On the acceptance by code reviewers of candidate security patches suggested by Automated Program Repair tools

Registered Report of the Experimental Design

Aurora Papotti
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
a.papotti@vu.nl

Ranindya Paramitha
University of Trento
Trento, Italy
ranindya.paramitha@unitn.it

Fabio Massacci
University of Trento
Trento, Italy
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
fabio.massacci@ieee.org

ABSTRACT

Background: Testing and validation of the semantic correctness of patches provided by tools for Automated Program Repairs (APR) has received a lot of attention. Yet, the eventual acceptance or rejection of suggested patches for real world projects by humans patch reviewers has received a limited attention.
Objective: To address this issue, we plan to investigate whether (possibly incorrect) security patches suggested by APR tools are recognized by human reviewers. We also want to investigate whether knowing that a patch was produced by an allegedly specialized tool does change the decision of human reviewers.
Method: In the first phase, using a balanced design, we propose to human reviewers a combination of patches proposed by APR tools for different vulnerabilities and ask reviewers to adopt or reject the proposed patches. In the second phase, we tell participants that some of the proposed patches were generated by security specialized tools (even if the tool was actually a 'normal' APR tool) and measure whether the human reviewers would change their decision to adopt or reject a patch.
Limitations: The experiment will be conducted in an academic setting, and to maintain power, it will focus on a limited sample of popular APR tools and popular vulnerability types.

KEYWORDS

Automated program repair, patch adoption, security patches, code review

ACM Reference Format:
Aurora Papotti, Ranindya Paramitha, and Fabio Massacci. 2022. On the acceptance by code reviewers of candidate security patches suggested by Automated Program Repair tools: Registered Report of the Experimental Design. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'22). available as ArXiv 2209.07211.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.48550/arXiv.2209.07211

1 INTRODUCTION

The current trend in software development pushes for shorter release cycles [48] and several tools support developers in this process [44]. Unfortunately, the very process that generates quick updates and increase business opportunities is also a source of security vulnerabilities [29]. To address the trade-off between business opportunities and security risks [28], qualitative studies with developers have shown that developers would appreciate automated tools to quickly and correctly patch security vulnerabilities [34].

In this respect, Automated Program Repair (APR) tools could be a promising avenue. The field is mature and the performance and correctness of the different tools have been the subject to significant automated benchmarking studies performed by independent researchers [9, 11, 17, 25]. The applicability of APR tools designed for security has also been the subject of some comparative case studies [35]. Recently, specialized tools for security vulnerabilities such as SeqTrans [8], VuRLE [24], and SEADER [54] have also been recently proposed, albeit not yet independently evaluated.

Yet, it has been shown that patches identified by APR tools may have passed all automatic tests and still be 'semantically incorrect' in which they do not actually fix the bug or introduce bugs in other functionalities when subject to a manual validation [22, 47]. The percentages of 'semantically incorrect' patches is significantly high even for mature and well studies tools: out of 395 bugs present in Defetcs4J, the best performing tool could semantically fix only 54 bugs, even when given the exact bug-fixing position [22, Table 3 vs Table 5].

Developers using APR tools might face candidate patches that are semantically incorrect: this is a *change-based code review* problem [4, 10] where the only difference is that the patch comes from an APR tool rather than a human developer as it happens in companies [40] or open source projects [37, 38].

Hence, a partially unexplored overarching research question [7, 13, 53] is to understand whether *human code reviewers will be able to discriminate between correct and incorrect security patches submitted by APR tools* (RQ1). Since different vulnerabilities require different patterns it might be that human patch reviewers would be able to discriminate more effectively for some vulnerabilities than others. There might be a difference also among tools using different patch strategies but to explore this option in details one would require that the number of patches generated by each tool is significantly larger than the one currently available [22, 47].

Consequential to the research question above is understanding *whether code reviewers decisions will be actually influenced by knowing that some patches come from a specialized security tool* (RQ2). There is already some evidence in the literature that providing this information can influence code reviewers. Such bias might

Registered Report at ESEM'22, June 30, 2022, Finland, FI
2022. ACM ISBN Available as ArXiv 2209.07211...$15.00
https://doi.org/10.48550/arXiv.2209.07211

not be necessarily for the good as APR tools designed for security (APR4Sec) might not be necessarily more accurate than the general APR tools.

The purpose of this registered report is to present the design of an experiment with human code reviewers to understand the answers to the above two research directions.

2 RELATED WORK

Bug repair. Durieux et al. [11] performed a large-scale evaluation of eleven APR tools on 2,141 general bugs from five different benchmarks. The result of their experiment is not promising as expected, the tools were able to fix only 0.7-9.9% of the bugs. This study's outcome is affected by patch overfitting in the Defects4J dataset [16]. However, this study did not assess the correctness of the patches, and they provide only a comprehensive review of tools repairability. There are several studies [17, 23, 25, 47, 51] that assess APR patches correctness. These studies show that several APR generated patches from APR tools are incorrect.

