Maestro: A Platform for Benchmarking Automatic Program Repair Tools on Software Vulnerabilities

Eduard Pinconschi
INESC-ID, and University of Porto
Portugal

Quang-Cuong Bui
Hamburg University of Technology
Germany

Rui Abreu
INESC-ID, and University of Porto
Portugal

Pedro Adão
IST-ULisboa and IT – Lisboa
Portugal

Riccardo Scandariato
Hamburg University of Technology
Germany

ABSTRACT

Automating the repair of vulnerabilities is emerging in the field of software security. Previous efforts have leveraged Automated Program Repair (APR) for the task. Reproducible pipelines of repair tools on vulnerability benchmarks can promote advances in the field, such as new repair techniques. We propose Maestro, a decentralized platform with RESTful APIs for performing automated software vulnerability repair. Our platform connects benchmarks of vulnerabilities with APR tools for performing controlled experiments. It also promotes fair comparisons among different APR tools. We compare the performance of Maestro with previous studies on four APR tools in finding repairs for ten projects. Our execution time results indicate an overhead of 23 seconds for projects in C and a reduction of 14 seconds for Java projects. We introduce an agnostic platform for vulnerability repair with preliminary tools/datasets for both C and Java. Maestro is modular and can accommodate tools, benchmarks, and repair workflows with dedicated plugins.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Vulnerability, program repair

ACM Reference Format:
Eduard Pinconschi, Quang-Cuong Bui, Rui Abreu, Pedro Adão, and Riccardo Scandariato. 2022. Maestro: A Platform for Benchmarking Automatic Program Repair Tools on Software Vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533767.3543291

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3543291

1 INTRODUCTION

In order to reduce development costs and avoid the exploitation of undiscovered (and unpatched) vulnerabilities that may cause substantial losses to organizations and users, detecting and fixing security vulnerabilities should be performed as early as possible in the software development life cycles. As an example, a recently critical vulnerability related to remote code execution, namely Log4Shell, was discovered in Apache Log4j 2 (CVE-2021-44228[1]) after being remained unnoticed for seven years. It caused a big disruption on the Web because Log4j has been used as the logging library for thousands of Java packages and products, including those from critical enterprise systems such as Amazon Web Services.

Automating the repair of vulnerabilities has a crucial role in software security to shift left. Prior efforts approach vulnerability repair with several techniques which are based on static analysis [8, 13–15, 20], symbolic execution [7, 11, 17], gray-box fuzzing [9], and dependency analysis [10]. Despite these efforts, most of the proposed approaches focus mainly on C programs and cover several specific types of vulnerabilities, such as buffer overflows. Machine learning approaches [3, 4, 16, 21] aim to generalize the repair of vulnerabilities; however, these are conditioned by the data. Meanwhile, automated program repair (APR) shares similar concerns to the automated repair of software vulnerabilities, that of correcting faulty software. APR has reached a reasonable maturity and addressed a broad range of software issues including security issues, however, in a limited amount. This limitation is mainly due to the lack of available benchmarks for reproducible vulnerabilities in the literature [1]. Furthermore, a standard and reproducible execution pipeline of repair tools on vulnerability benchmarks should be introduced to encourage researchers and practitioners to perform empirical studies and propose new repair techniques for software vulnerabilities. Existing work focuses on frameworks for the empirical studies of APR for vulnerabilities [12, 18, 19]. However, these are initial prototypes, limited in design to scale up to an industrial setting for repairing software vulnerabilities.

Therefore, we proposed a decentralized platform, namely Maestro, to perform the evaluation of the current state-of-the-art APR tools on the datasets of security vulnerabilities in multiple programming languages (i.e., C and Java). By leveraging containerization as the infrastructure, our platform is designed with a highly decentralized architecture, which benefits considerably the large-scale experiments of repair tools. Maestro also plays the role of a means

[1] https://nvd.nist.gov/vuln/detail/CVE-2021-44228

to support research and proposals for new repair techniques by providing benchmarks and core components for repair tools (e.g., fault localization, compiler, and executor of test cases). Currently, the two first-of-its-kind vulnerability datasets CB-Repair [18] and Vul4J [1], which are curated and devoted to the research community on program repair for security bugs, have been partially added in our platform. Moreover, an initial number of four repair tools (GenProg, MUT-APR, jGenProg, jMutRepair) have been integrated into Maestro to perform our preliminary evaluation.

