



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

Aurora Papotti¹ · Ranindya Paramitha² · Fabio Massacci^{1,2}

Accepted: 22 May 2024 / Published online: 3 August 2024
© The Author(s) 2024

Abstract

Objective We investigated whether (possibly wrong) security patches suggested by Automated Program Repairs (APR) for real world projects are recognized by human reviewers. We also investigated whether knowing that a patch was produced by an allegedly specialized tool does change the decision of human reviewers.

Method We perform an experiment with $n = 72$ Master students in Computer Science. 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.

Results It is easier to identify wrong patches than correct patches, and correct patches are not confused with partially correct patches. Also patches from APR Security tools are adopted more often than patches suggested by generic APR tools but there is not enough evidence to verify if ‘bogus’ security claims are distinguishable from ‘true security’ claims. Finally, the number of switches to the patches suggested by security tool is significantly higher after the security information is revealed irrespective of correctness.

Limitations The experiment was conducted in an academic setting, and focused on a limited sample of popular APR tools and popular vulnerability types.

Keywords Automated program repair · patch adoption · security patches · code review

Communicated by: Tayana Conte and Maria Teresa Baldassarre

This article belongs to the Topical Collection: *Registered Reports*.

✉ Aurora Papotti
a.papotti@vu.nl

Ranindya Paramitha
ranindya.paramitha@unitn.it

Fabio Massacci
fabio.massacci@ieee.org

¹ Foundational Security, Afdeling Informatica, Vrije Universiteit, Amsterdam, Netherlands

² Dipartimento di Ingegneria e Scienza dell’Informazione, Università di Trento, Trento, Italy

1 Introduction

The current trend in software development pushes for shorter release cycles (Wang et al. 2020a) and several tools support developers in this process (Shahin et al. 2017). Unfortunately, the very process that generates quick updates and increase business opportunities is also a source of security vulnerabilities (Massacci and Pashchenko 2021b). To address the trade-off between business opportunities and security risks (Massacci and Pashchenko 2021a), qualitative studies with developers have shown that developers would appreciate automated tools to quickly and correctly patch security vulnerabilities (Pashchenko et al. 2020).

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 (Durieux et al. 2019; Kechagia et al. 2021; Martinez et al. 2017; Chong et al. 2021). The applicability of APR tools designed for security has also been the subject of some comparative case studies (Pinconschi et al. 2021). Recently, specialized tools for security vulnerabilities such as SeqTrans (Chi et al. 2022), VuRLE (Ma et al. 2017), and SEADER (Zhang et al. 2021) have also been recently proposed, albeit not yet independently evaluated.

As argued by Pinconschi et al. (2021) different type of bugs have different type of impacts on program behaviour. while ‘normal’ bugs can cause unexpected failures that can be circumvented or avoided, security bugs makes it possible violation of system’s security properties. Security bugs require (beside the normal bugs of failure to increment a variable) also the existence of a control/data path from an externally controlled input to the bug in questions. The empirical evidence is that security bugs are typically harder to find and fix but are typically compactly located, a property that is consistent across the years (Zaman et al. 2011; Nguyen et al. 2016).

Most importantly from the perspective of automated repair, a ‘normal’ fix actually achieves the desired functionality whether a security fix bugs has to *both* eliminate the security vulnerability *and* maintain the functionality. For example, Bui et al. (2024) showed that for the tool SeqTrans (security APR tool) from 10 plausible patches only 4 are correct, 3 are fixing the vulnerability but they introduce functional errors, and 3 do not fix the vulnerability. Therefore, it is interesting to perform a human study the assess the correctness of the acceptance of the patches suggested by the APR tool.

The problem of identifying correct patches is of course not limited to security patches, ‘normal’ patches identified by APR tools may have passed all automatic tests and still be ‘semantically wrong’ in which they do not actually fix the bug or introduce bugs in other functionalities when subject to a manual validation (Liu et al. 2021; Wang et al. 2020b). The percentages of ‘semantically wrong’ patches are significantly high even for mature and well-studied tools: out of 395 bugs present in Defects4J, the best-performing tool could semantically fix only 54 bugs, even when given the exact bug-fixing position (Liu et al. 2021, Table 3 vs Table 5).

Developers using APR tools might face candidate patches that are semantically wrong: this is a *change-based code review* problem (Cohen 2010; Baum et al. 2016) where the only difference is that the patch comes from an APR tool rather than a human developer as it happens in companies (Sadowski et al. 2018) or open source projects (Rigby and Bird 2013; Rigby et al. 2014).

As Winter et al. (2022a) argues that few studies has been performed to investigate the interaction between developers and APR tools, it is important to perform experiments with human code reviewers.

Hence, a partially unexplored overarching research question (Fry et al. 2012; Zhang et al. 2022; Cambronero et al. 2019) is to understand whether *human code reviewers will be able to discriminate between correct and wrong 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 (Liu et al. 2021; Wang et al. 2020b).

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

1.1 Pre-registration and Data Availability Statement

The design of this experiment was published as a registered report in Empirical Software Engineering and Measurement 2022 (Papotti et al. 2022). The full protocol is available at the link <https://doi.org/10.48550/arXiv.2209.07211>.

2 Related Work

Bug Repair Durieux et al. (2019) 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 (Just et al. n.d.). 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 (Liu et al. 2020; Kechagia et al. 2021; Martinez et al. 2017; Ye et al. 2021; Wang et al. 2020b) that assess APR patches correctness. These studies show that several patches from APR tools are wrong.

Vulnerability Repair One of the first studies that assess generic APR tools' effectiveness for security vulnerabilities is the paper of Le Goues et al. (2012). 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 no complete evaluation of the tool's performance.

Several tools have been implemented specifically for vulnerability repair. ExtractFix (Gao et al. 2021) and Senx (Huang et al. 2019) use data from vulnerability exploits to generate vulnerability without any test-suites. Abadi et al. (2011) developed a tool specifically for fixing injection vulnerabilities by placing sanitizers on the right place in the code. Another study, regarding security vulnerabilities, performed by Pinconschi et al. (2021), 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 but it does not test the semantic correctness assessments of the generated patches which has only been recently done by Bui et al. for vulnerability patches in Java (Bui et al. 2024).

More recently, SeqTrans (Chi et al. 2022), VuRLE (Ma et al. 2017), and SEADER (Zhang et al. 2021) are tools that leveraged existing repair patterns to produce vulnerability patches also for the Java language. Despite these studies, an independent evaluation of the repair rate on a broader set of vulnerabilities in Java is still lacking as the source code of SeqTrans (Chi et al. 2022), and SEADER (Zhang et al. 2021) has been only recently published.

APR vs. Developers Even though there have been a lot of studies in APR evaluation, these machine-based evaluations have biases (Liu et al. 2021). This fact encourages human studies with APR tools (Zhang et al. 2022; Cambronero et al. 2019; Fry et al. 2012; Tao et al. 2014) to understand how effective actually the patches are in assisting the development process. Indeed, a recent study by Winter et al. (2022b) found that human studies are rare, and make only 7% in the Living Review (Monperrus 2018), even less for patch adoption experiments, let alone for vulnerability patches.

The concept of trustworthiness has been studied in the context such as social, business, etc. However, it is an important issue also in computer science, in particular it has an impact for organizations that intend to reuse, adapt, and extend code over time. Factors such reputation, transparency, and performance can influence the trustworthiness of code, moreover, well written code can facilitate the detection of malicious or poorly written code (Alarcon et al. 2017).

The first key question is whether human developers will actually accept an automatic suggested patch. To the best of our knowledge, the first work to study the adoption of automated patches by human developers has been the deployment of GetaFix at Facebook (Bader et al. 2019). In one case, for 250 suggested fixes the tool suggestions proved to be useful and accepted ‘as-is’ by 42% of the developers which raises to 61% if one consider adoption of semantically equivalent but stylistically different variations or lower ranked fixes. They authors also deployed GetaFix for Field Not Nullable and Return Not Nullable warnings, and the acceptance rate of fixes for both warnings is around 60%. Also, there is an increase in the fix rate when displaying displaying an auto-fix next to a warning resulted. Such success rate might be explained by one of the key feature of the GetaFix algorithm which mined *past human generated fixes* for similarity and pattern extraction.

Later studies did not show such wholehearted acceptance: Alarcon et al. (2020) performed a study with 51 programmers with at least 3 years’ experience and knowledge of the C programming language. The results show that the participants reported higher levels of perceived trustworthiness and intentions to reuse for human referents compared to automated code repair. Another study with 100 practitioners (Noller et al. 2022), showed that developer were willing to accept automated patches provided the number of suggested patch candidate be small (5-10 patches). This is also consistent with the observation of Bader et al. (2019) which actually recommended the candidate patches to be limited to one individual patch. Another requirement is that patches should be produced within a reasonable time. One needs to differentiate between what developers are considered acceptable and what is technically feasible. Indeed according to the technology acceptance model, developers may not intend to use a technology if their perceived usability is lower than

expected (Moody 2003). In this respect (Noller et al. 2022) suggest less than 1 hour, while Bader et al. (2019) set the boundary at 10 seconds. This might be a reasonable assumption as a preliminary study by Vu et al. (2021) has shown that even giving exponentially more time does not significantly improve the patch quality after the first few minutes.

In Ryan et al. (2019) they investigated programmer's perceptions of trustworthiness and trust in automated code repair, in particular, how those perceptions differ from code repaired by a human and by an APR tool. A half of the sample participants included experienced developers and the other half experienced developers. The results show that inexperienced developers find automated code repair more trustworthy than repairs made by a human, but the intentions of both experienced and inexperienced developers do not significantly differ to trust the automated code repair program after reviewing the repairs.

