# VTSE – Verification Tool based on Symbolic Execution

Nguyen Thi Van Anh*, Nguyen Thi Thuy, To Van Khanh*

*Faculty of Information Technology,*
*University of Engineering and Technology,*
*Vietnam National University, Hanoi, Vietnam*

**Abstract**

This paper gives an overview of modern symbolic execution techniques and presents a tool VTSE (Verification Tool based on Symbolic Execution) which allows users to verify some properties of C/C++ program based on symbolic execution technique. As two inputs including program's source code and user's assertion, VTSE reports whether user's assertion are always satisfied with the program. Results of experiments performed on two sets of benchmark which are Floats-cdfpl (SV-COMP) and Kratos (FBK-IRST) are relatively positive. As for the former, VTSE has a greater advantage in solving speed although not all the problems are solved. Moreover, VTSE is also able to verify problems in Kratos benchmark which have a large number lines of code with about 500 – 2000 LOC.

*Keywords*: Software verification, symbolic execution.

## 1. Introduction

Recent years, in response to the constantly changing technological environment, software quality assurance has evolved significantly and has become increasingly important, especially with classified safety- or business-critical software. In contrast of software testing which subjects the program to a series of tests with the intent of detecting potential software bugs, software verification ensure the safety of the whole software system. With the increasing complexity of system applications, using traditional technique likes model checking may easily lead to state space explosion even with SAT and SMT solvers scaling over the years [1].

This paper introduces the novel approach using Symbolic Execution [2] to verify some properties of C/C++ program. This method includes symbolizing programming source code into a first-order logic formula then incorporating with user's assertion to enter a SMT solver. A tool using the earlier technique is also presented – VTSE (Verification Tool using Symbolic Execution). VTSE has some applications in verifying C/C++ programs such as checking the return value of a program, verifying the pre-condition is sastified with the post-condition, verifying assertions provided by users or finding unreachable code.

Experiments are performed and compared to some well-known verifiers on two sets of benchmark which are Floats-cdfpl [3] and Kratos[4], which are used in SV-COMP 2017 and many other research publications.

The rest of this paperis organized as follows. Section 2 provides the background knowledge of this study. The architecture of VTSE is illustrated in section 3. Section 4

*Corresponding authors. E-mail: vananhnt97@gmail.com, khanhtv@vnu.edu.vn

describes the architecture and implemented technique of some related works. Experimental results are discussed in section 5. Finally, section 6 concludes this paper.

## 2. Background

### *2.1. Symbolic Execution*

Symbolic execution is a program analysis technique mostly used to automate software testing and software verification. The key idea of the technique is to use symbolic values as input instead of concrete data. It computes the result of program execution on symbolic states, which map variables to symbolic expressions. In concrete execution, a program is run on a specific input and a single control flow path is explored. Therefore, concrete execution in some cases can only under-approximate the analysis of the interested properties. On the contrary, symbolic execution simultaneously explore paths of the program under multiple inputs [5].

Symbolic execution can be divided into two approaches: dynamic symbolic execution (DSE) for testing and static symbolic execution (SSE) for verification. Dynamic approaches work by generating formula for certain path to test specific execution paths, while static approaches generates formulas over entire programs to verifying overall safety. Unlike DSE, SSE does not suffer from path explosion. All paths are encoded in a single formula that is then passed to the solver. In the end, a SMT solver (Satisfiability Modulo Theories solver) is used to generate test inputs or check whether there are any violations of the property on the execution paths.

### *2.2. Propositional logic*

Propositional logicis a branch of mathematical logic which studies the logical relationships between propositions. A proposition P is a collection of atoms or atomic formulas Pi. Each Pi has a truth value, which can be either True or False but never both.

