Partial Solution Based Constraint Solving Cache in Symbolic Execution

ZIQI SHUAI[*], State Key Laboratory of Complex & Critical Software Environment, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, China
ZHENBANG CHEN[*], State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, China
KELIN MA, State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, China
KUNLIN LIU, State Key Laboratory of Complex & Critical Software Environment, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, China
YUFENG ZHANG, College of Computer Science and Electronic Engineering, Hunan University, China
JUN SUN, School of Information Systems, Singapore Management University Singapore, Singapore
JI WANG, State Key Laboratory of Complex & Critical Software Environment, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, China

Constraint solving is one of the main challenges for symbolic execution. Caching is an effective mechanism to reduce the number of the solver invocations in symbolic execution and is adopted by many mainstream symbolic execution engines. However, caching can not perform well on all programs. How to improve caching's effectiveness is challenging in general. In this work, we propose a partial solution-based caching method for improving caching's effectiveness. Our key idea is to utilize the partial solutions inside the constraint solving to generate more cache entries. A partial solution may satisfy other constraints of symbolic execution. Hence, our partial solution-based caching method naturally improves the rate of cache hits. We have implemented our method on two mainstream symbolic executors (KLEE and Symbolic PathFinder) and two SMT solvers (STP and Z3). The results of extensive experiments on real-world benchmarks demonstrate that our method effectively increases the number of the explored paths in symbolic execution. Our caching method achieves 1.07x to 2.3x speedups for exploring the same amount of paths on different benchmarks.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**.

[*]Ziqi Shuai and Zhenbang Chen contributed equally to this work and are co-first authors. Zhenbang Chen and Ji Wang are the corresponding authors.

Authors' addresses: Ziqi Shuai, State Key Laboratory of Complex & Critical Software Environment, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, China, szq@nudt.edu.cn; Zhenbang Chen, State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, Changsha, China, zbchen@nudt.edu.cn; Kelin Ma, State Key Laboratory of Complex & Critical Software Environment, College of Computer, National University of Defense Technology, Changsha, China, kelinma@nudt.edu.cn; Kunlin Liu, State Key Laboratory of Complex & Critical Software Environment, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, China, klliu18@nudt.edu.cn; Yufeng Zhang, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, yufengzhang@hnu.edu.cn; Jun Sun, School of Information Systems, Singapore Management University Singapore, Singapore, Singapore, junsun@smu.edu.sg; Ji Wang, State Key Laboratory of Complex & Critical Software Environment, State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, China, wj@nudt.edu.cn.

ACM 2994-970X/2024/7-ART110
https://doi.org/10.1145/3660817

Additional Key Words and Phrases: Symbolic Execution, Constraint Solving, Cache

ACM Reference Format:
Ziqi Shuai, Zhenbang Chen, Kelin Ma, Kunlin Liu, Yufeng Zhang, Jun Sun, and Ji Wang. 2024. Partial Solution Based Constraint Solving Cache in Symbolic Execution. *Proc. ACM Softw. Eng.* 1, FSE, Article 110 (July 2024), 23 pages. https://doi.org/10.1145/3660817

1 INTRODUCTION

Symbolic execution [King 1976] is a precise program analysis technique powered by constraint solving [Baldoni et al. 2018; Cadar and Sen 2013]. It has a number of successful applications in software engineering, such as automatic test case generation [Cadar et al. 2011], bug detection [Cadar et al. 2006], and bound verification [Deng et al. 2006], to name a few. The effectiveness of symbolic execution relies on the underlying constraint solving techniques. The advancement of constraint solving has led to improved effectiveness and efficiency of symbolic execution, which enables its application in a broader range of scenarios.

Symbolic execution analyzes a program $\mathcal{P}$ by executing it symbolically. $\mathcal{P}$'s inputs are represented as symbolic values, each of which represents any concrete value in the input domain. During the analysis, symbolic execution maintains a path condition (PC) for each state S. PC is a quantifier-free first-order logic formula representing the requirements the inputs must satisfy to reach S. PC is $true$ for the initial state. Then, symbolic execution performs symbolic computations for statements such as addition and updates the symbolic store. The magic happens when executing branch statements. Suppose a branch statement (whose condition is c) is executed at state S with path condition PC. Symbolic execution calculates c's symbolic expression b_c and checks the feasibility of c's two branches by invoking the underlying constraint solver. If $PC \wedge b_c$ is satisfiable [Kroening and Strichman 2008], which indicates that the true branch is feasible, symbolic execution proceeds to analyze the code in the true branch and update the PC of S's successor to $PC \wedge b_c$. The other branch is similar. If a branch's condition is unsatisfiable, symbolic execution abandons the branch. If both branches are feasible, symbolic execution forks a new state from the current state to explore the paths corresponding to both branches. In this way, symbolic execution systematically explores $\mathcal{P}$'s path space.

However, constraint solving is a complex and time-consuming procedure, and it usually dominates the time spent by symbolic execution [Baldoni et al. 2018]. Moreover, due to the high complexity, constraint solver may fail to solve some formulas produced by symbolic execution within a given time budget. In such cases, symbolic execution usually abandons the state and may fail to explore certain program paths. Consequently, the effectiveness and efficiency of symbolic execution depend heavily on constraint solving. Optimizing the efficiency of constraint solving is critical for improving the performance of symbolic execution.

Caching is an effective optimization method to reduce the number of constraint solver invocations and then alleviate the burden of the constraint solving in symbolic execution. In fact, caching is commonly used in existing mainstream symbolic executors [Cadar et al. 2008; Pasareanu et al. 2008]. The basic idea of caching is as follows: a formula, in its normal form, and its solving result are cached, so that they can be reused when solving identical or similar formulas in the future. Usually, the normalized formula and the result form a key-value pair in a caching map. Caching proves highly effective for specific benchmarks. For example, KLEE's caching mechanism substantially reduces the number of solver invocations in Coreutils benchmarks [Cadar et al. 2008]. Nevertheless, its effectiveness might vary across different benchmarks. Besides, the effectiveness of cache may also depend on the search heuristic of symbolic execution. Therefore, enhancing the effectiveness of the caching mechanism presents a challenging task. There exists work that tries to improve the effectiveness of caching for the formulas of symbolic addresses [Trabish et al. 2021]. However, the approach is designed for specific kinds of formulas (or programs). *Improving caching's effectiveness in general is still a challenging problem*, similar to the cache hit rate problem [Hennessy and Patterson 2012] in computer architecture research.

We observe that constraint solving is also a search procedure over the input space. Given a path condition *PC*, the constraint solver tries different variable assignments and finally finds a solution if *PC* is satisfiable or proves *PC*'s unsatisfiability. During this procedure, the solver tries many intermediate values, called *partial solutions*. A partial solution satisfies only some sub-formulas of *PC*. Hence, a partial solution corresponds to another part of the input space, which may satisfy other path conditions of the program under symbolic execution [Zhang et al. 2020]. So, we can utilize partial solutions during constraint solving to improve the effectiveness of cache.

In this paper, we propose a partial solution-based caching mechanism to optimize the constraint solving in symbolic execution. Partial solutions exist extensively in different kinds of constraint solving techniques, *e.g.*, SAT/SMT solving. We have instantiated our idea of partial solution to SAT solving and record the partial solutions corresponding to the conflicts in the conflict-driven clause learning (CDCL)-based SAT solving, which is the mainstream SAT solving method [Kroening and Strichman 2008]. Based on it, our method supports three bit-vector SMT theories: QF_ABV, QF_BV, QF_ABVFP, all of which convert bit-vector formulas to SAT formulas and invoke a SAT solver for the real job. Like other caching mechanisms, our method also requires a trade-off between the effectiveness of caching and its overhead. In this paper, we use two parameters to achieve the trade-off: one for controlling the number of partial solutions returned by the solver and another for limiting the maximum size of the solution set attached to cache entries. We also provide some general guidelines for selecting parameter values. However, picking the optimal values for these parameters remains an open problem. We have implemented the partial solution-based caching on two symbolic executors (*i.e.*, KLEE [Cadar et al. 2008] and SPF [Pasareanu et al. 2008]) and two backend solvers (*i.e.*, STP [Ganesh and Dill 2007] and Z3 [de Moura and Bjørner 2008]). The experimental results on representative benchmark programs indicate the effectiveness of our caching method. In summary, the main contributions of this paper are as follows.

- We propose a partial solution-based caching method to improve caching's effectiveness in symbolic execution. Our method is general and orthogonal to existing search heuristics.
- We instantiated our method for two SMT solvers and two state-of-the-art symbolic executors.
- We have carried out extensive experiments to evaluate our method. The experimental results demonstrate the generality of our method. Our caching method achieves the speedups ranging from 1.07x to 2.3x for exploring the same amount of paths.

2 ILLUSTRATION

This section illustrates how our approach works with a small program shown in Figure 1a (denoted by $\mathcal{P}$). The function *foo* takes two ***8-bit signed integer*** inputs a and b and returns an integer.

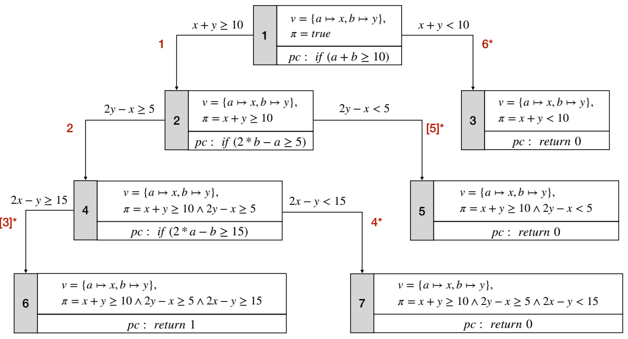
2.1 Symbolic Execution

The symbolic execution of $\mathcal{P}$ starts by assigning symbolic values x and y to inputs a and b, respectively. Then, the state space of $\mathcal{P}$ is explored in a state-forking manner. Assuming that we use the *depth-first search* (DFS) strategy and execute the *true* branch first, the resulting symbolic execution tree is shown in Figure 1b. Each node in the figure represents a symbolic state, and the gray block attached to the left shows its ID. The symbolic store, path condition, and program counter are denoted by v, π, and pc, respectively. The numbers in red attached to the edges indicate the order of constraint solving.

```c
// In total, foo has 4 paths.
int foo(int8_t a, int8_t b) {
  if(a + b >= 10) {
    if(2 * b - a >= 5) {
      if(2 * a - b >= 15) {
        printf("Path #1\n");
        return 1;
      }
    }
  }

  printf("Path #2-#4\n");
  return 0;
}
```

(a) A small example program.


