Cerberus: a Program Repair Framework

Ridwan Shariffdeen*, Martin Mirchev†, Yannic Noller‡, Abhik Roychoudhury§
National University of Singapore, Singapore
Email: *ridwan@comp.nus.edu.sg, †mmirchev@comp.nus.edu.sg, ‡yannic.noller@acm.org, §abhik@comp.nus.edu.sg

***Abstract*—Automated Program Repair (APR) represents a suite of emerging technologies which attempt to automatically fix bugs and vulnerabilities in programs. APR is a rapidly growing field with new tools and benchmarks being added frequently. Yet a language agnostic repair framework is not available. We introduce CERBERUS, a program repair framework integrated with 20 program repair tools and 9 repair benchmarks, coexisting in the same framework. CERBERUS is capable of executing diverse set of program repair tasks, using multitude of program repair tools and benchmarks.**

Video: https://www.youtube.com/watch?v=bYtShpsGL68

***Index Terms*—automated program repair, repair platform**

I. INTRODUCTION

Automated Program Repair (APR) [1] has many applications in software engineering, like supporting software developers in fixing bugs, particularly for security vulnerabilities and concurrency bugs, but also for guiding students to solve programming assignments in an educational context. Furthermore, APR has already been adopted to some initial extent in industry deployments [2], [3], in which they are usually added to the CI/CD pipelines, proposing patches for failing test cases. While we can observe a plethora of APR approaches [4], they greatly vary in required patch ingredients, target languages, and execution environments, including the dependencies and instrumentation requirements. This large diversity poses a challenge for bug and patch reproduction and technique comparisons.

The existing approaches for integrating APR tools into frameworks [5]–[7] fail to provide an environment and architecture that would allow covering multiple application domains and target languages. REPAIRTHEMALL [5] assumes that APR tools are provided as .jar-files and is customized for Java repair. SECURETHEMALL [6] is customized for security vulnerability repair and targets C/C++ programs. The most recent proposed work is MAESTRO [7], a benchmarking framework for automated program repair tools that supports multiple implementation and target languages. However, their design choices prevent the integration of semantic-based techniques like CPR [8] that require complex build infrastructures.

To close this gap, we present CERBERUS, a program repair framework that provides the means to integrate many different APR tools with diverse target languages and application domains, their execution environments, and their experiment data sets, resulting in a unified way of accessing the developed tools. In contrast to the existing works, CERBERUS does not make any assumption about implementation or target language, and is not customized to a specific application domain. In contrast to MAESTRO, it encapsulates the benchmark and the repair tool in a single container setup, which makes it straightforward to integrate tools with complex build infrastructure. In fact, we have already integrated 20 tools for C/C++ and Java with various repair methodologies covering search-based, semantic-based, and learning-based APR and multiple application domains, including test-based general-purpose repair, security repair, static-based concurrency repair, and student assignment repair.

CERBERUS is useful for software engineering researchers as well as for software developers. Researchers can integrate their new APR tool into our framework and perform evaluations with the already integrated tools. Thereby, they do not need to consider dependency issues or the technical setups of other tools because our framework makes them readily available. Moreover, we have already integrated the corresponding benchmarks and data sets, which makes it straightforward to run additional experiments. Software developers, who may not be familiar with APR and the existing tools, can use CERBERUS to apply different APR tools on their own (private) data set to see which technique is most suited for their needs. CERBERUS makes the existing APR tools accessible beyond their original experimental environment (i.e., as reported in the corresponding research papers), enables the reproducibility of APR studies, allows comparisons between tools, and enables practitioners to get easy access to the state of the art in APR.

To demonstrate the capabilities of CERBERUS in handling a diverse set of APR approaches, we used it to reproduce the experiments of VERIFIX [9], SEQUENCER [10], and RECORDER [11]. Both SEQUENCER and RECORDER are learning-based APR approaches, which require a specific environment (i.e., CUDA-compliant GPU) in order to work, and VERIFIX applies repair in the educational context, which makes it different from the standard general-purpose repair application. Our results show that CERBERUS can produce similar or the same results as the original works. We observed minor differences in the results for VERIFIX because we used the latest tool version, which the authors had improved since the original publication. We make the following contributions:

- CERBERUS, a fully agnostic repair platform with a layered architecture that allows the addition of new tools and benchmarks, including complex build infrastructures,
- the integration of 20 program repair tools and 9 repair benchmarks across multiple target languages and application domains, and
- the demonstration of CERBERUS's capabilities on executing repair in the educational context with VERIFIX and

in general-purpose repair with the learning-based APR techniques SEQUENCER and RECORDER.