The main features of Maestro are summarized as follows:

- **Connectivity between repair tools and benchmarks**: The source code of vulnerable project and dependency libraries is shared between the benchmark and repair tool containers, which allows the tool to manipulate the source code without running on the same machine serving the benchmark. Tools can assign fault location, compilation, and test execution tasks to benchmark handlers by *signal commands*.
- **Controlled environment for the execution of repair**: The repair tools and benchmarks are provided within the containerized machines, which ensures the same configuration and OS environment variables for every repair attempt on our platform.
- **Extensively distributed configuration and runtime architecture**: Repair tools and benchmarks are integrated into the platform as containers via *plug and play* manners, in which domain-specific language (DSL) files are used to describe the metadata for running the tools and serving the benchmarks. These containers are organized in a microservice architecture, in which RESTful APIs are geared to perform the interactions between these nodes.

2 PLATFORM ARCHITECTURE

Both APR tools and benchmarks of bugs depend on a specific configuration of their host environment to work correctly that, if not configured accordingly, may influence their behavior and results. Additionally, different tools and programs may conflict due to their OS/dependency requirements, which adds complexity to evaluating multiple APR tools across numerous benchmarks. Thus, APR requires controllable, reproducible, and independent environments that are easy to interact with to perform accurate experiments.

We propose Maestro, a platform that provides an accessible means of performing automated repair of software vulnerabilities in isolated environments and in an out-of-the-box manner. Maestro has a decentralized and microservice-based architecture based on Docker containers[2] as illustrated in Figure 1. Maestro is composed of an orchestration component, Nexus, and two components Orbisand Synapser, that respectively convert tools and benchmarks into microservices. To integrate a tool or benchmark in Maestro, a developer *only* needs to develop a plugin for its respective component, Synapser or Orbis. Then, the plugin for Nexus needs to connect with Synapser and Orbis for each tool/benchmark pair.

Nexus connects the tools to the benchmarks. Essentially, this component orchestrates the repair workflow through a scripted interaction among the tool/benchmark pairs. Nexus offers commands for provisioning, serving, and managing the instance of the

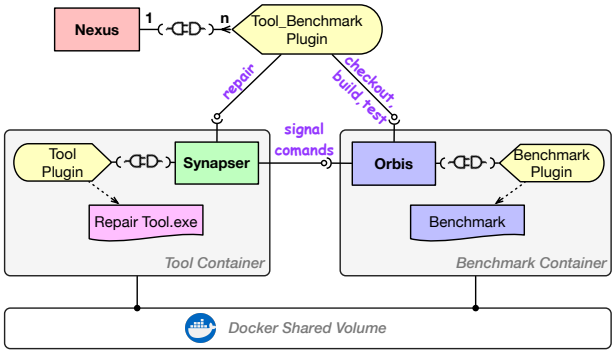

Figure 1: Platform Architecture and Components.

tools/benchmarks. The REST API of each component allows the containers to communicate between them. A global docker volume is shared between the containers for each component to read, write, and share persistent data. Our implementation works locally but can be easily extended to work remotely by mounting a remote file system.

Orbis builds benchmarks of software vulnerabilities as microservices. This component provides a set of methods to interact with and handle project software snapshots. The core methods allow checking out the project into a working directory, building it into an executable, and testing the project's functionality with the provided oracle. Other endpoints allow one to query the projects and vulnerabilities from the benchmark and generate the test oracle with provided scripts.

Synapser extends APR tools into portable microservices. It generalizes the APR tool through a plugin that offers an API with a set of methods for preparing, executing, and collecting the results generated by the tool. Synapser behaves as a proxy for the APR tool through the signal commands. The signals encapsulate the commands to be executed by the APR tool on the benchmark. Additionally, signals can have handlers to parse and process requests and responses. For instance, when it is necessary to map specific arguments between the tool and the benchmark.

