JLiSA: The Java Frontend of the Library for Static Analysis (Competition Contribution)

Vincenzo Arceri[1*], Luca Negrini[2*], Giacomo Zanatta[2*†], Filippo Bianchi[1‡], Teodors Lisovenko[2‡], Luca Olivieri[2‡], and Pietro Ferrara[2]

[1] University of Parma, Italy
vincenzo.arceri@unipr.it, filippo.bianchi5@studenti.unipr.it
[2] Ca' Foscari University of Venice, Italy
{luca.negrini, giacomo.zanatta, teodors.lisovenko, luca.olivieri, pietro.ferrara}@unive.it

Abstract. JLiSA is the extension to the Java programming language of LiSA, an analysis engine that works on a generic and extensible control flow graph representation of the program to analyze. LiSA implements several standard abstract domains and analyses aimed at approximating numerical values, strings, and heap structures. At the end of the analysis, it produces an abstract state for each program point. Then, checkers produce warnings indicating whether a property of interest is respected. JLiSA provides a front-end to translate Java programs into the internal LiSA control flow graph representation, the semantics of various parts of the Java standard library, and checkers to verify assertions and detect whether exceptions might be thrown and not caught. This paper presents our first participation in SV-COMP in the Java category, where we achieved 3rd place.

1 Verification Approach

JLiSA relies on LiSA [9,13,11] (Library for Static Analysis), a Java library that provides a complete infrastructure for the development of static analyzers based on abstract interpretation [6,7]. In particular, LiSA implements several standard components of abstract interpretation-based analyzers, including an extensible control flow graph (CFG) representation, a standard analysis framework for developing new static analyses, and fixpoint algorithms on LiSA CFGs. LiSA has been extended and applied to a large variety of properties, programming languages and contexts: Go smart contracts and blockchain software [21,22,17,19,18,16,14], Python robotic software [25,24], microservices [26], Michelson [20,15] and EVM bytecode [3,2], as well as for teaching purposes [12]. JLiSA builds on top of LiSA to analyze Java programs. In particular, it consists

[*] Equal contribution
[†] Jury member
[‡] Equal contribution

of a front-end that translates Java programs into LiSA's internal control flow graph representation. This representation is enriched with the semantics of a subset of the Java standard library, corresponding to the classes and methods exercised by the AutoStub category of the SV-COMP benchmarks[3]. Then, the parsed code is passed to LiSA, along with Java-specific semantic algorithms (e.g., class hierarchy traversal, call target matching, ...) and the analysis configuration. The latter includes, among others, the list of abstract domains to execute and the property checkers that will issue warnings. These can be either existing components provided by LiSA or custom ones defined in JLiSA. Regarding SV-COMP 2026 [4], JLiSA was configured with the following settings:

- *Analysis Settings*. We configured the interprocedural analysis, which computes a program-wide fixpoint, to start in the `main` methods and use a call-string-based approach for managing contexts at call sites [23], with a maximum depth of 150 nested calls. After evaluation, we found that this depth is well-suited for SV-COMP benchmarks, providing a good trade-off between precision and performance. For call chains longer than 150, the analysis resorts to an over-approximation (the top abstract state). Since call-string analysis alone does not resolve Java's virtual method targets, 0-CFA [10] was used to construct the call graph.
- *State Settings*. The state of the analysis was structured according to Ferrara et al. [8], employing: (i) a field-sensitive program-point based heap abstraction [1] to track objects and their fields, (ii) a reduced product between a constant propagation analysis (tracking numerical, string, and Boolean values) and the interval domain [6,7], for abstracting values, and (iii) type inference for considering the runtime types of expressions.
  The state was further wrapped into a *reachability* analysis, that distinguishes definitely reachable instructions from possibly reachable or unreachable ones relying on the traversed conditions; for instance, the true branch of an if-then-else statement is definitely reachable only if (i) the condition is reachable, and (ii) the condition is always satisfied — if the latter is instead sometimes satisfied, the branch is considered possibly reachable.

After the analysis, two property checkers were executed, one for runtime exceptions and one for failing asserts.

1.1 Tool pipeline and Software Architecture

The BenchExec framework [5] orchestrates the execution of our tool within the SV-COMP benchmarking infrastructure. A dedicated CLI tool preprocesses the Java benchmarks by translating property specifications and configuration files into JLiSA inputs. This preprocessing step sets the *checks options*, such as runtime safety and assertion violation checks, and other *configuration options*, according to the properties to be verified and the Java source files under analysis.