Vulnerability repair. Among the studies that assess APR tools' effectiveness for security vulnerabilities, there is the paper of Le Goues et al. [20]. This study evaluated GenProg (a generic APR tool enhanced for fixing vulnerabilities) on the fixes for CVE-2011-1148. This paper focuses on the evaluation of a single vulnerability, therefore, there is not a complete evaluation of the tool's performances.

Several tools have been implemented specifically for vulnerability repairing. Abadi et al. [1] developed a tool specifically for fixing injection vulnerabilities by placing sanitizers on the right place in the code. More recently, SeqTrans [8], VuRLE [24], and SEADER [54] are tools that leveraged existing repair patterns to produce vulnerability patches. Despite these studies, an independent evaluation of the the repair rate on a broader set of vulnerabilities is still lacking as the the source code of SeqTrans [8], and SEADER [54] has been only recently published.

Another study, regarding security vulnerabilities, performed by Pinconschi et al. [35], compares ten APR tools for C/C++ programs, using the concept of PoV (Proof of Vulnerability) test. The aim of the test is to measure the success rate of an APR technique. However, this study lacks of semantic correctness assessments of the generated patches.

APR vs. developers. Even though there have been a lot of studies in APR evaluation, these machine-based evaluations have biases [22]. This fact encourages human studies with APR tools [7, 13, 45, 53] to understand how effective actually the patches in assisting development process. However, a study by Winter et al. [49] found that this kind of human studies is still rare, and make only 7% in the Living Review [31], even less for patch adoption experiments.

Key Novelty: Building on these studies [7, 13, 22, 45, 49, 53], we design an experiment with humans code reviewers to determine if Automated Program Repairs tools effectively support the identification of correct vulnerability fixes.

Code Review and Secure Coding practices. Over the years several empirical studies on code inspections have been conducted [18]. Code reviewing occupies expensive developer time, therefore, nowadays organizations are adopting modern code review technique [10]. Modern code review or change-based review [4], is widely used across companies [3, 40], and community-driven projects [37, 38].

We investigated into studies which explicitly asked for implementing secure coding practices. Naiakshina et al. [32, 33] conducted two experiments with 40 computer science students, who have been divided into two halves. The two groups received a different task description. One description did not mention security, the other one explicitly gave security instructions; As result, the group without security instructions did not store passwords securely.

There are few studies on checklists for contemporary review. Rong et al. [39], through a study with students, found that checklists were helping them during code review. In addition, Chong et al. asserts that students were are able to anticipate potential defects and create a relatively good code review checklist [9]. Finally, a reports by Gonçalves et al. [14], explores whether review checklists and guidance improve code review performances.

Braz et al. [5] explores both aspects of explicitly asking for secure coding practices and providing checklists. She conducted an online experiment with 150 developers, setting up three different treatments. The baseline treatment consists in asking to the participants to perform a code review without any special instructions. In another treatment she explicitly asked to the developer to perform the review from a security perspective. Finally, the third treatment additionally asked developers to use a security checklist in their review. The results showed that asking participants to focus on security increases the probability of vulnerability detection, besides, the presence does not significantly improve the outcome further.

Key Novelty: On the basis to these experiments [5, 32, 33], we plan to randomly assign the participants to two different treatments. One treatment is supposed to give to the participants a real security information about the APR tools provided. Contrarily, we want to provide a bogus information to the participants assigned to the other treatment.

3 RESEARCH QUESTIONS

We structure our study around the two research questions:

RQ1. *Will human code reviewers be able to discriminate between correct and incorrect security patches submitted by the APR tools?*

As of today, all APR tools are research tools, with a great variety of user experience. Their 'users', who are not the tool's inventors or competing researchers, are essentially novices. From an internal validity perspective, this is an advantage, as our 'users' know about the domain but not about the tool inner workings so they don't have a prejudiced prior belief on what the tool expected output should be.

We hypothesize that the number of wrong patches identified as wrong, will be higher than the number of correct patches identified as correct, and therefore adopted.

*H*1.1: *Wrong patches are more easily identified as wrong than correct patches are identified as correct.*

The practical impact of this hypothesis is that code reviews of APR patches is an effective last line of defense for gross mistakes. However, we further assume that is much harder to distinguish a partially correct patch from a correct patch.

*H*1.2: *Partially correct patches are equally identified as correct patches as actually correct patches.*

A further natural hypothesis is that specialized tools perform better than general purpose tools and therefore an higher number of correct patches suggested by security designed tools will be actually adopted by the code reviewers as they would more closely match what a security patch should be.

*H*1.3: *Patches from APR tools designed for Security are adopted more often than than patches suggested by generic APR tools.*

We have not specified whether such adoption happens *irrespective of correctness* of the suggested patches. We suspect that correctness would not make a difference as the pattern of the patch rather than its actual semantic correctness would be a key measure of identification. This hypothesis leads to our second question.

