Q-KLEE: Quantitative Symbolic Execution with Redundant Path Pruning and Incremental Model Counting

ANONYMOUS AUTHOR(S)

Quantitative symbolic execution counts the number of program inputs if certain program properties hold or not. A common approach is to combine symbolic execution with model counting, where the symbolic execution engine enumerates all feasible paths, and a model counter computes the number of satisfying inputs for each path constraint. However, this approach faces two major limitations: (1) path explosion in symbolic execution and the expensive cost of model counting associated with each path constraint, and (2) every run of the model counting process is built from scratch. To overcome these limitations, we present *qklee*, a tool for efficient quantitative program analysis. First, we extend the *weakest precondition* based path pruning technique from qualitative to quantitative symbolic execution. Our approach relies on a novel technique to decompose the weakest precondition into two disjoint components, wp_{pass} and wp_{fail}, which precisely partition the input space of a pruned path. Second, we propose an incremental model counting technique that reuses SAT models from previously solved path constraints to guide the SAT solver in subsequent counting calls, avoiding cold starts and speeding up the counting process. We evaluate *qklee* on wildly-adopted benchmark programs and demonstrate that it achieves an average speedup of 3.70× over the baseline, with up to 15.88× on individual programs, while producing quantification results consistent with the baseline. We further conduct an ablation study to demonstrate the effectiveness of each optimization technique in *qklee*.

ACM Reference Format:
Anonymous Author(s). 2018. Q-KLEE: Quantitative Symbolic Execution with Redundant Path Pruning and Incremental Model Counting. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Quantitative symbolic execution goes beyond the traditional binary question of whether a safety property can be violated, but instead counts *how many* program inputs satisfy or violate the property. This capability has a broad range of practical applications. It serves as the foundation for probabilistic programming with symbolic execution [18, 40], where the goal is to compute the probability of certain execution results. In security analysis, it measures information leakage by counting how many inputs lead to a specific observable output, thus quantifying the amount of secret information an attacker can infer from the program [3, 4, 23, 28]. A common approach to implement quantitative symbolic execution is to combine an existing symbolic execution engine [5] with an off-the-shelf model counter [6, 33, 34, 41] together. The symbolic execution engine traverses the program and generates a path constraint for each feasible path, and the model counter computes the number of satisfying inputs for each constraint. The results are then aggregated across all paths to obtain the final quantification. Despite its effectiveness, this approach faces two major scalability limitations.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

The first limitation is path explosion. With the number of feasible paths grows exponentially in a larger program, symbolic execution usually suffers from the scalability issue and cannot finish exploring all paths within a reasonable time. This challenge is more severe in the quantitative setting, because each path requires a model counting call rather than a single satisifiability check to an SMT/SAT solver. State-of-the-art approximate model counters such as ApproxMC [6, 33, 34, 41] internally invoke a SAT solver multiple times per counting call, making model counting significantly more expensive than a SAT/SMT query. As a result, the total cost scales with both the number of paths and the per-path model counting cost. Prior work has proposed various techniques to mitigate path explosion in symbolic execution, including state merging [20, 30], program slicing [39], and *weakest precondition* (WP) based path pruning [15, 42, 43]. None of these techniques, however, can be directly applied to quantitative symbolic execution. State merging combines multiple execution states into one, which loses the precise per-path input counts needed for quantification. Program slicing removes statements that do not affect the property of interest, but such statements may still constrain the input domain and therefore affect the counts. Among the three, WP-based pruning is the most promising, because it specifically targets to remove redundant paths that share the same future execution behavior, thereby reducing both the symbolic execution overhead and the number of associated model counting calls. However, this technique was designed for *qualitative* analysis, where the goal is only to determine whether a safety violation is reachable. In the *quantitative* setting, it is unclear how to count the contribution of a pruned path, i.e., how many inputs executed along with the pruned path would lead to assertion pass or fail. Therefore, no existing path pruning technique is directly applicable to quantitative symbolic execution.

The second limitation is that in the baseline setting, each path constraint is passed to the model counter independently, so every counting call starts from scratch. This ignores the structural relationship between consecutive path constraints. Because symbolic execution forks new execution states at branch statements, consecutive path constraints often share a large common prefix and differ only in small portion of clauses added after the branching point. Although the SAT models of two sibling path constraints are mutually disjoint because they choose opposite branches at the same program location, they are structurally similar and often agree on the partial literal assignments for the shared prefix clauses. Therefore, there are opportunities to reuse the SAT models across path constraints to accelerate the SAT solving in the model counter.

To address both limitations, we present *qklee*, a tool for efficient quantitative symbolic execution. *qklee* makes two key contributions: (1) the WP-based quantitative path pruning, and (2) the incremental model counting technique. For **WP-based quantitative path pruning**, we extend the WP-based path pruning from qualitative to quantitative symbolic execution by decomposing the weakest precondition $wp[l]$ into two disjoint components, $wp_{pass}[l]$ and $wp_{fail}[l]$, which summarize path suffixes leading to assertion pass and assertion fail, respectively. When a path is pruned, we recover its contribution to the assertion outcomes by using the decomposed weakest preconditions with the strongest postcondition accumulated up to the pruning point. For **incremental model counting**, we store SAT models generated during each counting call and reuse them to bias the internal SAT solver of ApproxMC when processing the next path constraint, speeding up the SAT solving phase inside the model counter.

