ISBN: 978-981-18-7950-0
2023 the 13th International Workshop on Computer Science and Engineering (WCSE 2023)
doi: 10.18178/wcse.2023.06.003

VeriOover: A Verifier for Detecting Integer Overflow by Loop Abstraction

Jian Fang and Haipeng Qu[+]

School of Computer Science and Technology, Ocean University of China, Qingdao, China

Abstract. When verifying software properties by using automatic formal verification methods, loop structures require manually provided loop invariants to enable automatic verification. To solve the difficulty of manually writing loop invariants in automated verification, we propose a novel approach to abstract the behavior of the loop structure. Through this abstraction method, the trends and ranges of variables in loop structures can be obtained. The verifier utilizes the abstracted information to automatically verify whether integer overflow vulnerabilities exist in the program. We implemented an automated formal verification tool, VeriOover, based on this method for detecting integer overflow in C language programs. The tool parses input programs into abstract syntax trees, converts the nondeterministic values into symbolic variables, and leverages symbolic execution and loop abstraction for program state analysis. VeriOover generates program assertions based on each program state to determine if an integer overflow exists in the program at that time. VeriOover is implemented and publicly available as an open-source project at https://github.com/Rw1nd/VeriOover.

Keywords: software security, integer overflow, vulnerability detection, program verification

1. Introduction

As a general-purpose computer programming language, C lacks a security protection mechanism. Therefore, there are various types of vulnerabilities in C programs. The integer overflow is one of the software vulnerabilities [1]. Integer variables exist in a fixed range interval with limited computer memory [2]. An integer overflow occurs when the result of an arithmetic operation or an intermediate result exceeds the range of the integer type at runtime. Hence the program can produce an issue where the result of the computation does not match the expected result. This issue can threaten the security of the entire software. The attacker can trigger the overflow by constructing a particular input that makes the software fail to perform properly. Alternatively, the attacker could hijack the software execution logic or execute other malicious behavior by exploiting the integer overflow vulnerability [3].

Currently, the software vulnerability detection methods include software testing [4] and static analysis [5]. For software testing, fuzzing is one of the most popular methods [6]. Fuzzing utilizes the initial seeds as test samples, then mutates these seeds in the subsequent testing process to generate new test samples, and repeats the mutation and testing process to detect vulnerabilities. The most commonly used tools for fuzzy testing are AFL++ [7], VUzzer [8], etc. Generally, fuzzy testing is challenging to cover all tested code, and no guarantee that tested software is free of vulnerabilities. Static analysis is another software vulnerability analysis method. Through symbolic execution [9], taint analysis [10], program slicing [11] and other methods, static analysis methods can cover all of the code and evaluate the properties of a program. And it is capable of detecting software vulnerabilities through data flow analysis and pointing analysis [12] without executing the program. Some commonly used static analysis tools are SVF [13], CBMC [14], etc. However, static analysis may generate false negatives or false positives in the detection of vulnerabilities.

Formal verification is an alternative approach to ensuring program safety, whereby program properties are mathematically proven through proof codes [15]. However, this method requires a significant amount of manual effort in writing proof code. To achieve full automation, it is currently necessary to manually specify loop invariants when verifying programs with loop structures. Moreover, due to the large number of program states that need to be proven in automatic formal verification, some tools, like Frama-C [16] and VeriFuzz

[+] Corresponding author.
E-mail address: quhaipeng@ouc.edu.cn.

[17], are unable to produce proof results within an acceptable amount of time. In this paper, we propose a novel loop analysis technique that automates the verification of loop properties and leverages them to detect integer overflow vulnerabilities in programs. The contributions of this paper are as follows:

- We propose a novel method for abstracting the properties of loop structures. This method is capable of automatically abstracting the properties of variables in loop structures, which enables the detection of integer overflow vulnerabilities based on these properties and thus mitigates the difficulty in verifying loop structures during automated verification.
- We implement the tool VeriOover based on the idea of automatic formal verification. It combines symbolic execution and our loop abstraction methods to detect integer overflows in the C language.
- A formal verification approach is used to automatically detect existing programs and improve the credibility of the detection based on static analysis.

The following section is the design overview of VeriOover. Section 3 describes the method of the loop abstraction. Section 4 describes the evaluation of the method. Section 5 is the conclusion.

2. Design Overview

VeriOover verifier tool is depicted in Fig. 1. VeriOover accepts a C source code file for verification.