1
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = true$
$pc:\ if\ (a + b \geq 10)$
$x + y \geq 10$
1
$x + y < 10$
6*
2
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = x + y \geq 10$
$pc:\ if\ (2 * b - a \geq 5)$
$2y - x \geq 5$
2
$2y - x < 5$
[5]*
3
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = x + y < 10$
$pc:\ return\ 0$
4
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = x + y \geq 10 \wedge 2y - x \geq 5$
$pc:\ if\ (2 * a - b \geq 15)$
$2x - y \geq 15$
[3]*
$2x - y < 15$
4*
5
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = x + y \geq 10 \wedge 2y - x < 5$
$pc:\ return\ 0$
6
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = x + y \geq 10 \wedge 2y - x \geq 5 \wedge 2x - y \geq 15$
$pc:\ return\ 1$
7
$v = \{a \mapsto x, b \mapsto y\}$,
$\pi = x + y \geq 10 \wedge 2y - x \geq 5 \wedge 2x - y < 15$
$pc:\ return\ 0$

(b) Symbolic execution tree of *foo*.

Fig. 1. The example program and its symbolic execution tree. Without caching, the program requires 6 rounds of constraint solving. With the state-of-the-art caching technique, the 3rd and 5th (in bracket) times of solving can be saved under DFS strategy. With our partial solution-based caching method, the 3rd to 6th (noted with *) times of solving can be saved.

In total, symbolic execution needs to decide the satisfiability of the following six path constraints:

$$x + y \geq 10 \tag{1}$$

$$x + y \geq 10 \wedge 2 \times y - x \geq 5 \tag{2}$$

$$x + y \geq 10 \wedge 2 \times y - x \geq 5 \wedge 2 \times x - y \geq 15 \tag{3}$$

$$x + y \geq 10 \wedge 2 \times y - x \geq 5 \wedge 2 \times x - y < 15 \tag{4}$$

$$x + y \geq 10 \wedge 2 \times y - x < 5 \tag{5}$$

$$x + y < 10 \tag{6}$$

Hence, the traditional symbolic execution without any optimization has to invoke the constraint solver *six times* to solve the above path constraints. Next, we present the state-of-the-art constraint solving caching mechanism and our partial solution-based caching method.

2.2 Constraint Solving Caching

Many mainstream symbolic executors, such as KLEE, incorporate constraint solving caching and reusing mechanisms. Constraint solving results are reused when the symbolic executor needs to check a new path condition. The followings are commonly used caching and reusing mechanisms [Cadar et al. 2008; Visser et al. 2012].

- The first one is *strict* reusing, where a new constraint C must strictly match one of the key in cache map. It is important that C should be *normalized* to improve the caching's effectiveness. For example, the results of solving $x > 0$ can be reused for $y > 0$.
- The second one is *subset-based* reusing. Suppose that the new constraint is $C = \bigwedge_{0 \leq i \leq n} c_i$. We use $\mathcal{S}(C)$ to represent the set $\{c_i \mid 0 \leq i \leq n\}$. If there exists a cache pair $[C_1, R_1]$ such that $\mathcal{S}(C_1) \subset \mathcal{S}(C)$, we check whether R_1 satisfies C and return R_1 if the cache hits. If R_1 is UNSAT, we return UNSAT directly.
- The third one is *superset-based* reusing. If there exists a cache pair (C_1, R_1) such that $\mathcal{S}(C) \subset \mathcal{S}(C_1)$ and R_1 is a solution, then we return R_1 as the result.

If all these three fail to find a cache entry that can decide the satisfiability of new constraint, we call it a *cache miss*, and we invoke the solver to check the constraint. In principle, the effectiveness of these three mechanisms is determined by the program under analysis, the search heuristic, and the underlying solver.

Consider the program in Figure 1a. Recall that we use DFS search strategy for symbolic execution. The first solving is inevitable, *i.e.*, solving $x + y \geq 10$. Suppose that the solution is $(x : 10, y : 0)$. After solving, we have the following key-value pair in the cache map.

$$[x + y \geq 10, (x : 10, y : 0)] \tag{7}$$

Then, when we solve the second path constraint $x+y \geq 10 \wedge 2\times y-x \geq 5$, there does not exist exact matching in the cache. The cache miss occurs even though we can use the subset-based reusing technique since the formula $x + y \geq 10$ was satisfiable. So we need to invoke the solver to solve the formula. Assuming the solving returns $(x : 26, y : 17)$, we add the following new key-value cache pair to the cache map.

$$[x + y \geq 10 \wedge 2 \times y - x \geq 5, (x : 26, y : 17)] \tag{8}$$

Then, when solving the third path constraint, the solution in the above cache pair can satisfy the constraint because the solution satisfies $2 \times x - y \geq 15$. So, the third path constraint is hit. Hence, we have saved one time of constraint solving. Next, for the fourth constraint, none of the cache pairs can hit the constraint. So we add the following cache pair.

$$[x + y \geq 10 \wedge 2 \times y - x \geq 5 \wedge 2 \times x - y < 15, (x : 7, y : 31)] \tag{9}$$

Then, similar to the third constraint, the cache pair in equation (7) also hits the fifth path constraint $x + y \geq 10 \wedge 2 \times y - x < 5$. Solving the last constraint $x + y < 10$ cannot be hit by any cache pairs, and we need to invoke the solver. After solving, the following cache pair is added.

$$[x + y < 10, (x : 0, y : 9)] \tag{10}$$

In summary, also as indicated by Figure 1b, if we use the caching mechanisms, $\mathcal{P}$'s symbolic execution *needs four times constraint solving* and produces four cache pairs. The solvings of the 3rd and 5th path conditions are omitted. Both of the hitting cases are due to subset-based reusing.

2.3 Partial Solution-Based Caching

The partial solutions for solving a path constraint *PC* are the solver's intermediate solutions (*e.g.*, the conflicts during SAT solving) when searching for a *PC*'s solution. The partial solutions may satisfy some sub-formulas of *PC*. During the solving process, the solver tries the partial solutions but abandons them, which are not reported to the users. Because most constraint solving algorithms are search algorithms, partial solutions exist extensively in different constraint solvers. For example, *bit-vector* related SMT theories are widely adopted by the mainstream symbolic executors [Cadar et al. 2008; Poeplau and Francillon 2020; Yun et al. 2018] for precise program representation. Bit-vector SMT theories usually convert the bit-vector constraint to a SAT formula according to the machine number semantics and invoke the backend SAT solver for solving. SAT solving is a counterexample guided search procedure [Kroening and Strichman 2008], in which the assignments causing conflicts are the partial solutions. Hence, given a bit-vector formula *PC*, we can get the partial solutions by solving *PC*'s equisatisfiable [equ 2023] SAT formula and wrap them as *PC*'s partial solutions.

For the example program $\mathcal{P}$, if we use bit-vectors to represent $\mathcal{P}$'s input variables, we have the following partial solutions[1] when solving $x + y \geq 10 \wedge 2 \times y - x \geq 5$.

$$(x : 4, y : 64) \;\; (x : 0, y : 0) \;\; (x : 58, y : 81) \tag{11}$$

[1] All partial solutions in this subsection are collected by our variant of Z3 while solving the QF_BV formulas.

Note that the first partial solution $(x : 4, y : 64)$ actually satisfies the above formula, because we get the partial solution during the SAT solving process. *The solved SAT formula is equisatisfiable with the above bit-vector formula due to the conjunctive normal form (CNF) conversion for SAT solving* [Kroening and Strichman 2008]. *Hence, a partial solution of solving the SAT formula may satisfy the bit-vector formula.*

We observe that the search procedure of constraint solving is also searching the program's input space because the variables of the constraints are also the program's input variables, *e.g.*, a and b in the example program. *The partial solutions of solving one path constraint may satisfy the program's other constraints.* Hence, we can utilize partial solutions to enrich the constraint solving cache map during symbolic execution to improve the effectiveness of cache. Specifically, we can attach more solutions to each cache entry and construct more cache entries. Assuming we have a solved path constraint $PC = \bigwedge_{0 \le i \le n} c_i$, cache entries can be derived from the following two methods:

- Prefix(PC) : $\{\bigwedge_{0 \le j < i} c_j \mid 0 \le i \le n\}$, where each element represents a prefix of the path constraint PC.
- OffThePath(PC) : $\{\bigwedge_{0 \le j < i} c_j \wedge \neg c_i \mid 0 \le i \le n\}$, where each element represents the path constraint of a branch that is not taken in the current path, commonly referred to as an *off-the-path* branch.

For example, assuming PC is $A \wedge B \wedge C$, we can construct the following set of cache entries: $\{A,\ A \wedge B,\ \neg A,\ A \wedge \neg B,\ A \wedge B \wedge \neg C\}$.

Next, we will show how to leverage partial solutions to improve the caching of constraint solving. We assume that the search strategy is still DFS. The solving of the first path constraint $x + y \ge 10$ will only have the final solution but without any partial solutions. Hence, same as before, the cache map only has the following cache pair, where *the value of each pair is a solution set.*

$$[x + y \ge 10, \{(x : 10, y : 0)\}] \tag{12}$$

Then, when solving the second path constraint $x + y \ge 10 \wedge 2 \times y - x \ge 5$, besides the solution $(x : 26, y : 17)$, we will have the three partial solutions in (11). We can utilize these partial solutions to enrich the cache map. For each partial solution ps, we check whether ps can satisfy any prefix of the second path constraint or path constraint of any off-the-path branch (denoted by c). If ps can satisfy c, we add ps to the value (*i.e.*, solution set) of the cache pair whose key is c. Hence, the cache map after solving the second path condition contains the following cache pairs.

$$\begin{gathered}[x + y \ge 10, \{(x : 10, y : 0), (x : 4, y : 64)\}]\\ [\neg(x + y \ge 10), \{(x : 0, y : 0), (x : 58, y : 81)\}]\\ [x + y \ge 10 \wedge 2 \times y - x \ge 5, \{(x : 26, y : 17)\}]\end{gathered}$$

Then, same as before, the third path constraint will be hit by the second one's solution. However, when solving the fourth path constraint (4), the cached solution $(x : 4, y : 64)$ can satisfy the constraint, and the solution will be tried by the subset-based reusing mechanism. Same as before, the cache entries will hit the fifth path constraint. Besides, the last condition $x + y < 10$ will be hit by strict reusing mechanism. In summary, *we need two times of constraint solving if we enrich the cache map by partial solutions* (see Figure 1b), improving the efficiency of $\mathcal{P}$'s symbolic execution.