There is not agreement in the literature whether the developers prefer to be part of the loop in the correct patch identification. In the study (Noller et al. 2022) developers do not seem to be interested in a significant human-in-the-loop interaction with APR tools. The study by Winter et al. (2022a) instead declares that even though full automation may be a key a technical in terms of proving the efficacy of an APR system, developers do not necessarily be out of the loop. The authors highlight the need of more studies about the interaction between APR tools and developers as only few have been conducted.

Another key issue towards acceptance is whether the reputation of the tool might make a difference. In this respect there is no significant evidence in the literature. In the post experiment survey reported by Noller et al. (2022) only 9/103 developers indicated the reputation of an APR tool itself may be a factor that influences the trust for adopting a patch. To the best of our knowledge, the only studies that considered the reputation of the patch producers focused on the *human developer*. These studies considered the impact on the perceived trustworthiness of the patch's provenance in terms of the requesters' gender (Terrell et al. 2017), requesters' behaviour (Iyer et al. 2019), or social cues such as the participant's avatar (Ford et al. 2019). Reputational considerations based on contribution to well known repositories (e.g. by number of download or github stars) were not found even in eye tracking studies (Ford et al. 2019).

Key Novelty: Building on these studies (Liu et al. 2021; Zhang et al. 2022; Cambronero et al. 2019; Fry et al. 2012; Tao et al. 2014; Winter et al. 2022b; Bader et al. 2019), we design an experiment with human 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 (Kollanus and Koskinen 2009). Code reviewing occupies expensive developer time, therefore, nowadays organizations are adopting modern code review technique (Cohen 2010). Modern code review or change-based review (Baum et al. 2016), is widely used across companies (Bacchelli and Bird 2013; Sadowski et al. 2018), and community-driven projects (Rigby and Bird 2013; Rigby et al. 2014).

We investigated studies that explicitly asked for implementing secure coding practices. Naiakshina et al. (2017, 2018) conducted two experiments with 40 computer science students, who were divided into two halves. The two groups received different task descriptions. One description did not mention security, the other one explicitly gave security

instructions; As a result, the group without security instructions did not store passwords securely.

There are few studies on checklists for contemporary review. Rong et al. (2012), through a study with students, found that checklists were helping them during code review. In addition, Chong et al. asserts that students were able to anticipate potential defects and create a relatively good code review checklist (Chong et al. 2021). Finally, a reports by Gonçalves et al. (2020), explores whether review checklists and guidance improve code review performances.

Braz et al. (2022) 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 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 (Braz et al. 2022; Naiakshina et al. 2017, 2018), we randomly assigned the participants to two different treatments. One treatment is supposed to give the participants real security information about the APR tools provided. Contrarily, we provided 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 wrong 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 do not 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.

H1.1: Wrong patches are more easily identified as wrong than correct patches are identified as correct.

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

H1.2: 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 a 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.

H1.3: Patches from APR tools designed for Security are adopted more often than than patches suggested by generic APR tools.

We did not specify 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 use the same approach used for password testing (Naiakshina et al. 2018) with a minimal deception of the participants. We provide to a (random) subset of reviewers bogus information that one of the tools is a APR4Sec when in reality it is just a generic APR tool. We formulate our corresponding hypothesis as follows:

H2.1: Both experimental and treatment groups will have the 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 in 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 of 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 gave one group the true security information, and the other bogus security information (the tools are labeled wrongly). We decided to set up different treatments on the basis of Braz et al.'s study (Braz et al. 2022)

We have already done a pilot, in which we involved 42 computer science students (grouped in two) and after removing the answers without consent and unfinished answers, we are left with 16 sets of answers. We divided the participants into 2 groups to which we applied 2 different treatments: the first group got the complete version of the VSCode plugin that shows the unpatched and patched version side-by-side while the second group got the incomplete version that only highlights the vulnerable lines and show the patches but not side-by-side.

In this pilot experiment, we give the students 7 CVEs (the same CVEs used in the main experiment) and patches from 6 different tools (the same as the main experiment except

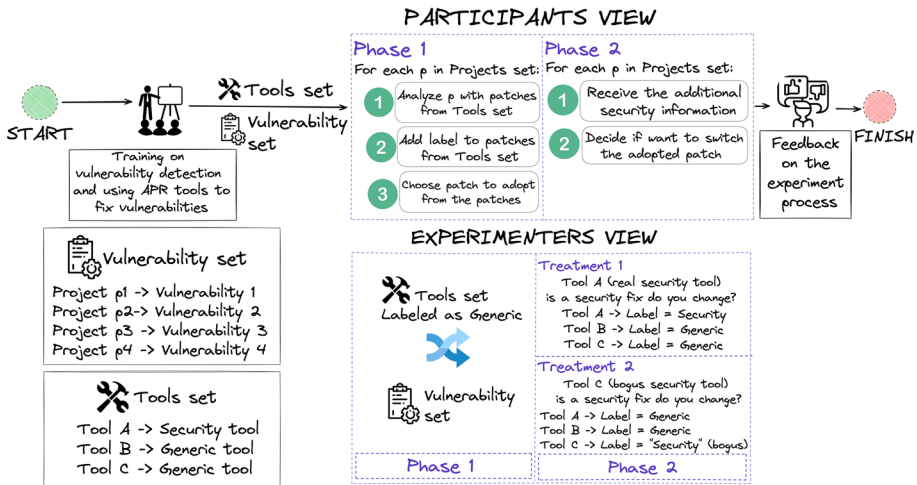


Fig. 1 Steps of our experiment

for SeqTrans). The result is summarized in Table 9 in the appendix. In our first pilot, we wanted to understand the usability of the plug-in, the time needed to fix the a vulnerability, and to show the different tools. Moreover, we wanted to make sure there were no confounding factors. Therefore, we performed a large scale pilot to assess the absence of designer issues (e.g. out of this we opted to fully integrate the patches into the Visual Studio Editor). Then we did a small scale pilot of the final experiment with PhD students to check the overall construction.

Overall, the experimental plan in this work is adapted from the lessons learned from the pilot, including: as there is no significant difference between the two treatments ($p > 0.05$), we decided not to use the same treatment difference in the main experiment and only pick one (using the complete version of the plugin) that is the same for everyone.

4.1 Ethical Approval

The students received a bonus point for the course for participating in the experiment. The ethical procedure was followed and it determined that a full ethical review was not necessary. In particular this was determined because (i) upfront, opt-in consent was asked, (ii) no personal or sensitive information was involved, (iii) it did not pose potential risks to either participants or researchers, (iv) the confidentiality of the participants was guaranteed by collecting data by GDPR compliant tool and removing their details before processing the data for the analysis, (v) the incentives to participate were minimal, (vi) and the participants were minimally deceived and thoroughly debriefed afterwards (they actually had full access to the anonymized data). There was no monetary compensation, and the participants received a compensation in terms of coursework's bonus. Such value was minimal (less than 2% of the final grade), and the participants could deny the consent, and still obtain the participation bonus. The personal details were only collected to grant the coursework's bonus.

4.2 Execution Plan

Training The participants of the experiments completed two training phases: (i) finding Vulnerabilities training, and (ii) using APR tools to fix vulnerabilities training. The purpose of the first phase was to help them to recognize vulnerabilities in the raw source code, and the second one on how to perform the same task when supported by APR tools (as we did not ask them to *fix* the vulnerability but to *adopt a fix*).

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 were therefore trained on the identification of vulnerabilities in code running for 1.5 hours (two academic hours of 45 minutes). The slides give a general introduction about vulnerabilities, and which ones 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 were 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 (Wuyts et al. 2014; Scandariato et al. 2015) or even days (Tuma and Scandariato 2018).

In the domain considered for this experiment, Chong et al. (2021) performed the experiment after several weekly lectures with 60 minutes a week. On the opposite side of the spectrum, among the cited works, Tao et al. (2014) provides only a 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 (Naiakshina et al. 2017, 2018; Cambronerio et al. 2019; Zhang et al. 2022; Fry et al. 2012). Other works (Rong et al. 2012; Gonçalves et al. 2020) do not mention any training for their participants. Braz et al. (2022) 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 (Allodi et al. 2020; Gramatica et al. 2015). This is essentially also a typical session of professional training session performed in industry¹. Also the survey by Kollanus and Koskinen (2009) 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”).

The second part of the training happened a week later than the first part. The participants received a general introduction on how APR tools work with the IDE that was used to perform the code reviews of the patches suggested by the APR tools. We provided the participants with 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. were provided to the participants and they can be consulted at any time.

¹ <https://www.secura.com/services/people/training-courses/secure-programming-training>



Fig. 2 View of the plug-in used by the participants. This an example of the view of the participants when using the plug-in into VS Code. The plug-in shows the identified faulty line and it shows the proposed the patches for that specific vulnerability

Experiment - Phase 1 The experiment consists of three hours of physical laboratory. The participants were 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 have to download a plugin in the software development IDE VSCode. The plugin suggests different patches from different APR tools for each vulnerability. The participants had several vulnerable code files to evaluate: they opened them in VSCode and used the plugin provided to analyze the identified vulnerability. The plug-in shows the faulty identified line, and it proposes a series of candidate patches for that vulnerability. Figure 2 shows an example of what the participants saw when using the plug-in. Among those, the participants had to select which one they thought it fixes the vulnerability. We did not provide a stack trace as it has no relevance: all the suggested patches satisfies the test cases, therefore, there is no need to use it. On the Qualtrics submission tool, the participants specified, for each suggested patch, whether it is correct, partially correct, or wrong.

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

Experiment - Phase 2 In the second phase, additional information about the tools' type is given to the participants. The students in the baseline treatment received correct information about the APR tools' type. Unlike, the students in the second treatment received 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 have to reply to a few questions regarding their perception of the task. The questions were answered through an ordinal scale, moreover, we also design an open question for those who want to give additional comments, and give suggestions.

Since the experiment was relatively long, we allowed breaks between the training and the experiment phases. Moreover, the participants could take breaks during the

execution of the experiment (e.g. going to the services), or finish the experiment earlier. We did not explicitly track whether the participants communicated with each other during the individual breaks. However, we did not observe groups of participants clustering together during the breaks..