An important form of proposition is conjunctive normal form. A compound statement is in conjunctive normal form if it is obtained by operating AND among variables (negation of variables included) connected with ORs

$$(p \vee \neg q) \wedge (\neg r \vee q) \wedge (\neg z \vee q)$$

### *2.3. First-order logic*

While propositional logic deals with simple declarative propositions, first-order logic additionally covers predicates and quantification. A predicate can be either True or False. Relationships between predicates can be stated using logical connectives ($\wedge$ for conjunction, $\vee$ for disjunction, $\rightarrow$ for implication, $\leftrightarrow$ for biconditional, $\neg$ for negation). Quantifiers ($\forall$ for universal quantifier and $\exists$ for existential quantifier) can be applied to variables in a formula.

$$(p \vee \neg q) \rightarrow (\neg r \vee q)$$

### *2.4. Satisfiability Modulo Theories and SMT solver*

Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory. There are now several powerful and sophisticated SMT solvers (e.g., Boolector [6], MathSAT5 [7] and Z3 [8]) which are being used in a rapidly expanding set of applications. Application areas currently include verification, equivalence checking, bounded and unbounded model checking, automated test case generation, … In the presented tool – VTSE, the SMT solvers are incorporated includes Z3 [8] and raSat [9].

## 3.VTSE Architecture

VTSE takes two inputs which are the program's source code and an assertion provided by user. Its output is a report on whether the user's assertion is satisfied.

After collecting the source code, VTSE symbolizes it using symbolic execution through a sequence of steps including generating

abstract syntax tree, building control flow graph, unwinding loops in the graph, then indexing variables in code to create a metaSMT formula which later will be transformed into a first-order logic formula. Finally, VTSE combines the source code's abstraction formula with user's assertion and enter the final formula into a SMT solver. Two candidate SMT solvers are Z3 and raSat.

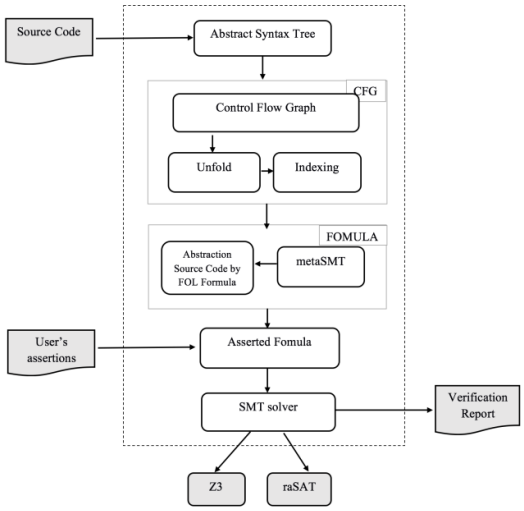



*Figure 1: VTSE Architecture*

### *3.1. Abstract Syntax Tree*

Abstract syntax tree [10] in Computer Science is a tree representation of the abstract syntax structure of source code with each node of the tree is a construct occurring in the source. Abstract syntax tree usually is the result of parser after translation and compiling. In VTSE, CDT (C/C++ Development Tool) is used to parse C/C++ source code to abstract syntax tree.

### *3.2.Control Flow Graph*

Control Flow Graph in VTSE is an one way directed graph including a collection of nodes, each node represents a component in the source code. Node types in CFG are the following:

*Plain node* represents a simple statement in source code like assignment, initialization, return statement.

*Decision node* contains condition in if-else, while-do, or do-while statement.

*Iteration node* contains an iterative statement and marks the appearance of the loop statement.

*Mark node* does not attach a statement but is used to set certain flags in the program such as if-else block beginning, if-else block ending.

**Building the Control Flow Graph**

Control flow graph is derived from abstract syntax tree of a function by iterating through tree.

For *compound statement*, algorithm creates a linked list of nodes corresponding with each statement in the compound.

For *branch (if-else, switch-case) statement*, a begin node is created to mark the beginning of if-else block of graph, then points to a DecisionNode with 2 outer links: one points to *then* block, one points to *else* block.

For *loop (for, while-do, do-while*), a begin node is created to mark the beginning of block of graph representing loop statement. After that, the begin node points to a DecisionNode with 2 outer links: one points to "then" block, one points to IterationNode - a node including the iteration statement.

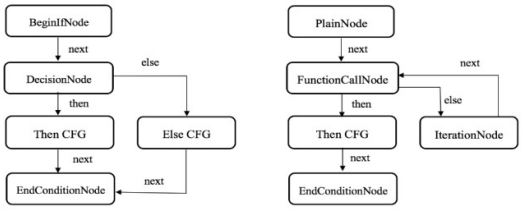



*Figure 2: Branch and loop CFG*

For *function call*, the algorithm creates a node including the parameters of the called function

For *go-to statement*, a GotoNode is linked to a corresponding LabelNode.