3 PARTIAL SOLUTION-BASED CACHING

3.1 Symbolic Execution with Solving Cache

Algorithm 1 shows the main procedure of our method. The procedure maintains a worklist of symbolic program states. Each state s is a tuple (l, PC, M), where l is the location of the current to-be-executed statement, PC is the path condition along the history of the current state, and M

Algorithm 1: Symbolic Execution Framework

```
SE($\mathcal{P}$)
Data: $\mathcal{P}$ is a program
begin
    $worklist \leftarrow \{(l_0, true, \emptyset)\}$                          ▷ entry point of $\mathcal{P}$
    $Cache \leftarrow (\emptyset, \emptyset)$
    while $worklist \neq \emptyset \wedge \neg stopCriterion$ do
        $(l, PC, M) \leftarrow$ Select($worklist$)                           ▷ search heuristic
        switch $stmt(l)$ do
            case $if(C)\ goto\ l'$ do
                $C_t, C_f \leftarrow PC \wedge M[C], PC \wedge \neg M[C]$
                if CacheAndSolve($Cache, C_t$) then
                    $worklist.insert((l', C_t, M))$
                end
                if CacheAndSolve($Cache, C_f$) then
                    $worklist.insert((l+1, C_f, M))$
                end
            end
            ...
        end
    end
end
```

is a map containing the symbolic expression of each variable. For brevity, we define M on the expressions of variables, *e.g.*, we define M on condition $a > b$ as $M[a > b] := M[a] > M[b]$. We use *Cache* to maintain the global cache of constraint solving results. The whole procedure is a while loop until all the states are covered or some stop criterion is satisfied (*e.g.*, timeout). One can choose a strategy to select an open state from the worklist (Line 5). The most commonly used strategies include depth-first search, breadth-first search, and so on [Baldoni et al. 2018]. Once a state (l, PC, M) is selected, the corresponding statement is executed symbolically. For brevity, we only show the case of branch statement. If the statement $stmt(l)$ is a conditional statement $if(C)\ goto\ l'$, the procedure checks the feasibility of both branches. A cache-based constraint solving method, *i.e.*, CacheAndSolve (Algorithm 2), is invoked to check the satisfiability of C_t (*i.e.*, $PC \wedge M[C]$) and C_f (*i.e.*, $PC \wedge \neg M[C]$). If one branch is feasible, a new open state is added to worklist. In the new state, the path condition is also updated as C_t or C_f accordingly.

Algorithm 2 shows how to use cache to reduce the times of constraint solving. *For simplicity, we do not differentiate a set of atomic constraints with their conjunction.* For clarity, we use two modules *RCache* and *SCache* to implement the *Cache* in Algorithm 2. Each key c of both caches is a set of atomic constraints. *RCache* stores solving results (*i.e.*, SAT and UNSAT) and *SCache* stores a set of solutions for each key. The solving target ϕ is compared with each key in the cache. Due to the existence of efficient algorithm for indexing and querying sets [Hoffmann and Koehler 1999], the time overhead of cache lookup is often not significant. If there exists a key c of *RCache* (denoted by $c \hat{\in} RCache$) such that $c = \phi$, then the corresponding result $SCache[c]$ can be used directly, without the need of invoking the constraint solver (Line 3). If $\phi \subset c$ and $RCache[c] =$ SAT (Line 5), it implies that ϕ is satisfiable because c contains more constraints than ϕ. In such case, we can use the solution to c as the result by restricting the solution to the variables in ϕ. In Line 6, we use $SCache[c] \downarrow \phi$ to denote such restriction. Additionally, we iterate through all prefixes of ϕ in descending order of

Algorithm 2: Cache and solve

```
CacheAndSolve($Cache, \phi$)
Data: $Cache$: a ($RCache : 2^{\mathcal{C}} \longrightarrow \{\text{SAT}, \text{UNSAT}\}$, $SCache : 2^{\mathcal{C}} \longrightarrow 2^{\mathcal{S}}$) pair, where $\mathcal{C}$ denotes a set of
        atomic constraints and $\mathcal{S}$ denotes a set of solutions, $\phi$: a path condition
1  begin
2      if $\exists c \hat{\in} RCache \cdot c = \phi$ then
3          return $RCache[\phi] = \text{SAT}$ ? (SAT, $SCache[c]$) : UNSAT        ▷ cache hit, strict reusing
4      end
5      if $\exists c \hat{\in} RCache \cdot \phi \subset c \wedge RCache[c] = \text{SAT}$ then
6          return (SAT, $SCache[c] \downarrow \phi$)        ▷ cache hit, superset-based reusing
7      end
8      for each $pref$ in Prefix($\phi$) do
9          if $\exists c \hat{\in} RCache \cdot c = pref$ then
10             if $RCache[c] = UNSAT$ then
11                 return UNSAT        ▷ cache hit, subset-based reusing
12             end
13             else
14                 $cr \longleftarrow$ Concretize($V(\phi) \setminus V(c)$)
15                 if $\exists s \in SCache[c] \cdot (s \cup cr) \models \phi$ then
16                     return (SAT, $s \cup cr$)        ▷ cache hit, subset-based reusing
17                 end
18             end
19         end
20     end
21     ($res$, $partial\text{-}solutions$) $\longleftarrow$ SmtSolving($\phi, V(\phi)$)        ▷ cache miss, solve and update cache
22     $RCache[\phi] \longleftarrow res$
23     $\varphi_c \longleftarrow$ Prefix($\phi$) $\cup$ OffThePath($\phi$)
24     for each $ps$ in $partial\text{-}solutions$ do
25         for each $c \in \varphi_c$ do
26             if $ps \models c$ and $SCache[c]$.size() $< K_s$ then
27                 $RCache[c] \longleftarrow$ SAT
28                 $SCache[c] \longleftarrow SCache[c] \cup \{ps\}$
29             end
30         end
31     end
32     ...        ▷ process solving result, omit for brevity
33 end
```

length, as the solution space of longer prefix more closely resembles that of ϕ. For each prefix ($pref$), we employ the subset-based reusing mechanism only when $c = pref$. Specifically, if $RCache[c]$ is UNSAT, ϕ must be unsatisfiable too because ϕ contains more constraints than c; Otherwise, we take a *quick test* by checking whether c's solution can satisfy ϕ (Line 15). Here one detail is that one can concretize variables not involved in c. For example, a common strategy is setting a variable to be 0. In Line 14, function Concretize returns the chosen values for the variables not involved in c, where $V(\phi)$ denotes the set of the variables in the constraint ϕ. If the solution of c (with other concretized value, if any) passes the test (denoted by $(s \cup cr) \models \phi$, Line 15), it means that we find a solution for ϕ with a very low cost, *i.e.*, cache hit. We return the solution.

Cache hits can reduce the constraint solving overhead effectively. At Line 21, the algorithm invokes an underlying SMT constraint solver supporting partial solutions. Here, $V(\phi)$ is also passed to trace symbolic variables and assist the constraint solver in filtering redundant partial solutions, which will be clarified in Section 3.2. The solving results include not only the solution (*i.e.*, SAT with a variable assignment or UNSAT) for ϕ but also a set of partial solutions. Then, the algorithm constructs a set of constraints φ_c to enrich the cache (Line 23). These constraints are from two sources: prefixes of ϕ (computed by Prefix) and path constraints of ϕ's off-the-path branches (computed by OffThePath). For each constraint c in φ_c, if a partial solution ps satisfies c and the size of $SCache[c]$ does not exceed an integer K_s (Line 26), we add ps into $SCache[c]$. It is intuitive that tracking the prefixes and off-the-path branches can significantly increase the effectiveness of the cache. This is because they are highly likely to appear as subsets in the subsequent path constraints, owing to the incremental analysis style employed in symbolic execution. Therefore, solutions to such carefully crafted sub-constraints are more likely to be reused. For instance, if we get a partial solution $(x : 1, y : 2)$ when solving $x + y \leq 5 \wedge y > 3$, we store $(x : 1, y : 2)$ into both $SCache[x + y \leq 5]$ and $SCache[x + y \leq 5 \wedge y \leq 3]$. In this way, a query $x + y \leq 5 \wedge y \leq 3 \wedge y > 0$ in the future would have a cache hit. This corresponds to the subset case at Line 16 of Algorithm 2. Since other sub-constraints do not serve as prefixes for any path constraint, we intentionally disregard them to minimize the cache size, thereby reducing the overhead of cache lookup. Given that constraint solving generates numerous partial solutions [Zhang et al. 2020], the algorithm enhances the effectiveness of the constraint solving cache compared to vanilla symbolic execution.

Algorithm 2 can be naturally extended to other caching techniques, such as Green [Visser et al. 2012], Utopia [Aquino et al. 2017], and so on. Regardless of the specific caching mechanism employed, whenever a cache miss occurs, the solver must be invoked to make the final decision. Therefore, it is always possible to enrich the cache using partial solutions obtained from the solver and then the improved cache can be queried. In summary, our caching method is orthogonal to other caching mechanisms, as further explained in Section 4.