3 PLATFORM DESCRIPTION AND USAGE

In this section, we describe in more detail the components in our platform, their general usage, and an evaluation of the time performance of our platform. Due to space limitations, we give real usage examples in our repository.[3]

We have implemented so far in Maestro plugins for tools and benchmarks for C and Java programming languages. We provide plugins for four tools (GenProg and MUT-APR for C, and jGenProg and jMutRepair for Java), and we consider our selection to promote a fair evaluation. Regarding the benchmarks, we provide plugins for two datasets, CB-Repair and Vul4J, and for both, we include five projects. We consider these datasets suitable because they have the necessary resources for vulnerability repair. Table 1 describes in detail the projects and their vulnerabilities that we have selected. For the reason of simplicity and due to the limitations of the targeted tools, we focused on versions of programs/projects that contain a single kind of vulnerability and that have a single file.

[2] https://www.docker.com/resources/what-container/

[3] https://github.com/epicosy/nexus/blob/main/TUTORIAL.md

Table 1: Datasets, execution times, and overheads.

Datasets	Projects	CWE	Execution Times (in seconds)				Overhead
			Previous		Maestro		
			GenProg	*MUT-APR*	*GenProg*	*MUT-APR*	
CB-Repair [18]	BitBlaster	824	247	394	270	419	24
	expression_database	119	1,184	106	1,279	144	66,5
	Scrum_Database	122	490*	115	488*	122	2,5
	WordCompletion	125	897	47	953	48	28,5
	yolodex	787	477*	157	467*	155	-6
			jGenProg	*jMutRepair*	*jGenProg*	*jMutRepair*	
Vul4J [1]	apache/tika	835	741	260	762	264	12,5
	cloudfoundry/uaa	200	421	320	348	282	-55,5
	jenkinsci/jenkins	835	556	286	523	259	-30
	spring-projects/spring-framework	22	146	155	131	139	-15,5
	x-stream/xstream	502	197	100	207	101	5,5

* *Repair succeeds with POV timing out.*

3.1 Setting the components

Each component in Maestro requires a schema file and a plugin file to work. The schema file is written in *YAML* with a specific specification language for the component, and the plugin file is written in *Python*. The components provide generic objects and methods to facilitate the implementation of the required plugins. The generic methods in the plugins are connected to the respective REST API endpoints in the component. The general workflow for instantiating the benchmark/tool in our platform consists in: (i) *provisioning*, to instantiate a Docker container with the tool or benchmark and their respective environment to function correctly (*Nexus* handles provisioning by building the container's image from a docker file or by pulling it from the Docker Hub library); (ii) *setup*, to install the component (Synapser for tools/Orbis for benchmarks) that interacts with Nexus along with the necessary plugin and its configuration. This step can be performed during the provisioning; and (iii) *serving*, to launch the REST API of the component, which allows remote interaction and transmission of structured data.

3.1.1 Setting up the benchmark. This requires to represent the benchmark in the schema file, and specifying the metadata for the vulnerable snapshots of each project. The essential attributes for each project are: (i) path to the GitHub repository where the project is hosted; (ii) name of the project and a unique identifier; (iii) build specification (system, version, architecture, arguments, and scripts); (iv) commit hash of the vulnerable version; (v) a unique identifier for the vulnerability (the same commit can have multiple vulnerabilities); and (vi) relative path to the vulnerable file(s), each with the vulnerable line number(s).

In addition to that, each project must contain two additional files with a schema to test the project. The tests file includes the test cases that ensure the correct functionality, and the proof of vulnerability (PoV) file specifies the exploit(s). Each file must specify: (i) the directory path of the script for testing, (ii) the command to run the script, (iii) the path to the test cases, (iv) general arguments if needed, and (v) the test cases. Each test case must have specified its name and file name. Other possible attributes are timeout limit, specific arguments, and order in which the tests should be run.