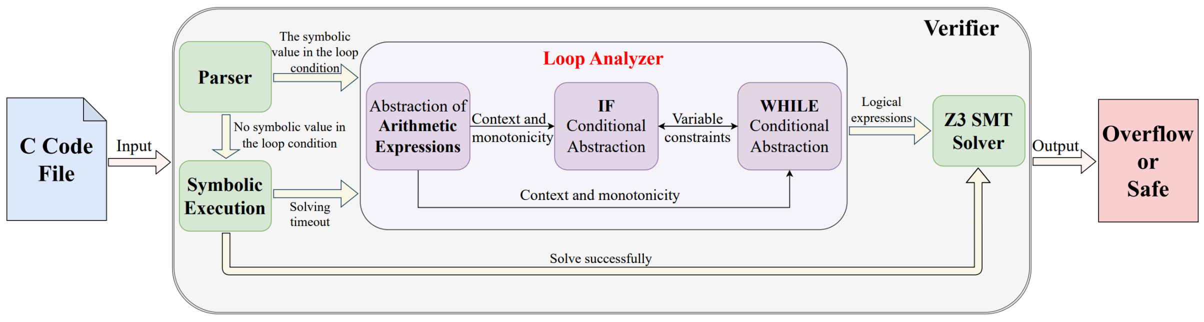


Fig. 1: Overview of VeriOover verifier.

The parser in the verifier first converts the input code into an abstract syntax tree. Then it uses static analysis methods to mark the symbolic values and their positions in the abstract syntax tree. If the symbolic value appears in the loop condition, the parser will provide the analyzed abstract syntax tree to the loop analyzer. The loop analyzer traverses each statement in the loop body and gradually abstracts the variation trend for each symbolic value based on the type of statement and the current abstraction context.

The verifier will generate logical assertions based on the variable information abstracted from the loop and the definition of integer overflow. Then the logical assertions are then fed into an SMT solver [18] for automated solving. If the assertions can not be satisfied, the integer overflow exists. Conversely, the program does not have an integer overflow vulnerability if the logical formulas can be satisfied. If the symbolic value is not in the loop condition, then we use the symbolic execution method to analyze the code. When symbolic execution does not produce a result within the specified time, VeriOover reanalyzes the program using the loop analyzer.

2.1. Symbolic execution

With symbolic execution, the input values of a program can be treated as symbols, and the program properties can be analyzed by constructing logical formulas. KLEE [19], angr [20], S2E [21] are currently popular symbolic execution tools. However, there are some challenges to symbolic execution. When analysing the loop, symbolic execution analyses all feasible paths. If there is no clear boundary in the loop condition, symbolic execution will generate an infinite number of states. This causes the symbolic execution tool to fail to continue with the analysis.

This paper uses a new loop abstraction method to analyze the range of symbolic variables in loops to avoid the path explosion problem that arises when analyzing loop structures in symbolic execution. We use static analysis to record all symbolic variables and their position in the programs. If the symbolic variable exists in the loop condition, then the loop analyzer will analyze the variables leveraging the loop abstraction method. In this paper, we employ this approach to abstract all symbolic variables and obtain logical constraints for the range of variables. We can determine whether the integer overflow exists by solving the logical formula.

2.2. Variable constraints

Since different systems define the range of shaping types differently, the integer types discussed in this paper are all 32-bit and are divided into signed and unsigned. Let $Prog$ denote the program to be analyzed, where v_s represents signed integer type variables, v_u represents unsigned integer type variables, σ represents the current state of the program during execution, Σ represents the set of all states of program $Prog$, and $V(\sigma)$ represents all variables in set σ. For signed integer variables with all states in the program, there is the theorem Q_{so} that holds:

$$Q_{so}: \forall v_s \in V(\sigma). -(2^{31}) \leq \sigma(v_s) \leq 2^{31} - 1 \tag{1}$$

For unsigned integer variables in any state of the program, there is the theorem Q_{uo} that holds.:

$$\mathrm{Q_{uo}}: \forall \mathrm{v_u} \in \mathrm{V}(\sigma). 0 \leq \sigma(\mathrm{v_u}) \leq 2^{32} - 1 \tag{2}$$

Then the integer overflow in the program can be defined formally as the theorem P_o:

$$P_o: \exists \sigma' \in \Sigma.\ \sigma' \vDash (\exists v_s \in V(\sigma').\ \neg Q_{so}) \vee (\exists v_u \in V(\sigma').\ \neg Q_{uo}) \tag{3}$$

