Proceedings of CONF-MPCS 2024 Workshop: Quantum Machine Learning: Bridging Quantum Physics and Computational Simulations

DOI: 10.54254/2753-8818/51/2024CH0178

Evaluating satisfiability modulo theorem solvers for code deobfuscation

Tinghan Zhu

University of Wisconsin–Madison, 702 W. Johnson St, Madison, WI 53715-1007, US

tzhu222@wisc.edu

Abstract. Automated static program analysis is crucial in cybersecurity research and malware identification. With antivirus software (like Microsoft Windows Defender) employing malware detection algorithms that statically analyze programs that the users interact with, the performance and efficiency of automatic static program analysis becomes more important than ever. Code obfuscation is a tactic of complicating the program expression and logic without affecting its semantics to prevent reverse engineering. The prevalence of code obfuscation makes static program analysis difficult. Symbolic execution is a powerful method of static program analysis that could be leveraged to optimize the obfuscation away and is powered by the solution of satisfiability modulo theorems (SMT). While research finding performant SMT-solving approaches has been highly focused, rarely have researchers looked into evaluating different SMT-solving algorithms for specific scenarios. This research aims to evaluate different mainstream state-of-the-art SMT solvers for their performance in the use of symbolic optimization and code deobfuscation. By analyzing common code obfuscation tactics and comparing the strategies that SMT solvers use and their performance in different benchmarking categories, the research concludes that linear arithmetics and bit vector solving are the most important aspects of an SMT solver and future SMT solver development should explore building solvers with hybrid architectures that combine the strengths different solvers at solving different types of theories for the most optimal efficiency.

Keywords: SMT Solver, Symbolic Execution, Symbolic Optimization, Software Reverse-Engineering, Code Obfuscation.

1. Introduction

Code obfuscation is the process of heavily reducing the human readability of some code without altering the semantics by complicating its existing logic and adding unnecessary operations to prevent third parties from reverse-engineering the program and understanding its underlying mechanisms. Obfuscations could aid in protecting paid commercial software from reverse-engineering and cracking and enforce trusted remote computing [1, 2]. However, it is also employed by malicious actors to protect malware. Therefore, efficient removal of obfuscation is important in the effort to protect digital systems as effective reverse-engineering of malware could help cybersecurity professionals understand how they spread and inflict negative effects and therefore allow them to develop prevention tactics like vulnerability fixes and anti-virus programs [3]. Although code obfuscation could both be done on source code (e.g. C/C++, Java source code) and compiled machine code (e.g. x86, ARM assembly

DOI: 10.54254/2753-8818/51/2024CH0178

code), this article would only focus on compile-time obfuscation (machine code obfuscation) as it is the most popular and effective obfuscation method [4].

Symbolic execution is an important tool in removing code obfuscation. By solving for symbolic values of execution conditions in a program rather than executing the code, symbolic execution could effectively determine under which explicit conditions a part of a program would execute [5]. By deducing explicit execution conditions of different parts of the code, one could use symbolic execution to effectively remove the logical and operational redundancy introduced by code obfuscation.

Symbolic execution requires the construction and solution of satisfiability modulo theories (SMT). SMT generalizes boolean satisfiability problems (SAT) by introducing more complex data types like real numbers, lists, and arrays. Since SMT could be transformed into SAT in polynomial time, SMT solving is an NP-complete problem. Therefore, algorithms designed to solve SMTs combine common techniques used to more efficiently solve other NP-complete problems like probabilistic methods, approximation, and parameterization. Different SMT solver approaches could perform differently under different scenarios due to this reason. Several attempts have been made to push for more performance SMT solving. The International Satisfiability Modulo Theories Solver Competition (SMT-COMP) is a competition hosted to compare the performance of SMT solvers and encourage the development of more efficient SMT-solving approaches [6]. Using multiple solvers and choosing the appropriate solver for each specific problem is also a popular approach to building an efficient SMT solver system [7]. Mach-SMT is a machine learning algorithm designed to choose the best solver for a certain SMT problem by approximating the run time of different solvers [8].

