An End-to-End Framework for Repairing Potentially Vulnerable Source Code

Judit Jász
University of Szeged
FrontEndART Ltd.
Szeged, Hungary
0000-0001-6176-9401

Péter Hegedűs
University of Szeged
FrontEndART Ltd.
Szeged, Hungary
0000-0003-4592-6504

Ákos Milánkovich
SEARCH-LAB Ltd.
Budapest, Hungary
akos.milankovich@search-lab.hu

Rudolf Ferenc
University of Szeged
FrontEndART Ltd.
Szeged, Hungary
0000-0001-8897-7403

***Abstract*—Nowadays, program development is getting easier and easier as the various IDE tools provide advice on what to write in the program. But it is not enough to implement a solution to a problem; it is also important that the non-functional properties, like the quality or security of the code, are appropriate in all aspects. One of the most widely used techniques to ensure quality is testing. If the tests fail, one can fix the code immediately. However, security issues are unexpected cases when implementing the program, which is why we do not write tests for them in advance. In many cases, security-relevant bugs can not only cause financial loss but also put human lives at risk, so detecting and fixing them is an important step for the reliability and quality of the program. The tool presented in this paper aims to generate automatic code repairs to potential vulnerabilities in the program. By integrating the recommended fixes, one can easily harden the security of their program early in the development process. A case study on six open-source Java subject systems showed that we were able to generate viable repair patches for 57 out of the 81 detected security issues (70%). For certain types (e.g., revealing private references of mutable objects), our tool reached close to perfect performance.**

***Index Terms*—automated code repair, vulnerability, ASG transformation**

I. Introduction

Today's world is increasingly dominated by computer programs, which means that more and more code is being developed and the speed of development is getting faster and faster. However, to ensure the quality and security of these programs it is very important to be able to detect and correct errors in them efficiently and quickly. Security issues (i.e., vulnerabilities) in particular can not only cause financial loss but also put human lives at risk.

Of course, manually localizing and correcting errors is a time-consuming process, which is why it is important to automate the steps of this process as much as possible. The topic of Automated Program Repair (APR) is a very active area within software engineering, both in terms of tools and the scientific literature. Most of the solutions in this area are relying on the unit tests of programs and target simple functional defects, where the goal of the repair tasks is defined as the automatic correction of the failing test cases by code modification.

However, security vulnerabilities are very specific sub-types of defects that are not anticipated by the developers (i.e., caused by an unexpected execution of the program), therefore tests are usually not prepared for them in advance. This means that we cannot apply the general APR approach of fixing security issues based on failing unit tests in most cases. As a potential solution, we can apply static code analysis tools, which are capable of indicating source code locations with potential defects without running the program. For example, consider the Ariane rocket case [1], where the rocket exploded because a 64-bit floating-point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. This error could have been detected by a quite simple static analyzer, and by correcting the right line of the source code in time, the tragedy could have been avoided.

Static code analyzers, especially static application security testing (SAST) tools are nowadays capable of detecting much more complex and profound issues than that by analyzing the source code. However, state-of-the-art tools stop at detecting coding problems and describing potential corrective actions (e.g., SonarLint [2]), and the actual fix of the issue remains the responsibility of the developers. It would make the developers' job a lot easier if there was a tool at hand that could suggest automatic fixes for the detected problems so that they could fix those immediately.

In this paper, we present an automated code repair framework based on ASG (Abstract Syntax Graph) transformation rules that automate the process of finding and fixing minor security flaws. Kirbas et al. [3] argue that practitioners are much more interested in tools that can fix such minor issues quickly and accurately in large quantities than in complex research tools. The framework detects security-related issues without test cases, using problems flagged by a static parser, and displays only fixes that ensure compiling and working code. During the implementation of the tool, we focused primarily on security issues that have been proven to occur typically in programming and can be easily and accurately identified. Since the fixes for these flaws are often simple code transformations, we could build on solutions applied in refactoring tools like [4]. A case study on 6 open-source subject systems showed that our tool was able to generate viable repair patches for 57 out of the 81 detected security issues (70%). For certain types (e.g., revealing private references of mutable

objects) our tool reached close to perfect performance.