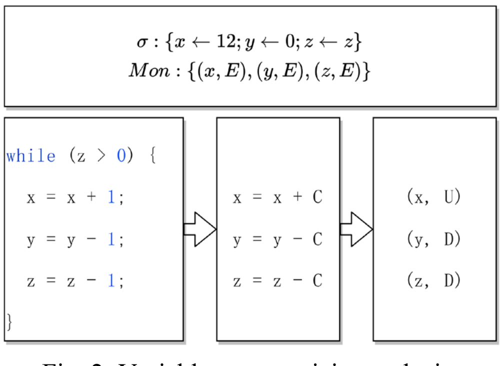
where the form $\exists x.p$ denotes the assertion p is satisfiable for some value of x. The meaning of P_o is that for all states of $Prog$, if there exists a state σ′ in which there exists a variable v_s or v_u with an assignment that makes $\neg Q_{so}$ or $\neg Q_{uo}$ true, then integer overflow occurs.

3. Loop Abstraction

We describe how to use loop abstraction techniques to abstract variables in a loop, how to obtain the program state after the loop, and how to use its results for integer overflow verification.

3.1. Abstraction of arithmetic expressions

Arithmetic expressions are one of useful expressions in programs. There are many types of arithmetic expressions, and the frequent arithmetic operations in C are addition, subtraction, multiplication, division, and modulo operations. Here we only consider arithmetic operations that may produce integer overflows, such as addition, subtraction, and multiplication. The primary purpose of arithmetic expression abstraction is to analyze the monotonicity of different variables and to analyze the range of different variables more precisely.



$\sigma : \{x \leftarrow 12; y \leftarrow 0; z \leftarrow z\}$
$Mon : \{(x, E), (y, E), (z, E)\}$
while (z > 0) {
x = x + 1;
y = y - 1;
z = z - 1;
}
x = x + C
y = y - C
z = z - C
(x, U)
(y, D)
(z, D)

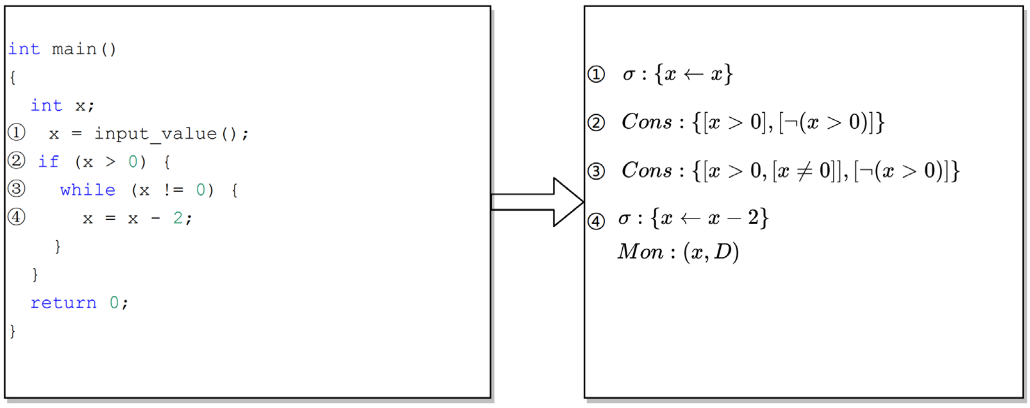
Fig. 2: Variable monotonicity analysis.

Fig. 2 represents a loop in which the variables x and y in the state σ are concrete values, and the variable z is a symbolic value. Mon indicates the set of recorded variable monotonicity, and E means that the variable monotonicity is not analyzed.

Suppose the concrete variable adds or subtracts a constant value. In that case, the trend of this variable can be either monotonically increasing(U) or monotonically decreasing(D) based on the constant value and the context. If a symbolic value is added to or subtracted from a constant value, then the trend of the variable can be interpreted in the same way. If two variables are calculated, the trend is judged based on the context and the analyzed trend. If the expression is too complex to be analyzed by the expression abstraction component, then the variable change range is judged as UNKNOW.