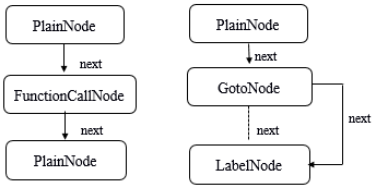



*Figure 3: Function call and Goto CFG*

### *3.3. Unwinding loops in Control Flow Graph*

To remove loops in control flow graph, iterating from start to end of graph, if BeginForNode or BeginWhileNode is caught, loop is unwinded into a certain number of if-else statement.

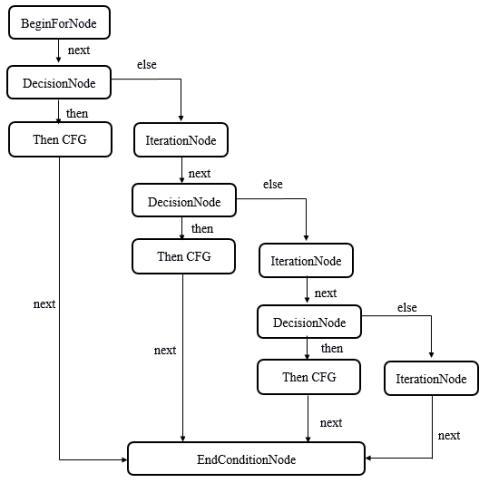



*Figure 4: Unwind loops in CFG*

If catching jump statement when iterating through graph, a new branch is replicated and linked to GotoNode.

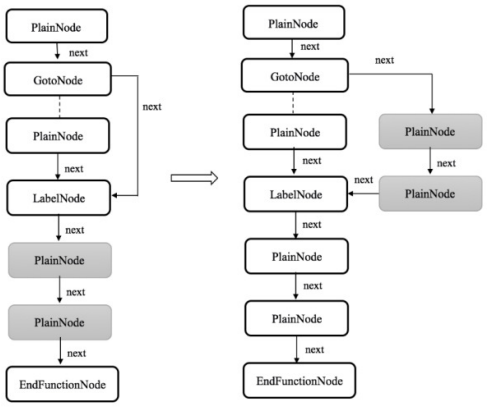



*Figure 5: Handle jump statements in CFG*

### *3.4. Indexing*

Because the input of SMT solver Z3 consists of first-order logic formula, a variable in Z3 can only be constant, not symbolic value. Therefore, to represent the change of variable's value, indexing is used to store all of variable states in program execution process.The initial index of a variable is -1. Each times variable's value is updated, its index is incremented.

**Syncing indexes**

A problem emerges when indexing is the uneven of variable's index in branching statement. To solve this, indexes in two branches of condition statement are synced using the below algorithm.

**Syncing Indexes Algorithm in VTSE**

**Input:** ConditionNode , Then - VarList, Else -VarList
**Output:** The SyncVarList

**1for** (with variable in the list)
**2** $i_1$ := index of v when variable is indexing in thenClause
**3** $i_2$ := index of v when variable is indexing in elseClause
**4if** ($i_1 < i_2$)
**5** Initialize SyncNode v_ $i_2$ = v_ $i_1$
**6** push SyncNode on the then clause
**7** Set the index value of the last variable in the ConditionNode is $i_2$
**8else if** ($i_2 < i_1$)
**9** Initialize SyncNode v_ $i_1$ = v_ $i_2$
**10** push SyncNode on the else clause
**11** Set the index value of the last variable in the ConditionNode is $i_1$

### *3.5. Generate first-order logic constraint formula*

Constraint formula is build following SMT solver's input format, as for Z3, the format is SMT-libv2. Each statement in source code is transformed into a first-order logic formula – a sub-formula. The constraint of the whole program is a conjunction of all sub-formulas.