4.3 Measurement Plan

The measurement plan describes where in the process we 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 (Braz et al. 2022; Naiakshina et al. 2017, 2018). Knowing the participant's background helps 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*, 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 Repair 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 means of a questionnaire and a log as an additional measure, e.g. we ask whether the patch for vulnerability 1 proposed by tool A is correct, partially correct, or wrong (Fig. 1).

Experiment - Phase 2 The purpose of the second phase of the experiment is to determine if APR4Sec are 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 *Patch-Adoption after security information by pair (Tool, CVE)*, and *Number of switches after security information by pair (Tool, CVE)*.

Feedback The experiment ends with a 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 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.

5.1 Independent Variables

Assignment to a Treatment For phase one, there were diverse pairs of types of vulnerabilities and APR tool patches with different types of answers. For phase two, this variable describes the random assignment of each participant to one of the two treatments. For the experiment, we designed two different treatments: (i) in one treatment true security information about the nature of the tools is given; (ii) in the other bogus security information about the nature of the tools is given.

5.2 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.

5.3 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.

Table 1 Experimental Variables

Name	Description	Operationalize
<i>Independent variables (design)</i>		
Assignment to a Treatment	Random assignments of participants to a treatment (true security information vs. bogus security information).	nominal (A)
<i>Background variables</i>		
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)
<i>Dependent variables</i>		
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)
<i>Experiment Validation Variables</i>		
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

5.4 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 by 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 by 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 measuring time (besides the previously mentioned timer as an 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 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 the participants if the time provided to complete the experiment is enough.

Process Training We ask 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 the participants to evaluate the usefulness of the experiment.

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 criteria.

Java Program Repairs A study on the repair of C++ vulnerabilities already exists (Pinconschi et al. 2021). We want to focus on Java given the recent studies on non security bug in Java (Durieux et al. 2019).

Test-based We want to check the performance in the presence of test cases present in industrial projects (Maven/Gradle).

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. Disclosure	cloudfoundry/uuaa
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

6.1 Choice of Vulnerabilities

We use the **Vul4J dataset** (Bui et al. 2022) 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 (Ponta et al. 2019).

We ran the chosen tools on all 79 vulnerabilities in the dataset. However, only 14 of them have generated patches from the tools (generated patches are the patch candidates that have passed the tool’s tests). From these 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 the sufficient in-house expertise to validate the correctness of the APR-generated patches for the experiment. We selected both generic APR tools and APR4Sec, using four criterion.

Accessible Source Code We want to eliminate some elements of uncertainty in 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.

Can Fix Multiple Kinds of Bugs or Vulnerabilities The selected tool should be able to fix different kinds of bugs or vulnerabilities and not limited to only one specific kind of bug/vulnerability.

Table 3 Tools used in experiment

Tool	Platform	Type
ARJA Yuan and Banzhaf (2018)	Arja	Generic
RSRepair-A Yuan and Banzhaf (2018)	Arja	Generic
Cardumen Martinez and Monperrus (2018)	Astor	Generic
jGenProg Martinez and Monperrus (2016)	Astor	Generic
jKali Martinez and Monperrus (2016)	Astor	Generic
TBar Liu et al. (2019)	Independent	Generic
SeqTrans Chi et al. (2022)	Independent	Security

We have considered the list of tools in the living review of Monperrus (2018) and analyzed the relevant literature on APR tools and APR4Sec. We then use the criteria listed above to choose the APR tools for our experiment, which we summarized in Table 3. 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 were put in the frame of mind that the eventual acceptance of the patch only depends on their code review. We also limit the generic tool scope to Java test-based APR tools which then we can validate the correctness before using the patches for the experiment.

For the APR4Sec, in the design of this experiment, we proposed either SEADER (Zhang et al. 2021) or SeqTrans (Chi et al. 2022). We decided to not use SEADER (Zhang et al. 2021) because it does not fulfill our last criteria as it targets only a specific kind of vulnerability (APW misuse). We instead chose SeqTrans (Chi et al. 2022) as it fulfills all the criteria we defined. SeqTrans is well-known in the community for being the class-leading and state-of-the-art one in Java APR4Sec world. While other tools have passed an end-to-end test in RepairThemAll framework (Durieux et al. 2019; Bui et al. 2024), we acknowledge that SeqTrans is not test-based, which means it can not be run with RepairThemAll. Therefore, to be fair on the ground-truth definition process, we ran end-to-end tests on SeqTrans patches ourselves with an automated framework mimicking the RepairThemAll framework (Durieux et al. 2019) that we use to evaluate the other APR tools. Other APR4Sec-tool such as ExtractFix (Gao et al. 2021) and Senx (Huang et al. 2019) are exploit based, which make them not fit our dataset which is test-based. Moreover, these tools cannot be used as they target a different programming language(C/C++ instead of Java). The latest does not have an available artefact.

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 was exposed to the same set of outputs corresponding to the patches of the vulnerabilities reported in Table 4. They received six files and when opening a file they received a warning in Visual Studio that a vulnerability has been found and they have to choose a fix identified by some APR tool. For each vulnerability *all* generated patches 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

Table 4 Preliminary Analysis of Possible Tools

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

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

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 others succeeded (albeit with possibly wrong or partially wrong patches). We can consider this process a sufficient proxy for a randomizing behavior (given the low success rate). We have considered the option of providing manually designed wrong patches but discarded them since they would not be generated by an APR tool and would not answer our research question as formulated.

We adopt the patch classification by Bui et al. (2024) to distinguish the patches used as our ground truth. Bui et al. (2024) did a systematic manual analysis by 3 experts to classify the patches (plausible ones) until they reached a consensus. The patches are classified in 3 classes as follows:

C = Correct A patch is considered correct if it passes the end-to-end test, fixes the security vulnerability, and preserves the functionality of the program (Bui et al. 2024).

PC = Partially Correct This type of patch refers to security-fixing patch in Bui et al. (2024): 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 the experimenters, it is clearly a wrong patch as it changes the semantics of the execution in a drastically different way.

We have not adopted in this paper the exact wording from Bui et al. (2024) because we have presented to our participants the choice between correct, partially correct, and wrong patch. Using a terminology such as ‘security-fixing’ would have created a bias that would have interacted with our main research question.

If a cell is empty, it does not mean that we did not select the tool, it means that there is no patch suggested by the tool for that vulnerability. Compared to the design in Papotti et al. (2022), in Table 4 we have added the evaluation of SeqTrans tool (Chi et al. 2022). Unfortunately, the tool performs poorly and was only able to produce a correct patch for one out of 7 vulnerabilities we used in our experimental setting.

For the very same lack of maturity in users' interfaces, participants did not actually *use* the tools. They received 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 (Naiakshina et al. 2017, 2018; Rong et al. 2012; Chong et al. 2021).

The experiment was 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 the design, execution, and measurements of security experiments.

In terms of learning outcomes, through this experiment, the students have the possibility to critically 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 have students as participants, this may affect the outcome of the experiment. However, we believe that master students have enough experience to participate in the experiment, and we think that we can collect interesting results and insights that can help us with 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

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 follow the same procedure for this experiment, and we determine the correct number of results, measuring the true positive, and false positive rates.

Statistical Tests As we do not expect normally distributed samples, we planned to use a non-parametric, Mann-Whitney test. To be conservative we also used the t-test since our data is not ordinal. We used the test from a python library: `scipy`² and similarly for Wilcoxon test in the same package. We tested the data for normality using the Kolmogorov-Smirnov test. Some of our hypotheses are about the equivalence of treatments. To answer them, we used TOST (Two One-Sided T) test as a test of equivalence which was initially proposed by Schuirmann (1981) and is widely used in pharmacological and food sciences to check whether the two treatments are equivalent within a specified range δ (Food and Drug Administration 2001; Meyners 2012). Therefore two distributions x and y will be considered equivalent if $x \cdot \delta < y < x \cdot 1/\delta$ for $\delta = 0.8$. The underlying directional test can be either the Mann-Whitney test or the t-test. We report here in the t-test because it was

² [scipy.stats.ttest_rel](https://docs.scipy.org/doc/scipy/reference/stats.html)

more conservative than Mann-Whitney test. The two tests never produced a qualitative difference (e.g. one test was significant and the other was not).

In case we had disproportionally many zero values (i.e. many participants failed to recognize even *some* lines of code) we planned to investigate the use of the combined test proposed by Lachenbruch (2002) but this case did not arise.

9 Results

In this section, we report the results of our investigation.

9.1 Validation of the Participants Data

A total of 80 Computer Science master's students participated in the experiment. As a population, this is consistent with several other studies (Naiakshina et al. 2017, 2018; Rong et al. 2012; Chong et al. 2021).

We did a preliminary check on the collected data. The participants were initially asked whether they agree to share their data for research purposes. The participants that denied this consent were removed from the data. Since the experiment was part of a course as a learning objective, the students that declined consent could still complete the activities of the experiment to obtain the credits for participating. We also removed clearly invalid submission attempts as measured by the process metrics. In particular, we deleted submissions from participants who failed the attention check by checking the time spent to complete the preparatory instruction videos before the experiment (Ejelöv and Luke 2020). We chose a minimum threshold of 15 minutes (the total length of the videos) to perform the attention check. As a final result, we counted $n = 72$ valid submissions for the experiment.

9.2 Demographics

To validate the participants' experience we looked first at their Java, Vulnerability Detection, Automated Testing, Automated Fault Localization, Security Testing, and Visual Studio Code expertise as collected during the training phase. The expertise scale values went from "*No experience*" to "*Experience through projects performed outside University*".

A consistent number of participants used Java (57%) and Visual Studio Code (69%) as part of university projects. An equal number of participants (16%) used both technologies outside of the university context, and developed several projects using Java (11%) and Visual Studio Code (10%). Only few participants do not have any experience with Visual Studio Code (5%). Therefore, we can conclude that participants would not have had any difficulty in using the proposed tools.