We have implemented *qklee* as a software based on the prestigious KLEE symbolic execution engine [5] and leverages the Z3 SMT solver [10] to determine the path feasibility. We further build the pipeline for counting the number of satisfying inputs for each path constraint following `csb` [32], where the STP [17] solver is used for bit-blasting and an incremental version of ApproxMC [6, 33, 34, 41] is used for model counting. The *qklee* tool is evaluated on a benchmark consists of 52 C programs, including 2 paper examples, 8 VerifyPIN variants [13, 14, 19, 29, 37], and 42 programs adapted from the SV-COMP benchmark suite [2]. Experimental results show that *qklee* achieves an average

speedup of 3.70× over the baseline with up to 15.88× on individual benchmarks, while producing quantification results consistent with the baseline (deviations within 3.2%, attributable to the inherent approximation of ApproxMC rather than unsoundness of our optimizations). An ablation study further confirms that both components individually contribute to the overall performance gain, and their combination yields an additional 16.9% improvement beyond the better single component.

The rest of the paper is organized as follows. Section 2 presents a motivating example, formalizes the quantitative analysis problem, and introduces the baseline approach along with its two limitations. Section 3 gives an overview of *qklee* and the running example used throughout the paper. Our two technical contributions are presented in Section 4 (WP-based path pruning for quantitative symbolic execution) and Section 5 (incremental model counting). Section 6 describes the experimental setup and evaluation results, followed by an ablation study in Section 7. Section 8 reviews related work, and Section 9 concludes.

2 Motivation

In this section, we use a motivating example to illustrate how a baseline approach works for the quantitative analysis of a program based on symbolic execution and model counting techniques. Then, we point out its limitations and the advantages of our new method.

2.1 A Motivating Example

We first present a motivating example in Fig. 1, which was originally used to quantify information leakage in the Cyclic Redundancy Check (CRC) algorithm [28]. To better illustrate our motivations, we slightly simplified the goal of the example program[1] and turn it into an assertion statement that checks whether the input byte collides with the computed checksum. Specifically, the program takes an 8-bit byte unsigned integer `ch` and the current CRC state `check` as input, and then updates `check` by folding `ch` into it for a fixed number of iterations (eight in standard CRC-8, one per bit).[2] At the end, the programs checks whether the updated `check` collides the original input byte `val`. In quantitative program analysis, we are interested in *counting* how many inputs make the assertion pass versus fail. For this example, our goal is to compute how many collisions would occur for all possible 8-bit input bytes with an existing CRC state.

```
1 void GETCRC6(unsigned char check, unsigned char ch) {
2    int i, stf = 3, val = ch;
3    for ( i=0; i < 6; ++i) {
4        if ( check & 0x80 ) {
5            check <<= 1;
6            if (ch & 0x80) { check = check | 0x01; }
7            else { check = check & 0xFE; }
8            check = check ^ 0x85;
9        } else {
10           check <<= 1;
11           if (ch & 0x80) { check = check | 0x1; }
12           else { check = check & 0xFE; }
13       }
14       ch <<= 1;
15   }
16   check >>= stf; // checksum obtained here
17   assert(check != val); // check collisions
18 }
```

Fig. 1. A motivating example to quantify the number of collisions between the input byte and obtained checksum the Cyclic Redundancy Check (CRC) algorithm, adapted from [28].

2.2 The Problem

To systematically introduce our goal, we formalize the quantitative problem as follows. Let P be a program with a set of symbolic inputs X (e.g., an integer array) ranging over a finite integer domain $\mathcal{D}(X)$, and φ be the safety property of P and expressed as an assertion statement `assert`(φ), embedded at the end of P. The goal is to compute the number of concrete inputs that make the

[1] In [28], the example in Fig. 1 quantifies information leakage about the input `ch` from the observed checksum value `check`.
[2] In the motivating example used in this paper, we use a reduced-round variant that performs 6 iterations to make collisions more observable. This modification is for demonstration purposes and should not be interpreted as standard CRC-8.

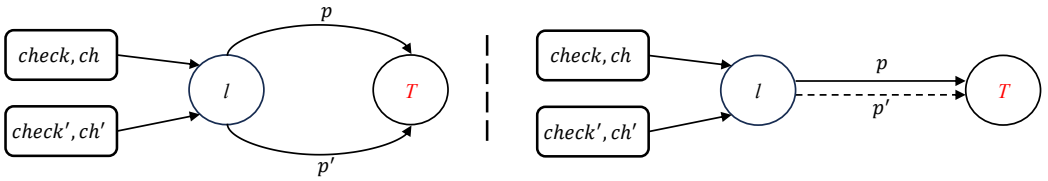


Fig. 2. Illustration for redundant paths in Fig. 1 during symbolic execution. Starting from a certain program location l, different inputs may result in distinct paths to the assertion statement at the end (Left), or they may share the same path from l to the program end (Right). In the latter case, the duplicate part would be pruned by existing work [15, 42, 43] of using the *weakest precondition* based path summary. "T" here denotes the assertion test and "p" indicates the execution path.

assertion pass or fail. Let $\mathcal{P}$ and $\mathcal{F}$ denote the sets of passing and failing inputs, respectively, where

$$\mathcal{P} = \{x \in \mathcal{D}(X) \mid P(x) \text{ reaches the assertion and } \varphi \text{ holds}\}$$
$$\mathcal{F} = \{x \in \mathcal{D}(X) \mid P(x) \text{ reaches the assertion and } \neg\varphi \text{ holds}\}\,.$$

Accordingly, $|\mathcal{P}|$ and $|\mathcal{F}|$ are the number of passing and failing inputs to be computed, respectively.

2.3 The Baseline Approach

To achieve the quantification defined above, a straightforward baseline approach is to combine symbolic execution and model counting techniques in the following manner. First, the input program P is passed to the popular LLVM [22] compiler and transformed into an Intermediate Representation (IR). Then, the LLVM IR is fed to an off-the-shelf symbolic execution engine, e.g., KLEE [5], to traverse all feasible paths on the Control Flow Graph (CFG) of P and generate a path constraint PC_π for each feasible path $\pi \in \Pi$. We denote the obtained set of path constraints by symbolic execution as $\mathcal{PC} = \{pcon_\pi \mid \pi \in \Pi\}$. Finally, for all path constraints in $\mathcal{PC}$, a state-of-the-art model counting solver, e.g., ApproxMC [6, 33, 34, 41], is invoked to compute the number of SAT models for each path constraint The final results are aggregated to obtain $|\mathcal{P}|$ and $|\mathcal{F}|$.

