Lazy Self-Composition for Security Verification

Weikun Yang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik

Princeton University

Abstract.

1 Introduction

AG: have added basic points to highlight, terminology needs to be cleaned up, will add more background and highlights after experiments

One approach for verifying secure information flow is to convert it into a safety verification problem on a "self-composition" of the program, where two copies of the program are created on which a safety property is checked [2]. For example, to check for information leaks, the low-security variables are initialized to identical values in the two copies of the program, while the high-security variables are unconstrained and can take different values. The safety check ensures that in all executions of the program with the two copies, the values of the low-security variables are identical at the end of the program, i.e., there is no information leak from high-security to low-security variables. The self-composition approach is a general approach for checking hyper-properties, and has been recently applied for checking constant-time implementations of secure code [1].

Although the self-composition reduction is sound and complete, it is challenging to perform safety verification, especially on two copies of a program. An improvement was suggested by Terauchi and Aiken [8], where they combined the self-composition approach with type analysis to make copies of (portions of) the program in a manner that is more friendly for software verifiers. For example, if a loop condition is of type low-security, then the loop bodies are duplicated inside a single loop. This has the effect of keeping corresponding variables in the two copies of the program near each other, which can be useful in deriving invariants to aid safety verification.

In this paper, we aim to further improve the self-composition approach by making it *lazier*, in contrast to an eager upfront translation into two copies of a program. This lazy duplication is enabled by dynamic taint propagation, which is performed along with an unrolling of the program during bounded model checking (BMC) [3]. Dynamic taint propagation has the benefit of being more precise than static type-based analysis. By combining it with BMC, it is guaranteed to cover all possible (bounded) program executions, unlike other dynamic approaches that cover only the tested executions. This also allows us to leverage existing interpolant-based verification methods for proving correctness over unbounded executions.

We also propose a specialized early termination check for the BMC-based approach. In secure programs, it is often the case that sensitive information is propagated in a

very localized context, and conditions exist that squash its propagation any further. An eager self-composition approach does not exploit this, but in effect delays the security property check to take place at the end of the two programs. Indeed, it depends on the software verifier to perform any property-related slicing or other optimizations. We believe that general software verifiers are not equipped to leverage such conditions. We formulate our early termination check in terms of taint queries on live variables during program unrolling. In practice, this check is often successful, leading to much saved effort in comparison to performing a full safety check.

To summarize, our lazy self-composition approach based on BMC provides the following advantages for security verification:

- It performs *dynamic taint propagation* to determine precise security types while unrolling the program during BMC. This information is more precise than static type-based analysis (which loses precision due to path-insensitive analyses), and covers all possible (bounded) program executions unlike dynamic taint analysis that covers only the tested executions.
- It performs *lazy self-composition* by querying security types during program unrolling and using it to avoid or optimize duplicated code. The goals are similar to the Terauchi-Aiken approach, but our dynamic type queries yield more precise information and our duplication optimizations are more effective. (AG: will need to justify this.)
- It performs an *early termination* check for information flow, whereby it is guaranteed that the security property is correct without any further unrolling.

Our security verifier targets rich specifications in terms of low-security and high-security variables/locations, predicates that allow information downgrading in certain contexts, checking for constant-time implementations, etc. It leverages modern SMT-based verification techniques, including (potential) use of interpolants to derive proofs of correctness. We have implemented the security verifier as part of the SeaHorn verification platform [4], which represents programs as CHC (Constrained Horn Clause) rules. It has a frontend based on LLVM [6], and backends to Z3 [7] and mu-Z3 [5].

2 Motivating Examples

- Example 1
  Path-sensitive analysis in MC gives more precise results than static analysis for security types.
- Example 2
  Dynamic taint analysis in BMC gives more precise results than static analysis on loops.
- Example 3
  Advantage of early termination

3 Preliminaries

3.1 Security Properties

3.2 CHC-based Program Verification

4 Lazy Self-Composition

4.1 Dynamic Taint Propagation

4.2 Lazy Duplication

4.3 Early Termination

4.4 Interpolant-based Verification

5 Implementation and Experiments

5.1 Implementation

5.2 Experiments: Synthetic Benchmarks

5.3 Experiments: Case Studies

6 Related Work

- Secure information flow
  confidentiality, integrity, non-interference
- Language-based security, static type analysis
- Dynamic taint analysis
  dynamic execution, symbolic execution
- Self-composition approaches
  Barthe et al., hyper-LTL checker, recent papers

7 Conclusions and Future Directions

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: 25th USENIX Security Symposium, USENIX Security. pp. 53–70 (2016)
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17). pp. 100–114 (2004)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS, Proceedings. pp. 193–207 (1999)
4. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 343–361 (2015)

5. Hoder, K., Bjørner, N., de Moura, L.M.: μZ- an efficient engine for fixed points with constraints. In: Computer Aided Verification - 23rd International Conference, CAV. Proceedings. pp. 457–462 (2011)
6. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO). pp. 75–88 (2004)
7. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: 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, Proceedings. pp. 337–340 (2008)
8. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Static Analysis, 12th International Symposium, SAS, Proceedings. pp. 352–367 (2005)