# A Static Analysis Tool with Optimizations for Reachability Determination

Yuexing Wang*†‡, Min Zhou*†‡, Yu Jiang*†‡, Xiaoyu Song§, Ming Gu*†‡, Jiaguang Sun*†‡
*Key Laboratory for Information System Security, Ministry of Education, China
†Tsinghua National Laboratory for Information Science and Technology (TNList), China
‡School of Software, Tsinghua University, China
§Electrical and Computer Engineering, Portland State University, USA

***Abstract*—To reduce the false positives of static analysis, many tools collect path constraints and integrate SMT solvers to filter unreachable execution paths. However, the accumulated calling and computing of SMT solvers are time and resource consuming.**

**This paper presents TsmartLW, an alternate static analysis tool in which we implement a path constraint solving engine to speed up reachability determination. Within the engine, typical types of constraint-patterns are firstly defined based on an empirical study of a large number of code repositories. For each pattern, a constraint solving algorithm is designed and implemented. For each program, the engine predicts the most suitable strategy and then applies the strategy to solve path constraints. The experimental results on some well-known benchmarks and real-world applications show that TsmartLW is faster than some state-of-the-art static analysis tools. For example, it is 1.32x faster than CPAchecker and our engine is 369x faster than SMT solvers in solving path constraints. The demo video is available at https://www.youtube.com/watch?v=5c3ARhFclHA&t=2s.**

***Index Terms*—Reachability determination, constraint pattern, path constraint solving**

## I. Introduction

Static program analysis (SPA) can determine run-time properties of programs automatically. The results of the technique may have errors that actually can not be reached. These errors are called false positives and they are generated because of the approximation nature of SPA [13].

Many static analysis tools filter false positives by collecting path constraints and using SMT solvers to get their satisfiability. Unreachable paths are dropped and false positives can be eliminated. For example, if we use a static analysis tool to analyze function $f$ in Fig. 1(a) and the analyzed path is $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$, the tool can collect path constraints (a-b>0 && b>a) and use SMT solvers to check their satisfiability. They are unsatisfiable and the error will not be reported. False positive is then eliminated and the path is dropped. The error in Fig. 1(b) is verified reachable.

Although SMT solvers can be used to filter false positives, the accumulated calling and computing time of SMT solvers can be long if SMT solvers are used too many times. CPAchecker [1] and CBMC [12] are two widely used open source verification tools for C and C++ programs that integrate SMT solvers, such as MathSAT5, Z3, Yices2. If the program to be analyzed has many branches, every branch contributes a set of path constraints and the calling and computing time of SMT solvers can occupy a large proportion of the execution time of CPAchecker or CBMC.

```
1 int f (int a, int b)
2   int c=a-b;
3   if(c>0)
4     c=b;
5     if(c>a)
6       ERROR;
7   else
8     c=a;
9   return c;
```
(a)

```
1 int g (int a, int b){
2   if(a+b>10){
3     if(a-b<5){
4       ERROR;
5     }
6   }
7   int c=a+b;
8   return c;
9 }
```
(b)

Fig. 1. Examples of using SMT

In this paper, we present TsmartLW, an optimized static analysis tool. A constraint solving engine (CSE) is designed and implemented for reachability determination. We define four constraint-patterns according to a preliminary empirical study. For each pattern, an especially designed constraint solving algorithm is presented and implemented. During the analysis process, the engine first predicts the most suitable strategy based on statistics. Then the strategy is applied to solve path constraints.

For evaluation, TsmartLW and CPAchecker are used to detect divide-by-zero errors in some commonly used benchmarks and real-world applications. The experimental results show that TsmartLW is faster than CPAchecker and the CSE is more efficient in solving path constraints compared with SMT solvers. On average, TsmartLW is 1.32x faster than CPAchecker and the CSE is 369x faster than SMT solvers in solving path constraints.

The rest of this paper is organized as follows. Section 2 introduces some static analysis tools and existing work on filtering false positives. The core components and algorithms of our tool are shown in Section 3. Section 4 presents the experimental results and Section 5 comes with conclusions.