II. DESIGN AND USAGE

In this section, we describe the design architecture and the components of our platform. The default mode of execution in CERBERUS is using containers, which allows to create isolated, modular, and easily reproducible experiments for empirical studies. However, CERBERUS is designed to cater repair tools in both containerized and non-containerized environments. This is because not all program repair tools are available as a Docker container, but also can be made available as a virtual machine (e.g., SENX [12]). The platform is built with the aim of providing flexibility in executing repair tools with less assumptions about the environment and the dependencies required. For brevity, the rest of the paper discusses the containerized mode of CERBERUS. Instructions for specifying the virtualization can be found in our repository.

Experiments in program repair require two main components to be configured and set up. One component is the APR tool itself, with all dependencies available during runtime. The second component is the benchmark which provides information on the bug that needs to be repaired and the mechanism to reproduce the bug in a new environment. CERBERUS abstracts the nuances of different experiments and alleviates the repetitive, tedious efforts required to set up an experiment by providing a single interface to the user.

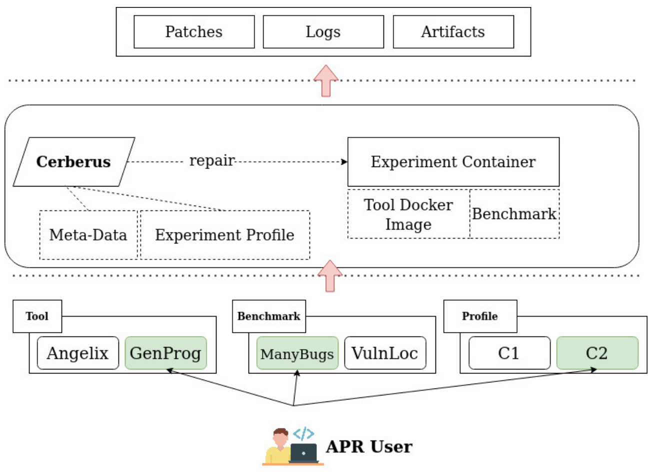

Fig. 1. Repair workflow of CERBERUS

Figure 1 illustrates the workflow for CERBERUS. The user selects a repair tool and a repair benchmark from a pre-configured list. Additionally, we provide experiment profiles that can be configured to control the experiment for different parameters (i.e., time duration for repair, number of test cases provided, etc.). Once the user makes the selection, CERBERUS will extract the relevant meta-data of the bug(s) that needs to be repaired. First, it will load the Docker image for the repair tool as the baseline image and extend the container by setting up the benchmark. Once the container is instantiated, CERBERUS will load the experiment profile to adjust the necessary parameters and invoke the repair module inside the container. Finally, CERBERUS extracts necessary artifacts, e.g., generated patches, debug logs, and other artifacts like repair constraints and additional generated test cases.

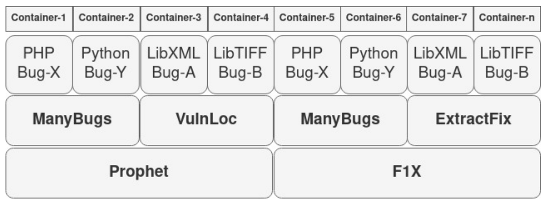

Fig. 2. Layered architecture for containers in CERBERUS

Figure 2 depicts the layered architecture CERBERUS follows to create containers for each experiment. As indicated previously, the baseline image is the repair tool that encapsulates all necessary dependencies to run the repair tool. CERBERUS would then extend the container by setting up the benchmark subject. For each experiment, a container will be spawned and can be used for a controlled experiment. This layered architecture provides several advantages, such as efficient space management, low latency for repeated experiments, and the re-usability of shared layers. Docker uses a union file system that uses a copy-on-write strategy to provide efficient space management. This means only the files modified by the write-layer (the top-most layer) are changed in the container. This strategy allows sharing of common files across different containers. Additionally, since Docker provides in-built caching of the layers, the consecutive re-run of experiments can be executed without rebuilding the complete container. For instance, consider a subject in the MANYBUGS benchmark (e.g., PHP-Bug-X) that we will run for one hour on PROPHET. Re-running the same experiment with a 2-hour timeout would be easily instantiated with the minor change to the repair profile, saving time and space for the new experiment.