The majority of the participants do not have any experience in Vulnerability Detection (58%), Automated Testing (74%), Automated Fault Localization (96%), and Security Testing (77%). Around a third of participants found vulnerabilities as part of university projects (37%), only the 3% did it for several projects, or outside of university projects (3%). Regarding Automated Testing and Security Testing an equal number of participants (18%) gained experience through university projects. Only the 5% have experience besides university projects, and really few participants (3%) used Automated Testing for several projects. Finally only two participants used Automated Fault Localization for university projects, and only one participant declared that used it also for projects outside of academia

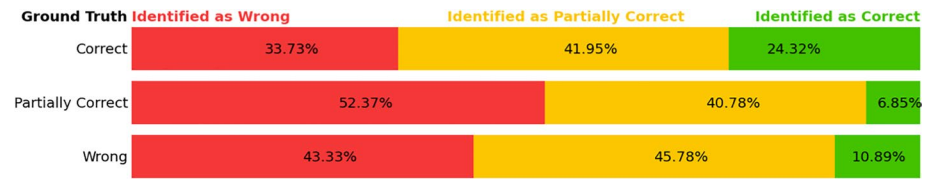


Fig. 3 The average percentage of patch identification. The picture shows the average of patches classification. We observe that 43.33% of wrong patches have been classified as wrong patches, and 24.32% of correct patches have been classified as correct patches

Table 5 Classification by students of each Tool per CVEs

CVE	ARJA			Card.			jGen.			jKali			RSRepair			TBar			SeqTrans		
	W	PC	C	W	PC	C	W	PC	C	W	PC	C	W	PC	C	W	PC	C	W	PC	C
CVE-2013-4378	57	11	5	—	—	—	—	—	—	—	—	—	59	10	4	36	25	12	17	27	29
CVE-2016-9878	—	—	—	58	12	3	57	6	10	63	10	0	—	—	—	33	25	15	40	22	11
CVE-2018-1192	45	12	16	35	29	9	53	15	5	—	—	—	—	—	—	62	7	4	57	15	1
CVE-2018-1324	40	14	19	—	—	—	—	—	—	42	20	11	41	14	18	62	7	4	48	20	5
CVE-2018-17202	60	11	2	57	11	5	—	—	—	57	13	3	56	13	4	—	—	—	30	29	14
CVE-2018-1000864	—	—	—	—	—	—	49	20	4	55	13	5	48	20	5	44	20	9	46	13	14
CVE-2019-10173	53	14	6	60	10	3	37	20	16	59	11	3	—	—	—	—	—	—	43	15	15

Each cell contains the total number of participants that classified a tool for each CVE as wrong, partially correct, and correct. The coloured cell with text in bold are the one that matches the ground truth of the patch. ‘—’ means there is no available patch from a specific tool for that CVE

context. Therefore, we can conclude there was no significant previous security experience that could bias the final result.

9.3 Preliminary Analysis

We performed a preliminary analysis by evaluating how participants classified each patch proposed, Table 5 shows the result we obtained. We observed that not always the participants have been able to classify correctly the patches as expected. Figure 3 shows the general picture of the average percentage of patch identification. We can observe that is easier to identify wrong patches as wrong patches rather than correct patches as correct patches.

Moreover, Table 6 shows the number of time that each patch tool has been adopted by the participants per CVE and we used the tools columns as contingency table to perform a chi-squared test³ python function. Since some cells contains zero as frequency we added two “successes” and “failures” to the sample (Agresti and Coull 1998). The result is not significant as expected, there is no difference among CVEs in terms of participants’ abilities to spot the correct patches.

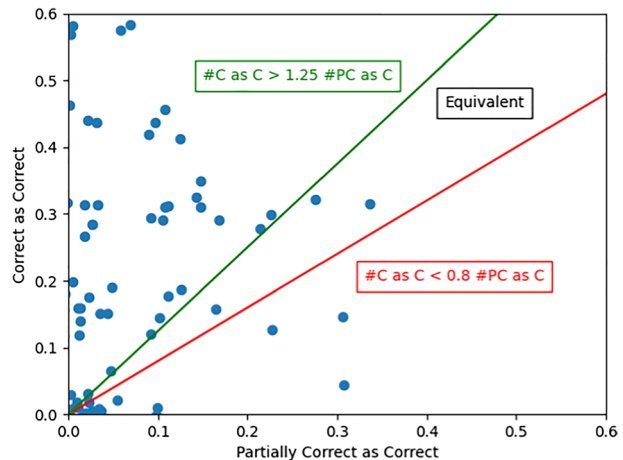
³ [scipy.stats.chi2_contingency](#)

Table 6 Tool Adoption per CVEs

CVE	Arja	Card	jGen	jKali	RSRepair	TBar	SeqTrans	Adopted		
								W	PC	C
CVE-2013-4378	4	—	—	—	4	20	45	0	28	45
CVE-2016-9878	—	8	13	2	—	28	22	73	0	0
CVE-2018-1000864	—	—	14	7	9	15	28	73	0	0
CVE-2018-1192	24	30	8	—	—	6	5	35	28	30
CVE-2018-1324	15	—	—	10	18	10	20	30	0	43
CVE-2018-17202	7	11	—	3	7	—	45	45	28	0
CVE-2019-10173	12	5	31	5	—	—	20	25	17	31
Total	62	54	66	27	38	79	185	281	101	149

Each cell contains the number of times a tool has been adopted by participants before the security information has been revealed. The last three columns represent the total of wrong, partially correct, and correct patches that have been adopted for each CVE

Fig. 4 Distribution behind TOST of H1.2. The green and the red lines corresponds to the values $x \cdot \delta < y < x \cdot 1/\delta$ for $\delta = 0.8$, where x is partially correct patches identified as correct patches, and y is correct patches identified as correct patches. As one can see the correct patches (y) are even higher than the 125% value of the partially correct patches. To improve visibility, the coordinates of data points have been slightly randomized by an offset in the range $[-0.01, 0.01]$



9.4 RQ1. Discrimination Between Correct and Wrong

For each participant, we calculated the number of wrong patches correctly identified by the participants divided by the total number of wrong patches. We applied the same procedure for the correct patches. As result we obtained $t = 15.12, p = 2.39 \cdot 10^{-24}$, therefore the data supports H1.1.

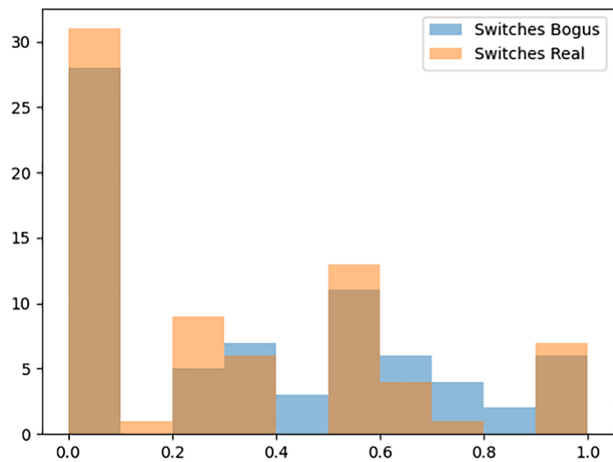
We used TOST test to investigate if partially correct patches are equally identified as correct patches as actually correct patches. We used again t-test ($\delta = 0.8$) to check the equivalence between the means of the two samples. As the result we obtained $p_{lower_bound} = 0.76 \cdot 10^{-08}, p_{upper_bound} = 0.99$, therefore the data does not support H1.2. Figure 4 shows that H1.2 is satisfied only for the lower bound but not for the upper bound.

We did an additional exploratory analysis by testing whether correct patches are identified as actually correct patches more than partially correct patches are identified as correct patches. Again we used t-test and as result we obtained $t = 4.57, p = 0.98 \cdot 10^{-07}$, therefore

Table 7 Results statistical t-test for RQ1 - H1.3

Tool	Statistic	<i>p</i> -value
Arja	−5.00	1.93e−06
Card	−3.31	7.36e−04
jGen	−0.71	2.39e−01
jKali	−8.17	3.64e−12
TBar	−2.81	3.19e−03
RSRepair	−8.88	1.73e−13

Fig. 5 Distribution behind TOST H2.1. On the *x* axes there is the proportion of actual switches with respect to potential switches available to the participants, and on *y* axes there is the frequency. As one can see there is a higher proportion for the bogus treatment rather than the real treatment. Moreover, we can notice a large number of zeros, which it represents no switches. However, even by removing the zero values the TOST was not significant



correct patches are more easily identified as correct patches rather than partially correct patches are identified as correct patches.

Finally, we explored if patches from APR tools designed for security are adopted more often than patches suggested by generic APR tools. At first, we performed a global test using the `scipy.stats.kruskal`⁴ python function to perform a Kruskal-Wallis test. We used seven arrays (one for each tool) where each value was the number of time a participant selected the specific tool and as result we obtained $k = 106.86, p = 0.93 \cdot 10^{-22}$.

Then we performed a post-hoc analysis by comparing how many times participants adopted a patch suggested by one of the generic tools and how many times they chose a patch suggested by SeqTrans. Table 7 reports the t-test results that we performed, which suggest that the data support H1.3.