## II. Related Work

To ensure the quality of the code, there are two directions, one is the qualified code generation techniques from the high-level model, and another is the code analysis and verification techniques. For the former, there are lots of tools for generating hardware and software codes from formal verified model [7]–[9]. For the latter, there are static analysis [1], [12] and dynamic analysis methods [5], [6], In this paper, we mainly focus on the static analysis tools for C code, such as CPAchecker [1] and CBMC [12]. CPAchecker integrates some SMT solvers.

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

We can choose whether to perform a constraint analysis to drop unsatisfiable paths using SMT solvers. CBMC can use SMT solvers to check the reachability of each error at the end of the analysis process.

Many works have been proposed for eliminating false positives of static analysis, by implementing more precise context analysis, by SMT solvers and so on. Huang *et al.* [4] and Chess *et al.* [2] believe the scope of the analysis is important because it determines the amount of context the tools consider. Kim *et al.* [11], Junker *et al.* [10] and Cordeiro *et al.* [3] use SMT solvers like CVC3, Boolector, Z3 to drop unreachable paths and filter false positives. Moreover, we can also use some techniques to enhance SMT solvers [15] [14].

Different from those works, we implement a static analysis tool TsmartLW and design a constraint solving engine for reachability determination to drop unreachable paths and filter false positives.

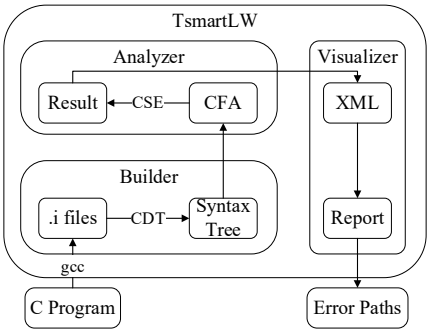


Fig. 2. Architecture and workflow of TsmartLW

## III. Design of the Tool

As presented in Fig. 2, TsmartLW contains three kernel components: *Builder*, *Analyzer* and *Visualizer*. The input of TsmartLW is the analyzed C program and the output is the error paths of the program.

*Builder* is used to preprocess the program to be analyzed. It parses a single .c file to a .i file using gcc. For real world applications, Builder requires them to contain makefiles. For each executable module of an application, Builder can capture the files that the module needs and parse them to .i files using gcc. If an application consists of many executable modules, each of them is captured to generate a task and each task will be analyzed separately by Analyzer. After the program is transformed to .i files, CDT is used to generate syntax tree.

*Analyzer* analyzes programs and outputs the results. It constructs control-flow automatons (CFA) of programs first using syntax trees. Then it analyzes programs based on CFAs. Going through all the possible edges from a node to another (to simulate an operation) iteratively, it can check all the possible execution paths of programs. The analyzer can find divided-by-zero errors. During the analysis process, we propose an optimized method for reachability determination. We use a CSE instead of SMT solvers to solve path constraints.

*Visualizer* shows all the errors to users. It transforms the result of Analyzer to XML format and generates a report. Then

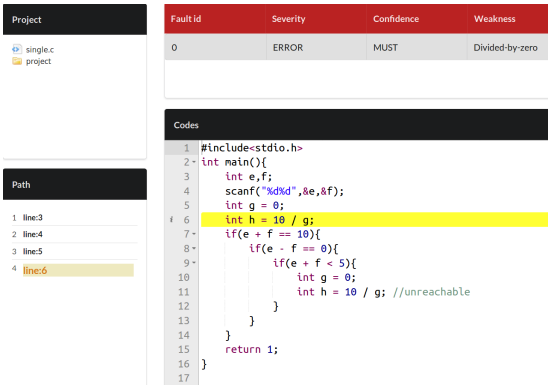


Fig. 3. The resulting interface of TsmartLW

the report is displayed in web form. All the errors and their paths in the source code are presented.

The builder and visualizer are implemented by ourselves and the analyzer is implemented based on CPAchecker. The resulting interface of TsmartLW is shown in Figure. 3. All the errors and their paths are presented.

In the next two parts, we present constraint-patterns and their corresponding constraint solving algorithms first and then show how our CSE utilizes the patterns and algorithms.

### A. Definitions of Patterns and Algorithms

This section gives the definitions of patterns and algorithms. The results of an empirical study are also presented. We define four kinds of constraint-patterns and the reasonableness of the four patterns can be seen from the empirical study. For each pattern, an especially designed constraint solving algorithm is presented. Before these concepts are introduced, we explain some basic definitions in Table I.