The rest of the paper is organized as follows. In Section II, we show related techniques in the area of automatic code repair and program refactoring, which form the basis of our tool. In Section III we introduce the approach to how we repair the source code. In Section IV, we present the essential elements of the framework, and in Section V, we discuss its evaluation with a study. Since the presented tool is still in a prototype state, we highlight our long-term goals with the tool in Section VI.

II. Related Work

Both the literature and the tools available offer a wealth of automatic program repair (APR) tools. Most APR systems are based on some kind of test base, where the goal is to automatically correct failing tests (i.e., improve the functionality of the program). These tools typically use genetic algorithms or heuristics to modify the source code. GenProg [5] and its Java version jGenProg, which is part of the ASTOR [6] program repair Java library, or ARJA [7] tools fall into this category. Besides genetic algorithms, there are of course also solutions based on different deep learning techniques, such as DeepRepair, another component of ASTOR, which tries to repair based on code similarities [8], DeepFix [9], GenPat [10], or solutions using mutations [6]. The disadvantage of the former approaches may be the large search spaces to examine and the difficulty in finding the exact fix for a specific case when it requires a more complex, larger change.

There are approaches where static analyzers like SpotBugs [11], SonarQube [12], PMD [13] are used to detect bugs and try to fix them automatically and quickly based on some patterns [14]. However, all these approaches still rely on failing unit tests as they use static analysis warnings for deriving fix patterns only, while we use static checkers both as bug detection and fix validation techniques. For security bugs, databases such as NIST-NVD or CVE provide suitable input to filter out the faulty code fragments using various machine learning methods [15], possibly recognizing structural and semantic similarities with vulnerable code.

In this paper, we present a repair tool that automates the process of finding and fixing minor security flaws. Our work is similar to those of Xuan-Bach et al. [16], Durieux et al. [17], and Jiang et al. [18] with the distinction that we rely on static code analysis based issue detection (i.e., no test cases are needed) and a rule-based ASG transformation for code repair, all integrated into an end-to-end framework ready for practical application.

III. Security Identification and Repair Approach

At the core of the repair framework is our CodeRepair module which is an open-source command-line application.[1] It implements a pattern-based code repair approach performed as graph transformation on the ASG (Abstract Syntax Graph) representation of the source code. The high-level schematic of the program's operation is shown in Figure 1.

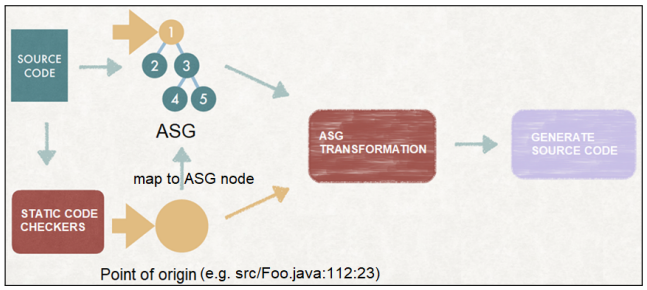

Fig. 1. The ASG transformation based repair algorithm

We first analyze the source code with the OpenStaticAnalyzer [19] open-source tool, which produces the ASG of the system. Each node of the ASG represents a single program element (e.g., statement, identifier, expression). As already outlined in Section I, we use static analysis tools to identify potentially vulnerable locations in the source code, therefore, together with the code analysis, we run a static code analyzer as well. We selected the SpotBugs tool for this purpose but it can be replaced by any other static analyzer or even some other (e.g., deep learning-based) solution that can identify potential vulnerabilities in the program. The output of such a tool should be the type of the detected issue and an exact location in the source code (i.e., path, line, and column of the detected issue). The input of the CodeRepair module is an XML file containing the list of such locations and the type of the detected issues.