This article will examine the performance of different SMT solvers for symbolic execution for code de-obfuscation.

2. Symbolic execution for code deobfuscation

Symbolic execution is a process of static program analysis. Symbolic execution assumes that the inputs given to the program are deterministic for the output of the program. Inputs are represented as symbols. By following the control flow of a program and assigning symbolic values to internal variables that depend on input values whenever needed, a mapping could be built between concrete input values to a program and parts of the program that the associated input would trigger. The benefit of program analysis symbolic execution is thus self-evident as this mapping essentially describes the behavior of the program. Symbolic execution is useful for code deobfuscation as representing execution conditions as symbolic values make it easy to condense the expression and remove the complications added by code obfuscation. Symbolic executors heavily rely on SMT solving to obtain simplified executing conditions for different parts of the program.

However, the task of tracing control flow is not simple for larger programs which take a wide range of input and involve complex conditional execution and nested loops. The number of possible paths to follow would grow exponentially as several branching and loops grow in larger programs. Oftentimes, the number of possible paths to follow becomes too large that it is unfeasible to follow every possible path in the program. This scenario is called a path explosion [9].

3. SMT solving principles

It is important to explore the reason why the problem of choosing an SMT solver exists and why the performance of SMT solvers is highly dependent on the theories it deals with. This section is dedicated to discussing the purpose and mechanisms of theory-specific solvers and their advantages compared to general brute-forcing methods.

3.1. Eager method: bitblasting

Since SMT are generalizations of SAT problems, all the higher order data structures and their operations that SMT introduces have corresponding encoding to transform themselves and their operations into bit vectors and bitwise operations. The simple and universal approach in SMT solving is to transform SMT into SAT and use SAT solvers to solve the transformed problem instead.

DOI: 10.54254/2753-8818/51/2024CH0178

However, this method comes with significant caveats. The solver eliminates the possibility of leveraging additional information given by higher-order constructs in SMT by transforming them into SAT. For example, the SAT solver would not be able to benefit from first-principle mathematical evaluations that are possible with SMT. Simplification given by realizations of algebraic identities would not be possible in the context of Boolean reasoning. The implication of this is that solvers that employ this strategy must adopt a strict evaluation (eager evaluation) method, all atomic bit vector expressions must evaluated individually before moving on to compound operations. Therefore, most SAT-based SMT solvers utilize stacks to evaluate large nested expressions. The eager evaluation nature of bit blasting makes SAT-based SMT solvers undesirable for use in symbolic execution for program analysis, as static analysis via symbolic execution driven by eager evaluation would offer little efficiency advantage over analyzing the program dynamically by actually executing them with specific inputs [10].

3.2. Lazy method: a layered approach

Most state-of-the-art SMT solvers today use a layered approach. Large SMTs are solved recursively by breaking them down into atomic reasoning problems and dealing with each of them individually using specific tactics [5]. Each tactic is explicitly defined and implemented algorithm for solving certain reasoning problems in the scope of SMT. Those tactics combine common strategies in solving NP-complete problems including parameterization and approximation with simple methods like exhaustion search (brute-forcing algorithm) to complete the reasoning step relatively efficiently. Like most other efficient NP-complete problem-solving algorithms, the performance of these tactics is highly scenario-dependent. Usually, there are no globally optimal tactics for a certain given reasoning problem (some tactics would outperform others in certain scenarios, and vice-versa). The combination of tactics that an SMT solver uses forms its problem-solving strategy.

4. Common code obfuscation techniques

Code deobfuscation is similar to compiler optimization as both are essenentially optimization processes that reduce redundancy in code. However, code obfuscation is for readability rather than performance. Also, code deobfuscation processes should be designed to deal with code that is intentionally manipulated to be unoptimized.

4.1. Opaque branching/predicates

Opaque branching (also called opaque predicates), is an obfuscation technique that masks the actual control flow of the program by adding redundant branching operations with conditions that are based on invalid predicates that are not affected by any runtime factors and thus do not alter the semantics of the program. Opaque branching is a powerful obfuscation strategy as reverse engineering requires the analysis of the control flow of the program for an effective understanding of a program's behavior. Control flow analysis could be done either using the debugger or symbolic execution and/or building a control flow graph (CFG) for easy human interpretation. The process of removing opaque branches is called control flow flattening, as removing the extra branching increases the linearity of the CFG and reveals the actual control flow with valid branching operations that determine the semantics of the program.