9.5 RQ2. Influenced on the Perceived Security Specialization

We used again TOST test to investigate if experimental and treatment groups have the same number of switches after revealing the security information. First we test whether the participants were shown equivalent numbers of bogus and real potential switches,

⁴ `scipy.stats.kruskal`

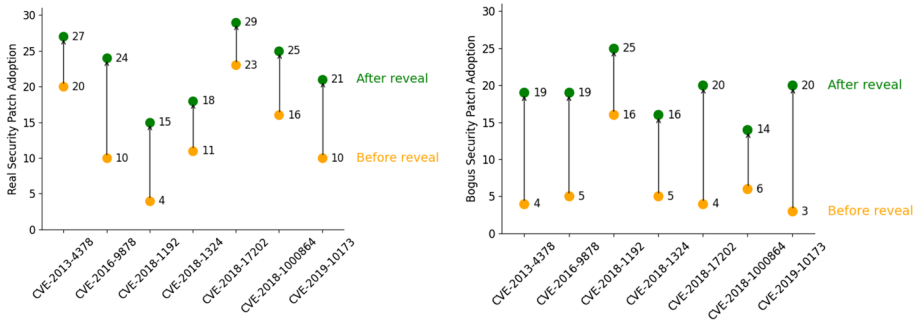


Fig. 6 Security Patch Adoption before and after revealing the security information. The plot on the left shows the number of security patches adopted for each CVE before and after the security information has been revealed. The plot on the right gives the same information for the bogus case. As one can see the number of the number of adoption of allegedly of “security” tools increases after their security quality is revealed

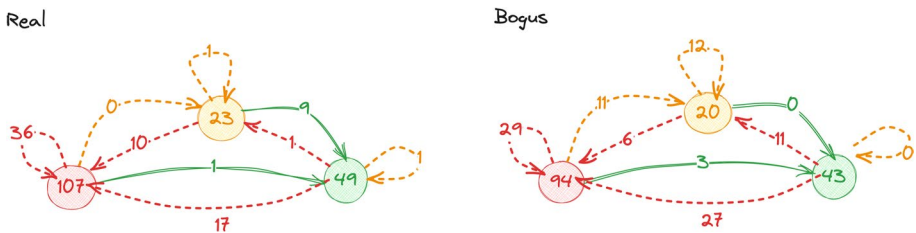


Fig. 7 Switches between Patches. The figure shows the switches performed by participants from a type of adopted patch to another adopted patch when they are told that a patch was generated by a security tool. On the left the patch was really generated by the security tool, on the other the tool was just a generic tool. Circles corresponds to a patch type (red for wrong, yellow for partially correct, green for correct). Edges corresponds to switches. The number inside the circles corresponds to type of adopted patches in which the user decided not to switch. As we can see several switches, represented by dashed red or yellow edges are useless or worsening. For example in the real case on the left 36 participants switched from another wrong patch to another wrong patch and 17 actually moved from a correct patch to a wrong patch

i.e. the two groups started from the same baselines of potential switches. As result we obtained $p_{lower_bound} = 0.01, p_{upper_bound} = 0.01$, which shows that in terms of potential choices the groups of participants were identical. Then we test whether the number of switches were equivalent between participants to whom a real security switch was proposed and participants to whom a bogus security switch was proposed. To do so, we computed the proportion of actual switches with respect to potential switches available to the participants. Figure 5 shows that there is a higher proportion of switches for the bogus treatments. The result for TOST test is $p_{lower_bound} = 0.37, p_{upper_bound} = 0.01$, which shows that there is not enough evidence to support H2.1. This might be explained by the large number of zeros. We considered the option of using Lachenbruch test but the TOST restricted to non-zero switches was already not significant.

Finally we performed t-test to verify if the number of adopted patches from known security designed tools will be higher after the security information is revealed. The result of the test is $t = -13.27, p = 5.65 \cdot 10^{-06}$, therefore the data support H2.2.

Figures 6 and 7 show the number of adopted patches from security tools (both for bogus and real treatments) before and after revealing the security information.

A possible alternative explanation of the switches could have been that participants actually switched to a better solution, for example from a wrong patch to a partially correct patch. Figure 7 shows this is not case. In the pictures we represent useless switches with red and yellow dashed edges. Such switches are those that do not improve the previous choice, for example switching from a partially correct patch to another partially correct patch, or from a wrong patch to a partially correct patch, or from a correct patch to another correct patch. In particular, useless switches are those that diminish the quality of the adoption choice, in particular, switching from a wrong patch or partially correct patch or a correct patch to a wrong patch, or also switching from a correct patch to a partially correct patch.

This qualitative evidence shows that the choice switching cannot be explained by the opportunity to reflect on the quality of patch to improve the quality of one's own choice. Hence, the most likely explanation is that such switch is due to authority bias (Silvester 2021).

9.6 Robustness Check

We performed a robustness check to verify that participants did not just select the patches according to the order they were showed. Indeed, we observed the frequencies of patches selected based on the order in which they were presented to participants as $\langle 104, 143, 115, 66, 78 \rangle$ for the number of time each patch P_i was adopted in the order in which it was presented. It might seem to suggest that participants selected more often one of the first three patches.

To understand whether such number is just by chance we needed to estimate what would be the probability that they would select the first patch that they meet that looks correct. We assume that participants will not adopt a wrong patch that they believe is wrong. However, if they believe that the patch they encounter it is correct they will select it. So we first assigned a probability of being chosen by a participant based on the classification of the patch. To choose the probability we relied on the results shown in Fig. 3. For example, 10.89% of wrong patches have been classified as correct patches, therefore, we considered that the probability that a participant adopt a wrong patch when they first meet it is $Pr[P_i \text{ correct} | P_i = W] = 0.11$. Similarly $Pr[P_i \text{ correct} | P_i = PC] = 0.07$ and $Pr[P_i \text{ correct} | P_i = C] = 0.24$.

So for each individual CVE we determined which patch was first presented to the participants. For example for CVE-2018-1192 the order was $\langle P_1 = C, P_2 = W, P_3 = PC, P_4 = C, P_5 = W \rangle$. Therefore, we have

$$\begin{aligned}
 Pr[P_1 \text{ adopted}] &= Pr[\text{correct} | P_1 = C] \\
 Pr[P_2 \text{ adopted}] &= (1 - Pr[P_1 \text{ adopted}]) \cdot Pr[P_2 \text{ correct} | P_2 = W] \\
 Pr[P_3 \text{ adopted}] &= (1 - Pr[P_1 \text{ adopted} \vee P_2 \text{ adopted}]) \cdot Pr[P_3 \text{ correct} | P_3 = PC] \\
 &\dots \\
 Pr[P_i \text{ adopted}] &= (1 - \sum_{j=1}^{i-1} Pr[P_j \text{ adopted}]) \cdot Pr[P_i \text{ correct} | P_i = \text{type}(P_i)]
 \end{aligned}$$

The final number might not add up to one. However, since a participant must eventually adopt the less worst patch in the current design we normalized everything by $\sum_i Pr[P_i \text{ adopted}]$. According to this estimation we finally obtained $\langle 132, 111, 114, 91, 58 \rangle$ as expected frequencies array. We performed Pearson's chi-squared test and the result is $\chi^2 = 28.94, p = 8.04 \cdot 10^{-06}$, which may mean that participants tend to adopt earlier patches in the list. Future experiments should control for this possibility.

To ensure that both treatments do not have an unequal distribution of experienced participants we performed a chi-squared test to compare the two different groups bogus and real. We focused on the knowledge of Java as it is one of the main skills necessary to be able to perform the experiment. We separated the data points in 3 main groups: (i) no experience, (ii) some experience (developed in Java for university projects, or developed some projects in Java), (iii) significant experience (developed several projects outside of University context). We used the `scipy.stats.chi2_contingency` python function, and we obtained $\chi^2 = 0.077, p = 0.96$ with 2 degrees of freedom $((2 - 1) \times (3 - 1))$. The result is not significant, which it means that the groups are not significantly unbalanced in terms of experienced participants.

10 Perception Results

We asked the participants their feedback about the experiment. We used the perception questions as a control to make sure that the experiment did not suffer from internal validity issues. In particular we asked them if they (i) clearly understood the task they had to perform, (ii) had enough time to perform the experiment, (iii) the training was sufficient enough.

Most of the participants somewhat agreed (40%) and agreed (27%) that they understood the task they had to perform for the experiment. A few percentage strongly disagreed (3%) and disagreed (8%) that the explanation of the experiment was clear. Few participants (21%) neither agreed nor disagreed.

The participants strongly agreed (48%) and somewhat agreed (38%) that they had enough time to perform the experiment. Only 1% strongly disagreed, and 10% disagreed that the time was not enough. Finally, 3% of the participants neither agreed nor disagreed.

Some participants suggested that a more detailed training could be beneficial. While 41% either agreed or strongly agreed that the training was sufficient, there were 21% that neither agreed nor disagreed. One of the participant declared that *"A live example of how such vulnerabilities are actually detected, maybe during the lecture we can be made to slowly work on this over the week so we can arrive at the experiment more prepared."*; another one suggested *"more information about the patches and the development"*, which is inline with another participants opinion: *"if time allows, more detailed sources would definitely be more helpful"*. This may also be due to the fact that for some instances of the experiment we made available the video recording of the training with a wrong format. Therefore, the students had to zoom in, affecting the ease of use of the training material.

Table 8 Summary of Findings and Implications

RQ	Main Findings	Implication for Research and Practice
RQ1	It is easier to identify wrong patches than correct patches. Correct patches are not confused with partially correct patches	Code review is an effective solution to filter wrong or partially correct patches proposed by tools. However it is an expensive one and therefore more research is needed to further filter them.
RQ1	Patches from APR4Sec are adopted more often than patches suggested by generic APR tools.	Since this choice did not depend neither on correctness nor on knowing that the tool was a security tool, we speculate it might be due to the presence of pat- terms in the proposed patches. This might require further investigation.
RQ2	There is not enough evidence to conclude that ‘bogus’ security claims are either indistinguishable or different from ‘true security’ claims.	Further investigations with more and diverse security tools is required to con- clude if there is a difference between the two treatments. It might also be that patches are patches and are essentially indistinguishable.
RQ2	Knowing a patch is from a security tool increases the chances of adoption irre- spective of correctness	Authority bias is an important phenomenon that can be behind tool adoption and perception of tool correctness. Coupled with the results of the RQ1, it shows that developers might be more willing to give a pass grade to a patch that is not correct. Independent validation of patches might be important.

