



# AISE: A Symbolic Verifier by Synergizing Abstract Interpretation and Symbolic Execution (Competition Contribution)

Zhen Wang <sup>1,2</sup> and Zhenbang Chen <sup>1,2\*</sup>

<sup>1</sup> College of Computer, National University of Defense Technology, Changsha, China

<sup>2</sup> State Key Laboratory of Complex & Critical Software Environment, National  
University of Defense Technology, China  
{wz,zbchen}@nudt.edu.cn

**Abstract.** AISE is a static verifier that can verify the safety properties of C programs. The core of AISE is a program verification framework that synergizes abstract interpretation and symbolic execution in a novel manner. Compared to the individual application of symbolic execution or abstract interpretation, AISE has better efficiency and precision. The implementation of AISE is based on KLEE and CLAM.

**Keywords:** Abstract Interpretation · Symbolic Execution · Program Verification.

## 1 Verification Approach

Given a program  $\mathcal{P}$  and a property  $\varphi$ , a software verification technique or tool verifies whether  $\mathcal{P}$  satisfies  $\varphi$ , *i.e.*, all the behavior (*e.g.*, the program paths) of  $\mathcal{P}$  satisfies  $\varphi$ . If  $\mathcal{P}$  does not satisfy  $\varphi$ , a counter-example (*e.g.*, a program input) will be given to demonstrate the violation of  $\varphi$ . Until now, many software verification techniques and tools have been developed and applied in different areas to result in successful stories [3,18,20,7,17].

AISE is a software verifier that verifies C programs with respect to reachability properties [5]. AISE's key idea is to synergize *symbolic execution* (SE) [4,21] and *abstract interpretation* (AI) [10,11]. In the main loop, our tool performs symbolic execution to analyze the program under verification. However, SE faces path explosion problem [23,16,9] when the program contains loops, which makes it infeasible for sound verification. AI can abstract a program in an over-approximation manner and automatically infer the program invariants at different program locations, which can be used to verify the property. However, the imprecision caused by over-approximation may result in false positives. AISE aims to combine these two techniques in a synergic manner to improve the verification's scalability as much as possible while ensuring precision. When doing SE, AISE carries out AI online to verify a part of the program, which can be used to prune the safe paths. On the other hand, SE can also improve the precision of AI. AISE only reports the violations detected by SE.

---

\* Jury member and corresponding author

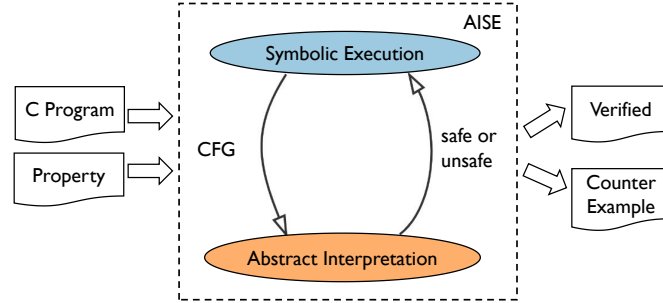


Fig. 1: AISE's verification framework

## 2 Framework

Figure 1 shows AISE's framework, which contains an AI module and a SE module. The two modules communicate by delivering control-flow graph (CFG) and verification results to help each other. On the one hand, SE constructs the sub-CFG on which AI is carried out; On the other hand, the verification results of AI are returned to SE to prune the redundant paths, *i.e.*, the paths that are guaranteed to satisfy the property.

### 2.1 Symbolic Execution Module

The SE module takes a C program as input and then executes the program with symbolic inputs. The SE procedure is a state-forking procedure [4]. The whole process is as follows. At the beginning of execution, the SE module constructs an initial symbolic state for the input program. As the state is executed, the data of the state is changed by executing instructions one by one. When the state encounters a branch instruction, a new state is forked based on the original state. A global state pool containing all forked states is maintained. After executing an instruction, the current state is paused, and another state is selected from the state pool to execute. When a state is terminated (*i.e.*, a state after executing the program exit instruction), AISE constructs a sub-CFG that contains all the instructions and the edges of the execution path that led to the state and carries out AI on the sub-CFG. Based on the AI's verification result, the state pool is updated, *i.e.*, adding the newly forked states or removing the pruned states. When SE finds a violation of an assertion, AISE reports the violation.