*3.6. Incorporate with user's assertion to compute result through SMT solver*

Let assume $F_{abstraction}$ is the constraint formula created in the previous step, $F_{assertion}$ is the assertion provided by the user.

$$F_{final} = F_{absrtaction} \wedge \neg -F_{assertion}$$

$F_{final}$ is the final input of SMT solver. The result is either SAT or UNSAT.

If the result is SAT (satisfaction): A set of values exists to satisfy $F_{final}$. In this case, the tool will inform users the result is ***unsafe***, the assertion is violated and a counter example is provided.

If the result is UNSAT (unsatisfaction): Not exists any values to satisfy $F_{final}$. In this case, the result is **safe**, which means user's assertion is always true.

## 4. Related work

During the last decades, there are several tools designed to assure quality of programs by testing or verification such as KLEE [11], CBMC [12], 2LS [13]. Among them, KLEE is a symbolic execution tool which has been widely used in software testing in both academic and industry sides, while CBMC and 2LS are verification tools which are both holding high ranking in SV-COMP [14]. SV-COMP is an annual thorough comparative evaluation of fully-automatic software verifiers. Each verification task in the competition belongs in a certain category and consists of a C program and a property (reachability, memory safety, termination). SV-COMP 2017 had 32 participating verification systems from 12 countries.

*4.1. KLEE*

KLEE [11] is a symbolic execution tool that is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. When KLEE runs the program, it tries to explore every possible path. This is done by executing the program symbolically, i.e. tracking all constraints on inputs marked symbolic as each instruction is reached.

The queries issued by KLEE are branch and counter example queries. The former are issued when a branch is reached to decide whether then, else or both sides of the branch are followed. Counterexample queries are used to request a solution for the current path, e.g. when KLEE needs to generate a test case at the end of a path.

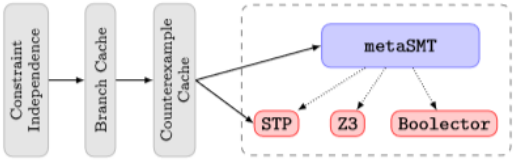



*Figure 6: Solver passes in KLEE [11]*

Before the metaSMT formula is created and entered a SMT/SAT solver, KLEE performs a series of constraint solving optimizations structured as a sequence of solver passes. The first pass in KLEE is the elimination of redundant constraints, which is called constraint independence. The other solver passes are concerned with caching. The branch cache stores the result of branch queries. The counterexample cache remembers constraint sets and their counterexamples if the set is satisfiable or a sentinel if the set is unsatisfiable.

Finally, when a path ends or an error is discovered, one can take all the constraints gathered along that path and ask the constraint solver for a concrete solution representing a test case that exercises the path. These test cases can be used to form high-coverage test suites, as well as to generate bug reports.

*4.2. CBMC*

CBMC (C Bounded Model Checking) [12] is a tool for the formal verification of ANSI-C programs using Bounded Model Checking. The tool supports almost all ANSI-C features

including pointer constructs, dynamic memory allocation, recursion, and float and double data types.

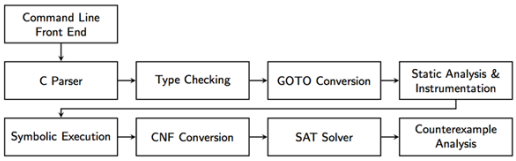



*Figure 7: Architecture of CBMC [12]*

In CBMC, the source code is symbolized into bit vector equations. The process has five steps:

- The source code is preprocessed.
- Loop constructs can be expressed using while statement, recursive function calls and goto statements. The while loops are un-winded by duplicating the loop body n times. After that, a unwinding assertion is added that assures that the program never requires more iterations.
- Loop in goto is un-winded using similar way.
- Function calls are expanded.
- The program is transform into static single assignment (SSA) form. The procedure produces two bit-vector equations: C (for the constraints) and P (for the property). In order to check the property, CBMC converts C ∧¬P into CNF.
- The property checking requires back end solvers.

In SV-COMP, CBMC took first place in 2015, third place in 2016 and 2017 in floats-cdfpl category.

### *4.3. 2LS*

2LS [13] is a static analysis and verification tool for C programs that can perform verification and refutation of assertions using an algorithm called kIkI (k-invariants and k-induction) [15] which combines bounded model checking, k-induction and invariant generation.