*1) Patterns and Their Corresponding Algorithms:* We define four kinds of constraint-patterns, SingleSymbolicEquality(SSE), SingleSymbolicInequality(SSI), MultiSymbolicSingleFunction(MSS) and MultiSymblicMultiFunction(MSM). We also design four corresponding algorithms, Substitution, Linear, LocalFrame and FullPath. These patterns and algorithms are defined below.

**SSE pattern and Substitution algorithm**: The constraints which have SSE pattern should satisfy the two properties.

1. $|SymVar(c)| <= 1$
2. Op(c) $\in \{=, \neq\}$

The corresponding constraint solving algorithm is Substitution. Suppose $C$ is a set of constraints and its pattern is SSE. There is at most one variable in each constraint and the operator is equal or not equal. Thus we can solve every constraint easily and record the variable-value tuples. If we meet a new constraint, variables in it can be substituted by values and we can get its satisfiability easily.

**SSI pattern and Linear algorithm**: Constraints of pattern $SSI$ should also satisfy some requirements.

1. $|SymVar(c)| <= 1$
2. Op(c) $\in \{<, \leq\}$

TABLE I
BASIC DEFINITIONS.

| Definition | Explanation |
|---|---|
| Constraint | $Constraint$ is an assumption statement with all its variables whose real values can be obtained from context substituted by their values. |
| Pattern | $Pattern$ describes the structural information about a constraint or a set of constraints. |
| Priority | $Priority$ denotes the complexity of a pattern. The pattern of a set of constraints is determined by the constraint which has the highest priority. |
| Global | $Global$ is the set which contains all the global variables appearing in the program to be analyzed. |
| Param | $Param_F$ is the set of all parameters of function $F$. |
| Ret | $Ret_F$ is the set of variables that are assigned returned values in function $F$. Returned values are the return values of other functions that are called in $F$. For example, if $F$ contains a statement like $int\ a\ =\ g()$, then $a$ is included in $Ret_F$. |
| SymVar | $SymVar(c)$ is the set that contains all the symbolic variables in constraint $c$. |
| Op | $Op(c)$ represents the relational operator in $c$ |
| Type | $Type(c)$ represents the type of constraint $c$ (equal, less, LessOrEqual). |

The corresponding algorithm is Linear. If the pattern of $C$ is SSI, it means all the constraints in $C$ are of pattern SSE or SSI. Each constraint has one variable and the operator is equal, not equal, less or LessOrEqual. The relations between variables in different constraints are linear. It is easy to get their satisfiability using their relations.

There are two reasons that make $Substitution$ and $Linear$ efficient in solving path constraints.

1. SMT solvers are integrated as external tools. The calling time for SMT solvers can be long if SMT solvers are called too many times. $Substitution$ and $Linear$ can be implemented inside these tools and the calling time can be saved.

2. $Substitution$ and $Linear$ are more targeted.

**MSS pattern and LocalFrame algorithm**: Let $f_c$ be the function from which the constraint c is extracted. To verify if c is of pattern $MSS$, we should check if $c$ satisfies the following requirements.

1. $|SymVar(c)| > 1$
2. $\forall v \in SymVar(c)$,
$v \notin Global\ \wedge\ v \notin Param_{f_c}\ \wedge\ v \notin Ret_{f_c}$

LocalFrame is used to solve constraints of pattern MSS. If the pattern of $C$ is MSS, SMT solvers should be used with some optimizations. If the function $f$ is being analyzed, the pattern of $C$ is MSS if there is no symbolic value that belongs to global variables, parameters of $f$ and variables that are assigned returned values in $f$. In this case, we can use only constraints in $f$ to determine the satisfiability of all the constraints. It is obvious that if we drop some constraints which are not in $f$, we can speed up SMT solvers.

**MSM pattern and FullPath algorithm**: Constraints of the pattern $MSM$ should satisfy the following two properties.

1. $|SymVar(c)| > 1$
2. $\exists v \in SymVar(c)$,
$v \in Global\ \vee\ v \in Param_{f_c}\ \vee\ v \in Ret_{f_c}$

FullPath is chosen if we can not use any other algorithm or even adapt some optimizations. We have to collect all path constraints and use an SMT solver to get their satisfiability.

Note that $PR$ is the priority function and $PR(SSE)$ = 1, $PR(SSI)$ = 2, $PR(MSS)$ = 3, $PR(MSM)$ = 4.