11 Discussion and Implications

Table 8 summarizes the main findings of our study. We first tried to investigate if human code reviewers are able to discriminate between correct and wrong patches submitted by the APR tools (RQ1).

The result are encouraging from the perspective of tool debugging as it shows is easier to identify wrong patches than correct patches. So code reviewers could potentially quickly discard wrong or partially correct solutions. From the perspective of the developers being able to recognize whether a patch is definitely wrong (RQ1) is an interesting finding. However, the recognition of partially correct patches is only the first step. What would be important to understand is what impact the delivery of a partially correct patch by an APR tool has on the actual fixing of the bug. If there is no difference then a wrong and partially correct patches would become one category for all practical purposes.

To investigate this issue we performed a qualitative analysis of the reasons proposed by the participants to select or discard the patches and the qualitative distinction between definitely wrong and partially correct emerged from the comments as semantically important. For example, partially correct patches gave suggestions to the participants to fix the vulnerabilities themselves. In one instance, a participant commented: *“I chose this one because it was improving the code by checking not only for null but also empty string but it is partially correct because session-id also must be checked if it does not have illegal characters”*. Another one wrote: *“I chose this one because it was improving the code by checking not only for null but also empty string but it is partially correct because session-id also must be checked if it does not have illegal characters”*. For another vulnerability a participant explicitly mentioned the patch as a good initial starting point: *“1 and 2 simply delete unsafe code, 3 and 4 encode. Not sure if encoding reduces risk of xss attack but looks like a good start”*. Fully answering the question would require a different experiment in which we propose wrong or partially correct patches and ask the participants to develop a fix based on the candidate patches. One of the difficulties behind this experiment is that both wrong and partially correct patches are candidate patches, which means that they all passed all the available tests. Therefore, we cannot use tests to discriminate among them and among the solutions proposed by the developers. Assessing the semantic correctness of patches would be therefore an expensive manual work.

Furthermore, code review remains expensive and time consuming. In fact, in another study (Cambronero et al. 2019) the authors observed that developers had difficulty in identifying correct patches in a set of plausible patches, and consequently spent a lot of time trying to find contextual information to assess the correctness of the candidate patches.

This is also supported by our qualitative analysis of the reason behind the failed choice among correct patches. For example for CVE-2018-1324 which has 3 correct patches (Table 4), some participants had difficulties to discriminate the semantically correct solution from the semantically wrong but thoroughly tested solution. In our setup testing is not helpful because all candidate patches satisfy all tests. For example a participants wrote: *“I don’t want to choose either of them. In patch 1, it will not enter the loop structure. In patch 2, syntax errors. In patch 3, 4, and 5, necessary code misses.”*, another one stated: *“Patch 2 changes the logic of the for loop however patches 3,4 and 5 just remove parts of the code which i do not believe is a good idea. Patch 1 changes the for loop condition which also does not look like the right option.”*. In contrast other participants interpreted correctly the very same proposed solution: one observed that *“There is a useless code and patch 3,4,5 found that”* and another one wrote that *“Patch 3 correctly removes unnecessary loops that*

serve no purpose, patch 5 does the same thing". Sieving among correct and partially correct patches seemed somehow easier. For example a participant analyzing CVE-2013-4378 reported that clearly: "P1: do nothing useful, just delete the part; P2: do not have a *javaScriptEncode* method; P3: fix the Information Disclosure vulnerabilities; P4: do nothing useful, just delete the part;".

A possible helping hand could be the use of machine learning tools to further prioritize patches to be recommended to developers as suggested by Long and Rinard (2016). Further experiments with trying to provide developers with this information may help to distinguish correct patches from wrong patches. Since we also observed that patches from APR4Sec are adopted more often than patches suggested by generic tools, and we may think that this choice does depend on the presence of patterns in the proposed patches, the study performed by Long and Rinard (2016) can also be applied in this case for further investigations.

In Section 3 we assumed that in both experimental and treatments groups there is the same number of switches after revealing the security information. From the results we obtained we did not get enough evidence that the patches claimed as produced by security tools are indistinguishable or different from patches actually produced by security tools. It would be necessary to use more diverse security tools to investigate if there is a difference between the two treatment groups, or if patches are patches and are essentially indistinguishable.

For our study we decided to propose a limited number of patches to the participants (no more than 4/5). Our choice is inline with the finding of Noller et al. (2022) that showed that developers would not review more than 5-10 patches. We did not focus on the importance of ranking in the resulting patches from the APR tools because we selected only the ones that passed all the test cases and restricted the result to one patch per tool as recommended by Bader et al. (2019), in our case the first patch. The pilot study that we performed also suggested that the number of shown patches had to be limited. Moreover, in our study, we wanted to focus on the evaluation of correctness of the patches from the participants. An interesting point for future works would be focusing on the ranking of the patches suggested by the APR tools and how these affect also the current findings of this study. The study by Bader et al. (2019) suggested that a better ranking would have increased the acceptance of patches by 10%.

Finally, we observed that knowing a patch is from a security tools increases the chances of adoption irrespective of correctness. In medicine authority figures and hierarchy affects the doctor-patient relationship, such phenomenon is called authority bias which predisposes people towards believing and obeying authority figures (Silvester 2021). We think that a similar situation of authority bias can be behind tool adoption and perception of tool correctness, which it might influence participants' choice. Therefore training programs for tools should make it clear when also when the tool may be wrong, and provide evidences to developers on how frequently tools can be wrong.

This finding is partly supported by the qualitative survey reported in Noller et al. (2022). A small number of interviewees (9/103 interviewees) reported that reputation of the tool was a key concept. However, whether patch adoption would vary depending on the tools' reputation was not tested in the paper which only reported traditional test measures of patch correctness on the ManyBugs' dataset (Le Goues et al. 2015). While 9/103 is too small to draw any conclusion, it suggests the possibility of performing an interesting experiment, similar to the one we propose here, in which the patch is attributed to a well known tool (either for real or by a bogus attribution). Such attribution could change the patch adoption. For example, in Iyer et al. (2019) is shown that pull requests from authors that are more

open and conscientious, but less extroverted, have a higher chance of approval. Terrell et al. (2017) also argues that women's contributions tend to be accepted more often than men's, however, for contributors who are outsiders to a project and their gender is identifiable, men's acceptance rates are higher. An eye-tracking study of students acceptance of pull-requests (Ford et al. 2019) showed that social cues are used to make such decisions (as measured by eye-fixation) yet almost no student looked at the popularity of repositories of past contribution and only few looked at Github stars. Our work seems therefore the only one which evaluated this aspect so far. More experiments or observational studies are required to conclude whether authority bias in the (security) tool is a general phenomenon.

12 Threats to Validity

Possibly not Representative Sample We acknowledge that our sample can be considered as not representative of all developers since we are considering only master students from a single course. More than 90 thousand professional developers replied to the 2023 developer survey of Stackoverflow⁵, and for 41.16% of them, the Bachelor's degree is the highest education they had. Also, a consistent part (20.11%) of developers' age range is between 18 and 24 years old. These statistics are well aligned with our population sample. However, the experience of the developers may be a significant factor, and academic education by itself may not be the most critical aspect in the participant outcome. Since running the experiment with novice vs expert professional could lead to different results for a well known technology (Salman et al. 2015), to generalize the results of our study, more studies should be designed and run with a different population sample and repeated training.

Participant Bias Students' background, knowledge, and practice may impact the experiment's results. However, as mentioned in Section 2, several studies have been performed with students (Naiakshina et al. 2017, 2018, 2020; Rong et al. 2012; Chong et al. 2021). Salman et al. (2015) 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. Therefore, for relatively new tools outside the research community (such as APR tools), the threat to external validity may be limited. However experiments should be run after extensive training to determine whether this similarity still holds for better known technology.

Patches Order Bias The order in which the patches are given to the participants may affect their choice as they may tend to check only few them selecting mostly the earlier patches shown. We tried to analyze this factor in Section 9.6, our results suggests that the order in which the patches are presented to participants affects their adoption choice. In the context of code review, tools such as GitHub present the files to review sorted in alphabetical order. Fregnan et al. (2022) provide evidence that the relative position in which files are presented has an impact on code reviews' outcome. Therefore, we think that similar investigation should be also conducted on the order and the number of candidate patches.

⁵ <https://survey.stackoverflow.co/2023>

Time Measurement Through Qualtrics we registered the time that each student took to perform the experiment. We set a lower and an upper bound to consider a submission valid enough to be significant. We acknowledge that the time measurement would not 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 significant results, that will give us a strong basis to explore further in the future, and replicate the experiment in different contexts; such as with developers from companies.

Generability Focusing on a limited sample of popular APR tools and popular vulnerability types may not capture the full spectrum of challenges associated with different tools and vulnerabilities. The limited scope of tools and vulnerabilities examined raises questions about the broader applicability of the findings. Therefore, future works could include a broader spectrum of vulnerable files. In particular, examples of path traversal with closer vulnerable lines.

Tools Performance The performance of SeqTrans, as shown in Table 4 is not very promising. It produced a correct patch only for one vulnerability, limiting a diverse investigation of the correctness aspect of the patches. However, SeqTrans was the only available tool when we performed the experiment, and it was not working properly. Bui et al. (2024) showed that the results claimed in the original paper were not good as expected. We plan to replicate the experiment in the future with more successful tools.

Tool Documentation and User Experience Among the potential factors that might influence the behavior of code reviewers in a real-life context is the role of tool documentation and user experience with different tools in the code review process. This was indeed the main reason for us to test the user interface in our pilot (§ 4).

13 Conclusions

The correctness of candidate patches proposed by APR Tools have been traditionally evaluated by automated means (Durieux et al. 2019) or by researchers to provide a ground truth (Liu et al. 2021).