2LS performs the following main steps which are outlined in Figure 2.

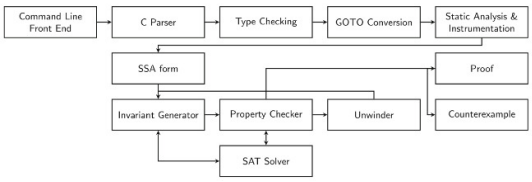



*Figure 8: Architecture of 2LS [13]*

*Front end.* The command line front end provides user with possible options and parameters, such as the bit-width. The C parser utilizes an C preprocessor (such as gcc) and parses a syntax tree from the source code. 2LS uses GOTO programs as an intermediate representation then as in CBMC, performs a light-weight static analysis resulting in a static call graph.

*Middle end.* The result of the static analysis is a static single assignment (SSA) form. Subsequently, 2LS refines this over-approximation by computing invariants.

*Backend.* Similar to CBMC, the SSA equation is translated into a CNF formula then a SAT solver is used for property check. If a property check is satisfiable, a human-readable counterexample is provided. Conversely, if the property check is unsatisfiable, the assertions have been proven.

In SV-COMP, 2LS took first place in 2016 and fourth place in 2017 in floats-cdfpl category.

## 5. Experimental results

The experiments are performed on a machine with a 2.20 GHz 64 bit Quad Core CPU Intel i5-5200U 2.2 GHz processor and 4GB of memory running Fedora 27. We set a 1500s timeout for the analysis of each subject. Two sets of benchmark are Floats-cdfpl [3] and Kratos [4]. Floats-cdfpl is a category of SV-COMP, including 40 verification tasks for checking programs with floating-point arithmetics. Kratos is a benchmark set provided by FBK-IRST, each problem usually has ~500-2000 LOC, therefore making many tools run

into path explosion problems. There are three possible answers for each problem [14]:

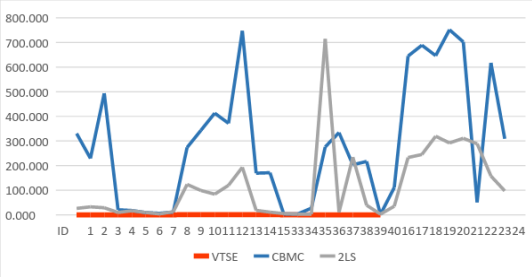
*Safe:*The specification is satisfied (i.e., there is no path that violates the specification).

*Unsafe*: The specification is violated (i.e., there exists a path that violates the specification) and a violation witness is produced.

*Timeout:*The tool cannot decide the problem or terminates by a tool crash, time-out, or out-of-memory.

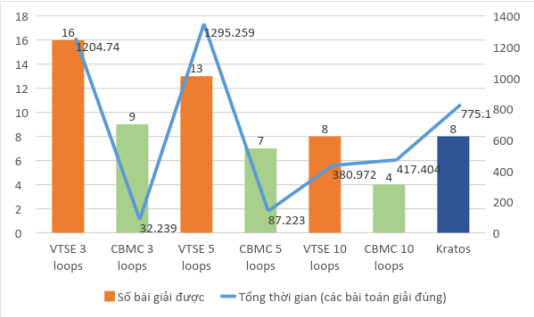
We compared VTSE with two C verifiers competing in SV-COMP 2017: CBMC-5.8 and 2LS-0.5, in terms of the accuracy and solving time.

**Table 1** compares the performance of VTSE, CBMC and 2LS on floats-cdfpl benchmark. In term of the number of correct results, VTSE is the only tool that cannot solve all 40 problems with 9 problems from *newton_3_1_*_unreach_call* to *newton_3_8_*_unreach_call*are timeout. However, in comparison of solving time, VTSE has better performance in remaining 31 problems than both CBMC and 2LS when the average time to solve those 31 problems of VTSE, CBMC and 2LS are 0.136s, 202.061 and 61.289s respectively. The illustration of result on floats-cdfpl benchmark is showed in **Graph 1**.