The module first maps the exact source code locations to the nodes of the ASG using a heuristic approach proposed by Szőke et al. [4]. Next, it determines the set of possible transformation patterns by the actual type of the detected issue (e.g., possible null dereference). The necessary changes to the ASG are defined by specifying the ASG nodes that need to be modified, deleted, or possibly rebuilt, and also the positions where the changes are to be included in the code, using the original source code. For this, we had to implement an ASG transformation API over the language schema used by OpenStaticAnalyzer, therefore we used a modified version of it.[1] Based on the modified ASG nodes and the information that determines the location of the modifications, we finally generate the modified source code (or a diff patch that can be applied to the original source code). We note that the module can produce several potential repair patches for the same issue if there are multiple fix patterns defined. The validation and verification of the produced patches are carried out by the repair framework described in Section IV.

IV. The Automated Repair Framework

To realize our dedicated goals (i.e., practically applicable security-related automated code repair), we have created a complete framework, which implements the proposed security identification and repair approach (see Section III) and integrates the entire repair process into an end-to-end solution. The framework is written in Java and targets the security-

[1] https://github.com/FrontEndART/OpenStaticAnalyzer/tree/CodeRepairTool/java/cl/CodeRepair

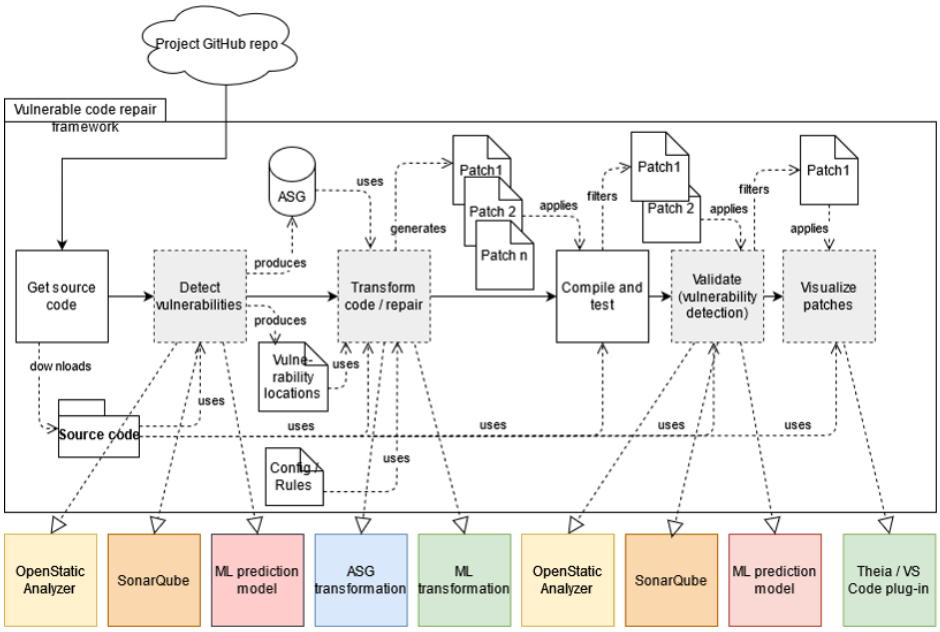

Fig. 2. The conceptual design of the repair framework and plugin

related automated code repair of Java subject systems.[2] The conceptual design of the framework is depicted in Figure 2.

The framework orchestrates the whole process of automated security issue repair, starting from the collection of subject system source code, localizing security-relevant issues, fixing and validating them, and presenting the candidate solutions to the developers. We designed the framework with a high level of abstraction in mind, therefore its architecture is modular and supports a plug-in mechanism. This is to ensure that the users can employ their security detection/repair/validation/visualization approaches still benefiting from the framework that binds all these steps together using well-defined interfaces and driver classes. The modules marked with gray and dashed lines in Figure 2 are exchangeable, they only define the appropriate interfaces for input and output and can be freely substituted with arbitrary implementations. For example, to get the vulnerability locations in the source code, one can apply a static analysis-based vulnerability detection but just as easily apply an ML prediction model to find vulnerable code parts.