For the motivating example in Fig. 1, we use the baseline approach to explore all feasible paths and obtain

$$|\mathcal{P}| = 65280 \quad \text{and} \quad |\mathcal{F}| = 256.$$

The symbolic execution tool explores 4,352 paths in total during the process, with calls to the model counter for each path constraint, using 403.4 seconds in total. While the baseline approach completes this analysis in reasonable time, it would become much more expensive for larger programs with wider input spaces due to path explosion.

2.4 Limitations of the Baseline Approach

In this section, we point out the two major limitations of the baseline approach.

2.4.1 Limitation #1: No Path Redundancy Elimination. One of the most well-known issues of symbolic execution is path explosion, i.e., the number of feasible paths grows exponentially with the number of branches in the program. For the motivating example in Fig. 1, while exploring about 4,000 paths appear to be manageable, this number can quickly grow beyond expectation for larger programs. This issue becomes even more severe for quantitative analysis, because each feasible path constraint is associated with a model counting call. Especially, model counting is significantly more expensive than a satisfiability check using a SAT/SMT solver (e.g., Z3 [10]) only, because state-of-the-art model counting techniques typically involve multiple calls to a SAT

solver internally. As a result, the overall cost grows with both the number of feasible paths and the per-path model counting cost. The baseline approach is therefore inefficient because it has to explore all feasible paths.

However, not all explored paths are necessary. As illustrated in Fig. 2, starting from a certain program location l during the symbolic execution, different inputs may lead to distinct paths to the assertion statement at the end (the left part), or they may share the same path from l to the program end (the right part). In the latter case, exploring the shared path multiple times is redundant, because it will lead to the exact same program behavior and the same assertion outcome, regardless of the concrete inputs that reach the location l. The redundant paths can not only increase the expense of the symbolic execution, which results in the notorious path explosion issue, but also significantly increase the cost of the quantitative symbolic execution because of the associated model counting calls to each path.

To mitigate the path explosion issue, prior works have proposed various techniques by various means, such as the forward-style state merging techniques [20, 30], backward-style path pruning techniques [15, 42, 43], or slicing program parts of interest [39]. However, none of these techniques can be directly applied to the quantitative analysis problem settings. For the state merging techniques, merging multiple paths into a single state would lose the precise count of individual inputs, as the merged path constraint represents multiple execution paths that need to be counted separately. The slicing-based techniques may remove program statements that do not directly affect the property of interest, but such statements may still contribute to the input domain and thus affect the counting results. Among these prior works, the *weakest precondition* (WP) based path pruning technique [15, 42, 43] is the most promising approach for our problem, because it aims to prune the redundant paths as shown in Fig. 2 that are guaranteed to satisfy or violate the same property. The key idea of this method is to leverage the *weakest preconditions* as a summary of explored path suffixes to prune the redundancy. However, the WP-based path pruning technique was designed for *qualitative* analysis, i.e., symbolic execution only checks whether inputs violate the safety property or not. Accordingly, the pruning only removes paths that will result in previously-explored suffixes, without considering the number of inputs associated with them. But in our *quantitative* analysis setting, the goal is to quantify how many inputs would (not) violate the safety property, so the existing WP-based path pruning is not sound. Therefore, an efficient counting-aware symbolic execution technique is needed for the quantitative analysis problem.

2.4.2 Limitation #2: Model Counting from Scratch. The second limitation of the baseline approach lies in the model counting. In the baseline approach, each path constraint is passed to the model counter independently, so every run starts from scratch, which is inefficient. Although different path constraints usually have disjoint sets of SAT models, they are often structurally related and share large common prefix parts, because of the nature of symbolic execution of forking paths at branch statements. For example, $a \wedge b \wedge c$ and $a \wedge b \wedge \neg c$ can be regarded as two consecutive path constraints sharing the common prefix $a \wedge b$ and are forked at statement $\texttt{if}(c)$. While the SAT models can not be directly shared to other path constraints, it would be still helpful if there is a way to reuse them to avoid cold starts in the model counter. To this end, an incremental model counting technique is needed to relieve the overhead of model counting calls.

2.4.3 Our Approach in a Nutshell. To address the aforementioned limitations of the baseline approach, we make two key contributions: (1) we extend the *weakest precondition* based path pruning from *qualitative* to *quantitative* analysis, by decomposing the weakest precondition into two disjoint components, wp_{pass} and wp_{fail}, that separately track passing and failing behaviors of the explored path suffixes; and (2) we propose an incremental model counting technique that reuses SAT models across different runs of the model counter. By incorporating these two techniques, our

approach can significantly improve the efficiency of the quantitative analysis for the motivating example in Fig. 1. Our approach obtains the same results as the baseline and reduces the execution time of the motivating example from 403.4 seconds to 100.2 seconds, achieving a 4× speed up. The rest of the paper presents the details of these techniques and evaluates their effectiveness.

3 Method Overview

In this section, we present an overview of *qklee*, an efficient quantitative program analysis technique using symbolic execution and model counting.

3.1 Top-Level Procedure

Algorithm 1 presents the top-level procedure of *qklee*, where the two contributions of this work are represented by two subroutines: WP-SE and INC-MC. Given an input program P with symbolic inputs over an integer domain $\mathcal{D}(X)$, the WP-SE subroutine symbolically traverse P in a depth-first manner and produces a set of path constraints for all feasible paths in P. The execution results of symbolic execution are saved as a set and denoted as C. Each element $(pc, status) \in C$ is a tuple and represents a path constraint pc and its execution outcome $status$, i.e., whether the input would violate the safety property φ or not. The tuple $(pc, status)$ is further passed to the INC-MC subroutine, which conducts approximate model counting in an incremental manner to compute the number of SAT models for pc. Finally, these number of model counts are aggregated based on the status of the explored path into $|\mathcal{P}|$ or $|\mathcal{F}|$, indicating the number of passing or failing inputs, respectively.