*2) Empirical Study:* To state the reasonableness of the four patterns, some benchmarks and real-world applications were analyzed and the number of constraints for each pattern is recorded. We select ten programs from five directories of sv-benchmarks (The benchmark of SV-COMP) randomly and each directory contributes two programs. They have 233K lines of C code in total. For real-world applications, we choose $grep, gzip, the\ silver\ searcher$, and $vim$.

TABLE II
THE NUMBER OF CONSTRAINTS FOR DIFFERENT PATTERNS.

| Program | SSE | SSI | MSS | MSM | Total |
|---|---|---|---|---|---|
| array1 | 8510 | 0 | 0 | 0 | 8510 |
| array2 | 9982 | 0 | 0 | 0 | 9982 |
| unreach1 | 4655 | 1460 | 2308 | 0 | 8423 |
| unreach2 | 5457 | 1603 | 1188 | 0 | 8248 |
| point1 | 2462 | 64 | 0 | 0 | 2526 |
| point2 | 12446 | 0 | 0 | 0 | 12446 |
| driver1 | 376 | 1 | 0 | 0 | 377 |
| driver2 | 517 | 0 | 0 | 0 | 517 |
| main1 | 4557 | 808 | 270 | 128 | 5763 |
| main2 | 3560 | 11149 | 50 | 0 | 14759 |
| grep | 16812 | 2266 | 3951 | 49 | 23078 |
| gzip | 5667 | 1450 | 1829 | 54 | 9000 |
| searcher | 7806 | 2170 | 47 | 0 | 10023 |
| vim | 306424 | 37038 | 46116 | 1188 | 390766 |
| Total | 389231 | 58009 | 55759 | 1419 | 504418 |
| Proportion | 77.17% | 11.50% | 11.05% | 0.28% | 100% |

The results are shown in Table II. The first column shows the programs and the next four columns present the number of constraints for each pattern. The last column is the total number of constraints that appear in the program. As we can see, 77.17% of all the constraints in these programs are of pattern SSE. SSI(11.5%) and MSS(11.05%) also occupy a large proportion. Only 0.28% of all the constraints have pattern MSM. The results prove the reasonableness of the four patterns and mean that most of the constraints can be solved by easier ways. Thus our CSE is worth being applied.

### B. Constraint Solving Engine

We implement the four algorithms in our CSE which is used in Analyzer to solve path constraints and show how our CSE applies the algorithms based on constraint-patterns. TsmartLW can perform a static analysis on C programs. Source files of the program to be analyzed are transformed to CFA first. Then the analysis process begins and our CSE is used to replace SMT solvers. During the analysis process, the engine predicts the most suitable strategy first based on statistics. Then the strategy is applied to solve path constraints.

Algorithm 1 shows how our CSE applies the algorithms. Source files are transformed to CFA. The predict function

$ST$ is used to predict the most suitable transfer function $f$ and initial constraint solving algorithm $a$. During the analysis process, if we meet an assumption edge, we can get a new constraint $c$. $c$ is added to path constraints $C$ and its pattern is used to update the current pattern and algorithm. Then the updated algorithm is used to check the satisfiability of $C$. If the constraints are unsatisfiable, the path is dropped. More details about the transfer function and predict function are listed below.

**Algorithm 1** Apply constraint solving algorithms

```
1: procedure APPLY ALGORITHMS
2:     CFA ← source files
3:     f, a ← ST(CFA)
4:     while analysis process continues do
5:         e ∈ CFA is being analyzed
6:         if e is an AssumptionEdge then
7:             c ← get constraint from e
8:             add(C, c)
9:             p' ← PA(c)
10:            if ⟨p, a⟩, PA(c), ⟨p', a'⟩ ∈ f then
11:                ⟨p, a⟩ ← ⟨p', a'⟩
12:                a → satisfiability checking
13:            end if
14:            if C → unsat then
15:                current path → drop
16:            end if
17:        end if
18:    end while
19: end procedure
```

*1) Transfer Function:* $f$ belongs to transfer function set $T$. For each $\rightsquigarrow \in T$, $\rightsquigarrow$ assigns each tuple $\langle p, a\rangle$ a new tuple $\langle p', a'\rangle$ and each transfer is labeled with $PA(c)$. $PA$ can get the pattern of a constraint or a set of constraints and $c$ is a constraint. If $(\langle p, a\rangle, PA(c), \langle p', a'\rangle) \in \rightsquigarrow$, the relation is denoted as $\langle p, a\rangle \overset{PA(c)}{\rightsquigarrow} \langle p', a'\rangle$. The relation means $PA(c) = p'$ and the current pattern and algorithm are changed to $p'$ $and$ $a'$.