A. *Extendibility*

CERBERUS provides necessary abstractions for tool developers and benchmark providers to easily integrate and extend the platform. To integrate a new repair tool, the tool developer needs to create a new driver for the tool. To invoke the repair process, the driver should provide the basic functionalities for CERBERUS. In particular, the driver should transform the meta-data provided by a benchmark into the expected form by the repair tool (i.e., creating configuration files). Once a repair driver is configured, CERBERUS can create the experiment container and execute the experiment with the configuration parameters defined in the repair profile and the information provided by the selected benchmark.

Integrating a new benchmark requires a separate driver and a schema file specifying the necessary meta-data for the defects in the benchmark. Different benchmarks provide different sets of information based on the complexity of the defects and the artifacts required for repair (i.e., test

TABLE I
DETAILS OF THE PROGRAM REPAIR TOOLS INTEGRATED WITH CERBERUS

#	Tool	Language	Methodology	Target Defect Type
1	Angelix	C/C++	Semantic	Test Failure
2	Prophet	C/C++	Learning	Test Failure
3	Darjeeling	C/C++	Search	Test Failure
4	CPR	C/C++	Semantic	Test Failure
5	VulnFix	C/C++	Semantic	Security Vulnerabilities
6	F1X	C/C++	Search	Test Failure
7	Fix2Fit	C/C++	Search	Test Failure
8	SenX	C/C++	Search	Security Vulnerabilities
9	GenProg	C/C++	Search	Test Failure
10	ExtractFix	C/C++	Semantic	Security Vulnerabilities
11	Verifix	C	Search	Student Assignments
12	Hippodrome	Java	Search	Concurrency Bugs
13	SequenceR	Java	Learning	Test Failure
14	ARJA	Java	Search	Test Failure
15	Cardumen	Java	Search	Test Failure
16	jMutRepair	Java	Search	Test Failure
17	jKali	Java	Search	Test Failure
18	jGenProg	Java	Search	Test Failure
19	Nopol	Java	Semantic	Test Failure
20	Recorder	Java	Learning	Test Failure

cases). For instance, VULNLOC only provides one failing test case since the benchmark is for vulnerability repair, while MANYBUGS includes a list of passing and failing test cases for each bug. The driver should provide the functionality to (configure/build/test/validate) the defects in the benchmark, capturing different stages in the repair process.

We provide extensive material with documentation, tutorials, and examples in our repository[1] to support the integration of new repair tools and new defect benchmarks.

III. IMPLEMENTATION

We have implemented CERBERUS with 20 program repair tools and 9 repair benchmarks consisting of real-world applications and student assignments. The repair tools represent different repair methodologies, including learning-based, semantic-based, and search-based techniques. In order to support containerized learning-based repair tools like RECORDER [11], we added support for the Nvidia Docker runtime [13], which can be activated through a command line argument. Table I details the program repair tools integrated with CERBERUS representing each methodology.

The benchmark programs consist of different classes of repair tasks, including but not limited to fixing concurrent bugs, generating feedback for student assignments, and repairing test suite failures. Table II describes the details of the benchmark programs integrated with CERBERUS. MANYBUGS [14], QUIXBUGS [15], BEARS [16], INTROCLASSJAVA [17] and DEFECTS4J [18] are benchmarks consisting of functionality errors reported as test case failures for C/C++ and Java programs. EXTRACTFIX [19] and VULNLOC [20] are benchmarks for C/C++ programs capturing a security vulnerability with a proof of concept exploit. ITSP [21] is a benchmark consisting of incorrect solutions for student assignments with a correct solution provided as a reference. HIPPODROME [22] is a benchmark for concurrency bugs, which does not include any test case.

[1]https://github.com/nus-apr/cerberus/blob/main/doc/Extending.md

TABLE II
DETAILS OF THE BENCHMARK SUBJECTS INTEGRATED WITH CERBERUS

#	Benchmark	Language	Type	# Projects	# Bugs
1	ManyBugs	C/C++	Test Failure	6	60
2	VulnLoc	C/C++	Vulnerabilities	11	43
3	ExtractFix	C/C++	Vulnerabilities	7	30
4	ITSP	C	Student Assignments	10	661
5	Hippodrome	Java	Concurrency Bugs	16	25
6	Defects4J	Java	Test Failure	17	835
7	QuixBugs	Java	Test Failure	40	40
8	Bears	Java	Test Failure	72	251
9	IntroClassJava	Java	Test Failure	6	297
Total					2242

IV. EVALUATION

We demonstrate the capabilities of CERBERUS by analyzing the performance improvement introduced for repair tasks and reproducing reported experimented values on literature. First, we analyze the performance improvement gained by using CERBERUS with respect to space and time to set up. For this purpose, we selected the VULNLOC benchmark [20], which consists of real-world applications with a single failing test. Table III shows the comparison of running F1X [23] and VULNFIX [24] on `Binutils` in the VULNLOC benchmark. Columns s and t indicate the space consumed by preparing the subject for repair and the time duration for setup, respectively.

