AISE v2.0: Combining Loop Transformations
(Competition Contribution)

Yao Lin[1,2], Zhenbang Chen[1,2(✉)]*, and Ji Wang[1,2]

[1] College of Computer Science and Technology, National University of Defense Technology, Changsha, China
[2] State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China
{linyao23,zbchen,wj}@nudt.edu.cn

Abstract. `AISE` is a C program verifier that synergizes symbolic execution and abstract interpretation. This year, `AISE` v2.0 introduces a loop transformation scheme based on recurrence analysis to handle programs involving nonlinear arithmetic. By combining loop transformations, `AISE` v2.0 achieved a score of 1031 and won first place in the *ReachSafety-Loops* category, demonstrating the effectiveness of the methods employed in `AISE` v2.0.

1 Verification Approach

The key idea of `AISE` [19] is to synergize abstract interpretation [12] and symbolic execution [7,15]. This synergy[1] enables `AISE` to handle the program with loops effectively. However, when the program involves nonlinear arithmetic, `AISE` still suffers from the path explosion problem. To address this, we introduce a loop abstraction method, which *over-approximates* the loop in the program based on the invariant generated by the recurrence analysis [17,18]. Additionally, we propose several other loop transformation methods to address different scenarios.

Figure 1 illustrates the workflow of `AISE` v2.0. Given a C program $\mathcal{P}$, we first compile it into `LLVM IR` format and then preprocess it using `LLVM`'s passes. Next, `AISE` will be used to verify $\mathcal{P}$. If `AISE` cannot finish within 60 seconds, the loops in $\mathcal{P}$ will be transformed, generating four variants, *i.e.*, `NL`, `A`, `ANs`, and `E`. `AISE` then sequentially verify `NL`, `A`, and `ANs` to determine whether the properties in $\mathcal{P}$ hold. If an unknown or violation occurs during this process, we will revert to verifying $\mathcal{P}$ or `E` until timeout. Below, we introduce each loop transformation and its role in verification.

Not entering the loop (`NL`). Figure 2b illustrates a scenario where the loop in $\mathcal{P}$ is not entered. Since symbolic execution often "gets stuck" in loops, prioritizing the analysis of this scenario may help reduce unnecessary overhead.

Loop abstraction (`A`). Inspired by the loop abstraction method [13], we use an invariant-based loop abstraction here to tackle the challenge brought by loops. Figure 2c shows the abstracted program `A`. Intuitively, `A` represents the set of

* Jury member
[1] Due to space limitations, further technical details can be found in [19].

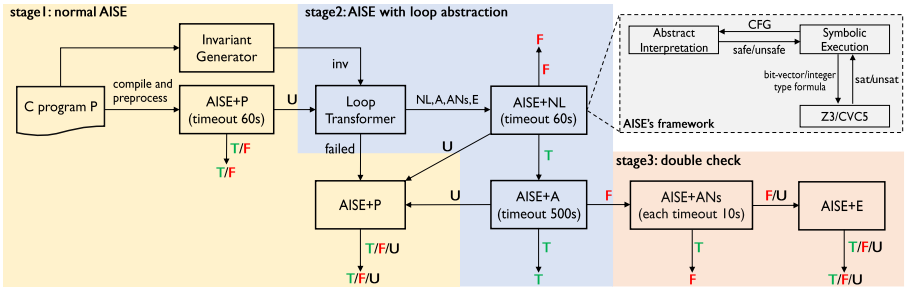
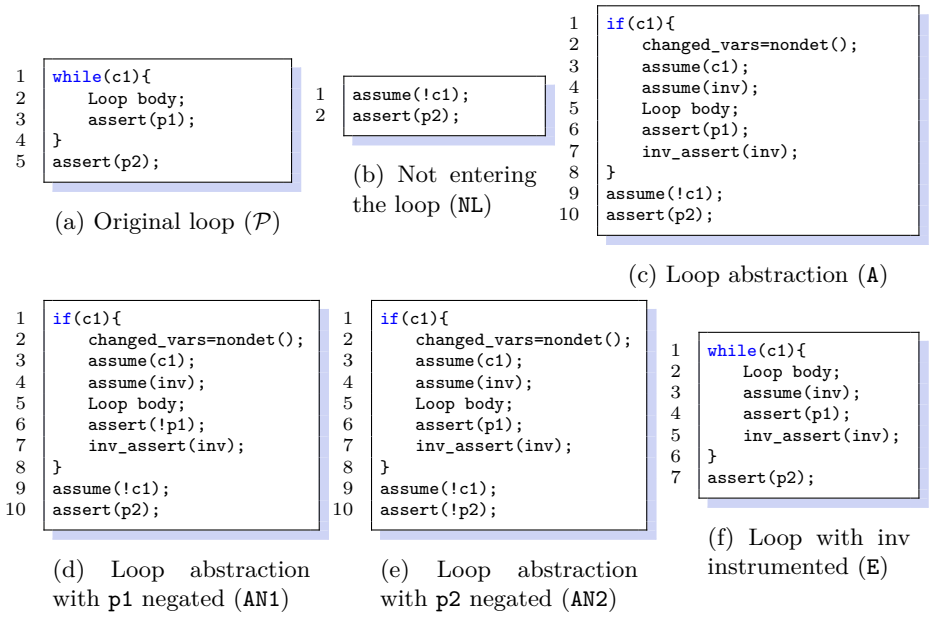