The first study to actually analyze the acceptance of an automated fix by developers has been the deployment at Facebook of Getafix (Bader et al. 2019) where 250 fixes for null deference warnings generated from patterns of previous fixes have been proposed to developers. While only 42% have accepted the fix, acceptance raised to more than half if we consider semantically equivalent fixes or fixes that were identified but not suggested as top by the tool. This paper continues in this direction by being the first study to analyze whether *vulnerability* patches proposed by SOTA APR tools for real-world projects are eventually accepted or rejected by human patch reviewers. In particular, we investigated whether (possibly wrong) security patches suggested by APR tools are recognized and possibly adopted by human reviewers. We also investigated whether being told that a patch was produced by an allegedly specialized tool does change the decision of human to adopt this particular patch.

To answer these questions, after an initial pilot with $n = 40$ students, we have run an experiment with $n = 72$ students enrolled in the Computer Science Master at the Vrije

Universiteit Amsterdam. We used a balanced incomplete design where we proposed to each participant a combination of patches proposed by APR tools for different vulnerabilities and ask them to adopt or reject the proposed patches. Participants could better identify wrong patches than correct patches thus showing that code reviews might actually work well.

In a second phase, we told participants that some of the proposed patches were generated by security-specialized tools (even if the tool was actually a generic APR tool) and measure whether the human reviewers would change their decision to adopt or reject a patch. The key finding is that the number of switches to the patches suggested by security tool is significantly higher after the security information is revealed irrespective of correctness. Several switches are actually switches moving from a correct to a wrong or a partially correct patch. This shows that there might be a potential authority bias in the decision of adopting what is recommended by a tool that claims to be specialized. To the best of our knowledge, the only studies that analyse the adoption of a patch based on *knowing* its provenance (as opposed to *testing* or *validating* its correctness) are observational studies focusing on the acceptance of pull-request based on the reputation or gender of the human developers proposing them (Iyer et al. 2019; Terrell et al. 2017; Ford et al. 2019). The practical implication is that authority bias might have an impact in the adoption of solutions that might turn out to be wrong.

Another interesting future work is to explore the difference between "Wrong" and "Partially Correct" as suggested by one of the Reviewer and supported by the analysis of the comments left by the participants when justifying their choice for the selected patch. Does the partially correct patch help the developer to fix the bug? Or are wrong and partially corrected patches in the same class? Answering this question requires to design a novel experiment, along the empirical, observational study on GetaFix (Bader et al. 2019), where we ask the participants to suggest different types of fixes and the actual (possibly different) fix suggested by the developer is validated in term of semantic correctness. We leave this experiment as a promising future work.

Appendix

Table 9 Results of the Pilot Study Classification of APR Tools

CVE	ARJA			Card.			jGen.			jKali			RSRepair			TBar		
	W	PC	C	W	PC	C	W	PC	C	W	PC	C	W	PC	C	W	PC	C
CVE-2013-4378	3	5	8	—	—	—	—	—	—	—	—	—	4	5	7	2	9	5
CVE-2016-9878	—	—	—	10	5	1	5	3	8	10	6	0	—	—	—	5	6	5
CVE-2018-1192	2	7	7	9	4	3	6	7	3	—	—	—	—	—	—	10	5	1
CVE-2018-1324	3	4	9	—	—	—	—	—	3	5	8	3	2	9	11	3	2	—
CVE-2018-17202	1	10	5	7	7	2	—	—	—	4	9	3	2	10	4	—	—	—
CVE-2018-1000864	—	—	—	—	—	—	5	8	3	4	7	5	3	10	3	2	8	6
CVE-2019-10173	5	8	3	9	4	3	7	2	7	10	5	1	—	—	—	—	—	—

Each cell contains the total number of participants that classified a tool for each CVE as wrong, partially correct, and correct. The colored cell with text in bold are the one that matches the ground truth of the patch. '—' means there is no available patch from a specific tool for that CVE

Acknowledgements This work has been partly supported by the European Union H2020 Program under the Grant 952647 (AssureMOSS) and the Horizon Europe Program under grant (Sec4AI4Sec) and the Dutch Research Council (NWO) under the grant NWA.1215.18.006 (Theseus) and grant KIC1.VE01.20.004 (HEWSTI). 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. We would also like to thank the reviewers of the registered report and the full version of the paper as their comments greatly helped to improve the quality of the paper.

Author Contributions Conceptualization: AP, RP, FM; Methodology: FM, AP, RP; Software: na; Validation: RP, AP, FM; Formal analysis: AP, FM, RP; Investigation: AP, RP; Resources: na; Data Curation: RP (Vulnerability tests in Table 4), AP (Python Colab); Writing - Original Draft: AP, RP. Writing - Review & Editing: AP, RP, FM. Visualization: AP, RP. Supervision: FM. Project administration: FM. Funding acquisition: FM.

Funding European Union H2020 Grant 952647 (AssureMOSS) and Horizon Europe Grant 101120393 (Sec4AI4Sec), Dutch Research Council (NWO) Grant NWA.1215.18.006 (THESEUS) and Grant KIC1.VE01.20.004 (HEWSTI).

Data Availability The replication package is available on [Zenodo](https://doi.org/10.5281/zenodo.11114305)(<https://doi.org/10.5281/zenodo.11114305>). It contains the raw data of the participants' responses in case the reviewers want to check the results, the python scripts for analyzing the data, and the training material.

Declarations

Competing Interest The authors have no financial or non-financial interest that are directly or indirectly related to the work submitted for publication.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Abadi A, Ettinger R, Feldman YA, Shomrat M (2011) Automatically fixing security vulnerabilities in Java code. In: Proc. OOPSLA'11, pp 3–4
- Agresti A, Coull BA (1998) Approximate is better than “Exact” for interval estimation of binomial proportions. *Am Stat* 52(2):119–126
- Alarcon GM, Militello LG, Ryan P, Jessup SA, Calhoun CS, Lyons JB (2017) A descriptive model of computer code trustworthiness. *J Cogn Eng Decis Mak* 11(2):107–121
- Alarcon GM, Walter C, Gibson AM, Gamble RF, Capiola A, Jessup SA, Ryan TJ (2020) Would you fix this code for me? effects of repair source and commenting on trust in code repair. *Systems* 8(1):8
- Allodi L, Cremonini M, Massacci F, Shim W (2020) Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. *Empir Softw Eng* 25(2):1063–1094
- Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proc. IEEE/ACM ICSE'13, pp 712–721. IEEE
- Bader J, Scott A, Pradel M, Chandra S (2019) Getafix: Learning to fix bugs automatically. Proceedings of the ACM on programming languages 3, OOPSLA, pp 1–27
- Baum T, Liskin O, Niklas K, Schneider K (2016) Factors influencing code review processes in industry. In: Proc. ACM SIGSOFT FSE'16, pp 85–96
- Braz L, Aeberhard C, Çalikli G, Bacchelli A (2022) Less is more: supporting developers in vulnerability detection during code review. In: Proc. IEEE/ACM ICSE'22, pp 1317–1329

- Bui QC, Scandariato R, Díaz Ferreyra NE (2022) Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In: Proc. IEEE/ACM MSR'22
- Bui Q-C, Paramitha R, Vu D-Ly, Massacci F, Scandariato R (2024) APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities. *Empir Softw Eng* 29(1):18
- Cambronero JP, Shen J, Cito J, Glassman E, Rinard M (2019) Characterizing developer use of automatically generated patches. In: Proc. VL/HCC'19. IEEE, pp 181–185
- Chi J, Qu Y, Liu T, Zheng Q, Yin H (2022) Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Trans Softw Eng*
- Chong CY, Thongtanunam P, Tantithamthavorn C (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, pp 20–29
- Cohen J (2010) Modern code review. *Making Softw: What Really Works, and Why We Believe It* 2010:329–336
- de Gramatica M, Labunets K, Massacci F, Paci F, Tedeschi A (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, pp 98–114
- Durieux T, Madeiral F, Martinez M, Abreu R (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, pp 302–313
- Ejelöv E, Luke TJ (2020) “Rarely safe to assume”: Evaluating the use and interpretation of manipulation checks in experimental social psychology. *J Exp Soc Psychol* 87(2020):103937
- Food and Drug Administration (2001) Guidance for industry: Statistical approaches to establishing bioequivalence
- Ford D, Behroozi M, Serebrenik A, Parnin C (2019) Beyond the code itself: how programmers really look at pull requests. In: 2019 IEEE/ACM 41st International conference on software engineering: software engineering in society (ICSE-SEIS). IEEE, pp 51–60
- Fregnan E, Braz L, D'Ambros M, Çalıklı G, Bacchelli A (2022) First come first served: the impact of file position on code review. In: Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, pp 483–494
- Fry ZP, Landau B, Weimer W (2012) A human study of patch maintainability. In: Proc. ACM SIGSOFT ISSTA'12, pp 177–187
- Gao X, Wang B, Duck GJ, Ji R, Xiong Y, Roychoudhury A (2021) Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Trans Softw Eng Methodol(TOSEM)* 30(2):1–27
- Gonçalves PW, Fregnan E, Baum T, Schneider K, Bacchelli A (2020) Do explicit review strategies improve code review performance?. In: Proc. IEEE/ACM MSR'20, pp 606–610
- Huang Z, Lie D, Tan G, Jaeger T (2019) Using safety properties to generate vulnerability patches. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 539–554
- Iyer RN, Yun SA, Nagappan M, Hoey J (2019) Effects of personality traits on pull request acceptance. *IEEE Trans Softw Eng* 47(11):2632–2643
- Just R, Jalali D, Ernst MD (n.d.) Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In: Proc. ACM SIGSOFT ISSTA'14, pp 437–440
- Kechagia M, Mechtaev S, Sarro F, Harman M (2021) Evaluating automatic program repair capabilities to repair API misuses. *IEEE Trans Softw Eng*
- Kollanus S, Koskinen J (2009) Survey of software inspection research. *Open Softw Eng J* 3:1
- Lachenbruch PA (2002) Analysis of data with excess zeros. *Stat Methods Med Res* 11(4):297–302
- Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In: Proc. IEEE/ACM ICSE'12, pp 3–13
- Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans Softw Eng* 41(12):1236–1256
- Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) TBar: Revisiting template-based automated program repair. In: Proc. ACM SIGSOFT ISSTA'19, pp 31–42
- Liu K, Wang S, Koyuncu A, Kim K, Bissyandé TF, Kim D, Wu P, Klein J, Mao X, Traon YL (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, pp 615–627
- Liu K, Li L, Koyuncu A, Kim D, Liu Z, Klein J, Bissyandé TF (2021) A critical review on the evaluation of automated program repair systems. *J Syst Softw* 171(2021):110817
- Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, pp 298–312
- Ma S, Thung F, Lo D, Sun C, Deng RH (2017) Vurle: Automatic vulnerability detection and repair by learning from examples. In: European symposium on research in computer security. Springer, pp 229–246