3.2. IF conditional abstraction

The primary purpose of conditional expression abstraction is to analyze the code logic in the two cases. Nested loops are possible in the IF expression, and the condition value directly affects the values of the variables in the loop. Consequently, it is desired that the analysis of IF expressions is separate. For IF expressions, we consider both cases where the branching condition is true and false. Firstly, we abstract the conditional constraints for both cases into logical formulas. And then, we consider adding the abstracted logical formulas to the set of variable constraints. Finally, the analyzed results are submitted to the WHILE conditional analysis component.



```
int main()
{
  int x;
① x = input_value();
② if (x > 0) {
③   while (x != 0) {
④     x = x - 2;
    }
  }
  return 0;
}
```

① $\sigma : \{x \leftarrow x\}$

② $Cons : \{[x > 0], [\neg(x > 0)]\}$

③ $Cons : \{[x > 0, [x \neq 0]], [\neg(x > 0)]\}$

④ $\sigma : \{x \leftarrow x - 2\}$

$Mon : (x, D)$

Fig. 3: Abstract results obtained by different expressions.

3.3. WHILE conditional abstraction

There are three purposes of the loop condition analysis. First, determine the possible times of loop executions based on the program state, the monotonicity of the variables and the loop condition expressions. Second, determine if there exists a variable that satisfies theorem (3), according to the monotonicity of the variables, the arithmetic expression in the loop body and the times of possible executions of the loop body. Third, form a logic assertion based on the previous conditional expression constraints, loop condition constraints and arithmetic expression constraints.

As shown in Fig. 3, we can obtain the following analytical results from the previous analysis: At position 1, it is clear that the variable x is a symbolic value through the Parser. Furthermore, this variable is in the loop condition after the analysis of Parser. The IF condition could be analyzed at position 2. The conditional expression is abstracted to obtain two constraints with the condition $true$ and $false$. Position 3 is the loop in the $ture$ branch of IF. The loop condition is abstracted and added to the corresponding constraint. Position 4 is an arithmetic expression. Depending on the content of this expression, the loop analyzer can update the program state and analyze the monotonicity of the variables.

The WHILE conditional abstraction component adopts the strategy of maximizing the loop count and calculates the loop count based on the monotonicity of the variables in the loop condition. If there is no upper or lower limit for the symbolic value in the loop condition, it is assumed that the loop will repeat as many times as possible. Moreover, based on the times of loop execution, the possible maximum or minimum values of each variable can be calculated and added to the constraints. Subsequently, the loop analyzer will compose all the previous constraints into a complete assertion, which is delivered to the Z3 SMT solver for solving. If the maximum value exists and is out of the range of the type, the verifier will assume that there is an integer overflow vulnerability in the program $Prog$, and vice versa is safe.

4. Evaluation

To verify the effectiveness of the proposed VeriOover, we use four datasets from SV-benchmarks: signedintegeroverflow-regression, termination-crafted, termination-crafted-lit, and termination-numeric. SV-benchmarks are the datasets used in competition on software verification. It contains the datasets for integer overflow. A C file in the datasets corresponds to a YML file. The YML file describes the properties of the program. We select the programs that contain the no-overflow property in the YML file as test input. This paper compares five tools, KLEE [19], VeriFuzz [17], Frama-C [16], CBMC [14], and 2LS [22], with VeriOover for automated formal verification of programs to detect integer overflow vulnerabilities. We specifies that the verification time for a single program must not exceed 60 seconds, and if the verification task can not be completed within the specified time, it will be considered as unable to detect the program and the result will be recorded as "Unknown".

As shown in Fig. 4, approximately 90.04% of the programs in the test set could be verified correctly by VeriOover. The complicated process of calculating variables in the test program led to judgment errors. The unknown result is mainly due to the presence of memory allocation functions such as malloc, which VeriOover can not analyze, or the program is overly complex and exceeding the abstraction capacity of VeriOover. Experiment results show that within the same time limit, VeriOover can correctly verify the most number of programs, has the fewest number of programs that can not be verified, and the number of programs that are correctly verified far exceeds that of other tools.