Fig. 1: The workflow of `AISE` v2.0 ($\mathcal{P}$: the original program, `NL`: not entering the loop, `A`: loop abstraction, `ANs`: loop abstractions with one negated property in each, `E`: the program with `inv` instrumented, **T**: True, **F**:False, **U**:Unknown)

all possible behaviors that could occur during any iteration of the loop in $\mathcal{P}$. In Figure 2c, Line 2 represents the non-deterministic assignments to the variables modified in the `Loop body` (denoted as `changed_vars`), which describes all possible states of all variables in the `Loop body` at the beginning of the loop's any iteration. Line 3 and Line 9 ensure the correct semantics of entering and exiting the loop, respectively. To improve the precision of abstraction, we use recurrence analysis [17,18], which captures the numerical relationships of variables using real arithmetic, to generate the loop invariant (`inv`) and add it in Line 4. However, when the loop involves integer division, rounding issues may cause the `inv` to fail to satisfy the inductiveness. Therefore, additional checks[2] (Line 7) are needed to verify the inductiveness of `inv`. If `inv` is violated, we will remove it from the transformed program and generate another invariant (which is included in each analysis step in Figure 1). Notably, recurrence analysis [17,18] only plays the role of generating `inv`, and `inv` can also be produced by other loop invariant generation techniques. Due to Line 1, the symbolic execution of `A` also covers the case not entering the loop (*i.e.*, `NL`).[3] For nested loops, we abstract the innermost loop first and then progressively abstract each outer loop before verification. By verifying that `p1` and `p2` hold in `A`, we can conclude that these properties also hold in $\mathcal{P}$. However, when verification fails, whether the property in $\mathcal{P}$ is violated cannot be determined, as it may be a false positive due to the *over-approximation* of the loop abstraction in `A`.

Loop abstraction with one negated property (`AN`). `AN` is generated from `A` with only one property negated. If a violation occurs when verifying `A`, we will first confirm it by verifying `ANs`. `AN1` (Figure 2d) and `AN2` (Figure 2e) are the `ANs` generated from `A` (Figure 2c). If we ignore assertions, we can also say that `ANs` are the *over-approximation* of $\mathcal{P}$. If the assertions in `AN1` and `AN2` are all reachable in $\mathcal{P}$ and the verification result of either is true, we can conclude that one property in $\mathcal{P}$ will be violated while the remaining properties will hold. For example, if

[2] `inv_assert` here functions like `assert`, but with a different name for distinction.
[3] Because of development time limitations, the scenarios described by `NL` and `A` overlap, which will be optimized in the future.



(a) Original loop ($\mathcal{P}$)

(b) Not entering the loop (`NL`)

(c) Loop abstraction (`A`)

(d) Loop abstraction with `p1` negated (`AN1`)

(e) Loop abstraction with `p2` negated (`AN2`)

(f) Loop with inv instrumented (`E`)

Fig. 2: Illustration of the loop transformation scheme

the assertions in `AN1` are all reachable in $\mathcal{P}$ and we can prove that `!p1` and `p2` hold in `AN1`, then we can conclude that `!p1` and `p2` also hold in $\mathcal{P}$, that is to say, the `assert(p1)` in $\mathcal{P}$ will be violated. During preprocessing, we use `LLVM`'s passes to remove the unreachable basic blocks in $\mathcal{P}$, ensuring that all assertions in `ANs` are reachable in $\mathcal{P}$. If unreachable assertions still remain in `ANs` after transformation, this approach may result in false positives.

Loop with invariant (`E`). This variant in Figure 2f is the last step of confirming the violation, *i.e.*, employing `AISE` to check the violation. However, different from $\mathcal{P}$, we add the invariant `inv` in Line 3, which improves the efficiency of symbolic execution by leveraging the invariant to accelerate constraint solving process.