We implement two transfer functions $\rightsquigarrow_{single}$ and $\rightsquigarrow_{standard}$ in our tool. $\rightsquigarrow_{single}$ means we should use just one constraint solving algorithm during the analysis process while $\rightsquigarrow_{standard}$ means we should adjust the constraint solving algorithm according to the pattern of current path constraints.

If $\rightsquigarrow_{single}$ is chosen, we should use just one constraint solving algorithm during the analysis process. Suppose $a$ is the initial algorithm, it should be used until the analysis process ends. We may treat unreachable paths as reachable and we may get some false positives because $a$ can not solve constraints whose pattern's priority is higher than a' corresponding pattern's priority. To solve this problem, we can mark all the errors and collect their path constraints. Then false positives can be filtered using SMT solvers at the end of the analysis process.

If $\rightsquigarrow_{standard}$ is chosen, we should use different algorithms for different path constraints based on their patterns. Thus we can always get the right answer and we do not need to filter false positives at the end of the analysis process. The workflow of standard method is presented in Figure. 4.

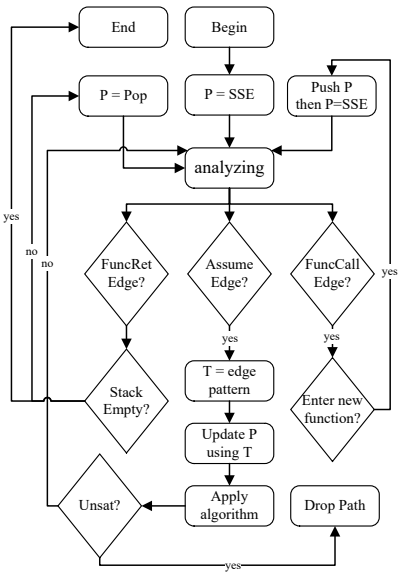


Fig. 4. How the pattern changes along the path.

We enter main function and set the pattern $P$ of current path constraints to SSE. For each path, these steps are performed.

- The analysis process continues until we meet a function return edge, an assumption edge or a function call edge.
- If we meet a function call edge, it means that we enter a new function. So we store the current $P$ and reset $P$ to SSE. Then we goto the first step.
- If an assumption edge is met, we can get a constraint from the edge and we add the constraint to path constraints. If the constraint's pattern is $T$. We use $T$ to update $P$. The algorithm should also be updated. After adding the constraint, if the new path constraints are unsatisfiable, we drop this path. Otherwise, we goto the first step.
- If a function return edge is met, it means that we leave the current function. We restore $P$ from stack and goto the first step.
- The analysis process of current path ends as soon as the return edge of main is met.

*2) Predict Function:* The function $ST$ in Algorithm 1 determines which transfer function should be used and $ST \subseteq CFA \times T$. Before the analysis process begins, we do not know which function is the most suitable. Since constraints can only be produced in the analysis process, we can not make a decision using constraints. Thus we propose a strategy to predict the most suitable function based on $CFA$ and statistics. Machine learning techniques are promising in predicting and these techniques can also be applied. After we get the most suitable transfer function, we can use it in the analysis process.

In our implementation, we use a simple method to predict the most suitable transfer function. We analyze many programs first and collect data that can influence our choice as statistics. Based on statistics, when analyzing new programs, we can predict the most suitable function.

To get statistics, we should analyze lots of programs first and record the number of each kind of constraint(Equal, Less, LessOrEqual) for different patterns (SSE, SSI, MSS, MSM). For example, the number of equal constraints that has SSE pattern is $E_{sse}$, the number of other three patterns are $E_{ssi}$, $E_{mss}$ and $E_{msm}$. For less constraints, the numbers are $L_{sse}$, $L_{ssi}$... and for LessOrEqual constraints, the numbers are $LE_{sse}$, $LE_{ssi}$... The data is treated as statistics for further use.