The plugin file represents the generic operations performed by the benchmark and it must implement at least the functionality for the: (i) *build method*: to build the project with the configuration in the schema file; and (ii) *test method*: to test the project with the testing scripts on the test cases. The default *checkout* in Orbis must be used in the plugin to download/update all project files in the vulnerable version to a working directory. Two additional methods can be implemented, *gen_povs* and *gen_tests*. The former generates the proof of vulnerability tests and can be done with program analysis tools. The latter generates the functionality tests and can be done with testing techniques, *e.g.*, model-based testing.

3.1.2 Setting up tool. This includes defining the schema file that represents the tool which must contain the following attributes: (i) name of the tool executable; (ii) directory path of the tool executable; (iii) signal/command pair (to bind custom build/test handlers to commands executed by the tool on the benchmark); and (iv) general arguments for the tool. The plugin file must implement the functionality for the following: (i) *repair method*: to provide the repair command that launches the execution of a repair instance on a specific project; and (ii) *get_patches method*: to list all the generated patches including the fixes (these are in the form of diffs between the target file and the generated file). Optionally, the plugin file can implement the *parse_extra* method to process extra arguments injected by the tools in the commands. Additionally, the plugin can implement custom handlers for the signal commands to process the responses returned by the benchmark through the Orbis API. Synapser also provides the *stream* method for querying information on the execution status of the repair, along with a web socket for streaming the execution output of the tool.

3.1.3 Running the tool on the benchmark. In Nexus, two schema files are necessary for defining the services for the tool and the benchmark. These must include the following attributes: (i) name of the component; (ii) type of the component (tool/ benchmark); (iii) Docker image tag; (iv) GitHub/DockerHub repository; (v) name of the Docker container; and (vi) API port number. The plugin file represents the repair workflow between the tool/benchmark pair, and it is implemented in the *run* method by using the API of each component. The generic workflow includes: (i) checking out the vulnerable snapshot of a particular project; (ii) setting up the *build* and *test* signals; (iii) setting up repair arguments, such as the number of tests; and (iv) sending the repair request to Synapser.

3.2 Evaluation

To demonstrate the time performance of Maestro, we execute our tools in C on projects from CB-Repair with successful repair

attempts—reported in SecureThemAll [18]. Similarly, for Java, we extend RepairThemAll [5] with Vul4J. We execute the experiments on the exact conditions—hardware, configurations, and tests. Table 1 includes information about the datasets, the projects, their associated vulnerability type, and the running times of each tool within the environment of previous studies and with Maestro. Table 1 also include an *overhead* column that describes the difference in the execution times on average for Maestro compared to the execution times with the environment of previous studies. Values in the *overhead* column indicate that the tools in C have a higher overhead. RepairThemAll performs post-repair tasks such as patches gathering while Maestro does not. This justifies the lower overhead results for Java projects and also gives a hint on the trade-off of introducing the *signal commands*. On average, an overhead of 23 seconds is introduced for C tools, and a reduction of 14 seconds for Java tools. The complete results can be found here. [4]

4 RELATED WORK

In this section, we first summarize the related work that proposed the platforms and frameworks dedicated to the empirical studies of program repair. Then we introduce the vulnerability datasets for C and Java that can be added to our platform.

Platforms: The closest to ours is BugZoo platform [19], which provides a decentralized and controlled environment for sharing data and interactions between the C datasets and the repair tools. Maestro complements BugZoo in several ways. First, Maestro allows repair tools to entrust core tasks of the repair process to dataset handlers via *signal commands* for more reliable performance, which BugZoo does not. Second, Maestro aims for vulnerabilities in multiple programming languages, while BugZoo focuses on general bugs in only C programs. RepairThemAll [5] is a monolithic framework used to perform a large-scale evaluation of eleven repair techniques on 2,141 generic Java bugs obtained from five different benchmarks. RepairThemAll also allows running multiple repair experiments in parallel (maximum number of threads in the local machine). SecureThemAll [18] follows a similar methodology to RepairThemAll, and it evaluated ten different C repair techniques on 55 security vulnerabilities from the CB-Repair dataset. APIARTy, proposed by Kechagia et al. [12], is a dockerized execution framework used to evaluate fourteen Java repair tools on a dataset of 110 API misuses related to security issues. APIARTy follows the simply monolithic architecture and focuses only on Java programs.