[3] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp26/java/autostub`

At the end of execution, JLiSA frontend may return an analysis result, containing warnings. The CLI tool then collects and post-processes the analysis results. Specifically, it uses the property file to determine which warnings to extract and process from JLiSA's analysis output and the selected warnings are used to produce a final report in the format required by SV-COMP. In particular, there are two main generic families of warnings (possible or definite). In addition, warnings about assertions report whether an assertion holds or does not. JLiSA reports that a test case is verified if either no warning is produced, or only definite warnings about assertions that hold are produced. Instead, it reports that a test case is not verified if only definite warnings about thrown exceptions or assertions that do not hold are produced. In all other cases (e.g., assertions that may or may not hold), JLiSA returns unknown to avoid false positives and false negatives. LiSA and JLiSA are both developed in Java (version 11 and 17, respectively), and rely on several standard libraries (such as log4j, JUnit, and reflections). JLiSA parses Java code through the Eclipse Java Development Tools. Instead, the CLI tool is written in Python and relies on PyYAML to read YAML files, and Typer to build the CLI application.

1.2 JLiSA Architecture

Figure 1 draws the overall architecture and an in-depth overview of JLiSA frontend components.

As shown in Figure 1, JLiSA (i) transforms the Java source code into its internal representation (middle left), (ii) performs a fixpoint computation of the semantics over the domains specified in the configuration (center), and (iii) produces an abstract state for each program point. All the standard components of the analysis rely on the LiSA library (grey background), while other components have been extended and customized in JLiSA (white background).

$\top$ and $\bot$ are special elements, representing *any value* and *no possible value*, respectively. Instead, CP, INT, R, DR denote *constant propagation* and *interval* abstract domains, *reachability*, and *definitely reachable*, respectively.

After the analysis, two semantic checks are executed to produce the final report (bottom right). The first verifies the validity of assert statements by reasoning about (i) their reachability and (ii) the satisfiability of their conditions. The second identifies statements that may throw runtime exceptions and determines whether such exceptions are properly caught. The two checks are always executed simultaneously, and JLiSA is unaware of which property is under investigation; therefore, both kinds of warnings may appear in its output. It is the CLI tool that uses the property file to determine which warnings to extract and process from JLiSA's analysis output.

2 Strengths and Weaknesses

The main strength and weakness of JLiSA are in its foundation in abstract interpretation theory. We strongly believe in the need for (coarse) approximation