*Graph 1: Comparison on Floats-cdfpl benchmark*

Experiment on Kratos benchmark is illustrated on **Table 2** and **Graph 2**. The comparison is set between VTSE and CBMC, both are running default options and are fixed on the number of unwind loops. When verifying program with large lines of code, VTSE has a general positive impact both on running times and on the number of problems solved.




*Graph 2: Comparison on Kratos benchmark*

## 6. Conclusion

This paper describes a technique using symbolic execution to apply in software verification and presents a tool VTSE (Verification Tool using Symbolic Execution) which allows users to verify some properties of C/C++ program using the proposed method. VTSE has several applications such as checking the return value of a program, verifying the pre-condition is sastified with the post-condition, verifying assertions provided by users or finding unreachable code. To compare VTSE with current C verifiers, experiments are performed on two sets of benchmark which are Floats-cdfpl (SV-COMP) and Kratos (FBK-IRST). The results show VTSE can produce answers in relatively fast solving time and also handle programs with large number lines of code.

In the future, the development of the study will focus on constraint optimization and loop invariant generation. We also aim to expand the handling of other C language's features such as array, allocate memory, or pointer.

## Acknowledgement

This work was supported by the research project QG.16.32. We thank to Assoc. Dr. Pham Ngoc Hung, University of Engineering and Technologies, VNU Hanoi, for his value comments.

**Table 1**: Verication results of Floats-cdfpl benchmark

| ID | Verification task | LOC | VTSE | | CBMC | | 2LS | |
|---|---|---|---|---|---|---|---|---|
| | | | V | Time | V | Time | V | Time |
| 1 | newton_1_1_true_unreach_call | 48 | S | 0.032 | S | 330.324 | S | 26.052 |
| 2 | newton_1_2_true_unreach_call | 48 | S | 0.029 | S | 229.267 | S | 32.393 |
| 3 | newton_1_3_true_unreach_call | 48 | S | 0.019 | S | 493.696 | S | 28.681 |
| 4 | newton_1_4_false_unreach_call | 48 | U | 0.018 | U | 20.559 | U | 9.481 |
| 5 | newton_1_5_false_unreach_call | 48 | U | 0.024 | U | 17.465 | U | 16.517 |
| 6 | newton_1_6_false_unreach_call | 48 | U | 0.017 | U | 9.142 | U | 9.499 |
| 7 | newton_1_7_false_unreach_call | 48 | U | 0.027 | U | 6.162 | U | 3.602 |
| 8 | newton_1_8_false_unreach_call | 48 | U | 0.038 | U | 11.217 | U | 12.596 |
| 9 | newton_2_1_true_unreach_call | 48 | S | 0.56 | S | 273.607 | S | 123.449 |
| 10 | newton_2_2_true_unreach_call | 48 | S | 0.528 | S | 343.148 | S | 98.819 |
| 11 | newton_2_3_true_unreach_call | 48 | S | 0.541 | S | 412.942 | S | 83.836 |
| 12 | newton_2_4_true_unreach_call | 48 | S | 0.503 | S | 371.692 | S | 119.9 |
| 13 | newton_2_5_true_unreach_call | 48 | S | 0.515 | S | 747.501 | S | 193.09 |
| 14 | newton_2_6_false_unreach_call | 48 | U | 0.58 | U | 169.122 | U | 18.057 |
| 15 | newton_2_7_false_unreach_call | 48 | U | 0.518 | U | 171.063 | U | 10.827 |
| 16 | newton_2_8_false_unreach_call | 48 | T | 0.785 | U | 110.768 | U | 34.741 |
| 17 | newton_3_1_true_unreach_call | 48 | *T* | *1500* | *S* | *644.915* | *S* | *232.768* |
| 18 | newton_3_2_true_unreach_call | 48 | *T* | *1500* | *S* | *689.052* | *S* | *244.979* |
| 19 | newton_3_3_true_unreach_call | 48 | *T* | *1500* | *S* | *646.655* | *S* | *319.251* |
| 20 | newton_3_4_true_unreach_call | 48 | *T* | *1500* | *S* | *751.86* | *S* | *291.928* |
| 21 | newton_3_5_true_unreach_call | 48 | *T* | *1500* | *S* | *702.424* | *S* | *310.889* |
| 22 | newton_3_6_false_unreach_call | 48 | *T* | *1500* | *U* | *50.856* | *U* | *288.599* |
| 23 | newton_3_7_false_unreach_call | 48 | *T* | *1500* | *U* | *617.201* | *U* | *157.526* |
| 24 | newton_3_8_false_unreach_call | 48 | *T* | *1500* | *U* | *309.053* | *U* | *97.281* |
| 25 | sine_1_false_unreach_call | 32 | U | 0.021 | U | 1.484 | U | 1.783 |
| 26 | sine_2_false_unreach_call | 32 | U | 0.024 | U | 8.683 | U | 2.377 |
| 27 | sine_3_false_unreach_call | 32 | U | 0.024 | U | 6.171 | U | 1.985 |
| 28 | sine_4_true_unreach_call | 32 | S | 0.011 | S | 1249.95 | S | 70.837 |
| 29 | sine_5_true_unreach_call | 32 | S | 0.02 | S | 74.886 | S | 7.463 |
| 30 | sine_6_true_unreach_call | 32 | S | 0.02 | S | 31.956 | S | 6.343 |
| 31 | sine_7_true_unreach_call | 32 | S | 0.01 | S | 3.807 | S | 6.343 |
| 32 | sine_8_true_unreach_call | 32 | S | 0.02 | S | 214.917 | S | 4.808 |
| 33 | square_1_false_unreach_call | 33 | U | 0.012 | U | 3.268 | U | 4.923 |
| 34 | square_2_false_unreach_call | 33 | U | 0.012 | U | 3.162 | U | 2.749 |
| 35 | square_3_false_unreach_call | 33 | U | 0.022 | U | 27.049 | U | 2.779 |
| 36 | square_4_true_unreach_call | 33 | S | 0.02 | S | 275.889 | S | 714.668 |
| 37 | square_5_true_unreach_call | 33 | S | 0.012 | S | 334.017 | S | 10.394 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 38 | square_6_true_unreach_call | 33 | S | 0.011 | S | 203.2 | S | 234.158 |
| 39 | square_7_true_unreach_call | 33 | S | 0.01 | S | 216.98 | S | 38.928 |
| 40 | square_8_true_unreach_call | 33 | S | 0.02 | S | 1.568 | S | 2.635 |
| | Total time (for 31 solved problems) | | | **4.218** | | **6263.894** | | **1899.972** |
| | Average solving time (for 31 solved problems) | | | **0.136** | | **202.061** | | **61.289** |