When a new program is analyzed, we can choose transfer function based on CFA and statistics. For equal constraints, if we get $E_{sse}$, $E_{ssi}$, $E_{mss}$, $E_{msm}$, then we believe that a new equal constraint has a possibility of

$$P_{equal}(SSE) = \frac{E_{sse}}{E_{sse} + E_{ssi} + E_{mss} + E_{msm}} \quad (1)$$

to has pattern SSE. The possibility of other patterns are also calculated in that way. The general formula is

$$P_{*}(PType) = \frac{*_{PType}}{*_{sse} + *_{ssi} + *_{mss} + *_{msm}} \quad (2)$$

* is the type of constraints (Equal, Less, LessOrEqual) while $PType$ is a kind of pattern. For constraint of type *, we can use formula(2) to get its possibility to has pattern $PType$.

Scores $SSEScore$, $SSIScore$, $MSSScore$, $MSMScore$ corresponding to the four patterns are defined. We should traverse the program's CFA first. If an AssumptionEdge is met, we identify its type (equal, less, LessOrEqual). If its type is equal. Then the edge has a probability of $P_{equal}(SSE)$ to has pattern SSE. Thus we believe that this edge increases SSEScore by $P_{equal}(SSE)$. $P_{less}(SSE)$ and $P_{lessORequal}(SSE)$ do the same. So we can use (3) to get SSEScore.

$$SSEScore = \sum_{e=AssumptionEdge}^{CFA} P_{type(e)}(SSE) \quad (3)$$

Others are calculated the same way. The general formula is

$$PTypeScore = \sum_{e=AssumptionEdge}^{CFA} P_{type(e)}(PType) \quad (4)$$

$PType$ denotes the pattern which you want to get its score.

We normalize the scores first and set four bounds empirically. Then the most suitable function $\rightsquigarrow = ST(S)$ is defined below. Note that if $\rightsquigarrow_{single}$ is chosen, the initial algorithm is determined by the pattern whose score crosses the threshold. Otherwise, we use $Substitution$ as the initial algorithm.

$$\mathrm{ST(S)} = \begin{cases} \rightsquigarrow_{single}, & if\ one\ score\ crosses\ threshold \\ \rightsquigarrow_{standard}, & otherwise \end{cases}$$

## IV. Experimental Results

In this section, the benchmarks and the real-world applications mentioned in section III.A are analyzed by TsmartLW and CPAchecker and the results are compared in Table III and Table IV. TsmartLW and CPAchecker are used to perform an inter-procedural analysis on benchmarks and an intra-procedural analysis on real-world applications. During the analysis process, TsmartLW uses our CSE to solve path constraints while CPAchecker uses SMT solvers. The time used for satisfiability checking and the total execution time of TsmartLW and CPAchecker are recorded.

Table III shows the time used for satisfiability checking of the two tools. Column M5 means CPAchecker uses MathSAT5 to solve path constraints and Z3 means CPAchecker uses Z3 solver. In the third to the ninth columns, TsmartLW is used to analyze the program and our CSE is applied to solve path constraints. Substitution, Linear, LocalFrame(M5) and LocalFrame(Z3) mean $\rightsquigarrow_{single}$ is applied and the initial constraint solving algorithm is Substitution, Linear, LocalFrame with MathSAT5 (LocalFrame and Standard need SMT solvers) and LocalFrame with Z3. Standard(M5) and Standard(Z3) mean $\rightsquigarrow_{standard}$ is used and MathSAT5 and Z3 are chosen. In the last column, we use our CSE to predict transfer function first and then apply the function in the analysis process. As we can see, Substitution, Linear and Standard(Z3) need the least time for satisfiability checking. On average, our CSE is 368.8x faster than SMT solvers. For example, Z3 needs 1688.4s for vim and our CSE costs only 3.1s. There are two reasons that make our CSE more efficient than SMT solvers in solving path constraints. The four algorithms in CSE are more targeted. Besides, SMT solvers are usually integrated as external tools. The calling time for SMT solvers can be very long if SMT solvers are called many times. CSE can be implemented inside these tools and the calling time can be saved.