RQ2. *Will code reviewers decisions be actually influenced by knowing that some patches come from a specialized security tool?*

To answer this question we need to perform a modest deception of participants that has been already used for password testing [33]. We need to provide to a (random) subset of reviewer the true a bogus information that the one of the tool is a APR4Secwhen in reality it just a generic APR tool. We formulate our corresponding hypothesis as follows:

*H*2.1: *Both experimental and treatment groups will have same number of switches after revealing the security information.*

In other words, knowing that a tool is a security tool (even if is actually not such a tool) will create a bias into the decision making process of the code reviewer. Further, we hypothesize that after revealing the security information, the participants will tend to adopt the patches suggested by the security designed tools. Therefore, our second formal hypothesis is:

*H*2.2: *The number of adopted patches from known security designed tools will be higher after the security information is revealed.*

4 EXPERIMENTAL PROTOCOL

Figure 1 summarizes the experimental protocol that we propose to address our research questions. We consider an APR tools set composed by a security tool A, and two generic tools B and C. In the first phase of the experiments all the APR tools are labeled as generic. In the second phase of the experiment we plan to give to one group the true security information, and to the other a bogus security information (the tools are labeled wrongly). We decided to set up different treatments on the basis of Braz et al.'s study [5][1].

4.1 Execution Plan

Training. The participants of the experiments have to complete two training phases: *(i)* Finding Vulnerabilities training, and *(ii)* Fixing Vulnerabilities training.

The first preliminary activity aims to demonstrate that giving to developers a slice of the file, instead of the full file (during code review), leads to finding more vulnerabilities. Participants will therefore be trained on the identification of vulnerabilities into code running for 1.5 hours two academic hours of 45 minutes. The slides give a general introduction about vulnerabilities, and which one are the most common ones. Then, we provide a more detailed explanation about injections, information disclosure, and denial of service vulnerabilities, and how to recognize them in the code. Then they will be asked to identify the vulnerabilities in both a fragment of the code and in the full file.

The duration of the training is an important experiment parameter. Comparing to other experimental activities this is considerably short: in the field of threat analysis and security requirements training sections last several hours [42, 50] or even days [46].

In the domain considered for this experiment, Chong et al. [9] performed the experiment after several weekly lectures with 60 minutes a week. On the opposite side of the spectrum, among the cited works, [45] provides only 10 minute tutorial. Several other works did some introduction or instruction or tutorial for their participants but they do not mention the length of the session [7, 13, 32, 33, 33, 53]. Other works [14, 39] do not mention any training for their participants. [5] mentions training as one of their control variable but they did not actually conduct any training as their experiment was an online experiment.

Eventually, we have opted for our construction by considering previous studies where professionals were involved and the minimum duration was indeed 1.5 hours [2, 15]. This is essentially also a typical session of professional training session performed in industry[2]. Also the survey by Kollanus et al. [18] mentions the use of "overview meetings" in most of the software inspection publications, which seems to imply that the duration was significant (as a 10 or 30 minutes presentation would hardly be considered a "meeting").

A second part of the training will happen a week later. The participants will receive a general introduction on how APR tools work with the IDE that will be used to perform the code reviews of the patches suggested by the APR tools. We will provide to the participants the instructions to install an IDE plugin. Once the plugin is installed, the vulnerable lines will be highlighted; then the students can choose which pair patch-tool to adopt.

Digital copies of lecture slides, technical documentation, etc. will be provided to the participants and it can be consulted at any time.

Experiment - Phase 1 The experiment consists of three hours of physical laboratory. The participants will be separated into different rooms according to their treatment (group) of belonging to avoid spillover effects. In this phase, there are no differences in the execution of the experiment between the two treatments. Each room is supervised by an experimenter whose role does not go beyond the supervision of the room and the technical support, s/he cannot reply to questions regarding the solution of the correct patches.

The participants in both treatments will have to download a plugin in the Software Development IDE VsCode; The plugin will suggest different patches from different APR tools for each vulnerability. The participants will have several projects to evaluate: they will run each project in their own environment, and use the plugin provided to analyze the highlighted vulnerability. Finally, they will select which patch to adopt to fix the vulnerability. On the Qualtrics submission tool, the participants will write, for each suggested patch, whether it is correct, partially correct, or wrong.

[1] We have already done a pilot, and the experimental plan in this work is adapted from the lesson-learned from the pilot.

[2] https://www.secura.com/services/people/training-courses/secure-programming-training