### 2.2 Abstract Interpretation Module

The AI module takes a sub-CFG as input and outputs safe or unsafe. Given the abstract domain [11], the AI module analyses the CFG to produce an invariant at each program location. The invariant describes the constraints of variables at the program location. Then, based on the invariant  $I$ , we can check the property  $\varphi$  by checking the validity of  $I \Rightarrow \varphi$ . If all assertions are checked, AISE can prune

states that can only reach the edges in the sub-CFG. Intuitively, all the possible paths start from these states are contained in the sub-CFG, so they are all safe paths. Therefore, we can prune all states from which only the nodes and edges of the sub-CFG can be reached. Pruning states reduces the path space of SE and improves the scalability of verification.

### 2.3 Example

Figure 2 gives an example<sup>3</sup> to illustrate the idea of AISE. This program contains a loop adding  $y$  to  $x$ . AI using interval domain [11] fails to verify this assertion because  $y$ 's invariant at Line 11 is  $(-\infty, 1000000]$ , which is not sufficient to prove the assertion. SE can verify this program by exploring all paths, but SE needs a long time as the paths of this program are numerous.

```

1 int main() {
2   int x=__VERIFIER_nondet_int();
3   int y=__VERIFIER_nondet_int();
4   if (!(y <= 1000000))
5     return 0;
6   if (y>0) {
7     while(x<100) {
8       x=x+y;
9     }
10  }
11  assert(y<=0 || (y>0 && x>=100));
12  return 0;
13 }

```

Fig. 2: C code segment

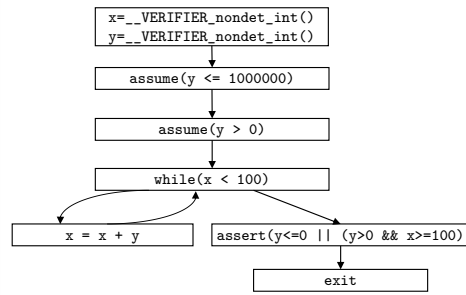


Fig. 3: CFG based on a path

AISE can verify this program successfully in a short time. The SE module only needs to explore a few paths because many can be pruned. After SE module explores the following path: 2→3→4→6→7→8→7→11→12, it constructs the sub-CFG in Figure 3 based on this path. For this sub-CFG, the AI module successfully verifies the assertion. Then, AISE framework updates the state pool in the SE module, killing all the states forked from line 7. These states are forked when encountering the loop head. Then, there are no more states in the pool and SE terminates, *i.e.*, a *safe* result. This also demonstrates that SE can improve the precision of AI by considering the sub-CFG of a symbolic path.

## 3 Implementation, Results and Discussion

AISE's implementation is based on the AI framework CLAM [2,17] and the SE tool KLEE [7]. STP [15] is the SMT solver of SE. AISE accepts the input in

<sup>3</sup> [https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/raw/main/c/loops/terminator\\_03-2.i](https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/raw/main/c/loops/terminator_03-2.i)

LLVM [1] intermediate representation. The AI module of AISE uses the polyhedron abstract domain [12], and we use the implementation in Apron library [19]. The search strategy of the SE module is *nurs:covnew*. Besides, AISE also integrates ESBMC [22] to handle floating-point programs because the SE’s module does not support the analysis of floating-point programs.

AISE participants in the *ReachSafety-Loops* category of SV-COMP 2024 [6]. Table 1 shows AISE’s results. AISE achieved 847 points in this category, and there were 4 tools ranked ahead of it: *Bubbaak* [8], *Symbiotic* [20], *VeriAbs* [13], *VeriAbsL* [14]. The figure<sup>4</sup> shows the score-based quantile plots in this category. When the time is less than about 100s, AISE achieved the highest score among all the tools. If the pruning method works, AISE can verify a program in a short time; otherwise, AISE may fail to finish the job. Many of the AISE’s failed cases are the programs with non-linear expressions. AI is limited for non-linear polynomials. Besides, AISE is also not efficient at handling large arrays. For example, AISE does not support symbolic size array, which is an inherited shortage from KLEE.