- Martinez M, Monperrus M (2016) Astor: A program repair library for java. In: Proc. ACM SIGSOFT ISSSTA'16, pp 441–444
- Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2017) Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empir Softw Eng* 22(4):1936–1964
- Martinez M, Monperrus M (2018) Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: Proc. SSBSE'18, pp 65–86
- Massacci F, Pashchenko I (2021a) Technical Leverage: Dependencies Are a Mixed Blessing. *IEEE Sec Priv* 19(3):58–62
- Massacci F, Pashchenko I (2021b) Technical leverage in a software ecosystem: Development opportunities and security risks. In: Proc. ACM/IEEE ICSE'21. IEEE, pp 1386–1397
- Meynens M (2012) Equivalence tests-A review. *Food Qual Prefer* 26(2):231–245
- Monperrus M (2018) The living review on automated program repair. Technical Report hal-01956501. HAL/archives-ouvertes.fr
- Moody DL (2003) The method evaluation model: a theoretical model for validating information systems design methods
- Naiakshina A, Danilova A, Tiefenau C, Herzog M, Dechand S, Smith M (2017) Why do developers get password storage wrong? A qualitative usability study. In: Proc. ACM SIGSAC CCS'17, pp 311–328
- Naiakshina A, Danilova A, Tiefenau C, Smith M (2018) Deception task design in developer password studies: Exploring a student sample. In: Proc. USENIX SOUPS'18, pp 297–313
- Naiakshina A, Danilova A, Gerlitz E, Smith M (2020) On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In: Proceedings of the 2020 CHI conference on human factors in computing systems, pp 1–13
- Nguyen VH, Dashevskiy S, Massacci F (2016) An automatic method for assessing the versions affected by a vulnerability. *Empir Softw Eng* 21(2016):2268–2297
- Noller Y, Sharifdeen R, Gao X, Roychoudhury A (2022) Trust enhancement issues in program repair. In: Proceedings of the 44th international conference on software engineering, pp 2228–2240
- Papotti A, Paramitha R, Massacci F (2022) On the acceptance by code reviewers of candidate security patches suggested by Automated Program Repair tools. [arXiv:2209.07211](https://arxiv.org/abs/2209.07211) [cs.SE] Available as <https://doi.org/10.48550/arXiv.2209.07211>
- Pashchenko I, Vu D-L, Massacci F (2020) A qualitative study of dependency management and its security implications. In: Proc. ACM SIGSAC CCS'20, pp 1513–1531
- Pinconschi E, Abreu R, Adão P (2021) A comparative study of automatic program repair techniques for security vulnerabilities. In: Proc. IEEE ISSRE'21. IEEE, pp 196–207
- Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C (2019) A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proc. IEEE/ACM MSR'19, pp 383–387
- Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proc. ACM ESEC/FSE'13, pp 202–212
- Rigby PC, German DM, Cowen L, Storey M-A (2014) Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans Softw Eng Methodol (TOSEM)* 23(4):1–33
- Rong G, Li J, Xie M, Zheng T (2012) The effect of checklist in code review for inexperienced students: An empirical study. In: Proc. IEEE CSEET'12. IEEE, pp 120–124
- Ryan TJ, Alarcon GM, Walter C, Gamble R, Jessup SA, Capiola A, Pfahler MD (2019) Trust in automated software repair: The effects of repair source, transparency, and programmer experience on perceived trustworthiness and trust. In: HCI for cybersecurity, privacy and trust: first international conference, HCI-CPT 2019, Held as Part of the 21st HCI international conference, HCII 2019, Orlando, FL, USA, July 26–31, 2019, Proceedings 21. Springer, pp 452–470
- Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at Google. In: Proc. IEEE/ACM ICSE-SEIP'18, pp 181–190
- Salman I, Misirlı AT, Juristo N (2015) Are students representatives of professionals in software engineering experiments?. In: Proc. IEEE/ACM ICSE'15, vol. 1. IEEE, pp 666–676
- Scandariato R, Wuyts K, Joosen W (2015) A descriptive study of Microsoft's threat modeling technique. *Req Eng* 20(2):163–180
- Schuurmann DL (1981) On hypothesis-testing to determine if the mean of a normal-distribution is contained in a known interval. *Biometrics* 37(3):617–617
- Shahin M, Babar MA, Zhu L (2017) Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5(2017):3909–3943
- Silvester C (2021) Authority Bias. Decision making in emergency medicine: biases, errors and solutions, pp 41–46
- Tao Y, Kim J, Kim S, Xu C (2014) Automatically generated patches as debugging aids: a human study. In: Proc. ACM SIGSOFT FSE'14, pp 64–74

- Terrell J, Kofink A, Middleton J, Rainear C, Murphy-Hill E, Parnin C, Stallings J (2017) Gender differences and bias in open source: Pull request acceptance of women versus men. *PeerJ Comput Sci* 3(2017):e111
- Tuma K, Scandariato R (2018) Two architectural threat analysis techniques compared. In: European conference on software architecture. Springer, pp 347–363
- Vu D-L, Pashchenko I, Massacci F (2021) Please hold on: more time= more patches? Automated program repair as anytime algorithms. In: 2021 IEEE/ACM international workshop on automated program repair (APR). IEEE, pp 9–10
- Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y (2020a) An empirical study of usages, updates and risks of third-party libraries in java projects. In: Proc. of ICSME'20. IEEE, pp 35–45
- Wang S, Wen M, Lin B, Wu H, Qin Y, Zou D, Mao X, Jin H (2020b) Automated patch correctness assessment: How far are we?. In: Proc. IEEE/ACM ASE'20, pp 968–980
- Winter E, Bowes D, Counsell S, Hall T, Haraldsson S, Nowack V, Woodward J (2022a) How do developers really feel about bug fixing? Directions for automatic program repair, *IEEE Trans Softw Eng*
- Winter ER, Nowack V, Bowes D, Counsell S, Hall T, Haraldsson SO, Woodward J (2022b) Let's talk with developers, not about developers: a review of automatic program repair research. *IEEE Trans Softw Eng*
- Wuyts K, Scandariato R, Joosen W (2014) Empirical evaluation of a privacy-focused threat modeling methodology. *J Syst Softw* 96(2014):122–138
- Ye H, Martinez M, Monperrus M (2021) Automated patch assessment for program repair at scale. *Empir Softw Eng* 26(2):1–38
- Yuan Y, Banzhaf W (2018) Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Trans Softw Eng* 46(10):1040–1067
- Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on Firefox. In: Proceedings of the 8th working conference on mining software repositories (MSR '11). Association for Computing Machinery, New York, NY, USA, pp 93–102. <https://doi.org/10.1145/1985441.1985457>
- Zhang Y, Kabir M, Xiao Y, Meng N et al (2021) Data-Driven vulnerability detection and repair in Java code. [arXiv:2102.06994](https://arxiv.org/abs/2102.06994)
- Zhang Q, Zhao Y, Sun W, Fang C, Wang Z, Zhang L (2022) Program repair: automated vs. manual. [arXiv:2203.05166](https://arxiv.org/abs/2203.05166)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Aurora Papotti is a Ph.D. student at Vrije Universiteit, Amsterdam, 1081 HV, The Netherlands. She attended the EIT Digital Master School in Cyber Security. She received her master's degree in informatics with distinction from the University of Turku, Turku, 2014, Finland, and the University of Trento, Trento, 38123, Italy. Her main research interest is in software security, focusing on the empirical assessment of tools that automatically detect and fix vulnerabilities, and how developers cope with these tools while using them to code review. She started to actively serve the research community, such as by being a student volunteer (ICSE'22). Contact her at a.papotti@vu.nl.

Ranindya Paramitha is a Ph.D. student at the University of Trento, Trento, 38123, Italy. She received her master's degree in informatics with distinction from Institut Teknologi Bandung, Bandung, 40132, Indonesia. Her main research interest is in software security, focusing on empirical analysis of secure software ecosystems, mining software repositories, and how developers can apply security. She is involved in European Projects named AssureMOSS and Sec4AI4Sec. She has also started to actively serve the research community in several IEEE/ACM International Conferences/Workshops, such as by being a student volunteer (ICSE'22) and program committee (ICSE SVM'23). Contact her at ranindya.paramitha@unitn.it.

Fabio Massacci is a professor at the University of Trento, Trento, 38123, Italy, and Vrije Universiteit, Amsterdam, 1081 HV, The Netherlands. Massacci received a Ph.D. in computing from the University of Rome "La Sapienza". He received the IEEE Requirements Engineering Conference Ten Year Most Influential Paper Award on security in sociotechnical systems. He participates in the FIRST special interest group on the Common Vulnerability Scoring System and the European pilot CyberSec4Europe on the governance of cybersecurity. He coordinates the European AssureMOSS and Sec4AI4Sec projects. He is a Member of IEEE, the Association for Computing Machinery, and the Society for Risk Analysis. Contact him at massacci@ieee.org.