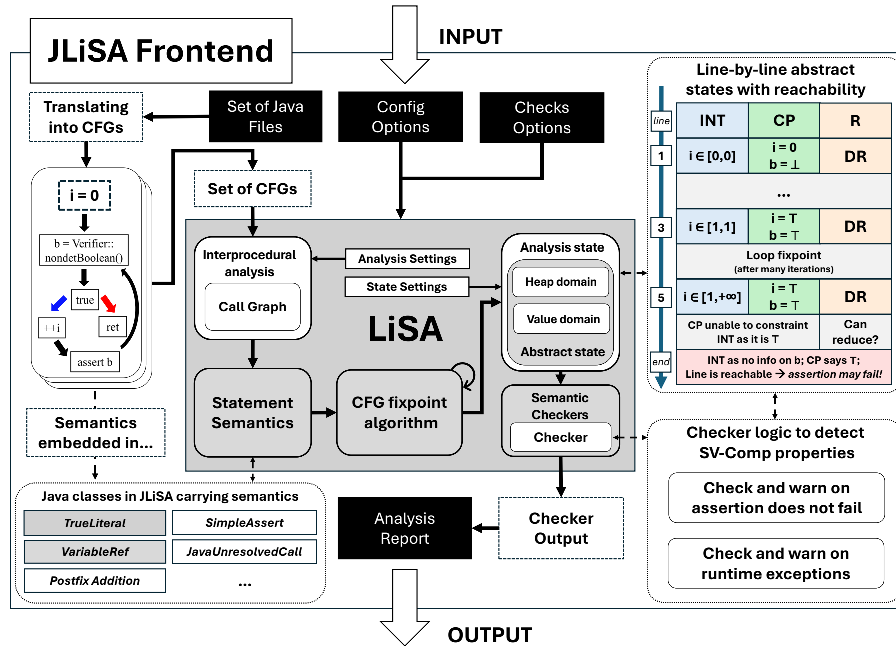

Fig. 1: JLiSA architecture

in order to make sound static analysis feasible. In practice, this means that (i) we can develop a wide variety of abstractions, modifying them based on our needs (strength), and (ii) we might produce false alarms (weakness). Point (i) gave us a strong advantage over other tools, since we could quickly adapt and modify the analyses we apply to the SV-COMP benchmark, especially when test cases were added (which was quite the case this year in the Java category). Point (ii) penalizes us since false alarms in SV-COMP are highly penalized. Generally speaking, a precision of 90% (that is, a pretty good precision for any generic abstract interpretation-based static analyzer) would be an excellent result, but this would end up in a negative score at SV-COMP (as a single false alarm incurs a -16 penalty versus a $+2$ benefit for a true alarm). To mitigate this weakness, we carefully post-processed the alarms generated by our analysis, as described in Section 1.2, and reported an alarms only when we were fully confident that it was a true positive.

The scores obtained by JLiSA at SVCOMP properly reflect our design choices. In particular, more than 95% of the points were awarded for correct true reports. Since the rules we applied to raise warnings were quite strict to avoid incorrect false reports, JLiSA did not produce any, thanks also to our conservative approach detailed in Section 1 for handling unproven cases. In addition, since JLiSA is sound, no incorrect true report was produced.

Data Availability Statement. The Unix binary of JLiSA that participated in SV-COMP 2026 is available at `https://doi.org/10.5281/zenodo.17609338`. The source codes are publicly available at `https://github.com/lisa-analyzer/jlisa` and `https://github.com/lisa-analyzer/lisa` for JLiSA and LiSA, repectively. JLiSA Gradle's configuration defines its dependency on specific LiSA versions. Both projects are released with an MIT license. The competition artifact was generated from commit ID ce67adb (tag svcomp-2026 of the JLiSA repository).[4] The JLiSA executable can be obtained either by downloading the competition archive or by building it from source as described in the README[5].

Acknowledgments. Work partially supported by SERICS (PE00000014 - CUP H73C2200089001), iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU; and by the EU—NGEU National Sustainable Mobility Center (CN00000023), Italian Ministry of University and Research Decree n. 1033—17/06/2022 (Spoke 10), Regione Veneto: RIR project SATCO (CUP D19J24000750007), RIR project SUPREME (CUP D19J24000810007), and by Bando di Ateneo 2024 per la Ricerca, funded by University of Parma (FIL_2024_PROGETTI_B_IOTTI - CUP D93C24001250005).

This work has benefited from the participation of the first and last authors, as well as the SV-COMP competition chair Dirk Beyer, in the Dagstuhl Seminar 25421 "Sound Static Program Analysis in Modern Software Engineering".

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Phd thesis, University of Copenhagen, DIKU (1994)
2. Arceri, V., Merenda, S.M., Dolcetti, G., Negrini, L., Olivieri, L., Zaffanella, E.: Towards a sound construction of evm bytecode control-flow graphs. In: Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs. p. 11–16. FTfJP 2024, Association for Computing Machinery, New York, NY, USA (2024). `https://doi.org/10.1145/3678721.3686227`
3. Arceri, V., Merenda, S.M., Negrini, L., Olivieri, L., Zaffanella, E.: Evmlisa: Sound static control-flow graph construction for evm bytecode. Blockchain: Research and Applications p. 100384 (2025). `https://doi.org/https://doi.org/10.1016/j.bcra.2025.100384`
4. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
5. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer **21**(1), 1–29 (Feb 2019). `https://doi.org/10.1007/s10009-017-0469-y`
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings

[4] In the Zenodo artifact, we explicitly added the commit ID to the JLiSA version (v0.2-ce67adbda) to distinguish it from other versions. Locally built versions may appear without the commit ID.