Input: P with inputs $\{x | x \in \mathcal{D}(X)\}$
Output: $|\mathcal{P}|$ and $|\mathcal{F}|$

1 **Procedure** *QKLEE(P)*:
2 Initialize $|\mathcal{P}| \leftarrow 0$ and $|\mathcal{F}| \leftarrow 0$
3 $C \leftarrow$ WP-SE(P)
4 **foreach** $(pcon, status) \in C$ **do**
5 $n \leftarrow$ INC-MC($pcon$)
6 **if** $status =$ *FAIL* **then**
7 $|\mathcal{F}| \leftarrow |\mathcal{F}| + n$
8 **else**
9 $|\mathcal{P}| \leftarrow |\mathcal{P}| + n$
10 **return** $(|\mathcal{P}|, |\mathcal{F}|)$

Algorithm 1: Top-level procedure.

3.2 A Running Example

Before diving into the details of our technique, we introduce the running example in Figure 3, which we will use throughout later this paper. Although our method also applies to the motivating example in Figure 1, we choose a simpler program here to better illustrate our approach and ease the understanding of readers. The running example in Fig. 3 has three input variables a, b, and c, which are all 32-bit unsigned integers bounded in $[0, 10^4)$. In total, there are $10^4 \times 10^4 \times 10^4 = 10^{12}$ possible inputs constrained by the three `assume` statements at lines 2-4. The safety property φ is abstracted as an assertion statement, and is further interpreted as an `if` statement at line 10, where an assertion failure occurs if the control flow falls through the *else* branch, i.e., when $c \geq 3000$.

We present all 8 feasible paths in the running example along with their path constraints in Table 1. The left part of the table shows the results of the baseline approach using the standard symbolic execution. Column 1 lists the path IDs, indexed by the depth-first search order. Column 2 indicates which branches are taken when symbolic execution encounters an `if` statement and forks new states. Column 3 presents the path constraints collected along each path. Note that we omit the bound constraints over the input variables, i.e., $0 \leq a, b, c < 10^4$, which are shared across all paths for brevity. Column 4 show

```
1 void foo(uint a, uint b, uint c) {
2    uint n = 0;
3    assume(0 <= a && a < 10000);
4    assume(0 <= b && b < 10000);
5    assume(0 <= c && c < 10000);
6    if (a > 5000) n += 1;
7        else n -= 1;
8    if (b > 5000) n += 2;
9        else n -= 2;
10   if (c < 3000) return;
11       else __assert_fail();
12 }
```

Fig. 3. A running example for demonstrating the efficient quantitative symbolic execution.

Table 1. Symbolic computation for running example in Fig. 3. The table shows the execution results of the standard symbolic execution and our *weakest precondition* based symbolic execution with path pruning. All model counts indicated by # should be scaled by 10^{11}. We omit this factor for brevity.

Path	Br No.	Standard Symbolic Execution			Weakest Precondition-based Symbolic Execution				
		Path Condition	Kind	#	wp_{pass}	wp_{fail}	$wp = wp_{pass} \vee wp_{fail}$	Kind	#
1	7	$a \geq 5000$	Fail	1.75	$false$	$false$	$false$	Fail	1.75
	9	$a \geq 5000 \wedge b \geq 5000$			$false$	$false$	$false$		
	11	$a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$			$false$	$false$	$false$		
2	7	$a \geq 5000$	Pass	0.75	$false$	$a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$	$a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$	Pass	0.75
	9	$a \geq 5000 \wedge b \geq 5000$			$false$	$b \geq 5000 \wedge c \geq 3000$	$b \geq 5000 \wedge c \geq 3000$		
	10	$a \geq 5000 \wedge b \geq 5000 \wedge c < 3000$			$false$	$c \geq 3000$	$c \geq 3000$		
3	7	$a \geq 5000$	Fail	1.75	$a \geq 5000 \wedge b \geq 5000 \wedge c < 3000$	$a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$	$a \geq 5000 \wedge b \geq 5000$	Pass	0.75
	8	$a \geq 5000 \wedge b < 5000$			$b \geq 5000 \wedge c < 3000$	$b \geq 5000 \wedge c \geq 3000$	$b \geq 5000$		
	11	$a \geq 5000 \wedge b < 5000 \wedge c \geq 3000$			$c < 3000$	$c \geq 3000$	$true$	Fail	1.75
4	7	$a \geq 5000$	Pass	0.75	-	-	-	(skip)	
	8	$a \geq 5000 \wedge b < 5000$			-	-	-		
	10	$a \geq 5000 \wedge b < 5000 \wedge c < 3000$			-	-	-		
5	6	$a < 5000$	Fail	1.75	$a \geq 5000 \wedge c < 3000$	$a \geq 5000 \wedge c \geq 3000$	$a \geq 5000$	Pass	1.5
	9	$a < 5000 \wedge b \geq 5000$			$c < 3000$	$c \geq 3000$	$true$		
	11	$a < 5000 \wedge b \geq 5000 \wedge c \geq 3000$			$c < 3000$	$c \geq 3000$	$true$	Fail	3.5
6	6	$a < 5000$	Pass	0.75	-	-	-	(skip)	
	9	$a < 5000 \wedge b \geq 5000$			-	-	-		
	10	$a < 5000 \wedge b \geq 5000 \wedge c < 3000$			-	-	-		
7	6	$a < 5000$	Fail	1.75	-	-	-	(skip)	
	8	$a < 5000 \wedge b < 5000$			-	-	-		
	11	$a < 5000 \wedge b < 5000 \wedge c \geq 3000$			-	-	-		
8	6	$a < 5000$	Pass	0.75	-	-	-	(skip)	
	8	$a < 5000 \wedge b < 5000$			-	-	-		
	10	$a < 5000 \wedge b < 5000 \wedge c < 3000$			-	-	-		