Datasets: Most vulnerability repair studies usually require the input datasets with the tests or exploits for the vulnerabilities. The Juliet Test Suite [6] is a dataset that contains 64, 099 small vulnerable programs covering 118 different categories from the Common Weakness Enumeration list (CWE). However, since it is a synthetic dataset, these programs contain the simplest forms of vulnerabilities which is rarely the case in well-established projects. An example of a dataset with programs close to real software is the CB-Repair [18]. It includes 55 vulnerable programs written in C and covers 36 different CWEs. Its programs are from the DARPA Challenge Sets [2], and their design allows the evaluation of vulnerability remediation systems. Vul4J [1] is the closest example of a dataset with real software and reproducible vulnerabilities. This dataset contains 51 open-source Java projects extracted from the "Project KB", covering 79 vulnerabilities and spanning over 23 different CWEs. The projects in Vul4J come from multiple domains, including libraries, web frameworks, data-processing apps, and CI/CD servers.

5 CONCLUSION

In this paper, we present Maestro, a platform for evaluating repair tools across different benchmarks with low overhead to tool developers. Our platform is designed to be used as an extensible, portable, and decentralized sandbox that assists researchers and practitioners in conducting empirical program repair research with ease. Maestro provides handy interfaces that allow integrating out-of-the-box repair tools and benchmarks via a plug and play approach. Our experiments showed that there is no considerable performance overhead of repair progress in our pipeline compared to the original. As a proof-of-concept, we integrated a total of four repair tools and two vulnerability datasets for the C and Java programming languages into our platform.

[4] https://github.com/epicosy/nexus/blob/main/patches

REFERENCES

[1] Q. C. Bui, R. Scandariato, and N. E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *MSR*.
[2] Brian Caswell. 2017. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus Accessed 19-April-2022.
[3] Z. Chen, S. Kommrusch, and M. Monperrus. 2022. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE TSE* (2022).
[4] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin. 2020. SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning. *arXiv:2010.10805* (2020).
[5] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *ESEC/FSE*.
[6] NSA Center for Assured Software. 2020. Test Suites. https://samate.nist.gov/SARD/testsuite.php Accessed 19-April-2022.
[7] F. Gao, L. Wang, and X. Li. 2016. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *ASE*.
[8] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *ICSE*.
[9] X. Gao, S. Mechtaev, and A. Roychoudhury. 2019. Crash-Avoiding Program Repair. In *ISSTA*.
[10] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM TOSEM* (2021).
[11] Z. Huang, D. Lie, G. Tan, and T. Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *S&P*.
[12] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman. 2021. Evaluating automatic program repair capabilities to repair API misuses. *IEEE TSE* (2021).
[13] J. Lee, S. Hong, and H. Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *ESEC/FSE*.
[14] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. 2007. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *ASIACCS*.
[15] F. Long, S. Sidiroglou, D. Kim, and M. Rinard. 2014. Sound Input Filter Generation for Integer Overflow Errors. In *POPL*.
[16] S. Ma, F. Thung, D. Lo, C. Sun, and R. H Deng. 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*. Springer, 229–246.
[17] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert. 2021. IntRepair: Informed Repairing of Integer Overflows. *IEEE TSE* (2021).
[18] E. Pinconschi, R. Abreu, and P. Adão. 2021. A Comparative Study of Automatic Program Repair Techniques for Security Vulnerabilities. In *ISSRE*.
[19] C. S. Timperley, S. Stepney, and C. L. Goues. 2018. Poster: BugZoo – A Platform for Studying Software Bugs. In *ICSE-C*.
[20] T. Wang, C. Song, and W. Lee. 2014. Diagnosis and Emergency Patch Generation for Integer Overflow Exploits. In *DIMVA*.
[21] Y. Zhang, Y. Xiao, M M A Kabir, D. Yao, and N. Meng. 2022. Example-Based Vulnerability Detection and Repair in Java Code. *arXiv:2203.09009* (2022).