Table IV shows the execution time of the two tools. TsmartLW using Substitution, Linear or Standard(Z3) has the least execution time and CPAchecker needs much more time for most programs except program main1(we do not lose much). On average, TsmartLW is 1.3167x faster than CPAchecker. For example, CPAchecker needs 7938s to analyze vim and TsmartLW only costs 3557.5s. TsmartLW is faster because it saves much time in satisfiability checking. Moreover, our CSE can always predict the best or near best strategies (transfer function and initial algorithm).

## V. Conclusion

This paper proposes TsmartLW, an optimized static analysis tool. TsmartLW can perform inter-procedural or intra-procedural analysis on programs to find divide-by-zero errors. During the analysis process, our CSE is applied for satisfiability checking. TsmartLW is compared with CPAchecker, a state-of-the-art static analysis tool. We use the two tools to perform an inter-procedural analysis on benchmarks and an intra-procedural analysis on real-world applications. The results reveal that TsmartLW is faster than CPAchecker.


## Acknowledgment

This research is sponsored in part by NSFC Program (No. 91218302, No. 61527812, 61402248), National Science and Technology Major Project (No. 2016ZX01038101), MIIT IT funds (Research and application of TCN key technologies ) of China, and The National Key Technology R&D Program (No. 2015BAG14B01-02).

The authors would like to thank Zuxing Gu for his advice.

TABLE III
THE TIME USED FOR SATISFIABILITY CHECKING BY CPACHECKER AND TSMARTLW.

| Program | M5 | Z3 | Substitution | Linear | LocalFrame(M5) | LocalFrame(Z3) | Standard(M5) | Standard(Z3) | Predict |
|---|---|---|---|---|---|---|---|---|---|
| array1 | 83.1s | 42.9s | **129ms** | 164ms | 63.7s | 61.4s | 227ms | 189ms | 129ms |
| array2 | 93.5s | 53.3s | **151ms** | 167ms | 73.7s | 77.3s | 256ms | 199ms | 151ms |
| unreach1 | 130.7s | 62.6s | **153ms** | 188ms | 78.2s | 75.9s | 73.5s | 56.4s | 153ms |
| unreach2 | 132.7s | 55.8s | **149ms** | 179ms | 105.4s | 81.6s | 84.6s | 50.3s | 179ms |
| point1 | 10.9s | 10.3s | 81ms | **74ms** | 10.4s | 15.9s | 142ms | 100ms | 81ms |
| point2 | 119.2s | 68.6s | 234ms | **192ms** | 123.6s | 96.8s | 263ms | **192ms** | 192ms |
| driver1 | 11.4s | 4.4s | **25ms** | 26ms | 4.5s | 4.4s | 41ms | 30ms | 25ms |
| driver2 | 492.3s | 88.8s | 66ms | **64ms** | 513.4s | 113.0s | 116ms | 99ms | 66ms |
| main1 | 49.9s | 40.6s | **187ms** | 199ms | 120.1s | 120.8s | 10.0s | 8.6s | 187ms |
| main2 | OOM[1] | 103.6 | 229ms | **214ms** | 220.4s | 133.4s | 1.1s | 0.9s | 229ms |
| grep | 44.3s | 22.7s | 435ms | **205ms** | -[2] | - | 49.4s | 40.2s | 205ms |
| gzip | 22.2s | 17.6s | 172ms | **131ms** | - | - | 9.8s | 12.9s | 131ms |
| seracher | 7.8s | 8.0s | **52ms** | 82ms | - | - | 1.2s | 1.7s | 82ms |
| vim | OOM | 1688.4s | 3.2s | **3.1s** | - | - | 788.6s | 685.3s | 3.1s |

[1] OOM means out of memory.
[2] An intra-procedural analysis is performed on real-world applications and there is no need to apply algorithm LocalFrame.
[3] We set four bounds 0.7, 0.2, 0.3, 0.1 for SSE, SSI, MSS, MSM as we have said in the last paragraph of III.C.a and they are used for prediction.