whether the path leads to an assertion failure (i.e., *Fail*) or not (i.e., *Pass*), and finally, Column 5 lists the number of satisfying inputs for each path, expected to be computed by a model counter. By aggregating the obtained numbers of each path, we have

$$|\mathcal{F}| = 4 \times 1.75 \times 10^{11} = 7 \times 10^{11}, \quad |\mathcal{P}| = 4 \times 0.75 \times 10^{11} = 3 \times 10^{11}.$$

The result sums to 10^{12}, consistent with the total input space $10^4 \times 10^4 \times 10^4$. This result also aligns with our manual computation: since whether the assertion fails depends solely on the value of c and is regardless of the branches taken on a and b, the number of failing inputs is $10^4 \times 10^4 \times 7000 = 7 \times 10^{11}$, confirming the correctness of the baseline approach. While the baseline approach would explore all 8 paths and invoke the model counter 8 times, our WP-based symbolic execution will only explore 4 paths and invoke the model counter 6 times. Among the 4 explores paths, 2 of them are completely executed while the other 2 are partially explored. We will illustrate how our method works in details in the following sections.

4 Efficient Quantitative Symbolic Execution

We present our efficient quantitative symbolic execution technique in this section. Our approach leverages a novel *weakest precondition* based path pruning technique to reduce the number of paths explored during the symbolic execution, while ensuring the soundness of the quantification result.

4.1 The Baseline Quantitative Symbolic Execution

Our efficient quantitative symbolic execution is illustrated in Algorithm 2. If ignoring the highlighted parts in blue, Algorithm 2 presents a standard symbolic execution procedure that visits the Control Flow Graph (CFG) of program P using depth-first search. The procedure starts with an initial state s_0 and explores all feasible paths by recursively invoking the Explore function. During exploration, the procedure calls the NextState function, shown in Algorithm 3, to compute the successor state from the current state and an event. An *event* is represented as a tuple $\langle l, inst, l' \rangle$, where l denotes the current program location, *inst* is the instruction to be executed at l, and l' is the next program location after executing *inst*. The NextState function maintains a symbolic memory map *mem*

Input: P with inputs $\{x | x \in \mathcal{D}(X)\}$
Output: C, a set of $(pc, status)$ pairs

1 **Procedure** *WP-SE(P)*:
2 Initialize $wp_{pass} \leftarrow false$, $wp_{fail} \leftarrow false$
3 Initialize $sp \leftarrow true$, $C \leftarrow \emptyset$, $S \leftarrow \{\}$
4 **Function** *Explore(s)*:
5 **if** *Pruning(s)* **then**
6 **return**
7 S.push(s)
8 **if** *s is a branching point* **then**
9 **foreach** $t \in s.branch$ **do**
10 $s' \leftarrow$ NextState(s, t)
11 Explore(s')
12 **else if** *s is an internal node* **then**
13 $s' \leftarrow$ NextState(s, t)
14 Explore(s')
15 **else**
16 status = CheckAssertion()
17 UpdateWP(s, status, false)
18 add $(pc, status)$ to C
19 TestGen(s)
20 S.pop()
21 Explore(s_0), s_0 is the initial state of P
22 **return** C

Algorithm 2: The procedure for efficient quantitative symbolic execution.

1 **Function** *NextState(s, t)*:
2 $\langle l, sp, mem\rangle \leftarrow s$
3 $\langle l, inst, l'\rangle \leftarrow t$
4 **if** *inst is* **assume**(c) **then**
5 $s' \leftarrow \langle l', sp \wedge c, mem\rangle$
6 **else if** *inst is* $v := expr$ **then**
7 $s' \leftarrow \langle l', sp[v/expr], mem[v/expr]\rangle$
8 **else**
9 $s' \leftarrow \langle l', sp, mem\rangle$
10 **return** s'

Algorithm 3: The NextState subroutine.

1 **Function** *Pruning(s)*:
2 $\langle l, sp, mem\rangle \leftarrow s$
3 **if** $\models (sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l]))$ **then**
4 $C \leftarrow C \cup \{(sp \wedge wp_{pass}[l], pass)\}$
5 $C \leftarrow C \cup \{(sp \wedge wp_{fail}[l], fail)\}$
6 UpdateWP(s, pass, true)
7 UpdateWP(s, fail, true)
8 **return** *true*
9 **else**
10 **return** *false*

Algorithm 4: The Pruning subroutine.

and the *strongest postcondition* formula sp along the execution path. For branching statements (i.e., **assume**(c) instructions), sp is updated by conjoining the branching condition c. For assignment statements (i.e., $v := expr$ instructions), both sp and mem are updated by substituting variable v with expression $expr$. When reaching the end of a path, the procedure invokes CheckAssertion to verify whether the safety property φ is violated and calls TestGen to generate test inputs for the executed path. These path constraints are then passed to a model counting subroutine to compute the number of satisfying assignments.

4.2 The WP-based Path Pruning

We now introduce the *weakest precondition* based path pruning technique as a foundation for our efficient quantitative symbolic execution.