TABLE III
ANALYSIS ON TIME AND SPACE IMPROVEMENT

Bug-ID	Original				Cerberus			
	F1X		VULNFIX		F1X		VULNFIX	
	s	t	s	t	s	t	s	t
CVE-2017-14745	849MB	23s	1.4GB	46s	849MB	27s	597MB	21s
CVE-2017-15020	840MB	30s	1.1GB	37s	840MB	23s	252MB	17s
CVE-2017-15025	851MB	27s	1.1GB	37s	851MB	24s	252MB	17s
CVE-2017-6965	840MB	29s	1.4GB	44s	840MB	24s	582MB	21s

Although both F1X and VULNFIX are repairing the same subjects, they both require different methods for preparing the subjects as observed in the space difference in Table III. This is because F1X does not require the binary files to be built; however, VULNFIX expects the binary executable to be provided as input to the repair. In a traditional environment where the user runs both VULNFIX and F1X in parallel, they would need to keep two copies of the setup, one for each tool. However, using the layered architecture in CERBERUS, we only need one copy of the source code, which saves significant space, as shown in Table III. For example, `CVE-2017-14745` setup would require 849MB amount of space, on top of which F1X will run the repair. VULNFIX for the same defect would require an additional 597MB space totaling to 1.4GB, before attempting to repair. In the native setup, the two experiments using F1X and VULNFIX would require 2.3GB. For CERBERUS, the total space required is 1.4GB saving 597MB for the two experiments. Similarly, using caching to re-use the previous setup CERBERUS saves time for consecutive repair on the same defect. Note that the space and time saving reported are for two consecutive repairs. For each additional repair, the savings would be a factorial of the initial saving.

Next, we successfully reproduce experimental results for two selected repair tasks. To demonstrate the diversity of repair tools CERBERUS can cater, we select two repair tasks. In contrast to traditional test failure repair, we select fixing student assignments. For this purpose, we executed the education repair tool VERIFIX [9] using CERBERUS and were able to generate similar results as reported in [9]. Table IV shows the results for the latest version of the tool VERIFIX. The repair percentages observed in our experiments are slightly better since we are using the latest version of the tool, which the authors have improved since the publication.

TABLE IV
EXPERIMENTAL RESULTS FOR VERIFIX IN ITSP BENCHMARK

Lab-ID	# Assignments	# Programs	Repair %	Average Time (s)
Lab-3	4	63	91.66	9.95
Lab-4	8	117	82.24	15.80
Lab-5	8	82	71.95	3.03
Lab-6	8	79	49.36	6.72

We also demonstrate that CERBERUS can support learning-based repair tools. Especially since learning-based repair tools require different infrastructures to run the repair task - for example, having a dedicated GPU capable of running machine learning frameworks that utilize the CUDA API. For this purpose, we selected SEQUENCER and RECORDER, popular learning-based tools, and reproduced the results reported in [10] and some of the experiments in [11]. Table V shows the results of SEQUENCER and RECORDER on a subset of DEFECTS4J benchmark. We are able to reproduce identical results in terms of candidate patches, compilable patches, plausible patches, and correct patches as reported in [10]. For [11], we examined less experiments and observed deviations from the reported numbers.

TABLE V
EXPERIMENTAL RESULTS FOR LEARNING-BASED TOOLS IN DEFECTS4J BENCHMARK

Tool	Metric	Observed	Reported
SEQUENCER	# Bugs candidate patches generated	57	57
	# Bugs compilable patches generated	52	52
	# Bugs plausible patches generated	19	19
	# Bugs correct patches generated	14	14
RECORDER	# Bugs candidate patches generated	25	25
	# Bugs compilable patches generated	25	25
	# Bugs plausible patches generated	19	25
	# Bugs correct patches generated	14	25

