayback Machine. https://archive.org/web/. [Online; accessed 12.12.2019].

[6] Prithvi Bisht, A. Sistla, and V. Venkatakrishnan. 2010. TAPS: automatically preparing safe SQL queries. *Lecture Notes in Computer Science* 6052, 645–647. https://doi.org/10.1007/978-3-642-14577-3_21

[7] Thomas Durieux, Fernanda Madeiral Delfim, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2, 141 Bugs and 23, 551 Repair Attempts. *CoRR* abs/1905.11973 (2019). arXiv:1905.11973 http://arxiv.org/abs/1905.11973

[8] F. Dysart and M. Sherriff. 2008. Automated Fix Generator for SQL Injection Attacks. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. 311–312. https://doi.org/10.1109/ISSRE.2008.44

[9] Elsevier. [n.d.]. Engineering Village. https://www.engineeringvillage.com/search/quick.url. [Online; accessed 16.12.2019].

[10] L. Gazzola, D. Micucci, and L. Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (Jan 2017), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[11] Google. [n.d.]. Google Scholar. https://scholar.google.com. [Online; accessed 16.12.2019].

[12] Isatou Hydara, Abu Bakar Md Sultan, Md Sultan, Hazura Zulzalil, and Novia Admodisastro. 2014. An Approach for Cross-Site Scripting Detection and Removal Based on Genetic Algorithms.

[13] IEEE. [n.d.]. IEEE Xplore. https://ieeexplore.ieee.org/Xplore/home.jsp. [Online; accessed 16.12.2019].

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[15] A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani. 2016. On the evaluation of automatic program repair techniques and tools. In *2016 24th Iranian Conference on Electrical Engineering (ICEE)*. 61–66. https://doi.org/10.1109/IranianCEE.2016.7585390

[16] Xianglong Kong, Lingming Zhang, W. Eric Wong, and Bixin Li. 2018. The impacts of techniques, programs and tests on automated program repair: An empirical study. *Journal of Systems and Software* 137 (2018), 480 – 496. https://doi.org/10.1016/j.jss.2017.06.039

[17] Yuzhen Liu, Long Zhang, and Zhenyu Zhang. 2018. A Survey of Test Based Automatic Program Repair. *JSW* 13 (2018), 437–452.

[18] Siqi Ma, David Lo, Teng Li, and Robert Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. 711–722. https://doi.org/10.1145/2897845.2897896

[19] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2013. Securing energy metering software with automatic source code correction. *IEEE International Conference on Industrial Informatics (INDIN)*, 701–706. https://doi.org/10.1109/INDIN.2013.6622969

[20] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2014. Automatic detection and correction of Web application vulnerabilities using data mining to predict false positives. *WWW 2014 - Proceedings of the 23rd International Conference on World Wide Web*, 63–74. https://doi.org/10.1145/2566486.2568024

[21] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2015. Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Transactions on Reliability* 65 (08 2015), 1–16. https://doi.org/10.1109/TR.2015.2457411

[22] Iberia Medeiros, Nuno Ferreira Neves, and Miguel Correia. 2016. Equipping WAP with WEAPONS to Detect Vulnerabilities: Practical Experience Report. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2016), 630–637.

[23] MITRE. [n.d.]. CWE VIEW: Weaknesses in OWASP Top Ten (2017). https://cwe.mitre.org/data/slices/1026.html. [Online; accessed 12.12.2019].

[24] Mahmoud Mohammadi, Bei-Tseng Chu, and Heather Richter Lipford. 2018. Automated Detecting and Repair of Cross-Site Scripting Vulnerabilities. *CoRR* abs/1804.01862 (2018). arXiv:1804.01862 http://arxiv.org/abs/1804.01862

[25] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[26] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[27] Raymond Mui and Phyllis Frankl. 2010. Preventing SQL injection through automatic query sanitization with ASSIST. *Computing Research Repository - CORR* 35, 27–38. https://doi.org/10.4204/EPTCS.35.3

[28] Springer Nature. [n.d.]. Springer Link. https://link.springer.com. [Online; accessed 16.12.2019].

[29] Hung Nguyen, Hoan Nguyen, Tung Nguyen, and Tien Nguyen. 2011. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, 13–22. https://doi.org/10.1109/ASE.2011.6100047

[30] NIST. [n.d.]. Test suites. https://samate.nist.gov/SRD/testsuite.php. [Online; accessed 12.12.2019].

[31] OWASP. [n.d.]. The Ten Most Critical Web Application Security Risks, 2017. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf. [Online; accessed 02.11.2019].

[32] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cedric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories*. https://arxiv.org/pdf/1902.02595.pdf

[33] Yuhua Qi, Wenhong Liu, Weixiang Zhang, and Deheng Yang. 2018. How to Measure the Performance of Automated Program Repair. *2018 5th International Conference on Information Science and Control Engineering (ICISCE)* (2018), 246–250.

[34] SAP. [n.d.]. Open-source vulnerability assessment knowledge base. https://github.com/SAP/vulnerability-assessment-kb. [Online; accessed 02.11.2019].

[35] Hafiz Shafiq and zaki Arshad. 2014. *Automated Debugging and Bug Fixing Solutions: A Systematic Literature Review and Classification*. Ph.D. Dissertation. https://doi.org/10.13140/RG.2.1.4730.8889

[36] Lwin Khin Shar and Hee Beng Kuan Tan. 2012. Automated removal of cross site scripting vulnerabilities in web applications. *Information and Software Technology* 54 (05 2012), 467–478. https://doi.org/10.1016/j.infsof.2011.12.006

[37] Sooel Son, Kathryn S. Mckinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing access-control bugs in web applications. In *In Network and Distributed System Security Symposium*.

[38] Stephen Thomas and Laurie Williams. 2007. Using Automated Fix Generation to Secure SQL Statements. 9–9. https://doi.org/10.1109/SESS.2007.12

[39] Stephen Thomas, Laurie Williams, and Tao Xie. 2009. On automated prepared statement generation to remove SQL injection vulnerabilities. *Information and Software Technology* 51, 3 (2009), 589 – 598. https://doi.org/10.1016/j.infsof.2008.08.002

[40] R. Tommy, G. Sundeep, and H. Jose. 2017. Automatic Detection and Correction of Vulnerabilities using Machine Learning. In *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*. 1062–1065. https://doi.org/10.1109/CTCEEC.2017.8454995

[41] K. Umar, A. B. Sultan, H. Zulzalil, N. Admodisastro, and M. T. Abdullah. 2014. Prevention of attack on Islamic websites by fixing SQL injection vulnerabilities using co-evolutionary search approach. In *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. 1–6. https://doi.org/10.1109/ICT4M.2014.7020604

[42] UNIT. [n.d.]. Oria.no. https://oria.no. [Online; accessed 16.12.2019].

[43] David A. Wheeler. [n.d.]. SLOCCount. https://dwheeler.com/sloccount/. [Online; accessed 02.11.2019].