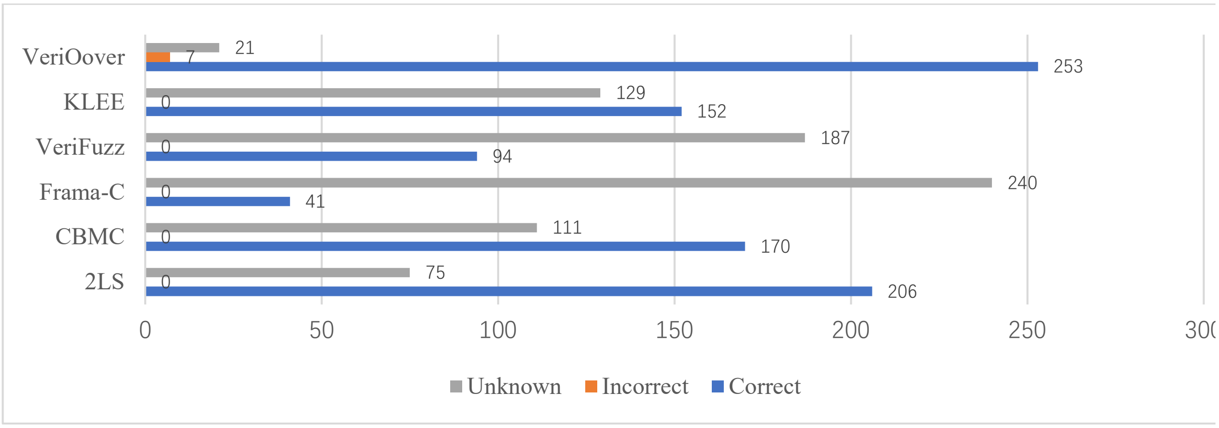


Fig. 4: verifier results comparison.

In the runtime testing for the verifier, we specified a maximum time limit of 60 seconds for each individual task for every tool. As shown in Table 1, the Verioover tool has an average runtime of 2.25 seconds, which is lower than the runtimes of Frama-c and VeriFuzz, which are 11.15 seconds and 4.04 seconds, respectively. VeriOover outperforms both KLEE and Frama-C in terms of maximum and minimum time consumption for a single verification task. Compared to the CBMC, 2LS, and KLEE tools, while VeriOover has a higher average runtime, it is capable of performing verification on a wider range of tasks. And VeriOover is implemented using the Python language, while CBMC and 2LS are developed using C/C++, which gives them some speed advantage at the language level.

Table 1: Verifier time consumption comparison

Tool	Average	Maximum	Minimum
VeriOover	2.25s	12.85s	0.30s
KLEE	1.72s	65.52	030s
VeriFuzz	4.04s	6.00s	3.68s
Frama-C	11.15s	50.45s	1.24s
CBMC	0.20s	3.80s	0.07s
2LS	0.59s	33.18s	0.05s

5. Conclusion

This paper presents a new loop abstraction method for detecting integer overflow vulnerabilities, which aims to alleviate the difficulty of analyzing loop structures in automatic formal verification. In this paper, based on the loop abstraction and symbolic execution methods, we first abstract the ranges of variables. Subsequently, assertions are generated based on all the abstracted expressions and the program properties. Finally, the SMT solver is considered for the verification of the properties of the program.

VeriOover can detect integer overflow vulnerabilities in C programs with the idea of formal verification. Automated verification provides a trade-off between reliability and usability than using interactive theorem prover such as Coq, Isabelle, etc. Compared to traditional software testing methods, the detection method based on formal verification can provide higher confidence and security. VeriOover is publicly available at https://github.com/Rw1nd/VeriOover.

6. References

[1] W.Dietz, P.Li, J.Regehr, and V.Adve, "Understanding integer roverflow in c/c++," *ACM Transactions on Software Engineering and Methodology (TOSEM*), vol. 25, no. 1, pp. 1–29, 2015.

[2] R. E. Bryant, O. David Richard, and O. David Richard, *Computer systems: a programmer's perspective*. Prentice Hall Upper Saddle River, 2003, vol. 2.

[3] G .Gong, "Exploiting heap corruption due to integer overflow in android libcutils," *Black Hat USA*, 2015.

[4] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[5] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis.* springer, 2015.

[6] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138.

[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[9] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[11] M. Chalupa and J. Strejcˇek, "Evaluation of program slicing in software verification," in *International Conference on Integrated Formal Methods*. Springer, 2019, pp. 101–119.

[12] A. Møller and M. I. Schwartzbach, "Static program analysis," *Notes. Feb*, 2012.

[13] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[14] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish et al., "sel4: Formal verification of an os kernel," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 207–220.

[16] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," Form. Asp. Comput., vol. 27, no. 3, p. 573–609, may 2015.

[17] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, "VeriFuzz: Program-aware fuzzing (competition contribution)," in Proc. TACAS (3), ser. LNCS 11429. Springer, 2019, pp. 244–249.

[18] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2016.

[19] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser.OSDI'08. USA:USENIXAssociation, 2008, p. 209–224.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[21] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications,"

ACM Trans. Comput. Syst., vol. 30, no. 1, feb 2012.

[22] V. Mal´ık, P. Schrammel, and T. Vojnar, “2ls: Heap analysis and memory safety (competition contribution),” in Proc. TACAS (2), ser. LNCS 12079. Springer, 2020, pp. 368–372.