The core idea of the weakest precondition based pruning is to identify and eliminate redundant paths that share common suffixes with previously explored paths. Let a path in symbolic execution be a sequence of state transitions from the initial state s_0 to a terminal state s_n, denoted as $\pi = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} s_n$, where each transition is associated with an event $e_i = \langle l, inst, l'\rangle$ that executes instruction $inst$ at location l and transitions to location l'. At any program location l along the path, a path π can be decomposed into a *prefix* π_{pre} and a *suffix* π_{post}, such that $\pi = \pi_{pre} \cdot \pi_{post}$. In prior work [15, 42, 43], a key observation is that different paths π and π' may usually have distinct prefixes, but share the same suffix when reaching a certain program location l, i.e.,

$$\pi = \pi_{pre} \cdot \pi_{post}, \quad \pi' = \pi'_{pre} \cdot \pi_{post}.$$

To avoid such redundancy, a key idea is to compute the *weakest precondition* wp at program location l to summarize all previously explored path suffixes from l. Let $\Pi^l_{post} = \{\pi^{(1)}_{post}, \pi^{(2)}_{post}, \ldots, \pi^{(k)}_{post}\}$ denote the set of path suffixes that have been explored from location l, where each $\pi^{(i)}_{post}$ represents a suffix starting from l from different path. The weakest precondition $wp[l]$ precisely characterizes the set Π^l_{post} by capturing all possible behaviors along these suffixes. Formally, $wp[l]$ is a logical formula that characterizes all program states at location l whose execution will only result in explored suffixes. In other words, for any state satisfying $wp[l]$, continuing execution from l will only lead to behaviors already covered by the suffixes in Π^l_{post}.

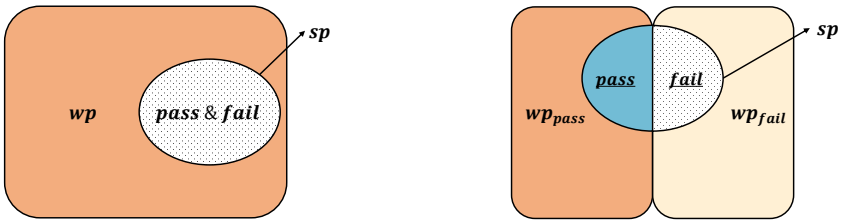
Once a new incoming path reaches location l, the pruning decision is determined by checking the validity of the implication $sp \rightarrow wp[l]$, i.e., $\models sp \rightarrow wp[l]$. Here, sp denotes the *strongest postcondition* accumulated along the incoming path up to location l, which precisely characterizes the exact program state reached by following the current path prefix. The term "strongest" indicates that sp describes the most restrictive (or precise) condition satisfied by states reachable via this specific path, while "weakest" in $wp[l]$ indicates the most permissive condition that still guarantees all suffixes have been explored. When the implication $sp \rightarrow wp[l]$ validates, it means the current state (characterized by sp) completely falls within the set of states (characterized by $wp[l]$) whose suffixes have already been explored. Therefore, continuing execution from the current state will only traverse path suffixes already in Π^l_{post}, making the incoming path redundant and safe to prune.

Returning to Algorithm 2, the highlighted parts in blue illustrate the key enhancements that integrate WP-based pruning into the baseline symbolic execution. Note that Algorithm 2 already presents our extended version for quantitative analysis, where wp is decomposed into wp_{pass} and wp_{fail} for quantification purposes. In existing WP-based pruning, there is only a single wp that merges all explored suffixes regardless of their assertion outcomes, and the pruning condition is simply checking $sp \rightarrow wp[l]$. Compared to the standard symbolic execution (non-highlighted parts), our approach introduces two main modifications: (1) at each branch location of the program, the Pruning function checks whether the current path can be pruned by validating the implication $sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$; and (2) upon completing each path, the UpdateWP function updates wp_{pass} and wp_{fail} to reflect the newly explored path suffixes, no matter whether the path was fully executed or pruned. The detailed implementations of Pruning and UpdateWP are presented in Algorithms 4 and 5, respectively, which we will explain in the following subsections. The soundness of the WP-based pruning will be discussed in Section 4.6.

4.3 The Challenges

However, the existing WP-based pruning technique cannot be directly applied to our quantitative settings. The main challenge arises when an incoming path is determined to be redundant and pruned. In traditional symbolic execution, pruning a path simply means skipping its exploration since the future path behaviors are already covered by previous paths. The primary goal of symbolic execution is to determine if there are any inputs that can trigger assertion violations, not *how many*. In contrast, quantitative symbolic execution aims to quantify the number of inputs that lead to certain outcomes, i.e., passing or failing the assertion. When a path is pruned, the existing WP-based approach provides no information about how the pruned path would contribute to the overall count. As a result, directly applying the existing WP-based pruning would lead to incomplete quantification results, making it unsound for our purpose.

Formally, when a path is pruned at location l with strongest postcondition sp, we need to determine how many inputs satisfying sp will eventually pass versus fail the assertion. That is, we need to compute the contributions to $|\mathcal{P}|$ and $|\mathcal{F}|$ from the pruned path. The existing weakest precondition $wp[l]$ only indicates that all behaviors from states satisfying sp are covered by



(a) Extracting all inputs from the pruned path using wp.

(b) Extracting pass and fail inputs using wp_{pass} and wp_{fail}.

Fig. 4. Extracting and decomposing inputs from pruned paths. The weakest precondition wp conjuncted with sp captures all inputs from the pruned path. The wp_{pass} and wp_{fail} further separate the inputs into (1) passing assertions and (2) failing assertions, respectively.

previously explored suffixes in Π^{l}_{post}, but it does not separate the contributions to $|\mathcal{P}|$ from those to $|\mathcal{F}|$. Without the precise breakdown, the pruning cannot correctly reflect the behaviors of the pruned path from the quantitative perspective, and it is thus unsound.