**Table 2**: Verication results of Kratos benchmark

| **ID** | **Problems** | **V** | **LOC** | **3 loops** | | | | **5 loops** | | | | **10 loops** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | **VTSE** | | CBMC | | **VTSE** | | CBMC | | **VTSE** | | CBMC | |
| 1 | pc_sfifo_1 | S | 360 | S | 0,15 | S | 0,1 | S | 3,5 | S | 2,571 | S | 8,7 | S | 44,27 |
| 2 | pc_sfifo_2 | S | 465 | S | 0,04 | S | 0,1 | S | 1,9 | S | 5,298 | S | 3,5 | S | 84,18 |
| 3 | token_ring_1 | S | 459 | S | 0,11 | S | 1,85 | S | 3,092 | S | 5,371 | S | 8,7 | S | 28,85 |
| 4 | token_ring_2 | S | 582 | S | 0,11 | S | 1,16 | S | 5,321 | S | 4,934 | S | 80,5 | S | 212,9 |
| 5 | token_ring_3 | S | 705 | S | 0,46 | S | 1,11 | S | 9,152 | S | 9,422 | T | 1500 | O | - - |
| 6 | token_ring_4 | S | 828 | S | 1,09 | S | 1,51 | S | 20,454 | S | 14,5 | T | 1500 | O | - - |
| 7 | token_ring_5 | S | 951 | S | 1,05 | S | 2,26 | S | 97,326 | S | 22,94 | T | 1500 | O | - - |
| 8 | token_ring_10 | S | 1566 | S | 6,78 | S | 9,24 | S | 1078,66 | T | 1500 | O | - - | O | - - |
| 9 | token_ring_12 | S | 1812 | S | 43,79 | S | 14,93 | T | 1500 | O | - - | O | - - | O | - - |
| 10 | transmitter_1 | U | 437 | U | 0,96 | S* | 0,19 | U | 1,14 | S* | 2,32 | U | 2,95 | S* | 19,43 |
| 11 | transmitter_2 | U | 557 | U | 1,8 | S* | 0,42 | U | 4,844 | S* | 2,771 | U | 17,42 | S* | 27,81 |
| 12 | transmitter_3 | U | 679 | U | 5,6 | S* | 0,77 | U | 5,716 | S* | 3,261 | U | 111,31 | O | - - |
| 13 | transmitter_4 | U | 801 | U | 7,4 | S* | 1,31 | U | 9,917 | S* | 5,348 | U | 147,89 | O | - - |
| 14 | transmitter_5 | U | 923 | U | 13,9 | S* | 2,05 | U | 54,237 | S* | 8,487 | T | 1500 | O | - - |
| 15 | transmitter_11 | U | 1655 | U | 106,6 | S* | 12,5 | T | 1500 | O | - - | T | 1500 | O | - - |
| 16 | transmitter_12 | U | 1777 | U | 1015 | S* | 16,79 | T | 1500 | O | - - | T | 1500 | O | - - |
| Total time (for solved problems) | | | | | 1205 | | 32,3 | | 1295,26 | | 87,22 | | **380,97** | | 417,4 |
| Number of solved problems | | | | | **16** | | 9 | | **13** | | 7 | | **8** | | 4 |