3.2 Partial Solutions in SAT solver

Bit-vector SMT theories are widely used in many symbolic execution engines to precisely represent programs. To solve a bit-vector formula, the classical pipeline, as shown in Algorithm 3, involves applying word-level simplification rules to the formula, converting the simplified formula to a propositional CNF formula through Tseitin transformation (known as *bit-blasting* [Kroening and Strichman 2008]), and then using a SAT solver for solving. Hence, we fetch the partial solutions during SAT solving to support the extraction of partial solution for all bit-vector SMT theories, including QF_ABV, QF_BV, QF_ABVFP, and so on. Since bit-blasting often introduces redundant propositional variables that are unnecessary for reconstructing bit-vector partial solutions (Line 5), Algorithm 3 traces the encoding of symbolic bit-vector variables at Line 3 and generates a vector of symbolic propositional variables, denoted as $[b_0 b_1 ... b_k]$. For example, when encoding a 64-bit bit-vector multiplier expression, the Tseitin transformation produces 20,417 propositional variables [Kroening and Strichman 2008]. However, only 128 (*i.e.*, the bits of two 64-bit bit-vector variables) of these variables are necessary to reconstruct bit-vector partial solutions. In such a case, Algorithm 3 generates a vector of size 128, *i.e.*, $[b_0 b_1 ... b_{127}]$. The SAT solver then uses this vector to optimize the generation of partial solutions (Line 4). *It is important to note that our method is not limited to bit-vector related theories and can be naturally extended to other theories, as the existence of partial solutions is widespread.*

Modern SAT solvers are usually based on the famous CDCL framework, which improves the classical DPLL framework by introducing backtracking with conflict-driven clause [Kroening and Strichman 2008]. Algorithm 4 shows the basic CDCL framework and how we collect partial

Algorithm 3: Bit-vector SMT solving

```
SmtSolving($\phi, V$)
Data: $\phi$: a bit-vector SMT formula, $V$: symbolic variables passed by symbolic execution
1 begin
2     $\phi' \leftarrow$ Simplify($\phi$)
3     $(\phi_{prop}, [b_0 b_1 ... b_k]) \leftarrow$ bitblast($\phi', V$)          ▷ trace encoding of symbolic variables
4     $(res, partial\text{-}solutions_{sat}) \leftarrow$ CDCL($\phi_{prop}, [b_0 b_1 ... b_k]$)
5     $partial\text{-}solutions_{bv}$ = reconstruct($partial\text{-}solutions_{sat}$)
6     return $(res, partial\text{-}solutions_{bv})$
7 end
```

Algorithm 4: CDCL reporting partial solutions

```
CDCL($\phi_{prop}, [b_0 b_1 ... b_k]$)
Data: $\phi_{prop}$: a propositional CNF formula, $[b_0 b_1 ... b_k]$: a vector of traced propositional variables
1  begin
2      $PS \leftarrow \emptyset$                                   ▷ a set of partial solutions
3      while True do
4          while BCP() = $\bot$ do
5              if $curModel \downarrow [b_0 b_1 ... b_k] \notin PS$ then
6                  $PS \leftarrow PS \cup \{curModel \downarrow [b_0 b_1 ... b_k]\}$
7              end
8              $backtrack \leftarrow$ AnalyzeConflict()
9              if $backtrack < 0$ then
10                 return (UNSAT, $PS$.size() $> K_p$ ? UniformSample($PS, K_p$) : $PS$)
11             end
12             Backtrack($backtrack$)
13         end
14         if $\neg$Decide() then
15             return (SAT, $PS$.size() $> K_p$ ? UniformSample($PS, K_p$) : $PS$)
16         end
17     end
18 end
```

solutions (PS) within the CDCL framework. The procedure explores the solution space (essentially, a binary tree) systematically. The procedure of *boolean constraint propagation*, *i.e.*, BCP, utilizes *unit propagation* [Kroening and Strichman 2008] to analyze whether the current intermediate assignment ($curModel$) can cause a conflict. If there is no conflict, Decide selects a variable to assign a value (true or false). If BCP detects a conflict, a learned clause reflecting the conflict's root cause is generated and added into the formula to prevent the conflicts of the same reason in the future search. Then, the search procedure backtracks to the level suggested by the conflict-driven learning procedure. When a conflict arises, we store the current intermediate assignment, projected onto the traced propositional variables (denoted by $curModel \downarrow [b_0 b_1 ... b_k]$), as a SAT partial solution (Line 6). Tracking only these bits consumes significantly less memory than monitoring the entire intermediate assignment. A conflict is considered valuable if it can generate a distinct projection, yielding a unique partial solution. We employ a hash-based search algorithm to identify such valuable conflicts, treating the projection as an integer vector to compute its hash value (Line 5). This optimization allows us to discard redundant projections resulting in the same partial solution, thereby further reducing memory consumption and alleviating the burden on quick test due to the smaller number of partial solutions. If all the variables are assigned and there is no conflict, the formula is satisfiable. As we can see, the SAT solving may return partial solutions despite of the formula's result, which means that we can always increase the number of cache entries. Moreover, if the final size of PS exceeds an integer K_p, the algorithm performs uniform sampling [Forsyth 2018] on PS and select K_p of them to return (Line 10&15).

3.3 Discussion

In the constraint solving process of the path condition ϕ, our caching method leverages partial solutions produced by the constraint solver to expand the constraint solving cache in symbolic execution. Firstly, our method uses prefixes of ϕ (*i.e.*, $\mathsf{Prefix}(\phi)$) and path constraints of ϕ's off-the-path branches (*i.e.*, $\mathsf{OffThePath}(\phi)$) to enrich the cache entries. Secondly, our method expands the single solution associated with each cache entry into a set of solutions (Line 28 in Algorithm 2). Due to the widespread availability of partial solutions [Zhang et al. 2020], our method is a *universal* method to improve caching's effectiveness, independent of the specific constraint solver used, the format of formulas, or the chosen search heuristic. However, our method only improves the cache hits for satisfiable constraints. Unsatisfiable constraints is not our target. Besides, our method also introduces overhead during the generation and utilization of partial solutions.

As explained in Section 3.2, we acquire partial solutions from conflicts within the CDCL framework. Consequently, SAT formulas whose solvings involve numerous conflicts can generate a substantial quantity of partial solutions, potentially causing significant overhead for our approach. Fortunately, by tracking symbolic variables designated by the symbolic executor ($V(\phi)$ in Algorithm 2), we can effectively identify valuable conflicts, namely those that yield unique partial solutions (Line 5 in Algorithm 4), instead of considering all conflicts. This helps to reduce memory and time overhead in partial solution generation and utilization.

When utilizing partial solutions in our cache, there are two types of time overhead that can potentially result in unfavorable outcomes. On the one hand, for each partial solution returned by the solver, our method needs to check which constraint in φ_c the partial solution can satisfy (Line 26 in Algorithm 2). However, this process may be time-consuming when the partial solution set is large. On the other hand, in the subset-based reusing, our method takes a quick test by checking whether some solutions of a cache entry can satisfy the path constraint (Line 15 in Algorithm 2). The quick test may also introduce overhead when too many solutions are attached to the cache entry. Hence, there exists a *trade-off* between the caching's effectiveness and its time overhead. With regard to the two types of time overhead mentioned earlier, our method configures two key parameters, K_p and K_s, to achieve a trade-off for each. Specifically, K_p controls the number of partial solutions returned by the solver, while K_s represents the maximum size of the solution set for each cache entry. Intuitively, a larger K_p or K_s allows us to expand the cache more effectively, but it also introduces greater time overhead. Therefore, the selection of these parameter values is crucial and closely related to the format of formulas and the specific constraint solver being used. While there is no one-size-fits-all solution for choosing parameter values, there are some common standards that can be applied. Generally, as the constraint-solving process becomes more challenging and generates more partial solutions, the value of K_p should be set higher. The value of another parameter, K_s, depends on the structural complexity of the path constraints in symbolic execution because the quick test essentially involves iterating through constraints. Since constraints with simpler structures incur lower time overhead during scanning, the value of K_s could be set higher to increase cache hits.

4 IMPLEMENTATION AND EVALUATION

4.1 Implementation

We have implemented partial solution-based caching method on multiple existing mainstream SMT solvers and symbolic execution engines.

SMT Solvers. We have extended two state-of-the-art constraint solvers to support partial solutions on the level of SAT solving. STP [Ganesh and Dill 2007] is an SMT solver mainly aimed at QF_BV and QF_ABV logic. Since STP employs an external backend SAT solver, *i.e.*, Minisat [Eén and Sörensson 2003], we modified Minisat to generate SAT-based partial solutions. In STP, these SAT-based partial solutions are reconstructed for SMT formulas. We also extended STP's interfaces for symbolic executors to obtain the partial solutions. Z3 [de Moura and Bjørner 2008] is a general purpose SMT solver for many logics. We use Z3 for solving QF_BV and QF_ABVFP constraints. We modified the self-customized SAT solver in Z3 to support partial solutions. Similarly, we added the interfaces for partial solutions to Z3. The versions of SMT solvers are STP 2.1.2 and Z3 4.8.8, respectively.

Symbolic executors. We have implemented the method on KLEE [Cadar et al. 2008] and SPF [Pasareanu et al. 2008], *i.e.*, two state-of-the-art symbolic executors for C and Java programs, respectively. Both of the engines have cache-based solving optimization. We built our method on KLEE 2.2-pre. Specifically, we modified its counterexample caching mechanism to support the caching based on partial solutions. Our implementation is based on KLEE's existing cache data structure, *i.e.*, a map from a constraint to its solution. We replace the single solution of each constraint with a solution set. To support the analysis of floating-point programs, we also have implemented our caching method on KLEE-Float [Liew et al. 2017], which is a variant of KLEE that supports reasoning about floating-point arithmetic. KLEE-Float uses the extended Z3 to collect partial solutions when solving QF_ABVFP formulas. We have implemented our method on SPF with Green [Visser et al. 2012] as the caching layer. We integrated our method into the Green framework. The partial solutions were obtained from the extended Z3. Specifically, we implemented our method on Green's Grulia service [Taljaard 2019; Taljaard et al. 2020], which implements a SAT-delta value [Aquino et al. 2017] based caching mechanism. We have developed our method based on the commit a1261b0 in a fork of Green.

4.2 Research Questions

We conducted extensive experiments to answer the following three research questions:

- *Effectiveness*: Can our method improve the efficiency of symbolic execution? If our method is effective, it should explore more paths or states than the vanilla within the same time budget.
- *Efficiency*: How efficient is our caching method? Here, efficiency means consuming less time to explore the same number of paths during symbolic execution.
- *Impact of parameter tuning*: If we change the values of K_p and K_s, can our method still consistently improve the efficiency of symbolic execution?

4.3 Experimental Setup

Intuitively, the increased number of explored paths directly shows the effectiveness of our method. Therefore, we use each of the previously mentioned symbolic executors to analyze the benchmark programs with different configurations and then collect the number of explored paths during symbolic execution. To alleviate the nondeterminism of symbolic execution, we use two deterministic search strategies to analyze programs, *i.e.*, depth-first search (DFS) and breadth-first search (BFS). For experiments on KLEE and KLEE-Float, we further use the default random-cover new (RCN)

Table 1. C benchmark programs in Experiment 1.

Name	**SLOC**	**Brief Description**
`json-c`	7288	A JSON implementation in C
`apr`	61201	Apache portable runtime framework
`cmark`	21658	C implementation of CommonMark language
`fribidi`	8816	GNU Unicode bidirectional algorithm
`gas`	52130	GNU Assembler
`libinjection`	13296	A library to detect SQL injection
`libtommath`	19682	A multiple-precision integer library
`m4`	93771	Unix macro processor
`discount`	5880	C implementation of Markdown language
`pacparser`	959	A library to parse proxy auto-config files
`ptx`	2049	GNU permuted index generator
`sha1-cd`	2095	A tool to detect SHA-1 collisions in files
`smaz`	274	Small strings compression library
`sqlite3`	134768	A popular SQL database system
`sundown`	3611	Markdown parser based on Upskirt library
Total	**427478**	**15 real-world open-source C programs**

search strategy to demonstrate the compatibility between our method and search heuristics. The timeout of each analysis task is 30 minutes.