4.2. Data obfuscation

Adding redundant operations is a common obfuscation technique as it masks the code which performs the actual operations in invalid codes which has no side effects and does not alter the behavior of the former code. Invalid sets of operations include blank operations which have no side effects like the nop instruction in x86, or dead stores (operations performed on unused variables) and combinations of operations that have negating side effects are common tactics of adding redundant operations.

Redundant operations could also be performed on variables or registers that are referenced by the underlying operational code, given that the alternation would not result in a change of behavior, doing

DOI: 10.54254/2753-8818/51/2024CH0178

so would not only complicate the mechanics of the obfuscated code, it would also complicate the symbolic expression of a variable. To illustrate an example, given that in a section of an x86 program, register eax is assigned a value of 2, the instruction mov eax, 2 is performed to assign literal value 2 to eax. Without data obfuscation, the intent of this single instruction and the expected value of eax is very clear. Data obfuscations could be applied to transform a single mov instruction into more complicated series of operations. After the execution of this transformed section of code, the value 2 should still stored on eax. Since$2 = (1 + 1) * (1 + 1) - (3 - 1)$ (any mathematical identity could be used, of course), the code could then be transformed into:

In this version of code, more types of instruction (add, sub, and mul on top of mov) and more registers (ebx, ecx on top of eax) are used to perform this simple operation. While ebx and ecx are used to perform valid arithmetic operations for the value of eax, after this section of code, ebx and ecx could function as deadstores if it is not referenced in the upcoming sections of code, otherwise, the value of ebx and ecx would have to be saved before performing obfuscated operations to prevent unexpected side-effects. Data obfuscations introduce a heavy need for expression simplification for SMT solvers when attempting to leverage SMT solvers for code deobfuscation. Thus, the tactics used for expression simplification play an important role in determining the solver's efficiency in symbolic optimization.

String obfuscation is an important aspect of data obfuscation. Strings are often stored as-is in unprotected distributed software. Strings referenced in code would often reveal information regarding the purpose of a certain piece of code. Looking at the following piece of example C code:

```
result = func1(var1)

IF result == 0 THEN
    PRINT "Password is incorrect"
ELSE
    PRINT "Welcome"

END IF
```

The purpose of func1 and var1 is revealed by the string parameters of the print calls. Given that func1 takes variable var1 and returns an integer value, the code prints 'Password is correct' if the return value is 0 and 'Welcome' otherwise, it is reasonable to assume that func1 is a password validation function that takes the user input as a parameter and returns a boolean value of password correctness, and var1 is responsible for storing said user input. In this case, the behavior and mechanism of func1 are partially revealed without the analysis of its code. Simple string obfuscation techniques use similar techniques as the one shown above, while more advanced techniques employ cryptographic methods for string obfuscation.

5. SMT solvers

A few popular SMT solvers are explored in this section. Examining the underlying strategies used by these popular SMT solvers helps find the optimal set of tactics to use for SMT solving. This article would example Z3, Bitwuzla, and MathSAT5. All three solvers use the lazy-solving approach and have a variety of theory solvers and utilize different strategies.

5.1. MathSAT5

MathSAT5 is a state-of-the-art open-source SMT solver. It is composed of a preprocessor, a SAT-solving instance, and the MathSolve theory reasoning system. It supports quantifier elimination to remove quantifiers generated by data obfuscations. MathSAT5 utilizes Craig interpolation for formal

DOI: 10.54254/2753-8818/51/2024CH0178

verification, which is a powerful technique of logical reasoning that would aid in expression simplification from known unsatisfiability [11].

5.2. Z3

Z3 is an open-source SMT solver that features a preprocessor and uses heuristic techniques to perform efficient simplification on arithmetic theories. It could efficiently combine results for different theory solvers with partial models. Z3 also supports efficient quantifier elimination. It can choose from a variety of strategies and heuristics depending on the theories given [12].