V: Verification result
LOC: Lines of code
S: safe, U: unsafe, O: overflow, T: timeout, S*: incorrect results

**References**


[1] W. K. M. N. a. P. Z. Edmund M. Clarke, "Model Checking and the State Explosion Problem," Springer-Verlag, Berlin, 2012.

[2] E. C. D. C. D. D. a. I. F. Roberto Baldoni, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.51,* vol. 0, no. https://doi.org/0000001. 0000001, p. 37, 2018.

[3] SV-COMP, "Collection of Verification Tasks," Microsoft, 2013. [Online]. Available: https://github.com/sosy-lab/sv-benchmarks. [Accessed 25th May 2017].

[4] A. G. A. M. I. N. a. M. R. A. Cimatti, "Kratos - A software model checker for SystemC," Fondazione Bruno Kessler — Irst, 2008.

[5] P. G. K. S. N. T. S. K. C. S. P. W. V. Cristian Cadar, "Symbolic Execution for Software Testing in Practice – Preliminary Assessment," ICSE, Waikiki, Honolulu, HI, USA, 2011.

[6] M. P. a. A. B. Aina Niemetz, "Boolector at the SMT Competition," Johannes Kepler Univesity, Linz, Austria, 2017.

[7] N. E. Niklas Sorensson, "MiniSat v1.13 – A SAT Solver with Conflict-Clause," Chalmers University of Technology, Sweden, 2005.

[8] L. d. M. a. N. Bjørner, "Z3: An Efficient SMT Solver," Microsoft Research, Redmond, WA, USA, 2008.

[9] T. V. K. a. M. O. Vu Xuan Tung, "raSAT : an SMT Solver for Polynomial Constraints," in *International Joint Conference on Automated Reasoning*, Ishikawa, Japan, 2016.

[10] O. T. Thomas Kuhn, "Abstract Syntax Tree," Sun Microsystems, Inc, USA, 2006.

[11] D. D. D. E. ∗. Cristian Cadar, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Stanford University, San Diego, California, 2008.

[12] D. K. a. F. L. Edmund Clarke, "A Tool for Checking ANSI-C Programs," Carnegie Mellon University, TACAS, Berlin Heidelberg, 2004.

[13] P. S. a. D. Kroening, "2LS for Program Analysis (Competition Contribution)," TACAS. Springer, University of Oxford, 2017.

[14] D. Beyer, "Software Verification with Validation of Results (Report on SV-COMP 2017)," TACAS, Springer, LMU Munich, Germany, 2017.

[15] S. J. D. K. a. P. S. Martin Brain, "Safety Verification and Refutation by k-Invariants and k-Induction," Springer-Verlag, Berlin Heidelberg, 2015.