The role of the different modules and our current choice of their implementation are as follows.

Get source code. This module acquires the source code from a local directory or a remote repository (e.g, git).

Detect vulnerabilities. The module takes the source code as input and produces the list of source code positions where vulnerabilities are detected. We implement this module with the open-source code analysis tool called OpenStaticAnalyzer [19] that integrates many checkers, like SpotBugs, SonarQube, and PMD that can detect vulnerabilities as proposed in Section III. The module can also produce different artifacts upon code analysis, like representation of the source code (ASG, embedding, etc.) to be used later in the repair step.

Transform code / repair. This is the core module of the framework that performs the actual code repair task. It can use the source code of the project and the intermediate representations produced in the previous step (e.g. an ASG). We implement an Abstract Semantic Graph (ASG)-based transformation that performs code changes at the ASG level and generates source code (Java) based on this transformed ASG as described in Section III.

Compile and test. After the code repair candidates are generated, we need to validate that the proposed code changes keep the system in a syntactically valid state with all the unit tests passing. This module performs these sanity checks. The supported build systems can be extended as well, by implementing the proper interfaces of this module. Currently, the framework supports the Maven and Gradle build systems. After the module is executed, some patches might be thrown away that produce failing tests or break the syntax of the code.

Validate. After filtering out invalid patches in the previous step, we need to validate that the originally detected vulnerability disappears after applying the generated patch. We achieve this by re-running the same analysis as in the step of vulnerability detection and confirming that the targeted vulnerability is not found anymore in the source code version after applying the fix patch. The output of this module is the filtered list of patches that indeed remove the vulnerability in question. If the validation fails for all potential fixes, the framework will not offer to fix a given bug.

Visualize patches. This module implements the last step of the repair process, the visualization of the code changes. The input of this module is the final set of patches. We currently provide a Visual Studio Code IDE plug-in as the means for results visualization (see more details in Section V-A). The

[2]https://github.com/FrontEndART/AIFix4SecCode

VSCode (Visual Studio Code) plugin of the framework is capable of visualizing the output and triggering the execution of the analysis. The patches are presented in a "diff view" and the user can navigate between patch candidates where they can decide to apply or decline the patches.

The framework can be used as a command-line tool as well (i.e., running the executable jar directly) that can support DevOps by integrating the framework into CI/CD pipelines. The whole repair process can be started from the IDE plug-in also, which is more convenient for the developers.

V. Tool Evaluation

A. Data Mining and Implemented Fixes

To find out the most relevant issues to be fixed and derive their repair recipes, we followed a data-driven, empirical approach. We collected real-world fixes by mining GitHub for a set of commits with SonarQube and SpotBugs in commit messages, issues, and source code of JAVA projects. The source code has been examined by hand to filter out duplicates and false positives.

We have collected the originally vulnerable and fixed versions of the code along with metadata (commit message, commit links, the position of the issue) grouped by the selected issues. The first iteration of the dataset contains 1902 instances of bugs and their fixes accompanied by a machine-processable JSON metadata descriptor. The type-wise distribution of the collected samples can be seen in Figure 3, the S1444 ("public static" fields should be constant) issue comes from SonarQube, and the rest are detected by SpotBugs.[3]

These are the vulnerability-related warning types that are most frequently fixed within the open-source community, therefore we identified them as our primary target for providing automated repairs. As a first step, our tool is capable of fixing the following issues:

1) EI_EXPOSE_REP / EI_EXPOSE_REP2 (SpotBugs) – three fix strategies are implemented: (i) fix with object cloning; (ii) fix with array copying; (iii) fix with instantiating a new DateTime object.
2) MS_SHOULD_BE_FINAL / SQUID_S1444 (SpotBugs / SonarQube) – one fix strategy is implemented: fix with adding the "final" keyword.
3) NP_NULL_ON_SOME_PATH / NP_NULL_ON_SOME_PATH_EXCEPTION (SpotBugs) – one fix strategy is implemented: add a null check for the variable in a ternary operator.