Table 1: AISE’s results

	number time(s)	
total tasks	790	
total correct	491	9400
correct true	356	6200
correct false	135	3200
total incorrect	0	0
score	847	

## 4 Software Project, Setup and Contributors

AISE only participats in the *ReachSafety-Loops* category of SV-COMP benchmarks. The usage of AISE is as follows.

```
./bin/aise <program>
```

The `<program>` is the input program. AISE only needs the input program as the parameter because all the properties in the *ReachSafety-Loops* benchmarks are the same, *i.e.*, (`unreach-call`, `ILP32`), and these properties are built in AISE.

AISE can be found at <https://github.com/zbchen/aise-verifier>. AISE is a prototype project developed by National University of Defense Technology. The license of AISE is GPL 3.0. People involved in the project are fully listed as the authors of this paper.

## Data-Availability Statement

AISE’s artifact is available at Zenodo: <https://doi.org/10.5281/zenodo.10201159>.

**Acknowledgement** This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (No. 62172429).

<sup>4</sup> <https://sv-comp.sosy-lab.org/2024/results/results-verified/quantilePlot-ReachSafety-Loops.svg>

## References

1. LLVM. <https://llvm.org>, accessed 2023-12-17
2. CLAM repository. <https://github.com/seahorn/clam> (2022)
3. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER with strategy selection (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
4. Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Comput. Surv.* **51**(3) (may 2018). <https://doi.org/10.1145/3182657>, <https://doi.org/10.1145/3182657>
5. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., Mckenzie, P.: Reachability Properties, pp. 79–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). [https://doi.org/10.1007/978-3-662-04558-9\\_6](https://doi.org/10.1007/978-3-662-04558-9_6), [https://doi.org/10.1007/978-3-662-04558-9\\_6](https://doi.org/10.1007/978-3-662-04558-9_6)
6. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
7. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
8. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 535–540. Springer Nature Switzerland, Cham (2023)
9. Christakis, M., Müller, P., Wüstholtz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proceedings of the 38th International Conference on Software Engineering. p. 144–155. ICSE ’16, Association for Computing Machinery, New York, NY, USA (2016). <https://doi.org/10.1145/2884781.2884843>, <https://doi.org/10.1145/2884781.2884843>
10. Cousot, P.: Abstract interpretation. *ACM Comput. Surv.* **28**(2), 324–328 (jun 1996). <https://doi.org/10.1145/234528.234740>, <https://doi.org/10.1145/234528.234740>
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings 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>, <https://doi.org/10.1145/512950.512973>
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 84–96. POPL ’78, Association for Computing Machinery, New York, NY, USA (1978). <https://doi.org/10.1145/512760.512770>, <https://doi.org/10.1145/512760.512770>
13. Darke, P., Agrawal, S., Venkatesh, R.: Veriabs: A tool for scalable verification by abstraction (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 458–462. Springer International Publishing, Cham (2021)
14. Darke, P., Chimdyalwar, B., Agrawal, S., Kumar, S., Venkatesh, R., Chakraborty, S.: Veriabsl: Scalable verification by abstraction and strategy prediction (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and

- Algorithms for the Construction and Analysis of Systems. pp. 588–593. Springer Nature Switzerland, Cham (2023)
15. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) *Computer Aided Verification*. pp. 519–531. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
  16. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. p. 23–33. ISSTA '11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/2001420.2001424>, <https://doi.org/10.1145/2001420.2001424>
  17. Gurfinkel, A., Navas, J.A.: Abstract interpretation of LLVM with a region-based memory model. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 13124, pp. 122–144. Springer (2021). [https://doi.org/10.1007/978-3-030-95561-8\\_8](https://doi.org/10.1007/978-3-030-95561-8_8)
  18. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: *ULTIMATE AUTOMIZER 2024* (competition contribution). In: *Proc. TACAS. LNCS*, Springer (2024)
  19. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) *Computer Aided Verification*. pp. 661–667. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
  20. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: *SYMBIOTIC 10: Lazy memory initialization and compact symbolic execution* (competition contribution). In: *Proc. TACAS. LNCS*, Springer (2024)
  21. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (jul 1976). <https://doi.org/10.1145/360248.360252>, <https://doi.org/10.1145/360248.360252>
  22. Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: *ESBMC v7.4: Harnessing the power of intervals* (competition contribution). In: *Proc. TACAS. LNCS*, Springer (2024)
  23. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. p. 225–236. ISSTA '09, Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1572272.1572299>, <https://doi.org/10.1145/1572272.1572299>