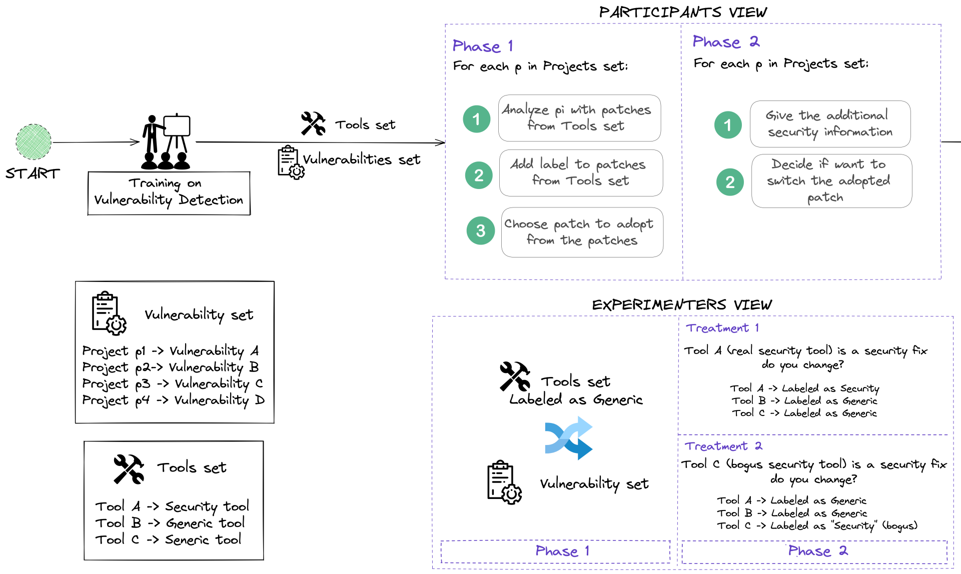
PARTICIPANTS VIEW
Phase 1
For each p in Projects set:
1
Analyze pi with patches from Tools set
2
Add label to patches from Tools set
3
Choose patch to adopt from the patches
Phase 2
For each p in Projects set:
1
Give the additional security information
2
Decide if want to switch the adopted patch
START
Training on Vulnerability Detection
Tools set
Vulnerabilities set
Feedback on the experiment process
FINISH
Vulnerability set
Project p1 -> Vulnerability A
Project p2-> Vulnerability B
Project p3 -> Vulnerability C
Project p4 -> Vulnerability D
Tools set
Tool A -> Security tool
Tool B -> Generic tool
Tool C -> Seneric tool
EXPERIMENTERS VIEW
Tools set
Labeled as Generic
Vulnerability set
Treatment 1
Tool A (real security tool) is a security fix do you change?
Tool A -> Labeled as Security
Tool B -> Labeled as Generic
Tool C -> Labeled as Generic
Treatment 2
Tool C (bogus security tool) is a security fix do you change?
Tool A -> Labeled as Generic
Tool B -> Labeled as Generic
Tool C -> Labeled as "Security" (bogus)
Phase 1
Phase 2

Figure 1: Steps of our experiment

Once the participants have analyzed all projects, they can move to the second phase of the experiment.

Experiment - Phase 2. In the second phase an additional information about tools' type is given to the participants. The students in the baseline treatment will receive a correct information about the APR tools' type. Unlike, the students in the second treatment will receive a bogus information. Once the students receive the additional information, they can review their answers for each project and decide whether to change their response, or keep their choice made during the first phase of the experiment.

Feedback The participants will have to reply few questions regarding their perception of the task. The questions will be answered through an ordinal scale, moreover, we plan to design an open question for who wants to give additional comments, and give suggestions.

Since the experiment may be long, we plan breaks between the experiment phases. Moreover, the participants can take breaks during the execution of the experiment, or stop at any time they want, and finish the experiment earlier.

4.2 Measurement Plan

The measurement plan describes where in the process we will collect the various variables that are further described in Section 5.

Randomization. We randomly separate the participants into two groups (treatments), in the variable *Assignment to a Treatment.*

Training. The purpose of the training is to collect information regarding the background of the participants as in the studies [5, 32, 33]. Knowing the participant's background help us to better evaluate the results that we collect in the experiment part. We collect the variables *Knowledge of Security Vulnerabilities*, *Knowledge of Java*, *Knowledge of Software Development IDE* by means of a Qualtrics questionnaire. For example we ask whether students attended the tutorial, a full course, have done vulnerability assessment in an internship or have true professional assessment.

We collect the *Time spent on the video*, whether to check if the students watched the video/material provided for the training part.

Experiment - Phase 1. The purpose of the first phase of the experiment is to determine if Automated Program Repairs tools effectively support the developer in the identification of correct vulnerability fixes. We collect the dependent variable to measure correctness for proposed (tool,patch) whether *Patch Classification* adoption by mean of a questionnaire and a log as an additional measure, i.e. we ask whether the patch for CVE-2013-4379 proposed by Arja is correct, partially correct, or wrong.

Experiment - Phase 2. The purpose of the second phase of the experiment is to determine if APR4Secare better than generic APR tools. Also, we want to know if the 'security' information will lead to switches irrespective of correctness. We collect the dependent variables *Path-Adoption after security information by pair (Tool, CVE)*, and *Number of switches after security information by pair (Tool, CVE)*.

Feedback. The experiment ends with few questions regarding the perception of the tasks. We collect three variables: *Process Understanding* to measure whether the participants had a clear understanding of the experiment, therefore, of the tasks, *Process Time* to check whether the participants had enough time to complete the experiment, *Process Training* to control whether the training material was sufficient to carry out the tasks and on open ended *Process Material* as a feedback if there was any other material that could have been useful or necessary to complete the tasks.

5 VARIABLES

Table 1 presents all the variables we consider in our experiment.