The utilization of the dataset will go beyond identifying issues and their repair strategies, it can be used for training machine learning-based fixing of detectable issues in the framework or for deriving prioritization among the various repair patch candidates.

B. Evaluation Study

We used the IDE-based analysis usage scenario to evaluate the tool's performance on 6 subject systems. We selected popular open-source systems that contained potential security issues that the OpenStaticAnalyzer tool could detect. The set of subject systems we used in the study is shown in Table I.

TABLE I
Study subject systems

System	Version	Repo URL	Java Lines
ANTLR4	4.2	https://github.com/antlr/antlr4	40K
Arduino	1.8.19	https://github.com/arduino/Arduino	27K
EasyExcel	3.0.5	https://github.com/alibaba/easyexcel	25K
Guava	31.0.1	https://github.com/google/guava	360K
MapDB	0.9.6	https://github.com/jankotek/mapdb	47K
Titan	0.5.1	https://github.com/thinkaurelius/titan	80K

For all the subject systems, we downloaded the source code for the selected releases and loaded them into the VSCode plug-in provided as part of the repair framework. We started the analysis from the plugin and used its diff view to observe the generated patches. For 2 out of the 6 subject systems, two of the paper authors with more than 15 years of experience also evaluated all the generated patch candidates manually. They either accepted or declined them providing also the reasoning of their decision in a textual form.

The overall results of the repair framework on the 6 subject systems are shown in Table II. The framework detected 81 security-relevant issues altogether that it can potentially fix in the 6 subject systems and was able to provide a fix patch for 57 of them (70%). The generated patches and statistics are available online.[4]

The repair framework was able to correctly fix all the `MS_SHOULD_BE_FINAL` issues detected. The repair of this issue requires a single "final" keyword insertion, which is a fairly simple transformation. Nonetheless, fixes for the `EI_EXPOSE_REP` and `EI_EXPOSE_REP2` are almost always successfully generated (92.3% and 91.7% of the cases, respectively). The only exception is the EasyExcel system, where the framework was unable to generate a fix patch for 1-1 such problems. Looking into the results, we found that the problem is that the framework could not locate the appropriate source code location where to apply the code repair. This was because EasyExcel is using the Lombok[5] library for generating getter and setter methods. The source code contains only the data members of the Java beans and the necessary constructors, getters, and setters are generated during compile time based on various Lombok annotations. However, both `EI_EXPOSE_REP` and `EI_EXPOSE_REP2` were firing to getter and setter methods of such Java beans. Since SpotBugs uses the class files for detecting issues, it could successfully locate the generated getter and setter methods, while the repair framework uses the source code, where it could not locate the appropriate methods, therefore no viable repair patches could be generated.

Our framework was the least efficient for finding correct fixes for potential null pointer dereference issues (`NP_NULL_ON_SOME_PATH` and `NP_NULL_ON_SOME_PATH_EXCEPTION`). We investigated