To address this challenge, we decompose the weakest precondition $wp[l]$ into two separate components: $wp_{pass}[l]$ and $wp_{fail}[l]$, rather than maintaining a single wp that merges all explored suffixes regardless of their outcomes. With this decomposition, $wp[l]$ can be expressed as $wp_{pass}[l] \vee wp_{fail}[l]$. For path pruning, we still validate the implication $sp \rightarrow wp[l]$, which is equivalent to $sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$. For quantifying the contributions from a pruned path, we compute $|\mathcal{P}|$ and $|\mathcal{F}|$ using $sp \wedge wp_{pass}[l]$ and $sp \wedge wp_{fail}[l]$, respectively. We detail the implementation of this idea in the next subsection.

4.4 The Solutions

We present the *weakest precondition* based path pruning for quantitative symbolic execution in this section, as our solutions to the challenges discussed in Section 4.3.

Recall the Algorithm 2, which illustrates our efficient quantitative symbolic execution with WP-based path pruning. Our pruning procedure starts with initializing the decomposed weakest precondition, wp_{pass} and wp_{fail}, as $false$, to summarize the previously explored path suffixes that lead to passing and failing the assertion, respectively. During exploration, the Pruning function is invoked at each branch to check whether the current path can be pruned. If the pruning condition holds, i.e., $\models sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$, the path is pruned and two path constraint tuples $(sp \wedge wp_{pass}[l], pass)$ and $(sp \wedge wp_{fail}[l], fail)$ are added to C to quantify the contributions from the pruned path. When a path is fully executed without pruning, one path constraint tuple is added to C based on the assertion outcome. In both cases, the UpdateWP function is invoked to update wp_{pass} and wp_{fail} based on the explored path suffix. The detailed implementations of the Pruning and UpdateWP functions are presented in Algorithms 4 and 5, respectively.

We now explain the Pruning function in Algorithm 4. The Pruning function still follows the same principle of pruning a path as we discussed in Section 4.2, i.e., validating $sp \rightarrow wp[l]$. Here, since the wp has been decomposed into wp_{pass} and wp_{fail} in the quantitative settings, the pruning condition is thus checking whether $sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$ holds.

Definition 4.1 (Pruning Condition). Given a state $s = \langle l, sp, mem \rangle$ at program location l, the path is eligible for pruning if and only if:

$$\models sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l]).$$

When the implication is valid, it indicates that all possible behaviors from the current state have been covered by previously explored path suffixes. However, it is still unclear how the pruned path contributes to the overall quantification, i.e., how many inputs from the pruned path will pass versus fail the assertion. To overcome this challenge, we leverage the strongest postcondition sp to help decompose the path inputs. Formally, when the pruning condition validates, we know that continuing the execution of the current path would only result in known behaviors from previously explored paths. To precisely quantify how many inputs will satisfy the known behaviors from the pruned path, a natural approach is to compute the conjunction $sp \wedge wp[l]$, which captures all program states that (1) satisfy the current path prefix characterized by sp, and (2) have their suffixes already explored as summarized by $wp[l]$. The decomposition is illustrated in Figure 4a. In the figure, the rectangle represents the wp, and the oval represents the sp. The oval is fully contained within in the rectangle, indicating the pruning condition validates. The overlapped area thus indicates the logical formula $sp \wedge wp[l]$, which captures all inputs from the pruned path. However, this conjunction alone only yields the total number of inputs from the pruned path without distinguishing the inputs between passing and failing the assertion.

To address this limitation, we further used two disjoint components: $wp_{pass}[l]$ and $wp_{fail}[l]$ that are decomposed from $wp[l]$.

Definition 4.2 (Weakest Precondition Decomposition). We decompose the weakest precondition $wp[l]$ into two disjoint formulas:

$$wp[l] = wp_{pass}[l] \vee wp_{fail}[l], \quad \text{where} \quad wp_{pass}[l] \wedge wp_{fail}[l] \equiv false,$$

where $wp_{pass}[l]$ characterizes the set of program states at location l from which all execution paths lead to assertion passing outcomes, while $wp_{fail}[l]$ characterizes states leading to assertion failures.

This decomposition allows us to precisely partition the inputs from the pruned path according to their assertion outcomes. As illustrated in Figure 4b, by computing $sp \wedge wp_{pass}[l]$, we extract exactly the inputs that follow the current path prefix and will eventually pass the assertion based on previously explored suffixes, indicated by the blue region that are shared by wp_{pass} and sp in the figure. Similarly, $sp \wedge wp_{fail}[l]$ extracts inputs that will fail the assertion, indicated by the hatched region that are shared by wp_{fail} and sp.

Quantification via Conjunction. For a pruned path at location l with strongest postcondition sp, we compute the contributions to passing and failing outcomes as:

$$\mathcal{P} = \{x \in \mathcal{D}(X) \mid x \models (sp \wedge wp_{pass}[l])\}$$
$$\mathcal{F} = \{x \in \mathcal{D}(X) \mid x \models (sp \wedge wp_{fail}[l])\}$$

where $\mathcal{P}$ and $\mathcal{F}$ denote the sets of inputs from the pruned path that pass and fail the assertion, respectively. The model counting subroutine then computes $|\mathcal{P}|$ and $|\mathcal{F}|$ to quantify these contributions.

Lemma 4.3 (Properties of Decomposition). *The decomposition satisfies the following properties:*

$$\mathcal{P} \cap \mathcal{F} = \emptyset \quad \textit{(disjoint)}$$
$$\mathcal{P} \cup \mathcal{F} = \{x \in \mathcal{D}(X) \mid x \models sp\} \quad \textit{(complete)}$$

ensuring that we partition all inputs from the pruned path without overlap or omission.

Our key insight is that the conjunction between sp and the decomposed wp_{pass} and wp_{fail} enables us to leverage information from previously explored paths to precisely quantify the outcomes of an incoming path without actual execution, which not only reduces computational overhead in

symbolic execution and model counting but also ensures soundness in quantitative analysis. This is the core novelty of our approach: while existing WP-based pruning simply discards redundant paths, our technique extracts quantitative information from the pruned paths by decomposing the weakest precondition in an outcome-specific manner.