TABLE IV
THE EXECUTION TIME OF CPACHECKER AND TSMARTLW (SECONDS).

| Program | M5 | Z3 | Substitution | Linear | LocalFrame(M5) | LocalFrame(Z3) | Standard(M5) | Standard(Z3) | Predict |
|---|---|---|---|---|---|---|---|---|---|
| array1 | 184.9 | 119.7 | 54.2 | **52.8** | 141.9 | 133.2 | 66.5 | 53.8 | 54.2 |
| array2 | 184.8 | 136.0 | 55.6 | **55.2** | 157.1 | 155.7 | 67.7 | 56.0 | 55.6 |
| unreach1 | 517.8 | 460.8 | 553.6 | 546.0 | 458.1 | 429.3 | 504.3 | **428.6** | 553.6 |
| unreach2 | 336.9 | 248.8 | **186.3** | 187.6 | 307.1 | 275.8 | 306.3 | 246.0 | 187.6 |
| point1 | 295.3 | 304.1 | 281.3 | 271.1 | 291.7 | 287.9 | 323.1 | **270.4** | 281.3 |
| point2 | 1125.6 | 1097.2 | 880.0 | 870.3 | 1225.9 | 1025.1 | 998.6 | **853.0** | 870.3 |
| driver1 | 138.7 | 133.5 | **120.0** | 127.1 | 158.6 | 134.2 | 155.6 | 126.1 | 120.0 |
| driver2 | 526.7 | 124.0 | 34.6 | **33.1** | 561.9 | 154.9 | 43.6 | 36.0 | 34.6 |
| main1 | 150.2 | **121.8** | 136.6 | 138.7 | 296.2 | 266.6 | 176.2 | 147.2 | 136.6 |
| main2 | OOM | 237.4 | 153.9 | **150.8** | 415.2 | 284.2 | 185.0 | 158.7 | 153.9 |
| grep | 527.5 | 297.9 | **124.9** | 125.1 | - | - | 185.8 | 175.6 | 125.1 |
| gzip | 82.3 | 76.8 | 57.0 | **55.5** | - | - | 69.3 | 68.1 | 55.5 |
| seracher | 48.6 | 48.2 | 42.8 | 42.5 | - | - | 43.9 | **40.2** | 42.5 |
| vim | OOM | 7938.0 | 3580.6 | **3557.5** | - | - | 6440.8 | 6743.5 | 3557.5 |

[1] The time used for verifying marked errors generated by applying single algorithms is included in execution time.

## REFERENCES


[1] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518. Springer, 2007.

[2] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[3] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.

[4] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[5] Yu Jiang, Han Liu, Hui Kong, Rui Wang, Mohammad Hosseini, Jiaguang Sun, and Lui Sha. Use runtime verification to improve the quality of medical care practice. In *2016 38th ACM International Conference on Software Engineering(ICSE)*. ACM, 2016.

[6] Yu Jiang, Houbing Song, Rui Wang, Ming Gu, Jiaguang Sun, and Lui Sha. Data-centered runtime verification of wireless medical cyber-physical system. *IEEE Transactions on Industrial Informatics*, 2016.

[7] Yu Jiang, Yixiao Yang, Han Liu, Hui Kong, Ming Gu, Jiaguang Sun, and Lui Sha. From stateflow simulation to verified implementation: A verification approach and a real-time train controller design. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[8] Yu Jiang, Hehua Zhang, Zonghui Li, Yangdong Deng, Xiaoyu Song, Ming Gu, and Jiaguang Sun. Design and optimization of multiclocked embedded systems using formal techniques. *IEEE transactions on industrial electronics*, 62(2):1270–1278, 2015.

[9] Yu Jiang, Hehua Zhang, Huafeng Zhang, Han Liu, Xiaoyu Song, Ming Gu, and Jiaguang Sun. Design of mixed synchronous/asynchronous systems with multiple clocks. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2220–2232, 2015.

[10] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. Smt-based false positive elimination in static program analysis. In *International Conference on Formal Engineering Methods*, pages 316–331. Springer, 2012.

[11] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. Filtering false alarms of buffer overflow analysis using smt solvers. *Information and Software Technology*, 52(2):210–219, 2010.

[12] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[13] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing model checking and static program analysis: A case study in error detection approaches. *Proceedings of SSV*, 2010.

[14] Min Zhou, Fei He, Xiaoyu Song, Shi He, Gangyi Chen, and Ming Gu. Estimating the volume of solution space for satisfiability modulo linear real arithmetic. *Theory of Computing Systems*, 56(2):347–371, 2015.

[15] Min Zhou, Fei He, Bow-Yaw Wang, Ming Gu, and Jiaguang Sun. Array theory of bounded elements and its applications. *Journal of automated reasoning*, 52(4):379–405, 2014.