[3]https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html

[4]https://doi.org/10.5281/zenodo.6778637

[5]https://projectlombok.org/

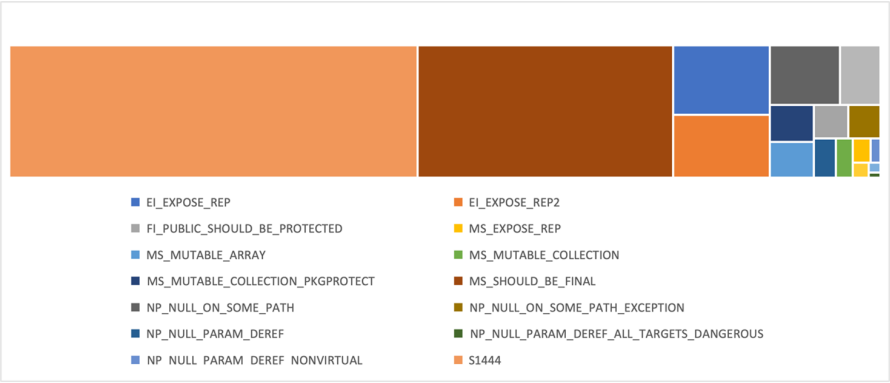

Fig. 3. Ratio of entries in the dataset grouped by warning types

TABLE II
CODE REPAIR RESULTS ON 6 SUBJECT SYSTEMS

Issue	System	ANTLR4	Arduino	EasyExcel	Guava	MapDB	Titan	Repair ratio
EI_EXPOSE_REP	Found	1	0	1	0	5	6	
	Fixed	1	0	0	0	5	6	92.3%
EI_EXPOSE_REP2	Found	0	4	1	0	2	4	
	Fixed	0	4	0	0	2	4	91.7%
MS_SHOULD_BE_FINAL	Found	3	3	12	0	2	3	
	Fixed	3	3	12	0	2	3	100%
NP_NULL_ON_SOME_PATH	Found	3	1	0	22	2	5	
	Fixed	0	0	0	9	1	2	36.4%
NP_NULL_ON_SOME_PATH_EXCEPTION	Found	1	0	0	0	0	0	
	Fixed	0	0	0	0	0	0	0%
	Total time (s)	399	578	1,768	1,140	894	2,037	

all the failed cases manually and found that there were different reasons why the framework was unable to generate repair patches.

For ANTLR4, the `NP_NULL_ON_SOME_PATH_EXCEPTION` and two out of the three `NP_NULL_ON_SOME_PATH` issues are located in generated sources. Since SpotBugs analyzes the compiled bytecodes, it can detect the issue, while the framework works on the source code of the project, therefore could not locate the position where to apply the repair rules. For the last `NP_NULL_ON_SOME_PATH`, the framework was not able to transform the expression in a syntactically correct way, therefore there is still room for improving the repair rules. We note, however, that this particular issue is a false positive SpotBugs finding since there is a null check earlier in the code.

In Arduino, the possible null pointer dereference is located in a negated condition of an `if` statement, which the framework could not transform correctly.

For Guava, the framework could not fix 13 `NP_NULL_ON_SOME_PATH` out of the 22. In three cases, the framework could not locate the exact column information of the code element potentially being null (i.e., SpotBugs reports only line information for a warning but the repair framework needs the exact source code location, which it tries to extract). All these are located in inner classes within complex expressions. All the remaining cases are such that the possibly null variable is not on the right-hand side of an assignment operation (which we assume when generating the null check in ternary as we observed only such cases in the collected dataset), therefore the generated repair will not form a valid statement. It suggests that we might need to consider adding further transform rules to this issue type.

In the case of MapDB, the unsuccessfully fixed `NP_NULL_ON_SOME_PATH` issue is fired for the `log` variable in a complex condition of an if statement, where there are multiple checks concatenated together with boolean || operators. The framework cannot handle the case properly, where there are multiple references to the same variable that might be potentially null. It suggests that further fix strategies are needed for this particular issue type.

For Titan, the framework generated a fix for two out of the three `NP_NULL_ON_SOME_PATH` issues, but they could not be compiled. Our ternary operator fix is generated inside another ternary in these cases, which would not work. For the last one, the issue was similar to the `log` case, a `file` variable was used multiple times in a large condition, which the framework could not handle.

The running time of the repair tool is reported in the last row of Table II. Even for the largest systems the complete running time of detection, repair, and validation is within half an hour. The running time depends on the number of generated repair candidates but these are realistic subject systems that reflect the expected analysis time.

