SWAT: Modular Dynamic Symbolic Execution for Java Applications using Dynamic Instrumentation (Competition Contribution)

Nils Loose(✉)*, Felix Mächtle, Florian Sieck, and Thomas Eisenbarth

Institute for IT Security, University of Lübeck, Lübeck, Germany
{n.loose,f.maechtle,florian.sieck,thomas.eisenbarth}@uni-luebeck.de

Abstract. SWAT is a novel dynamic symbolic execution engine for Java applications utilizing dynamic instrumentation. SWAT's unique modular design facilitates flexible communication between its symbolic explorer and executor using HTTP endpoints, thus enhancing adaptability to diverse application scenarios. The symbolic executor's ability to attach to Java applications enables efficient constraint generation and path exploration. SWAT employs JavaSMT for constraint generation and ASM for bytecode instrumentation, ensuring robust performance. SWAT's efficacy is evaluated in the Java Track of SV-COMP 2024, achieving fourth place.

Keywords: Dynamic Symbolic Execution · Java · Dynamic Instrumentation

1 Verification Approach

The symbolic execution of a System-under-Test (SuT) is a well-known verification technique where the state space is systematically explored by using constraint modeling to compute new valid inputs for the SuT. Dynamic Symbolic Execution (DSE), in particular, has shown recent successes with JDart [15] winning the Java track of SV-COMP 2022 [4] as the first DSE tool and GDart [16] achieving second place in 2023 [5]. Generally, DSE utilizes a symbolic executor to evaluate a SuT by observing the concrete execution for a given assignment of the symbolic variables. Constraints are recorded during execution, reflecting all operations involving symbolic variables. In particular, each branching point that depends on a symbolic variable is modeled as a path constraint. After the execution terminates, the symbolic explorer can select a previously unexplored branch. Given the recorded constraints, an SMT solver is used to determine whether a model for the symbolic variables under the given constraints exists. If so, a concrete instantiation for each value can be obtained to drive execution to previously unexplored regions of the state space. By repeating this process, the state space of the SuT can be systematically explored.

JDart, the winning candidate from 2022, relies on Java Pathfinder (JPF) [9] and its implementation of the Java Virtual Machine (JVM) for symbolic

* Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 399–405, 2024.
https://doi.org/10.1007/978-3-031-57256-2_28

execution [15]. While the JPF-JVM offers robust analysis tools, it limits JDart's applicability and causes a significant overhead. Coastal [8], on the other hand, relies on a standard JVM. The symbolic execution is realized using dynamic instrumentation. While Coastal provides a loosely coupled design between the symbolic execution engine and the symbolic explorer, both components are still located in the same Java program used as the driver to start and execute the SuT symbolically. GDart extends the notion of modularity introduced by Coastal with a fully decoupled explorer and executor that communicate using a custom protocol [16]. SWAT offers a fully modular design comparable to GDart while relying on HTTP endpoints for communication between the symbolic explorer and executor. In addition, GDart relies on the GraalVM [20] for driving symbolic execution while SWAT attaches to the SuT, thus enabling symbolic execution inside native JVM implementations.

2 System Architecture

SWAT's decoupled design allows for a persistent symbolic explorer that receives relevant information from instances of the symbolic executor. The executor observes the SuT by attaching to the JVM and adding symbolic capabilities using dynamic instrumentation. An overview of the design and interaction between the different components is shown in Figure 1 and described in more detail below.

Symbolic executor The executor attaches to the JVM running the SuT via the Java agent interface and dynamically instruments each class at load time with additional (non-interfering) instructions that dynamically build and manage a symbolic shadow state responsible for maintaining the symbolic constraints. This leads to a symbolic executor that does not actively drive symbolic execution and instead records relevant information during normal execution. SWAT utilizes the ASM framework [6] for bytecode manipulation via the `Java.lang.instrument` API [17]. Historically, this part builds on CATG [19] as a basis for dynamic symbolic execution. Significant parts of CATG are reworked, and the language support is lifted to Java 17, including most of its features. The symbolic shadow state and the symbolic constraint handling are extended and wholly rewritten to utilize the API offered by JavaSMT [1] as an abstraction layer between constraint generation and the solver. The symbolic scope and variables, as well as the entry and exit points for symbolic tracking, are fully configurable, allowing for broad applicability of the system. The instrumentation logic is also modularized, allowing us to easily extend SWAT to various use cases, such as the SV-COMP.