Independent variables

Assignment to a Treatment. For phase one, there will be diverse pairs of type of vulnerabilities and APR tool patches with different type of answers. For phase two, this variable describes the random assignment of each participant to one of the two treatments. For

Table 1: Experimental Variables

Name	Description	Operationalize
Independent variables (design)		
Assignment to a Treatment	Random assignments of participants to a treatment (true security information vs. bogus security information).	nominal (A)
Background variables		
Knowledge of Java	Self-reported concrete experience on Java	Ordinal scale (B)
Knowledge of Security Vulnerabilities	Self-reported concrete experience on vulnerability assessment	Ordinal scale (B)
Knowledge of SW Development IDE	Self-reported concrete experience on software development IDE	Ordinal scale (B)
Dependent variables		
Patch-Classification by pair (Tool, CVE)	Classification of patch by the participants.	Ordinal scale (C)
Patch-Adoption after security information by pair (Tool, CVE)	Classification of patch by the participants after receiving the security information.	Ordinal scale (C)
Switches after security information	Number of patches classified differently by a participant before or after receiving information on the alleged security nature of tool.	ratio (A)
Experiment Validation Variables		
Time spent on training video	time (in minutes) to watch the video	ratio (A)
Time spent on task	time (in minutes) to complete the task	ratio (A)
Process Understanding	the participants had a clear understanding of the experiment	Ordinal scale (D)
Process Time	the participants had enough time to complete the experiment	Ordinal scale (D)
Process Training	the training material was sufficient to carry out the tasks	Ordinal scale (D)
Process Material	material suggestions that can be useful to complete the tasks	Open Text
Perceived Usefulness (PU)	self-reported usefulness of the prescribed technique	Ordinal scale (D)

(A) Automatically performed by the Qualtrics submission tool
(B) Multiple choice: no experience, attended a tutorial, attended a course, company internship, professional practice
(C) Multiple choice: adopted, correct, partially correct, wrong
(D) 5-point Likert scale: strongly disagree, disagree, neither-agree-nor-disagree, agree, strongly agree

the experiment, we designed two different treatments: *(i)* in one treatment a true security information about the nature of the tools is given; *(ii)* in the other a bogus security information about the nature of the tools is given.

Background Variables

The purpose of background variables is to ascertain if they have any experience in different contexts (e.g. University projects, personal developed projects, or professional experience) that might impact the result of the experiment.
Knowledge of Java. Java experience of the participants.
Knowledge of Security Vulnerabilities. Vulnerability assessment experience of the participants.
Knowledge of Software Development IDE. Software Development IDE experience of the participants.

Dependent Variables

Patch-Classification by pair (Tool, CVE). This variable describes how participants classify the different patches proposed by the selected tools for the experiment. It is used to answer to **RQ1**.
We designed the next two variables to answer **RQ2**.
Patch-Adoption after security information by pair (Tool, CVE). This variable describes what happens once the participants receive the security information.
Number of switches after security information by pair (Tool, CVE). This variable is 1 if a user has changed his/her decision to adopt a patch after receiving the information of which patches have been suggested by a security tool and 0 if no change has been made.

Experiment Validation variables

We collect these variables in order to verify the correctness of the experimental process.
Time spent on training video With this variable we measure the time spent from each participant on the training video. Also, we verify how many participants actually completed the training.
Time spent on task We designed this variable to measure the time spent from each participant to classify the patches proposed for each project. We measure the time by how long the participants spend on each page in the online form assessment. We do not envisage to measure time (beside the previously mentioned timer as overall interval) as this is not a fully controllable variable. We realistically assume that in 20 minutes per vulnerability one should have made a choice.
Process Understanding We ask to the participants if they had a clear understanding of the experiment. We designed this variable to measure the overall clearness of the experiment, and therefore what is necessary to improve for the future.
Process Time We ask to the participants if the time provided to complete the experiment is enough.
Process Training We ask to the participants if the training is sufficient to complete the experiment. We measure this variable to collect feedback for future improvements of the experiment.
Process Material We measure if the provided material is useful to carry out the tasks.
Perceived usefulness (PU) We ask to the participants to evaluate the usefulness of the experiment.

Table 2: Vulnerabilities used in experiment.

CVE	CWE	Keyword	Project
CVE-2013-4378	79	XSS	javamelody/javamelody
CVE-2016-9878	22	Path Trav.	spring-projects/spring-framework
CVE-2018-1192	200	Info. Disc.	cloudfoundry/uaa
CVE-2018-1324	835	Inf. Loop	apache/commons-compress
CVE-2018-17202			apache/commons-imaging
CVE-2018-1000864			jenkinsci/jenkins
CVE-2019-10173	502	Data deserialz.	x-stream/xstream

Table 3: Tools used in experiment.

Tool	Platform	Type
ARJA [52]	Arja	Generic
RSRepair-A [52]		
Cardumen [27]	Astor	
jGenProg [26]		
jKali [26]		
TBar [21]	Independent	
SEADER [54]	Independent	Security
SeqTrans [8]		

6 GROUND TRUTH DATASET

We built a dataset that we intend to use for the execution of the experiment. The dataset contains a set of vulnerabilities and APR tools that have been selected according to two criterion.
Java Program Repairs. A study on the repair of C++ vulnerabilities already exists [35]. We want to focus on Java given the recent studies on normal fixes on Java [11].
Test-based. We want to check the performance in presence of test cases present in industrial projects (Maven/Gradle).