5.3. Bitwuzla

Bitwuzla is an open-source SMT solver that specializes in bit-vector solving. Bitwuzla has a comprehensive bit-blasting strategy which also enhances its performance of floating-point arithmetics by transforming floating point operation into bit-vectors. Bitwuzla also features a diverse set of strategies for the solver instance to choose from [13].

6. Evaluation

From the previous analysis, it could be seen that linear arithmetics, bit-vector handling, and quantifier-free string theory solving are crucial in symbolic execution for code deobfuscation. This article would only focus on the sequential performance of solvers as symbolic execution and optimization are sequential. From results in SMT-COMP 2021, Z3 generally outperforms MathSAT at non-quantifier-free arithmetic theories and bit vectors. Meanwhile, for qualifier-free floating point arithmetics, Bitwuzla significantly outperforms Z3 [14].

7. Conclusion

It could be seen that efficient SMT solving for code deobfuscation requires a combination of different solvers for the most optimal efficiency. Many solvers act as wrappers that combine other solvers and select them depending on the scenario. Machine learning algorithms exist for solver choosing, while these algorithms have the potential to improve general solving performance, the resources those algorithms use would also have to be taken into account. For resource-limited or time-sensitive scenarios like edge computing, the most optimal way of SMT solving for code deobfuscation would be to use different solvers to tackle different sub-problems depending on their type. The limitation of this research is that it did not concern the performance impact of the process of lifting from machine codes to SMTs for symbolic execution, which is also a computationally intensive process. Further studies should focus on developing solver wrappers to combine general-purpose SMT solvers and improve performance for specific scenarios.

References

[1] Behera, C. K., & Bhaskari, D. L. (2015). Different obfuscation techniques for code protection. Procedia Computer Science, 70, 757–763.
[2] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2016). Protecting software through obfuscation. ACM Computing Surveys, 49(1), 1–37.
[3] Canfora, G., Di Penta, M., & Cerulo, L. (2011). Achievements and challenges in software reverse engineering. Communications of the ACM, 54(4), 142–151.
[4] You, I., & Yim, K. (2010). Malware obfuscation techniques: A brief survey. In 2010 International Conference on Broadband, Wireless Computing, Communication and Applications.
[5] Naug, R. (2020). Useless code identification with symbolic execution. Journal of Advanced Research in Dynamical and Control Systems, 12(SP3), 17–20.
[6] Barrett, C., de Moura, L., & Stump, A. (2005). SMT-Comp: Satisfiability modulo theories competition. In Lecture Notes in Computer Science (pp. 20–23).

DOI: 10.54254/2753-8818/51/2024CH0178

[7] Scott, J., Niemetz, A., Preiner, M., Nejati, S., & Ganesh, V. (2021). MACHSMT: A machine learning-based algorithm selector for SMT solvers. In Lecture Notes in Computer Science (pp. 303–325).
[8] Scott, J., Niemetz, A., Preiner, M., Nejati, S., & Ganesh, V. (2023). Algorithm selection for SMT. International Journal on Software Tools for Technology Transfer, 25(2), 219–239.
[9] Sen, K. (2016). Technical perspective: Veritesting tackles path-explosion problem. Communications of the ACM, 59(6), 92.
[10] Bruttomesso, R., et al. (2005). A lazy and layered SMT(BV) solver for hard industrial verification problems. In Lecture Notes in Computer Science (pp. 547–560).
[11] Bozzano, M., et al. (2005). MathSAT: Tight integration of SAT and mathematical decision procedures. In SAT 2005 (pp. 265–293).
[12] De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In Lecture Notes in Computer Science (pp. 337–340).
[13] Niemetz, A., & Preiner, M. (2023). Bitwuzla. In Lecture Notes in Computer Science (pp. 3–17).
[14] SMT-COMP. (2021). Results single query. SMT-COMP 2021. Retrieved August 6, 2024, from https://smt-comp.github.io/2021/results/results-single-query