When the execution of the SuT reaches a symbolic entry point, the symbolic executor records control-flow information as well as the constraints, and after the exit point has been reached, both the trace and the corresponding constraints are sent to the symbolic explorer using HTTP requests. Constraints are serialized using the SMT-LIB v2 [3] format.

Symbolic explorer The explorer, written in Python using the FastAPI [18] web framework, receives the language agnostic trace and constraint information. These are stored in a binary execution tree. The tree can be searched using a

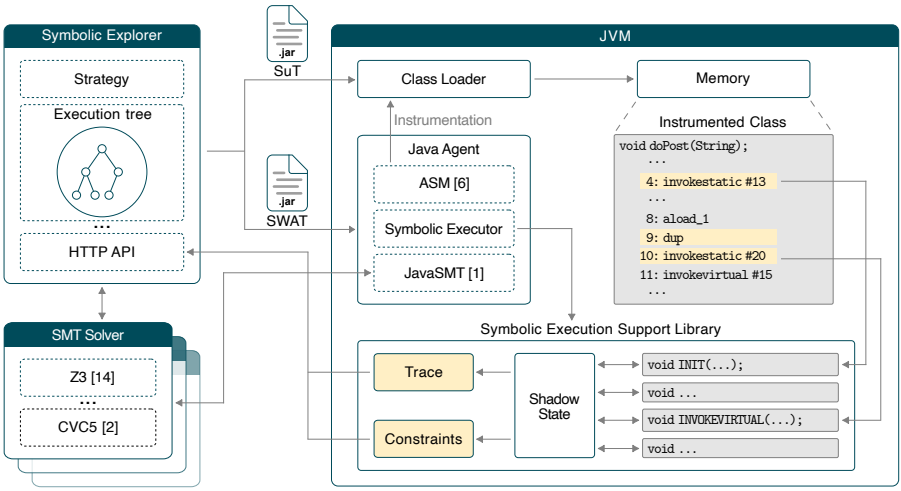
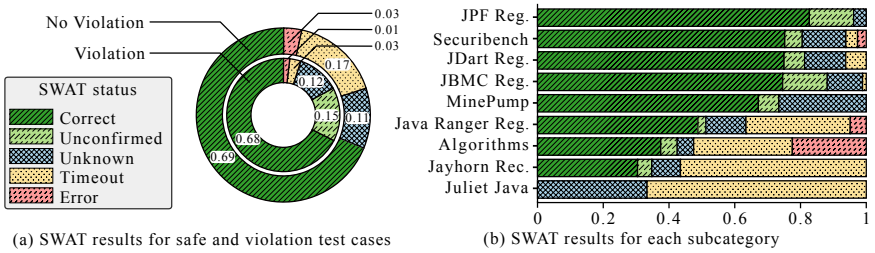


Fig. 1. Schematic overview of SWAT's modular architecture.

configurable and modularized strategy to select unexplored branches. To obtain new inputs, the constraints are sent to Z3 [14]. The inputs can either be made available to external drivers, such as fuzzers, using an endpoint or, in the case of SV-COMP, are directly used to initiate a new concrete execution. This structure makes SWAT widely applicable and even enables straightforward testing of web services, for example, where each controller is configured as the entry and exit point and user-controlled values are tracked symbolically. This allows the same JVM to keep running in between symbolic runs and even allows for multiple (non-interfering) executions in parallel.

3 Evaluation

In the first participation on the Java category of SV-COMP 2024, SWAT reached fourth place with 566 out of 828 total points while MLB [7], the winning candidate, scored 676 points. Overall, SWAT correctly classified 68% of test cases. Figure 2a visualizes the result distribution for test cases containing violations and those without. The number of correctly classified cases is similar for both groups. However, due to issues during witness generation, several correctly identified violations did not produce correct witnesses. Hence, without considering the witnesses, the number of identified violations rises significantly from 68% to 83%. Generally, DSE frameworks are expected to identify violations (one concrete path) better than proving their absence (full state-space exploration). This is also reflected in the distribution of timeouts, with a five times increase between violation and safe test cases. Roughly 10% of test cases are labeled as unknown by SWAT. This case comprises several possibilities: Out-of-scope invocations



(a) SWAT results for safe and violation test cases

(b) SWAT results for each subcategory

Fig. 2. SWAT results divided based on the ground truth of each test case (a) and results for each subcategory of the Java category (b).

without a symbolic model, inability to determine satisfiability or unsupported behavior such as uncaught exceptions.