6.1 Choice of Vulnerabilities

We use the **Vul4J dataset** [6] as it is the only existing benchmark that satisfies our requirements. The dataset contains **79 vulnerabilities** from 51 real-world open-source Java projects (libraries, web frameworks, data-processing desktop apps, and CI/CD servers); it is extracted from the 'Project KB' knowledge base [36].

We ran the tools on all 79 vulnerabilities in the dataset. However, only 14 of them actually generated patch/es from the tools. From this 14 vulnerabilities, we chose 7 in Table 2 because *(i)* they cover different vulnerability classes and *(ii)* they have more than two (successfully-tested) patches from the selected tools.

6.2 Choice of Tools

We limit the tools scope to Java test-based APR tools for which we have sufficient in-house expertise to validate the correctness of the APR generated patches for the experiment. We selected both generic APR tools and, using four criterion.
Accessible Source Code. We want to eliminate some elements of uncertainty in the vulnerability identification, and want to use Maven testing pipeline.
Executable. We do not intend to use tools that are no longer maintained, or not executable due to technical issues.
Extensible to any dataset. We prefer to use tools that do not require major efforts to make it work with a different dataset than the tool's own recommended dataset.
Generating (security) patches. Each tool should possibly generate some patch for a vulnerability, even if the patch is not semantically correct.

Table 3 shows the APR tools that we selected. The applicability of the Maven test suite criteria is important for realism. We must be able to say to the participants that *the selected patches passed all the tests*. In this way the participants will be put in the frame of mind that the eventual acceptance of the patch only depends on their code review.

The source code for the two APR4Sectools SEADER [54] and SeqTrans [8] have just been released. Therefore, we did not have yet the chance to fully analyze them. We intend to use at least one of them for the experiment.

We have considered the list of tools in the living review of Monperrus [31] and analyzed the relevant literature on APR tools and APR4Sec. We chose the tools described in Table 3 as we mentioned in 6.2. We also limit the tool scope to Java test-based APR tools which then we can validate the correctness before using the patches for the experiment. We will add the APR tools designed for security (either SeqTrans [8] or SEADER [54]) to Table 4 after we get their correctness result on the chosen CVEs.

6.3 Applying Tools on Vulnerabilities

We have already started a preliminary analysis of the tools and the vulnerabilities that we intend to use. Table 4 describes the patches classification of the APR tools for each vulnerability (from the dataset experiment) made by the experimenters. A cell of the table can assume three different values. Each participant will be exposed to the same set of outputs corresponding to the patches of the vulnerabilities reported in Table 4. They will receive six files and when opening a file they will receive a warning in Visual Studio that a vulnerability has been found and they will have to choose a fix identified by some APR tool. For each vulnerability *all* generated patched will be shown. This design choice was made because some tools did *not* produce any patch for several vulnerabilities. Therefore, we do not have enough patches per tool to run an experiment where we expose participants to different tools. Since the participants are exposed to the same set of outputs, there is no real 'randomization'. However, the generation of patches was attempted from a wider sample of APR tools and some tools failed to report a patch while other succeeded (albeit with possibly a wrong or partially incorrect patches). We can consider this process a sufficient proxy for a randomising behavior (given the low success rate). We have considered the option of providing manually designed wrong patches but discarded it since they would not be generated by an APR tool and would not answer our research question as formulated.
C = Correct. The experiment organizers considered the vulnerability patch correct.
PC = Partially Correct. The vulnerability patch fixes the vulnerability but might introduce other functional errors as it changes the semantics of the execution in some non-trivial ways (in the non-vulnerable case).
W = Wrong. The vulnerability patch is just plausible, but it is wrong. It might have passed all regression tests and the specific test showing that the vulnerability is present, but according to

Table 4: Preliminary Analysis of Possible Tools

Card.: Cardumen. jGen: jGenProg. APR tools designed for security will be added by the time the experiment is carried out

CVE	ARJA	Card.	jGen.	jKali	RSRepair	TBar
CVE-2013-4378	PC				PC	PC
CVE-2016-9878		W	W	W		W
CVE-2018-1192	C	W	PC			C
CVE-2018-1324	C			C	C	W
CVE-2018-17202	PC	PC		PC	PC	
CVE-2018-1000864			W	W	W	W
CVE-2019-10173	PC	W	C	PC		

the experimenters, it is clearly a wrong patch as it changes the semantics of the execution in a drastically different way.

If a cell is empty, does not mean that we did not select the tool, it means that there is no patch suggested by the tool for that vulnerability. Note that Table 4 does not report the classification for the tools SEADER [54] and SeqTrans [8] because the source code has just been published. Thus, we did not have the chance yet to classify them; We plan to perform a further analysis, and include these security designed tools in the APR tools set for the experiment.

For the very same lack of maturity in users' interfaces, participants would not actually *use* the tools. They will receive the code changes recommended by the tool in a standardized and well known interface such as Visual Studio.

7 PARTICIPANTS

Our population is Master Computer Science students, with some differences in the elective courses and program choices. All the participants are students enrolled in the course Security Experiments and Measurements from VU Amsterdam. The course is taught by the experimenters. We decided to perform the experiment with students as in the studies [9, 32, 33, 39].