V. RELATED WORK

Several works have been proposed in the literature to address the gap of a standard platform for empirical evaluation in automated program repair. MAESTRO [7] is a recently proposed platform to evaluate automated program repair tools across different benchmarks with a low overhead to the user. The platform is designed to work as micro-service containers communicating via RESTful APIs to perform repair tasks for different benchmark programs. The proposed decentralized approach creates separate micro-service containers, each for the benchmark program and for the program repair tool itself. In contrast, CERBERUS creates a single container encapsulating the benchmark program and the repair tool. This design choice allows CERBERUS to generate on-the-fly Docker containers customized for a repair tool and a benchmark program pair. However, a decentralized approach as proposed in MAESTRO [7] is difficult to extend with repair tools that require a large dependent toolchain. For example, semantic-based repair tools such as CPR [8] require LLVM build infrastructure with KLEE [25] runtime support. Therefore, separating the benchmark program and the repair tool prevents running repair using semantic-based repair tools.

Similar to our approach, REPAIRTHEMALL [5] proposed a framework to evaluate multiple program repair tools across multiple sets of benchmarks. REPAIRTHEMALL implements a monolithic architecture targeted for Java. SECURETHEMALL [6] follows a similar design but focuses on security vulnerabilities. SECURETHEMALL is implemented for C programs and specifically designed to cater only security vulnerabilities. In contrast, CERBERUS is not restricted to a specific language or a class of defects. We have shown that CERBERUS is capable of catering to multiple programming languages, multiple classes of defects, and multiple types of repair techniques (i.e., learning-based, semantic-based, and search-based).

VI. CONCLUSION

We presented a platform that is capable of executing a diverse set of program repair tasks using a multitude of program repair tools and benchmarks. We implemented our solution in CERBERUS and demonstrated the capability to reproduce previously reported results in the literature. Our experiments also showed there is a significant cost saving in terms of setup time. Our vision is that CERBERUS will be extended to be used as a repair service constituting a variety of repair capabilities. As future work CERBERUS will be integrated with testing and analysis tools such as fuzzers, static analyzers, and symbolic execution engines. This will combine bug detection and repair in a single framework.

CERBERUS is open-source and available for use via:
https://github.com/nus-apr/cerberus

ACKNOWLEDGMENT

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

REFERENCES

[1] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, nov 2019. [Online]. Available: https://doi.org/10.1145/3318162

[2] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 269–278.

[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*, vol. 38, no. 4, pp. 43–51, 2021.

[4] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3105906

[5] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 302–313. [Online]. Available: https://doi.org/10.1145/3338906.3338911

[6] E. Pinconschi, R. Abreu, and P. Adão, "A comparative study of automatic program repair techniques for security vulnerabilities," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 196–207.

[7] E. Pinconschi, Q.-C. Bui, R. Abreu, P. Adão, and R. Scandariato, "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*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 789–792. [Online]. Available: https://doi.org/10.1145/3533767.3543291

[8] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 390–405. [Online]. Available: https://doi.org/10.1145/3453483.3454051

[9] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, "Verifix: Verified repair of programming assignments," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, jul 2022. [Online]. Available: https://doi.org/10.1145/3510418

[10] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[11] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[12] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 539–554.

[13] Nvidia, "Nvidia/nvidia-docker: Build and run docker containers leveraging nvidia gpus." [Online]. Available: https://github.com/NVIDIA/nvidia-docker

[14] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[15] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–56. [Online]. Available: https://doi.org/10.1145/3135932.3135941

[16] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An extensible java bug benchmark for automatic program repair studies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 468–478.

[17] T. Durieux and M. Monperrus, "IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs," Universite Lille 1, Tech. Rep. hal-01272126, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01272126/document

[18] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[19] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, feb 2021. [Online]. Available: https://doi.org/10.1145/3418461

[20] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 537–549. [Online]. Available: https://doi.org/10.1145/3433210.3437528

[21] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 740–751. [Online]. Available: https://doi.org/10.1145/3106237.3106262

[22] A. Costea, A. Tiwari, S. Chianasta, K. R, A. Roychoudhury, and I. Sergey, "Hippodrome: Data race repair using static analysis summaries," *ACM Trans. Softw. Eng. Methodol.*, jul 2022. [Online]. Available: https://doi.org/10.1145/3546942

[23] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, oct 2018. [Online]. Available: https://doi.org/10.1145/3241980

[24] Y. Zhang, X. Gao, G. J. Duck, and A. Roychoudhury, "Program vulnerability repair via inductive inference," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 691–702. [Online]. Available: https://doi.org/10.1145/3533767.3534387

[25] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX OSDI*, 2008, p. 209–224.