[5] `https://github.com/lisa-analyzer/sv-comp#build-sv-comp-2026-executable`

of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973

7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA , January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778
8. Ferrara, P.: A generic framework for heap and value analyses of object-oriented programming languages. Theoretical Computer Science **631**, 43–72 (Jun 2016). https://doi.org/10.1016/j.tcs.2016.04.001
9. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 1–6. SOAP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460946.3464316
10. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 108–124. OOPSLA '97, Association for Computing Machinery, New York, NY, USA (Oct 1997). https://doi.org/10.1145/263698.264352
11. Negrini, L.: A generic framework for multilanguage analysis. Phd thesis, Universitá Ca' Foscari Venezia (2023)
12. Negrini, L., Arceri, V., Olivieri, L., Cortesi, A., Ferrara, P.: Teaching through practice: Advanced static analysis with lisa. In: Sekerinski, E., Ribeiro, L. (eds.) Formal Methods Teaching - 6th Formal Methods Teaching Workshop, FMTea 2024, Milan, Italy, September 10, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14939, pp. 43–57. Springer (2024). https://doi.org/10.1007/978-3-031-71379-8_3
13. Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: Lisa: A generic framework for multilanguage static analysis. In: Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (eds.) Challenges of Software Verification, Intelligent Systems Reference Library, vol. 238, pp. 19–42. Springer (2023). https://doi.org/10.1007/978-981-19-9601-6_2
14. Olivieri, L.: Detection of cross-channel invocation risks in hyperledger fabric. In: 2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE). pp. 107–118 (2025). https://doi.org/10.1109/ISSRE66568.2025.00023
15. Olivieri, L., Jensen, T., Negrini, L., Spoto, F.: Michelsonlisa: A static analyzer for tezos. In: 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). pp. 80–85 (2023). https://doi.org/10.1109/PerComWorkshops56833.2023.10150247
16. Olivieri, L., Negrini, L.: Don't panic: Error handling patterns in go smart contracts and blockchain software. In: 2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 1–9 (2025). https://doi.org/10.1109/BRAINS67003.2025.11302935
17. Olivieri, L., Negrini, L., Arceri, V., Chachar, B., Ferrara, P., Cortesi, A.: Detection of phantom reads in hyperledger fabric. IEEE Access **12**, 80687–80697 (2024). https://doi.org/10.1109/ACCESS.2024.3410019
18. Olivieri, L., Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Detection of read-write issues in hyperledger fabric smart contracts. In: Hong, J., Battiato, S., Esposito, C., Park, J.W., Przybylek, A. (eds.) Proceedings of the 40th ACM/SIGAPP

Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025. pp. 329–337. ACM (2025). https://doi.org/10.1145/3672608.3707721

19. Olivieri, L., Negrini, L., Arceri, V., Ferrara, P., Cortesi, A., Spoto, F.: Static detection of untrusted cross-contract invocations in go smart contracts. In: Hong, J., Battiato, S., Esposito, C., Park, J.W., Przybylek, A. (eds.) Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025. pp. 338–347. ACM (2025). https://doi.org/10.1145/3672608.3707728
20. Olivieri, L., Negrini, L., Arceri, V., Jensen, T.P., Spoto, F.: Design and implementation of static analyses for tezos smart contracts. Distributed Ledger Technol. Res. Pract. **4**(2), 13:1–13:23 (2025). https://doi.org/10.1145/3643567
21. Olivieri, L., Negrini, L., Arceri, V., Tagliaferro, F., Ferrara, P., Cortesi, A., Spoto, F.: Information flow analysis for detecting non-determinism in blockchain. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 23:1–23:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.23
22. Olivieri, L., Tagliaferro, F., Arceri, V., Ruaro, M., Negrini, L., Cortesi, A., Ferrara, P., Spoto, F., Talin, E.: Ensuring determinism in blockchain software with golisa: an industrial experience report. In: Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 23–29. SOAP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3520313.3534658
23. Pnueli, M., Sharir, M.: Two approaches to interprocedural data flow analysis. Program flow analysis: theory and applications pp. 189–234 (1981)
24. Zanatta, G., Caiazza, G., Ferrara, P., Negrini, L.: Inference of access policies through static analysis. Int. J. Softw. Tools Technol. Transf. **26**(6), 797–821 (2024). https://doi.org/10.1007/S10009-024-00777-8
25. Zanatta, G., Caiazza, G., Ferrara, P., Negrini, L., White, R.: Automating ROS2 security policies extraction through static analysis. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024, Abu Dhabi, United Arab Emirates, October 14-18, 2024. pp. 3627–3634. IEEE (2024). https://doi.org/10.1109/IROS58592.2024.10802507
26. Zanatta, G., Ferrara, P., Lisovenko, T., Negrini, L., Caiazza, G., White, R.: Sound static analysis for microservices: Utopia? A preliminary experience with lisa. In: Stefano, L.D. (ed.) Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2024, Vienna , Austria, 20 September 2024. pp. 5–10. ACM (2024). https://doi.org/10.1145/3678721.3686229