The experiment will be performed during class time of the course, and the purpose of this research methodology course is also to introduce the students to the critical issues behind design, execution, and measurements of security experiments.

In terms of learning outcome, through this experiment, the students have the possibility to critical review the results of the experiment and evaluate its statistical and practical significance. In the course we do not evaluate the number of correct responses given, but the student's capability to review and analyze the experiment results that we obtained.

As we plan to have students as participants, this may affect the outcome of the experiment. However, we believe that master students have enough experience to participate into the experiment, and we think that we can collect interesting results and insights that can help us with the future research. Moreover, obtaining significant results with students may suggest that we designed a relevant experiment that we can carry out with developers from companies in the future.

8 ANALYSIS PLAN

Data cleaning. We plan to perform a preliminary check on the collected data. All submissions without an explicit consent by the participants will be removed. Moreover, we will remove clearly invalid submission attempts if any, as measured by the process metrics.

Ground Truth. For the previous experiment we have manually evaluated all patches generated by the tools in advance and we compared them with the results of the participants. We plan to follow the same procedure for this experiment, and we plan to determine the correct number of results, measuring the true positive, and false positive rates.

Statistical Tests. As we think we will not obtain distributed samples, we plan to use a non-parametric, Mann-Whitney test. Some of our hypothesis are about equivalence of treatments. To answer them, we will use TOST as a test of equivalence which was initially proposed by [43] and is widely used in pharmacological and food sciences to check whether the two treatments are equivalent within a specified range δ [12, 30]. The underlying directional test will be again the Mann-Whitney test. In case we have too many zero values (i.e. many participants failed to recognize even *some* lines of code) we will investigate the use of the combined test proposed by Lachenbruch [19].

Validity threats. We acknowledge that students' background, knowledge, and practice may impact the experiment's results. However, as mentioned in Section 2, several studies have been performed with students [9, 32, 33, 39]. Moreover, Salman et al [41] shows a comparison between students and professionals to understand how well students represent professionals as experimental subjects in SE research. The results show that both subject groups perform similarly when they apply a new approach for the first time. We also acknowledge that the time measurement would not be exactly reflect the actual time the participants spend on the tasks. We plan to investigate these limitations with further studies. However, we believe that we can still get significance results, that will give us some strong basis to explore further in the future, and replicate the experiment in different contexts; such as with developers from companies. We also acknowledge that our sample is not representative of all developers since we are considering only master students from a single course. Therefore, to consider our study extensible and generalizable, more studies should be designed and run.

ACKNOWLEDGMENTS

This work has been partly supported by the European Union H2020 Program under the Grant 952647 (AssureMOSS - www.assuremoss.eu). We would like to thank Quang-Cuong Bui, Duc-Ly Vu, and Riccardo Scandariato for many useful discussions on APR tools and their assessment and Ákos Milánkovich for providing the test Visual Studio plug-ins we have used in the pilot.

CRediT statements

Conceptualization: AP, RP, FM; *Methodology:* FM, AP, RP; *Software:* not yet; *Validation:* not yet; *Formal analysis:* not yet; *Investigation:* AP, RP; *Resources:* not yet; *Data Curation:* RP (Vulnerability tests in Table 4); *Writing - Original Draft:* AP, RP *Writing - Review & Editing:* AP, RP, FM *Visualization:* AP *Supervision:* FM *Project administration:* FM *Funding acquisition:* FM

REFERENCES

[1] Aharon Abadi, Ran Ettinger, Yishai A Feldman, and Mati Shomrat. 2011. Automatically fixing security vulnerabilities in Java code. In *Proc. OOPSLA'11*. 3–4.

[2] Luca Allodi, Marco Cremonini, Fabio Massacci, and Woohyun Shim. 2020. Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. *Empir. Softw. Eng.* 25, 2 (2020), 1063–1094.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proc. IEEE/ACM ICSE'13*. IEEE, 712–721.

[4] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. Factors influencing code review processes in industry. In *Proc. ACM SIGSOFT FSE'16*. 85–96.

[5] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. 2022. Less is More: Supporting Developers in Vulnerability Detection during Code Review. In *Proc. IEEE/ACM ICSE'22*. 1317–1329.

[6] Quang Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *Proc. IEEE/ACM MSR'22*.

[7] José Pablo Cambronero, Jiasi Shen, Jürgen Cito, Elena Glassman, and Martin Rinard. 2019. Characterizing developer use of automatically generated patches. In *Proc. VL/HCC'19*. IEEE, 181–185.

[8] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* (2022).

[9] Chun Yong Chong, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2021. Assessing the students' understanding and their mistakes in code review checklists: an experience report of 1,791 code review checklist questions from 394 students. In *Proc. IEEE/ACM ICSE-SEET'21*. IEEE, 20–29.

[10] Jason Cohen. 2010. Modern code review. *Making Software: What Really Works, and Why We Believe It* (2010), 329–336.

[11] Thomas Durieux, Fernanda Madeiral, 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. In *Proc. ACM ESEC/FSE'19*. 302–313.