2 Implementation

The `Loop Transformer` of `AISE` v2.0 is based on `LLVM` [1]. `AISE`'s abstract interpretation module is `CLAM` [2,14], configured as in [19]. `AISE`'s symbolic execution module is based on `BUBAAK-LEE` [3] (a fork of `KLEE` [10] used in the `BUBAAK` [11]) with some improvements. We improve the solver part by supporting floating-point programs and employing the theory of integer as an additional option (`KLEE` uses bit-vector theory by default) for solving path conditions. The symbolic execution module of `AISE` first uses `Z3` [16] to check formulas' satisfiability, and switches to `CVC5` [8] if `Z3` times out. The `Invariant Generator` is based on `c_convertor` [4], which summarizes loops by computing closed-form solutions for each variable in the loop body [17] or for the polynomial expressions

Table 1: The contribution of each method

	total	AISE+P	AISE+NL	AISE+A	AISE+ANs	AISE+E
total correct	603	528	0	66	8	1
correct true	428	362	0	66	0	0
correct false	175	166	0	0	8	1
total correct-unconfirmed	73	49	0	23	0	1
total incorrect	0	0	0	0	0	0
score	1031	890	0	132	8	1

extracted from the loop body [18]. Besides fixing some bugs, we have optimized the `Invariant Generator` by considering branch conditions and the initial assignments of variables.

3 Result and Discussion

This year, by combining (1) the synergy between abstract interpretation and symbolic execution and (2) a recurrence analysis based loop transformation scheme, `AISE` v2.0 scored 1031 points and won first place in the *ReachSafety-Loops* category, demonstrating its capability in verifying the programs with loops. Table 1 presents the contribution of each method. As shown by the table, the synergy in `AISE` is effective for most programs, and when it fails within 60 seconds, loop transformations can further enhance `AISE`'s ability. Due to lack of invariant in correctness witnesses, we still have 73 unconfirmed results. `AISE` v2.0 could not produce any results in the `AISE+NL` phase due to the program's small size, and it may perform better on larger programs. When handling programs with arrays or loops containing multiple branches, both `CLAM` and `Invariant Generator` may not work, causing `AISE` v2.0 to still suffer from the path explosion problem. Additionally, the SMT solvers encounter difficulties when solving nonlinear constraints, further limiting the verification capabilities of `AISE` v2.0. Since we only consider whether the program is correct when it terminates, `AISE`'s verify method currently cannot handle programs with non-terminating loops.

4 Software Project, Tool setup and Contributors

`AISE` v2.0 can run on the Ubuntu 22.04/24.04 LTS and is licensed under GPL 3.0. This year, `AISE` v2.0 participates in the *ReachSafety-Loops* category of SV-COMP 2025 [9]. The execution command of `AISE` v2.0 is as follow:

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

Where `<data_model>` is the program's data model (32-bit or 64-bit) and `<program>` is the input program. The contributors of `AISE` v2.0 are all from the National University of Defense Technology. A complete list of contributors is available at [5].

Data-Availability Statement The artifact for `AISE` v2.0 has been archived and is available on Zenodo [6].

Acknowledgement This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62032024).

References

1. LLVM. `https://llvm.org`, accessed 2024-12-11
2. CLAM repository. `https://github.com/seahorn/clam` (2022)
3. BUBAAK-LEE repository. `https://github.com/mchalupa/bubaak-lee` (2022)
4. c_convertor repository. `https://github.com/psy054duck/c_convertor` (2024)
5. Contributors list of AISE. `https://github.com/zbchen/aise-verifier/blob/master/Contributors.txt`, accessed 2025-1-23
6. AISE artifact. `https://zenodo.org/records/14203693` (2024)
7. Baldoni, R., Coppa, E., Delia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques **51**(3) (May 2018). `https://doi.org/10.1145/3182657`
8. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
9. Beyer, D., Strejek, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
10. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association
11. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers: (competition contribution). p. 535540. Springer-Verlag, Berlin, Heidelberg (2023). `https://doi.org/10.1007/978-3-031-30820-8_32`
12. Cousot, P.: Abstract interpretation. ACM Comput. Surv. **28**(2), 324328 (Jun 1996). `https://doi.org/10.1145/234528.234740`
13. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1407–1412 (2015). `https://doi.org/10.7873/DATE.2015.0245`
14. Gurfinkel, A., Navas, J.A.: Abstract interpretation of llvm with a region-based memory model. p. 122144. Springer-Verlag, Berlin, Heidelberg (2021). `https://doi.org/10.1007/978-3-030-95561-8_8`
15. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (Jul 1976). `https://doi.org/10.1145/360248.360252`
16. de Moura, L.M., Bjørner, N.S.: Z3: An efficient smt solver. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2008)
17. Wang, C., Lin, F.: Solving conditional linear recurrences for program verification: The periodic case. Proc. ACM Program. Lang. **7**(OOPSLA1) (Apr 2023). `https://doi.org/10.1145/3586028`

18. Wang, C., Lin, F.: On polynomial expressions with c-finite recurrences in loops with nested nondeterministic branches. In: Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 2427, 2024, Proceedings, Part I. p. 409430. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-65627-9_20
19. Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 347–352. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_19