Even though we know that all of the patches compile and remove the underlying SpotBugs warnings, to evaluate the correctness of the patches semantically as well two of the authors validated all the proposed patches for the MapDB and the Titan systems. In general, the manual evaluation confirmed the previous analysis. Both evaluators

marked all the generated patches for the EI_EXPOSE_REP, EI_EXPOSE_REP2, and MS_SHOULD_BE_FINAL issues as acceptable and semantically correct. They agreed that the single fix of NP_NULL_ON_SOME_PATH in MapDB is invalid as it changes the expected behavior of the code. The fix inserted a ternary null check into a one-liner compare method (consisting of a ternary expression itself), where the null value was already handled properly. They agreed that the issue reported by SpotBugs can be considered a false positive. The two NP_NULL_ON_SOME_PATH fixes in Titan were syntactically and semantically correct according to the evaluators, however, both of them marked the fix to be invalid. This was again because the actual variable could not be null due to a Preconditions.checkArgument() call earlier in the code. Therefore, the fix was incorrect due to a false positive issue report.

VI. Conclusion and Future Works

In this paper, we have presented an end-to-end framework supporting the automated repair of security-relevant static analysis warnings. Our goal was to provide a practical and efficient way of automatically fixing code problems that could lead to vulnerabilities. The framework relies on so-called repair recipes defined as graph transformations on the ASG of the underlying subject system.

Even though the framework is a work-in-progress prototype, it is already capable of fixing 6 different types of security-related issues. A VSCode plug-in for visualizing and evaluating the generated patches is also part of the framework. Both the issues to fix and the corresponding repair recipes were collected by an extensive data mining process from open-source project code histories.

With a case study on 6 well-known open-source subject systems, we showed that the framework is applicable in practice, as it was able to generate viable repair patches for 57 out of the 81 detected coding issues (70%). Moreover, the framework's efficiency on the EI_EXPOSE_REP, EI_EXPOSE_REP2, and MS_SHOULD_BE_FINAL issues was close to 100%. It struggled a bit with repairing potential null dereference issues (fixed 12/34, 35%), which indicates that further improvement and additional recipes are required. Nonetheless, in many cases, the underlying problem was that the identified security issue was a false positive, which depends entirely on the code analysis tool we integrated. The manual evaluation of two subject systems by two of the authors confirmed that the vast majority of the generated viable patches are acceptable by humans as well (both syntactically and semantically).

We plan to further improve our framework by gathering even more data on bug-fix pairs to support the repair generation process. The repair patch generation algorithm can provide multiple candidate patches, which all have to pass compilation and unit tests. However, the developers have to select the best candidate to apply the patch. The evaluation of the tool revealed that prioritization of the patches would be beneficial to help this process, which we intend to add soon. Furthermore, we will continuously implement newer repair strategies for other, not yet supported security-relevant issues.

Acknowledgement

The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004) and by project TKP2021-NVA-09, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. The research was partly supported by the EU-funded project AssureMOSS (Grant no. 952647) as well.

Furthermore, Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences.

References

[1] Ariane 5. [Online]. Available: "https://iansommerville.com/software-engineering-book/case-studies/ariane5"
[2] SonarLint. [Online]. Available: "https://www.sonarlint.org/"
[3] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "on the introduction of automatic program repair in bloomberg," *IEEE Software*.
[4] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 253–258.
[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 364–374.
[6] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of ISSTA*, 2016.
[7] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.
[8] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 479–490.
[9] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI conference on artificial intelligence*, 2017.
[10] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring Program Transformations From Singular Examples via Big Code," ser. ASE, 2019.
[11] Spotbugs. [Online]. Available: "https://spotbugs.github.io/"
[12] Sonarqube. [Online]. Available: "https://www.sonarqube.org/"
[13] Pmd. [Online]. Available: "https://pmd.github.io/"
[14] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2019, pp. 456–467.
[15] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018.
[16] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER) 2016*. IEEE, 2016.
[17] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017.
[18] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," ser. ISSTA, 2018.
[19] OpenStaticAnalyzer. [Online]. Available: "https://openstaticanalyzer.github.io/"