4.3.1 Experiment 1: QF_ABV based analysis. The queries issued by KLEE are QF_ABV formulas. We use the extended STP as the backend constraint solver for evaluation. Table 1 shows the benchmark programs, which are collected from existing studies [Shuai et al. 2021; Trabish et al. 2021] and open-source repositories like GitHub and SourceForge. Since STP's word-level simplifications are quite powerful, many trivial queries are decided during simplification and do not reach the backend SAT solver. Therefore, we chose benchmarks that are challenging to symbolic execution, *i.e.*, they issue lots of complex QF_ABV formulas whose solvings produce many partial solutions during the analysis. In total, we collected 15 real-world C libraries or programs with various applications for evaluation. For each library, we customized test driver based on the attached test suites. The values of K_p and K_s are set to 250 and 50, respectively. The timeout of STP is 30s.

4.3.2 Experiment 2: QF_ABVFP based analysis. We use KLEE-Float and the extended Z3 for evaluation. Z3 is used for solving QF_ABVFP formulas. In this experiment, all benchmarks are from the GNU scientific library (GSL) [gsl 2023]. GSL is a renowned scientific computing numerical library written in C, offering a wide range of valuable features like fundamental mathematical functions. As a result, the implementation of GSL involves numerous floating-point operations, notably non-linear ones like multiplication. Hence, reasoning about GSL functions is highly challenging for symbolic execution. We randomly selected 32 functions from GSL as the entry points and constructed a driver for each function to symbolize the input data. The detailed benchmark list can be found in Table 3. In this case, both K_p and K_s are set to 2500. The timeout of Z3 is 200s.

4.3.3 Experiment 3: QF_BV based analysis. To validate the generalization ability, we apply our method in the context of symbolic execution for Java programs. We use SPF equipped with Green and Z3 for evaluation. Here, we use Z3's QF_BV theory for solving. All benchmark programs used in this experiment are sourced from the standard benchmarks within the Green framework

[Taljaard 2019; Taljaard et al. 2020], comprising a total of 23 Java programs[2]. Among them, 10 programs were excluded because they either did not produce any partial solutions during analysis due to the simple structure of path constraints, or the vanilla Grulia service performed exceptionally efficiently on them, *i.e.*, the analysis was completed within 1 minute. The remaining 13 programs shown in Table 4 constitute the benchmark set for this experiment, with a total of 3,999 lines of code. In this experiment, both K_p and K_s are set to 100. The timeout for Z3's QF_BV solving is 5s.

4.3.4 Determining K_p and K_s. In Experiment 2, the values of K_p and K_s are much larger compared to the other two experiments. As clarified in Section 3.3, the value of K_p is directly proportional to the complexity of constraint solving, whereas the value of K_s is inversely proportional to the length of path constraints in symbolic execution. In general, solving floating-point constraints is more challenging than solving other types of constraints [Kroening and Strichman 2008], often resulting in a larger number of partial solutions. Hence, the value of K_p is set much larger in Experiment 2 to capture more partial solutions. At the same time, since the floating-point constraints that cause the solver to timeout are likely to be structurally simple, leading to extremely low traversal overhead, the value of K_s is also set larger in Experiment 2. Specifically, the magnitudes of these two parameters in the three experiments are chosen based on empirical experiments. To evaluate the impact of parameter tuning, we also conducted extensive experiments with other parameter configurations. In each experiment, the value of each parameter is selected from a set of three specific values to reflect the impact of parameter tuning. In total, there are 9 parameter configurations for each experiment.

All experiments were performed on a multi-core server with 2.5GHz Intel(R) Xeon(R) Platinum 8269CY CPU. The operating system is Ubuntu 20.04 LTS shipped with Linux version 5.4.0-144-generic. We carried out the experiments in parallel while restricting the number of parallel tasks to ensure sufficient resources. Additionally, all experiments were repeated five times to minimize the randomness. The experimental results are the average of five runs.

4.4 Experimental Results

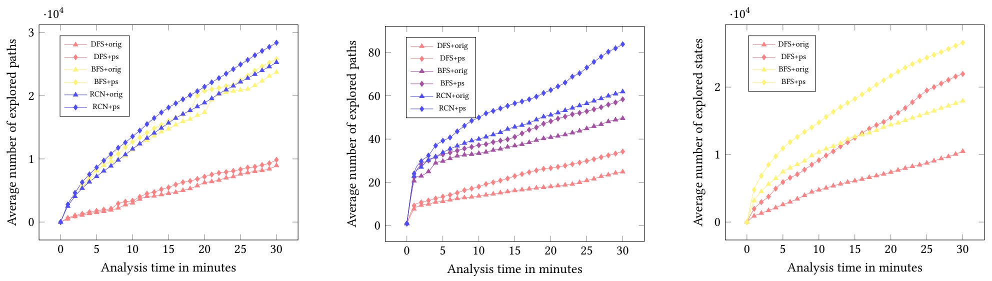
Tables 2&3&4 show the detailed results of three experiments, respectively. "**P**" indicates the configuration with our partial solution based caching method, while "**O**" indicates the original configurations. Under all search strategies, we collected the number of explored paths (**#Paths**) or states (**#States**), cache hit rate (**CHR**) and time overhead (**TO(s)**) for each task. Specifically, **CHR** is the proportion of queries that hit the constraint solving cache, out of the total number of queries. Since many analysis tasks in Experiment 3 completed within the 30-minute time budget, we also collected the analysis time (**T(s)**). Additional data, such as the number of solved constraints and the count of generated partial solutions, that provide a deeper understanding of the proposed method are available in our artifact. We do not present these data here for the sake of space. For the convenience of readers, our method's best results are presented in bold fonts in these tables.

4.4.1 Results of Experiment 1. As shown in Table 2, our caching method improves the numbers of paths for 9 tasks under all search strategies. On average, our method improves the explored paths by 7.0%, 16.0% and 23.1% under DFS, BFS, and RCN, respectively. The limited improvement observed in our method during this experiment is attributed to KLEE's counterexample cache optimizations, which are highly effective for QF_ABV solving, particularly in the case of satisfiable queries. Indeed, the counterexample cache primarily operates at the level of satisfying assignments [Palikareva and Cadar 2013], making it more suitable for confirming the satisfiability of queries. As a result, queries that do not hit the cache are more likely to be unsatisfiable, which is not the target

[2]All 23 Java programs are available at https://bitbucket.org/Developer_Jan/green/src/master/jpf_exp/exs/voorbeelde/.

Table 2. Detailed results of QF_ABV Experiment. For the convenience of readers, our method's best results are presented in bold fonts.

Programs	Mode	DFS			BFS			RCN		
		#Paths	CHR	TO(s)	#Paths	CHR	TO(s)	#Paths	CHR	TO(s)
apr	P	**107(33.8%)**	0.847	1.24	84(10.5%)	0.703	0.77	87(29.9%)	0.765	0.87
	O	80	0.793	0.0	76	0.671	0.0	67	0.721	0.0
cmark	P	1790(-7.3%)	0.485	213.49	1569(3.8%)	0.528	28.84	**2569(5.9%)**	0.576	28.01
	O	1931	0.463	0.0	1511	0.502	0.0	2425	0.546	0.0
fribidi	P	336(23.1%)	0.924	9.75	**8427(24.3%)**	0.975	6.09	6925(2.2%)	0.971	4.19
	O	273	0.909	0.0	6782	0.966	0.0	6773	0.966	0.0
gas	P	312(2.3%)	0.888	6.03	**1541(12.2%)**	0.895	1.09	2563(-13.3%)	0.906	0.9
	O	305	0.876	0.0	1373	0.87	0.0	2955	0.906	0.0
json-c	P	409(3.0%)	0.565	65.39	569(28.7%)	0.557	71.71	**492(41.8%)**	0.556	67.1
	O	397	0.35	0.0	442	0.236	0.0	347	0.209	0.0
libinjection	P	1007(-15.7%)	0.936	1.87	**27373(1.7%)**	0.936	4.3	16088(1.5%)	0.928	2.77
	O	1194	0.946	0.0	26918	0.939	0.0	15856	0.924	0.0
libtommath	P	38006(5.1%)	0.933	24.62	27214(33.7%)	0.923	8.36	**42238(98.7%)**	0.96	6.63
	O	36151	0.915	0.0	20347	0.917	0.0	21260	0.964	0.0
m4	P	14938(-26.4%)	0.965	124.29	75509(1.2%)	0.996	131.61	**44140(31.4%)**	0.99	129.03
	O	20304	0.969	0.0	74645	0.995	0.0	33586	0.983	0.0
discount	P	**86474(24.0%)**	0.974	25.74	222984(9.8%)	0.999	3.68	274828(4.3%)	0.998	2.62
	O	69764	0.972	0.0	203125	0.998	0.0	263373	0.997	0.0
pacparser	P	36(2.9%)	0.663	12.02	3668(5.9%)	0.96	5.13	**5655(9.9%)**	0.971	6.95
	O	35	0.641	0.0	3465	0.95	0.0	5145	0.96	0.0
ptx	P	**3312(28.4%)**	0.985	10.67	151(-5.0%)	0.696	63.74	446(-6.3%)	0.778	29.74
	O	2579	0.982	0.0	159	0.682	0.0	476	0.779	0.0
sha1-cd	P	80(9.6%)	0.596	38.26	**1772(68.0%)**	0.591	266.96	3180(34.3%)	0.568	551.15
	O	73	0.52	0.0	1055	0.525	0.0	2367	0.52	0.0
smaz	P	52(26.8%)	0.622	34.65	154(51.0%)	0.628	17.81	**106(89.3%)**	0.619	19.84
	O	41	0.601	0.0	102	0.564	0.0	56	0.58	0.0
sqlite3	P	42(-10.6%)	0.85	8.68	225(-6.2%)	0.823	20.96	**173(8.8%)**	0.685	11.95
	O	47	0.839	0.0	240	0.801	0.0	159	0.581	0.0
sundown	P	1524(6.6%)	0.792	74.05	16244(0.0%)	0.84	50.07	**26906(7.9%)**	0.883	46.97
	O	1430	0.723	0.0	16241	0.831	0.0	24929	0.87	0.0



(a) Experiment 1 (b) Experiment 2 (c) Experiment 3

Fig. 2. Path exploration trends in three experiments

of our method. For example, using the vanilla mode **O**, the average ratios of unsatisfiable queries among those reaching the solver are 0.7, 0.6 and 0.61 under DFS, BFS and RCN, respectively.