Further dividing the results based on the different subgroups (see Figure 2b) highlights differences in the status distributions. SWAT generally performs well for regression test categories, as these usually test specific functionalities, resulting in small programs that do not lead to a state space explosion. With the increasing complexity of test suites, the number of timeouts is expected to rise. The Jayhorn recursive test cases cause many timeouts as SWAT currently does not support advanced recursion handling. Lastly, SWAT is holistically unable to solve the test cases provided by the Juliet test suite due to the extensive use of socket connections, which require explicit mocking.

While the results demonstrate the impact of state space explosion on the performance of DSE engines, generally, the results highlight the potential of SWAT, especially when considering the overhead incurred by starting a new JVM instance for each run of the test case. In SWAT's current form, this causes instrumentation at each iteration whereas test cases that can be re-initiated without restarting the JVM would result in significantly faster executions.

4 Software Project

SWAT is developed by the Institute for IT Security at the University of Lübeck and published on GitHub [12] under the BSD 2-Clause. Installation instructions, documentation, and examples can be found on our GitHub Page [11]. Global configuration options chosen for the participation include the exclusive usage of the Z3 [14] solver, a breadth-first search strategy, and an SV-COMP specific driver modules inside the symbolic explorer and executor.

Data-Availability Statement The version of SWAT used for the SV-COMP 2024 Java category is available at Zenodo [13] and on GitHub [10].

5 Acknowledgments

This work has been supported by the Bundesministerium für Bildung und Forschung (BMBF) through the PeT-HMR project.

References

1. Baier, D., Beyer, D., Friedberger, K.: Javasmt 3: Interacting with SMT solvers in java. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 195–208. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_9, https://doi.org/10.1007/978-3-030-81688-9_9
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)
4. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20, https://doi.org/10.1007/978-3-030-99527-0_20
5. Beyer, D.: Competition on software verification and witness validation: Sv-comp 2023. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 495–522. Springer Nature Switzerland, Cham (2023)
6. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems **30**(19) (2002)
7. Bu, L., Liang, Y., Xie, Z., Qian, H., Hu, Y., Yu, Y., Chen, X., Li, X.: Machine learning steered symbolic execution framework for complex software code. Formal Aspects Comput. **33**(3), 301–323 (2021). https://doi.org/10.1007/S00165-021-00538-3, https://doi.org/10.1007/s00165-021-00538-3
8. Geldenhuys, J., Visser, W.: Coastal. https://github.com/DeepseaPlatform/coastal, accessed 12/2023
9. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA pathfinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000). https://doi.org/10.1007/S100090050043, https://doi.org/10.1007/s100090050043
10. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT Competition Version. https://github.com/SWAT-project/SWAT/tree/SV-COMP-Submission-2024, accessed 12/2023
11. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT Documentation. https://swat-project.github.io/docs/, accessed 12/2023

12. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT Repository. https://github.com/swat-project/swat, accessed 12/2023
13. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: Swat (2023). https://doi.org/10.5281/zenodo.10418643, https://doi.org/10.5281/zenodo.10418643
14. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
15. Mues, M., Howar, F.: Jdart: Dynamic symbolic execution for java bytecode (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 398–402. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_28, https://doi.org/10.1007/978-3-030-45237-7_28
16. Mues, M., Howar, F.: Gdart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 435–439. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_27, https://doi.org/10.1007/978-3-030-99527-0_27
17. Oracle: Java Instrumentation. https://docs.oracle.com/en/java/javase/17/docs/api/java.instrument/java/lang/instrument/package-summary.html, accessed 12/2023
18. Ramírez, S.: FastAPI, https://github.com/tiangolo/fastapi, accessed 12/2023
19. Tanno, H., Zhang, X., Hoshino, T., Sen, K.: Tesma and CATG: Automated test generation tools for models of enterprise applications. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. pp. 717–720. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.231, https://doi.org/10.1109/ICSE.2015.231
20. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Hosking, A.L., Eugster, P.T., Hirschfeld, R. (eds.) ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013. pp. 187–204. ACM (2013). https://doi.org/10.1145/2509578.2509581, https://doi.org/10.1145/2509578.2509581