[12] Food and Drug Administration. 2001. Guidance for industry: Statistical approaches to establishing bioequivalence.

[13] Zachary P Fry, Bryan Landau, and Westley Weimer. 2012. A human study of patch maintainability. In *Proc. ACM SIGSOFT ISSTA'12*. 177–187.

[14] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2020. Do explicit review strategies improve code review performance?. In *Proc. IEEE/ACM MSR'20*. 606–610.

[15] Martina de Gramatica, Katsiaryna Labunets, Fabio Massacci, Federica Paci, and Alessandra Tedeschi. 2015. The role of catalogues of threats and security controls in security risk assessment: an empirical study with ATM professionals. In *Proc. REFSQ'15*. Springer, 98–114.

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *Proc. ACM SIGSOFT ISSTA'14*. 437–440.

[17] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. 2021. Evaluating automatic program repair capabilities to repair API misuses. *IEEE Transactions on Software Engineering* (2021).

[18] Sami Kollanus and Jussi Koskinen. 2009. Survey of software inspection research. *The Open Software Engineering Journal* 3, 1 (2009).

[19] Peter A Lachenbruch. 2002. Analysis of data with excess zeros. *Statistical methods in medical research* 11, 4 (2002), 297–302.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. IEEE/ACM ICSE'12*. 3–13.

[21] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proc. ACM SIGSOFT ISSTA'19*. 31–42.

[22] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817.

[23] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proc. ACM/IEEE ICSE'20*. 615–627.

[24] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*. Springer, 229–246.

[25] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[26] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proc. ACM SIGSOFT ISSTA'16*. 441–444.

[27] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Proc. SSBSE'18*. 65–86.

[28] Fabio Massacci and Ivan Pashchenko. 2021. Technical Leverage: Dependencies Are a Mixed Blessing. *IEEE Sec. & Priv.* 19, 3 (2021), 58–62.

[29] Fabio Massacci and Ivan Pashchenko. 2021. Technical leverage in a software ecosystem: Development opportunities and security risks. In *Proc. ACM/IEEE ICSE'21*. IEEE, 1386–1397.

[30] Michael Meyners. 2012. Equivalence tests–A review. *Food quality and preference* 26, 2 (2012), 231–245.

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

[32] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why do developers get password storage wrong? A qualitative usability study. In *Proc. ACM SIGSAC CCS'17*. 311–328.

[33] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. 2018. Deception task design in developer password studies: Exploring a student sample. In *Proc. USENIX SOUPS'18*. 297–313.

[34] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proc. ACM SIGSAC CCS'20*. 1513–1531.

[35] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A Comparative Study of Automatic Program Repair Techniques for Security Vulnerabilities. In *Proc. IEEE ISSRE'21*. IEEE, 196–207.

[36] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proc. IEEE/ACM MSR'19*. 383–387.

[37] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proc. ACM ESEC/FSE'13*. 202–212.

[38] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–33.

[39] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. 2012. The effect of checklist in code review for inexperienced students: An empirical study. In *Proc. IEEE CSEET'12*. IEEE, 120–124.

[40] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at Google. In *Proc. IEEE/ACM ICSE-SEIP'18*. 181–190.

[41] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *Proc. IEEE/ACM ICSE'15*, Vol. 1. IEEE, 666–676.

[42] Riccardo Scandariato, Kim Wuyts, and Wouter Joosen. 2015. A descriptive study of Microsoft's threat modeling technique. *Requirements Engineering* 20, 2 (2015), 163–180.

[43] DL Schuirmann. 1981. On hypothesis-testing to determine if the mean of a normal-distribution is contained in a known interval. *Biometrics* 37, 3 (1981), 617–617.

[44] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.

[45] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: a human study. In *Proc. ACM SIGSOFT FSE'14*. 64–74.

[46] Katja Tuma and Riccardo Scandariato. 2018. Two architectural threat analysis techniques compared. In *European Conference on Software Architecture*. Springer, 347–363.

[47] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proc. IEEE/ACM ASE'20*. 968–980.

[48] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *Proc. of ICSME'20*. IEEE, 35–45.

[49] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Saemundur O Haraldsson, and John Woodward. 2022. Let's Talk With Developers, Not About Developers: A Review of Automatic Program Repair Research. *IEEE Transactions on Software Engineering* (2022).

[50] Kim Wuyts, Riccardo Scandariato, and Wouter Joosen. 2014. Empirical evaluation of a privacy-focused threat modeling methodology. *Journal of Systems and Software* 96 (2014), 122–138.

[51] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 1–38.

[52] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering* 46, 10 (2018), 1040–1067.

[53] Quanjun Zhang, Yuan Zhao, Weisong Sun, Chunrong Fang, Ziyuan Wang, and Lingming Zhang. 2022. Program Repair: Automated vs. Manual. *arXiv preprint arXiv:2203.05166* (2022).

[54] Ying Zhang, Mahir Kabir, Ya Xiao, Na Meng, et al. 2021. Data-Driven Vulnerability Detection and Repair in Java Code. *arXiv preprint arXiv:2102.06994* (2021).