We also evaluated the efficiency achieved by our method. Figure 2a shows the trend of explored paths during symbolic execution. The X-axis represents the analysis time in minutes, while the Y-axis displays the average number of explored paths for all programs. As shown in the figure, our method consistently increases the number of explored paths under any search strategy. Specifically, our caching method achieves 1.07x, 1.11x and 1.15x speedup for exploring the largest number of paths using the original caching method under DFS, BFS and RCN strategy, respectively.

Table 3. Detailed results of QF_ABVFP Experiment. For the convenience of readers, our method's best results are presented in bold fonts.

GSL Functions	Mode	DFS			BFS			RCN		
		#Paths	**CHR**	**TO(s)**	**#Paths**	**CHR**	**TO(s)**	**#Paths**	**CHR**	**TO(s)**
`gsl_cdf_beta_P`	**P**	19(0.0%)	0.506	5.3	**44(37.5%)**	0.617	3.4	63(21.2%)	0.605	5.9
	O	19	0.455	0.0	32	0.465	0.0	52	0.466	0.0
`gsl_cdf_beta_Pinv`	**P**	24(0.0%)	0.566	3.2	40(0.0%)	0.661	2.4	**64(30.6%)**	0.669	4.7
	O	24	0.458	0.0	40	0.503	0.0	49	0.479	0.0
`gsl_cdf_cauchy_Pinv`	**P**	17(21.4%)	0.565	3.9	44(10.0%)	0.63	5.0	**47(27.0%)**	0.663	6.1
	O	14	0.517	0.0	40	0.469	0.0	37	0.466	0.0
`gsl_cdf_chisq_P`	**P**	15(7.1%)	0.581	1.8	**39(25.8%)**	0.661	3.8	43(22.9%)	0.643	4.1
	O	14	0.478	0.0	31	0.5	0.0	35	0.49	0.0
`gsl_cdf_gumbel1_Q`	**P**	14(16.7%)	0.571	1.1	**27(42.1%)**	0.634	1.3	33(17.9%)	0.602	1.8
	O	12	0.462	0.0	19	0.573	0.0	28	0.538	0.0
`gsl_cdf_gumbel1_Qinv`	**P**	**16(23.1%)**	0.552	1.2	30(11.1%)	0.604	0.7	40(11.1%)	0.614	1.7
	O	13	0.471	0.0	27	0.474	0.0	36	0.476	0.0
`gsl_cdf_gumbel2_Pinv`	**P**	24(26.3%)	0.588	0.8	45(7.1%)	0.651	1.5	**64(45.5%)**	0.675	1.9
	O	19	0.519	0.0	42	0.558	0.0	44	0.507	0.0
`gsl_cdf_weibull_Qinv`	**P**	23(9.5%)	0.608	1.1	40(0.0%)	0.629	1.9	**44(10.0%)**	0.639	1.4
	O	21	0.522	0.0	40	0.562	0.0	40	0.552	0.0
`gsl_complex_exp`	**P**	**27(58.8%)**	0.604	1.7	238(11.7%)	0.841	2.5	310(13.1%)	0.895	3.0
	O	17	0.516	0.0	213	0.794	0.0	274	0.848	0.0
`gsl_complex_log`	**P**	**27(58.8%)**	0.603	1.5	238(11.7%)	0.845	2.6	295(4.6%)	0.892	3.1
	O	17	0.513	0.0	213	0.794	0.0	282	0.85	0.0
`gsl_complex_sinh`	**P**	29(45.0%)	0.604	1.5	143(5.1%)	0.721	5.4	**284(68.0%)**	0.759	8.2
	O	20	0.516	0.0	136	0.681	0.0	169	0.633	0.0
`gsl_deriv_forward`	**P**	11(22.2%)	0.589	1.6	**13(30.0%)**	0.625	1.9	15(25.0%)	0.647	1.6
	O	9	0.462	0.0	10	0.459	0.0	12	0.457	0.0
`gsl_diff_forward`	**P**	10(11.1%)	0.541	0.2	10(42.9%)	0.57	0.7	**15(50.0%)**	0.604	0.6
	O	9	0.513	0.0	7	0.506	0.0	10	0.493	0.0
`gsl_eigen_genv_sort`	**P**	17(30.8%)	0.617	1.4	75(27.1%)	0.833	2.9	**50(56.2%)**	0.656	2.9
	O	13	0.459	0.0	59	0.783	0.0	32	0.519	0.0
`gsl_integration_glfixed`	**P**	11(10.0%)	0.569	3.8	22(15.8%)	0.628	3.2	**29(20.8%)**	0.582	4.3
	O	10	0.477	0.0	19	0.493	0.0	24	0.5	0.0
`gsl_integration_qawc`	**P**	138(137.9%)	0.776	195.2	24(0.0%)	0.614	1.0	**243(170.0%)**	0.796	13.1
	O	58	0.489	0.0	24	0.557	0.0	90	0.628	0.0
`gsl_integration_qng`	**P**	12(9.1%)	0.579	3.2	**70(311.8%)**	0.679	2.8	109(22.5%)	0.739	6.8
	O	11	0.469	0.0	17	0.509	0.0	89	0.619	0.0
`gsl_linalg_PTLQ_decomp`	**P**	44(18.9%)	0.538	7.0	**113(41.2%)**	0.553	2.1	34(13.3%)	0.508	2.1
	O	37	0.494	0.0	80	0.52	0.0	30	0.444	0.0
`gsl_linalg_complex_LU_lndet`	**P**	**30(57.9%)**	0.726	6.7	79(14.5%)	0.731	2.3	47(-2.1%)	0.627	2.8
	O	19	0.476	0.0	69	0.543	0.0	48	0.491	0.0
`gsl_poly_complex_solve_cubic`	**P**	23(1050.0%)	0.776	3.0	25(1150.0%)	0.74	2.1	**28(1300.0%)**	0.753	2.5
	O	2	0.286	0.0	2	0.286	0.0	2	0.286	0.0
`gsl_poly_complex_solve_quadratic`	**P**	20(66.7%)	0.746	4.9	**60(160.9%)**	0.676	5.8	58(123.1%)	0.697	5.6
	O	12	0.459	0.0	23	0.482	0.0	26	0.484	0.0
`gsl_poly_solve_quadratic`	**P**	19(46.2%)	0.719	5.1	**26(62.5%)**	0.65	2.7	27(50.0%)	0.66	2.9
	O	13	0.463	0.0	16	0.475	0.0	18	0.477	0.0
`gsl_sf_airy_Ai_deriv_e`	**P**	92(0.0%)	0.623	26.8	13(0.0%)	0.516	0.1	**178(85.4%)**	0.827	7.5
	O	92	0.514	0.0	13	0.516	0.0	96	0.5	0.0
`gsl_sf_bessel_In_scaled_e`	**P**	**95(61.0%)**	0.717	30.2	58(28.9%)	0.559	4.9	48(11.6%)	0.614	2.3
	O	59	0.481	0.0	45	0.5	0.0	43	0.51	0.0
`gsl_sf_bessel_Inu_e`	**P**	12(0.0%)	0.526	1.4	**37(60.9%)**	0.633	4.0	35(9.4%)	0.616	4.4
	O	12	0.459	0.0	23	0.496	0.0	32	0.497	0.0
`gsl_sf_bessel_Inu_scaled_asymp_unif_e`	**P**	**27(107.7%)**	0.762	5.7	11(10.0%)	0.647	1.1	18(63.6%)	0.697	1.4
	O	13	0.471	0.0	10	0.468	0.0	11	0.465	0.0
`gsl_sf_bessel_Inu_scaled_e`	**P**	12(9.1%)	0.538	1.4	**36(50.0%)**	0.624	4.3	37(27.6%)	0.638	4.6
	O	11	0.457	0.0	24	0.496	0.0	29	0.496	0.0
`gsl_sf_bessel_cos_pi4_e`	**P**	**25(92.3%)**	0.573	1.2	106(3.9%)	0.685	4.0	139(37.6%)	0.744	5.0
	O	13	0.515	0.0	102	0.636	0.0	101	0.582	0.0
`gsl_sf_bessel_sin_pi4_e`	**P**	**23(64.3%)**	0.572	1.2	106(2.9%)	0.687	3.8	139(44.8%)	0.744	5.0
	O	14	0.511	0.0	103	0.634	0.0	96	0.581	0.0
`gsl_sf_debye_1_e`	**P**	165(217.3%)	0.827	32.5	79(68.1%)	0.78	11.4	**144(585.7%)**	0.831	28.6
	O	52	0.492	0.0	47	0.492	0.0	21	0.491	0.0
`gsl_sf_elljac_e`	**P**	94(-1.1%)	0.605	88.6	71(9.2%)	0.584	3.3	**108(16.1%)**	0.634	5.6
	O	95	0.497	0.0	65	0.502	0.0	93	0.493	0.0
`gsl_sf_gamma_inc_P_e`	**P**	14(-6.7%)	0.49	1.9	49(14.0%)	0.653	4.8	**47(20.5%)**	0.665	5.6
	O	15	0.468	0.0	43	0.512	0.0	39	0.497	0.0

4.4.2 Results of Experiment 2. The detailed results of using KLEE-Float equipped with our partial solution-based cache to analyze the GSL functions are shown in Table 3. The table shows that our method increases the number of explored paths for 29 tasks under all search strategies. The average improvement in the number of explored paths is 71.0% under DFS, 70.8% under BFS, and 93.8% under RCN, respectively. In this experiment, the cache hit rates (**CHR**) of the vanilla mode for various tasks are often below 0.5. Besides, the average ratios of unsatisfiable queries among those reaching the solver are 0.4 for DFS, 0.47 for BFS and 0.4 for RCN, respectively. Clearly, when compared to array constraints, the counterexample cache performs less effectively on floating-point constraints, which is attributed to the large search space involved in solving floating-point constraints. Hence, our method is more competitive in this experiment.

As shown by Table 3, there are some cases where our method produce negative results, such as `gsl_sf_gamma_inc_P_e`. The reason is that solving floating-point constraints is notoriously challenging, especially for path constraints under the DFS strategy, where Z3 often times out. In such cases, KLEE-Float fails to reach deep parts of the program and explores only the shallow states, where constraint solving is faster. However, with the help of our caching method, KLEE-Float is able to explore deeply into the program. The more complex path constraints may introduce more timeouts, slowing down constraint solving and path exploration, making our method less effective. Also, for the same reason, our method does not have an advantage under DFS compared to other search strategies in this experiment. On the other side of the coin, vanilla KLEE-Float sometimes terminates early because Z3 times out at the beginning of the analysis, such as `gsl_poly_complex_solve_cubic`. In contrast, our method enables KLEE-Float to continue its analysis by leveraging the solving cache enriched with partial solutions. Consequently, the path spaces explored by our method and the baseline method differ. Our method consistently delves more deeply into the program's path space, which is usually desirable in practical applications.

Same as the first experiment, we also inspect the path exploration trend of analyzing floating-point programs. Figure 2b shows the trend of explored paths in this experiment during symbolic execution. Similarly, our caching method consistently increases the number of explored paths under all the experimented search strategy. Specifically, our method achieves 1.67x, 1.36x and 1.5x speedups under DFS, BFS and RCN, respectively. These results indicate that our partial solution-based caching method can also help the symbolic execution of floating-point programs.

4.4.3 Results of Experiment 3. In addition to the symbolic execution of C programs, we also applied our method to the symbolic execution of Java programs. Table 4 shows the detailed results of this experiment. Because the analysis of some programs was completed quickly, in this experiment, we compare our method and the baseline using two metrics: **#States** and **T(s)**. Under DFS, there are 3 programs whose path exploration is finished within 30 minutes in all runs, *i.e.*, `Remainder`, `Triangle` and `Operations`. Compared with the vanilla, our method achieves speedups of 1.72x, 12.5x and 2.12x on the three programs, respectively. For the other programs, our method improves the number of explored states by 114.3%. Similarly, under BFS, there are 4 programs whose path exploration is finished within 30 minutes in all runs, *i.e.*, `Remainder`, `Triangle`, `Operations` and `MagicIndex`. The speedups are 1.85x, 5.63x, 2.25x and 80.67x, respectively. The improvements in the number of states on other programs are 56.8%. On `MagicIndex`, our method significantly enhances cache hit rates, thereby improving the efficiency of path exploration under both search strategies. Unlike the second experiment, our caching method achieves better results on DFS for Java programs. The reason is that the constraints generated from the benchmark programs are not particularly challenging for the solver. Constraint solving timeout never occurs, and our method explores the same path space as the baseline, indicating that constraint solving tasks faced by our method and the baseline are identical. Consequently, DFS generates longer constraints that result in more partial solutions, leading to an increased number of cache entries and improved effectiveness. Another contributing factor is that the formulas under DFS are more uniform and therefore more likely to be cached. Figure 2c shows the trend of explored states, where our method achieves 2.3x and 2.0x speedups under DFS and BFS, respectively. As shown by the table and the figure, the partial solution-based caching method also works for the symbolic execution of Java programs.

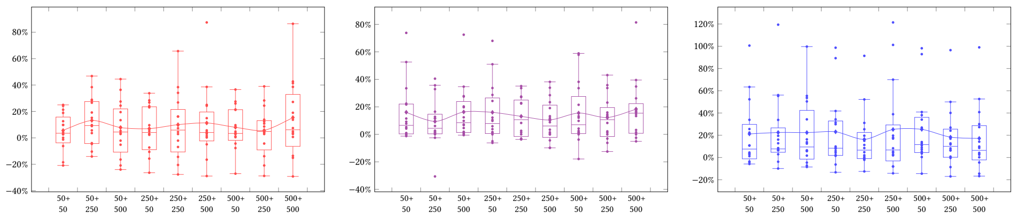
Table 4. Detailed results of Experiment 3. For the convenience of readers, our method's best results are presented in bold fonts. Here, we focus on two metrics: **#States** and **T(s)**.

Programs	Mode	DFS				BFS			
		#States	CHR	TO(s)	T(s)	#States	CHR	TO(s)	T(s)
`Remainder`	**P**	957(0.0%)	0.761	160.6	**285.8**	957(0.0%)	0.771	178.6	298.6
	O	957	0.24	0.0	491.0	957	0.156	0.0	551.6
`BubbleSort`	**P**	11777(17.9%)	0.283	312.5	1802.0	**18255(41.8%)**	0.323	159.7	1800.2
	O	9992	0.044	0.0	1800.0	12876	0.144	0.0	1800.0
`Dijkstra`	**P**	8242(60.5%)	0.325	113.1	1800.0	**8294(63.5%)**	0.347	105.3	1800.0
	O	5135	0.116	0.0	1800.0	5072	0.125	0.0	1800.0
`NanoXML`	**P**	**48520(227.3%)**	0.66	6.7	1800.0	99426(51.6%)	0.536	9.1	1800.0
	O	14822	0.362	0.0	1800.0	65598	0.533	0.0	1800.0
`SortedListInt`	**P**	**79704(377.2%)**	0.846	57.8	1800.0	38491(95.5%)	0.817	11.8	1800.0
	O	16703	0.686	0.0	1800.0	19684	0.695	0.0	1800.0
`BinTree`	**P**	**15227(47.1%)**	0.911	14.8	**912.6**	15227(21.6%)	0.899	17.2	1204.4
	O	10353	0.829	0.0	1800.0	12522	0.842	0.0	1800.0
`Triangle`	**P**	2207(0.0%)	0.502	1.0	**7.2**	2207(0.0%)	0.474	1.5	19.8
	O	2207	0.354	0.0	90.0	2207	0.303	0.0	111.4
`Operations`	**P**	15619(0.0%)	0.52	59.7	**256.4**	15619(0.0%)	0.465	58.7	262.8
	O	15619	0.188	0.0	544.2	15619	0.222	0.0	591.2
`Sorting`	**P**	**24219(80.0%)**	0.561	127.3	1801.0	18714(25.1%)	0.36	48.6	1800.0
	O	13459	0.256	0.0	1800.0	14959	0.181	0.0	1800.0
`MagicIndex`	**P**	**7202(52.4%)**	0.944	0.2	**9.2**	7202(0.0%)	0.956	0.2	10.8
	O	4726	0.551	0.0	1788.6	7202	0.786	0.0	871.2
`BinomialHeap`	**P**	**47461(106.5%)**	0.939	86.7	**1435.4**	47303(21.1%)	0.939	65.6	1493.2
	O	22978	0.885	0.0	1800.0	39063	0.925	0.0	1800.0
`TreeMap`	**P**	24640(120.2%)	0.942	58.7	1800.0	**61411(131.7%)**	0.961	30.1	1800.0
	O	11188	0.888	0.0	1800.0	26499	0.93	0.0	1800.0
`Median`	**P**	12452(53.5%)	0.29	313.7	1802.0	**17504(59.2%)**	0.325	156.9	1800.0
	O	8111	0.046	0.0	1800.0	10994	0.14	0.0	1800.0

4.4.4 Impact of parameter tuning. Intuitively, K_p and K_s play a key role in our method. A natural question arises: what impact do different parameter configurations have on the effectiveness of our method? Figures 3&4&5 shows the results of parameter tuning in the three preceding experiments. The X-axis shows the values of K_p and K_s, *e.g.*, 250 + 50 indicates that K_p is 250 and K_s is 50. The Y-axis shows the percentage increase in the number of explored paths or states. Each circular point represents a individual subject, while each diamond-shaped point represents the average value across all subjects under the corresponding parameter configuration. Note that subjects that terminate early in vanilla mode within the given time budget are ignored, such as `gsl_poly_complex_solve_cubic` in Experiment 2, and `Remainder`, `Triangle` and `Operations` in Experiment 3. Furthermore, we connect all the average values to indicate the fluctuations of these values with parameter configurations.

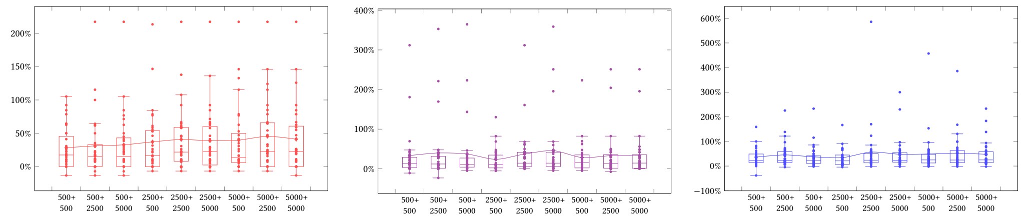
From these three figures, it can be observed that regardless of the search strategy and parameter configuration, the average values are always positive. Therefore, in each experiment, our method consistently demonstrates the ability to enhance the average efficiency of symbolic execution across all subjects. Similarly, the median is always positive in any situation, indicating that our method is consistently beneficial on more than half of the subjects. However, there are also some subjects for which the path growth is negative under certain parameter configurations. This suggests that picking inappropriate values for parameters may incur negative outcomes. In addition, if we focus on the impact of an individual parameter like K_s, *i.e.*, keeping the value of K_p fixed and changing the value of K_s, we do not observe any correlation between the experimental results and the value of K_s. The same holds for K_p. Therefore, the impact of these parameters is a combined effect.

Figure 6 presents the *standard deviations* [Forsyth 2018] of the increasing percentages in the number of explored paths or states for individual subjects under 9 parameter configurations. A



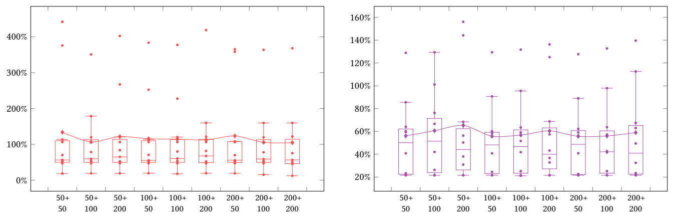
(a) DFS (b) BFS (c) RCN

Fig. 3. Results of parameter tuning in Experiment 1.



(a) DFS (b) BFS (c) RCN

Fig. 4. Results of parameter tuning in Experiment 2.



(a) DFS (b) BFS

Fig. 5. Results of parameter tuning in Experiment 3.

larger standard deviation suggests that our method's effectiveness on the corresponding subject is more sensitive to parameter tuning. Figure 6 indicates that the impact of parameter tuning is relatively small in Experiment 3. Due to the lower difficulty of constraint solving in Experiment 3, our method generates fewer partial solutions on many subjects. As a result, the effectiveness and overhead of our method are similar across different parameter configurations. On the contrary, the benefits of our method are significantly influenced by parameter tuning on certain subjects, especially in Experiment 2. Therefore, selecting appropriate values for K_p and K_s is crucial for these subjects, which remains an open problem.

4.4.5 Time & Memory overhead. Thanks to the tracing of symbolic variables (*c.f.*, discussion in Section 3.3), the time and memory overheads introduced by our method are generally minor compared to the 30-minute time budget and several gigabytes of memory budget. As Table 2&3&4 show, the time overheads of our method on major analysis tasks are typically less than 5.6%, 0.6% and 5.6% in the three experiments, respectively. The analysis tasks in Experiment 2 incur extremely low time overhead, which is primarily due to the shorter length of floating-point constraints. Furthermore, because the number of solver invocations in Experiment 2 is limited (owing to the

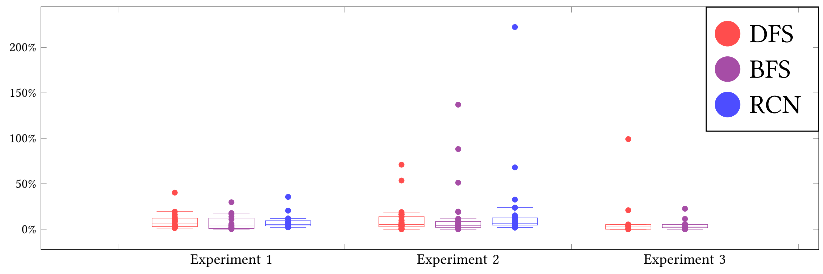


Fig. 6. Standard deviations of individual subjects in three experiments.

time-intensive nature of floating-point constraint solving), there are fewer cache entries available for expanding the partial solution-based cache. As a result, the size of the cache remains quite small, leading to naturally low query overhead. Similarly, the memory overhead introduced by partial solutions is rather low, never exceeding 3.9%, 2.3% and 5.0% in a single analysis task in the three experiments. This is because each partial solution only records a small fraction of bits corresponding to symbolic variables.

4.5 Threats to Validity

One internal threat of our work is the randomness brought by partial solution generation. Due to the use of many heuristics in the CDCL framework, the search process of SAT solving is not deterministic. Hence, solving the same query may result in different partial solutions, which may affect the experimental results. To ensure the soundness of our evaluation, we employed deterministic search strategies and repeated our evaluation five times. The external threat is that our benchmarks may be not representative. We will conduct more extensive experiments on various benchmarks in the future.

5 RELATED WORK

Modern symbolic execution engine usually uses a cache module. For example, KLEE uses both branch cache and counterexample cache [Palikareva and Cadar 2013]. The branch cache simply stores the solving results (SAT or UNSAT) of branch queries, while the counterexample cache makes good use of satisfying assignments, as clarified in Section 2. In addition, KLEE employs many query optimizations including expression rewriting, constraint set simplification, implied value concretization, and constraint independence [Cadar et al. 2008]. These optimizations simplify queries effectively and thereby increase the query cache's hit rate. Pex [Tillmann and de Halleux 2008], another well-known symbolic execution engine, also uses constraint cache and independent constraint optimization before invoking constraint solver.

Green [Visser et al. 2012] is a general constraint caching framework that can share results across runs, programs and tools. *Green* factorises the formula into independent sub-formulas and then canonizes the formula before storing into the cache. These steps can increase the cache hit rate effectively. *GreenTrie* [Jia et al. 2015] extends Green to check the implication relationship between constraints and store constraint solutions into tries. If the queried formula is implied by a satisfiable formula, or implies an unsatisfiable formula, it returns without invoking the constraint solver. *Recal* [Aquino et al. 2015] is another caching framework. *Recal* encodes simplified formula as a matrix to support efficient checking whether a formula is contained in the cache. *Recal+* [Aquino et al. 2015] extends *Recal* by supporting particular logical implication between formulas as like *GreenTrie*.

Most of existing caching frameworks only support rigorous form matching or logical implication checking. A recent proposed framework *Utopia* [Aquino et al. 2017] extends existing caching framework by introducing imprecise matching between formulas. Instead of checking the form of formulas, *Utopia* defines the distance between formulas based on the similarity of the solution spaces of formulas. Formulas with small distance tend to share their solutions although they may not be equivalent. *Utopia* can increase the reusability of cached results beyond equivalent or contained formulas. Cashew [Brennan et al. 2017] investigates the caching method in the context of model-counting constraint solving for quantitative program analysis. The existing cache frameworks focus on improving the usability of existing solutions. Our work is orthogonal to them. We increase the cache items with the same amount of constraint solving computation. With the help of our work, the cache can hit more formulas even when these formulas have not been met in the past analysis.

There also exist approaches that try to reduce the times of constraint solving or improve the solving's efficiency in symbolic execution. Muse [Zhang et al. 2020], which inspires our caching method, utilizes the partial solutions to generate multiple inputs by solving once for dynamic symbolic execution. However, our method aims to improve constraint solving caching that works with traditional symbolic execution in the state-forking style. This differs from Muse, which operates within the context of concolic execution. Speculative symbolic execution (SSE) [Zhang et al. 2012] proposes to execute the instructions speculatively by ignoring branch feasibility. Until a limit is reached, SSE invokes the constraint solver to check the feasibility. In case of feasibility, the solver invocations of the speculatively executed branches are saved. Liu *et al.* [Liu et al. 2014] propose to utilize stack-based incremental solving to boost the solving in symbolic execution. KLEE-Array [Perry et al. 2017] tries to eliminate the array constraints during symbolic execution to improve the efficiency. Shuai *et al.* [Shuai et al. 2021] propose to synergize symbolic execution and constraint solving by using the array index information calculated in symbolic execution to remove redundant array axioms during array constraint solving. Chen *et al.* [Chen et al. 2021] propose to online synthesize a solving strategy [de Moura and Passmore 2013] during symbolic execution to improve the efficiency of constraint solving.

6 CONCLUSION

Caching optimizes the constraint solving in symbolic execution. We present a partial solution-based constraint solving caching method to improve symbolic execution's constraint solving. Our method utilizes the partial solutions inside the constraint solver to improve caching's effectiveness. Our method is orthogonal with different search strategies and the existing caching mechanisms. Besides, due to the widespread existence of partial solutions, our method is applicable for many constraint solvers. To balance the benefits of caching and its overhead, our method relies on two key parameters. However, determining the optimal values for these parameters remains an open problem. We have implemented our method on two mainstream symbolic executors and SMT solvers. The experimental results demonstrate our partial solution-based caching method's effectiveness and efficiency. The future work lies in the following three aspects: (1) Apply our method on other SMT theories and symbolic executors; (2) Utilize the domain knowledge in constraint solving to improve the quality of partial solutions; (3) Design an automated parameter tuning algorithm that adaptively adjusts the values of K_p and K_s at runtime.

Data Availability.

The data of the experimental results is available at the following URL: https://github.com/zbchen/pscache.git.

ACKNOWLEDGMENTS

This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429, 62032024, and 62372162).

REFERENCES

2023. Equisatisfiability. https://en.wikipedia.org/wiki/Equisatisfiability. Accessed September 25, 2023.

2023. GNU Scientific Library. https://www.gnu.org/software/gsl/. Accessed September 25, 2023.

Andrea Aquino, Francesco A. Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. 2015. Reusing constraint proofs in program analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015.* 305–315. https://doi.org/10.1145/2771783.2771802

Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. 2017. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 427–437. https://doi.org/10.1109/ICSE.2017.46

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. 2017. Constraint normalization and parameterized caching for quantitative program analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 535–546. https://doi.org/10.1145/3106237.3106303

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings.* 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006.* 322–335. https://doi.org/10.1145/1180405.1180445

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* 1066–1071. https://doi.org/10.1145/1985793.1985995

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

Zhenbang Chen, Zehua Chen, Ziqi Shuai, Guofeng Zhang, Weiyu Pan, Yufeng Zhang, and Ji Wang. 2021. Synthesize solving strategy for symbolic execution. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021.* 348–360. https://doi.org/10.1145/3460319.3464815

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Leonardo Mendonça de Moura and Grant Olney Passmore. 2013. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune.* 15–44. https://doi.org/10.1007/978-3-642-36675-8_2

Xianghua Deng, Jooyong Lee, and Robby. 2006. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan.* 157–166. https://doi.org/10.1109/ASE.2006.26

Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers.* 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

David A. Forsyth. 2018. *Probability and Statistics for Computer Science.* Springer. https://doi.org/10.1007/978-3-319-64410-3

Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings.* 519–531. https://doi.org/10.1007/978-3-540-73368-3_52

John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann.

Jörg Hoffmann and Jana Koehler. 1999. A New Method to Index and Query Sets. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, Thomas Dean (Ed.). Morgan Kaufmann, 462–467. http://ijcai.org/Proceedings/99-1/Papers/067.pdf

Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015.* 177–187. https://doi.org/10.1145/2771783.2771806

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures - An Algorithmic Point of View.* Springer. https://doi.org/10.1007/978-3-540-74105-3

Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-point symbolic execution: a case study in n-version programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017.* 601–612. https://doi.org/10.1109/ASE.2017.8115670

Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. 2014. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. In *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings.* 284–299. https://doi.org/10.1007/978-3-319-13338-6_21

Hristina Palikareva and Cristian Cadar. 2013. Multi-solver Support in Symbolic Execution. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 53–68. https://doi.org/10.1007/978-3-642-39799-8_3

Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008.* 15–26. https://doi.org/10.1145/1390630.1390635

David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017.* 68–78. https://doi.org/10.1145/3092703.3092728

Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020.* 181–198.

Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and interval aware array constraint solving for symbolic execution. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021.* 361–373. https://doi.org/10.1145/3460319.3464826

Jan Taljaard. 2019. Optimised constraint solving for real-world problems.

Jan Taljaard, Jaco Geldenhuys, and Willem Visser. 2020. Constraint Caching Revisited. In *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12229)*, Ritchie Lee, Susmit Jha, and Anastasia Mavridou (Eds.). Springer, 251–266. https://doi.org/10.1007/978-3-030-55754-6_15

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4966)*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer, 134–153.

David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. Address-Aware Query Caching for Symbolic Execution. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021.* 116–126. https://doi.org/10.1109/ICST49551.2021.00023

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012.* 58. https://doi.org/10.1145/2393596.2393665

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761.

Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Tianqi Zhang, Kenli Li, and Ji Wang. 2020. Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* 846–857. https://doi.org/10.1145/3324884.3416645

Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative Symbolic Execution. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012.* 101–110. https://doi.org/10.1109/ISSRE.2012.8

Received 2023-09-28; accepted 2024-04-16