Returning to the PRUNING function in Algorithm 4, if a path is determined to be pruned, the PRUNING function further generates two tuples: $(sp \wedge wp_{pass}[l], pass)$ and $(sp \wedge wp_{fail}[l], fail)$, representing quantification for the inputs of passing and failing the assertion from the pruned path, respectively. These tuples would be subsequently passed to a model counter, and the function calls the UPDATEWP subroutine twice, to update both wp_{pass} and wp_{fail} based on the explored path.

We now explain the UPDATEWP function in Algorithm 5. The weakest precondition is computed following Dijkstra's method [11]. As stated in Section 4.2, the path in symbolic execution is represented as a sequence of state transitions, where each transition is associated with an event $t_i = \langle l, inst, l' \rangle$. We denote the sequence of events along a path as $\tau = t_1, t_2, \ldots, t_n$. The UPDATEWP function takes three parameters: the state s, the assertion outcome $status$ (either $pass$ or $fail$), and a boolean flag $pruned$ indicating whether the path was pruned. The function first selects the part of WP to update (wp_{pass} or wp_{fail}) based on the $status$ parameter, then computes the weakest precondition by traversing the event sequence τ backward from the current location to the initial state.

```
1  Function UpdateWP(s, status, pruned):
2      path = ⟨t_0, t_1, ..., t_n⟩ ← s
3      ⟨l, sp, mem⟩ ← s
4      if status = pass then
5          wp ← wp_pass
6      else
7          wp ← wp_fail
8      if ¬pruned then
9          ŵp ← true
10     else
11         ŵp ← wp[l]
12     while t = path.pop() do
13         ⟨l, inst, l'⟩ ← t
14         if inst is assume(c) then
15             ŵp ← ŵp ∧ c
16             wp[l] ← wp[l] ∨ ŵp
17         else if inst is v := expr then
18             ŵp ← ŵp[v/expr]
19     return
```

Algorithm 5: The procedure for updating *weakest precondition*.

The function initializes $\hat{wp}$, a local variable that serves as the base case for the backward computation. For a path that completes execution without pruning ($pruned = false$), the function sets $\hat{wp} = true$, reflecting that the execution of the path has successfully reached a terminal state without errors. For a pruned path ($pruned = true$), the function sets $\hat{wp} = wp[l]$, using the existing weakest precondition value at the pruning location l. The backward computation then propagates $\hat{wp}$ to earlier locations along the path. Note that both wp_{pass} and wp_{fail} are initialized as $false$ in Algorithm 2, indicating that no path suffixes have been explored yet before starting the symbolic execution.

During backward traversal, for **assume**(c) statements, the local variable is first conjoined with the branching condition, i.e., $\hat{wp} \leftarrow \hat{wp} \wedge c$, and then the weakest precondition at the location is updated by taking the disjunction with the local variable, i.e., $wp[l] \leftarrow wp[l] \vee \hat{wp}$. For assignment statements, the local variable is updated by substituting the assigned variable with the assignment expression, i.e., $\hat{wp} \leftarrow \hat{wp}[v/expr]$, where $[v/expr]$ denotes the substitution of all occurrences of variable v with expression $expr$ in $\hat{wp}$. This backward propagation continues until all events in the path have been processed, updating the weakest precondition summary at each branch location along the path.

4.5 Applied to the Running Example

To illustrate how our WP-based pruning with decomposition technique works, we apply it to the running example in Figure 3. Table 1 compares standard symbolic execution with our WP-based approach. For our method (right half), the first three columns track wp_{pass}, wp_{fail}, and their

disjunction $wp = wp_{pass} \vee wp_{fail}$ at each branch location in the program along the execution. The fourth column shows the execution outcome (Pass, Fail, or skip for pruned paths), and the fifth column gives the number of models that satisfy the path constraints, scaled by 10^{11}.

We now demonstrate how our new WP-based pruning reduces the number of paths to be executed and the number of calls to the model counter from 8 to 4 and 8 to 6, respectively. Throughout the execution, we evaluate the pruning condition at each branch location by checking whether $\models sp \rightarrow (wp_{pass} \vee wp_{fail})$ holds. This validity check can be equivalently transformed into a SAT check:

$$\models sp \rightarrow (wp_{pass} \vee wp_{fail}) \Leftrightarrow UNSAT(sp \wedge \neg(wp_{pass} \vee wp_{fail})). \quad (1)$$

If the formula $sp \wedge \neg(wp_{pass} \vee wp_{fail})$ is UNSAT, the pruning condition holds and the path can be pruned; otherwise, the path continues execution. Note that in the following discussion, we omit variable n from sp formulas for brevity, as they do not affect the model counting because the modelc counting would be conducted over the input variables a, b, and c.

The execution starts with Path#1, which takes the *else* branches of all three if statement at branches 7, 9, and 11, respectively, reaching the assertion failure with path constraint $a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$. Since this is the first path executed, both wp_{pass} and wp_{fail} are initialized as $false$ at each branch point. Applying the SAT check from Equation 1, we have:

$$UNSAT(sp \wedge \neg(false \vee false)) \Leftrightarrow UNSAT(sp \wedge true) \Leftrightarrow UNSAT(sp).$$

Since sp is the path constraint of a feasible path, it is SAT. Therefore, the pruning condition fails at all three branch locations and Path#1 is fully executed. Upon completion, the path reaches the assertion failure and the path constraint $a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$ is sent to the model counter, yielding 1.75×10^{11} failing inputs. Note that the execution of Path#1 is identical to standard symbolic execution, as no prior path information is available for pruning.

After Path#1 completes, the weakest preconditions are updated via backward propagation from the terminal state. Since Path#1 reaches the assertion failure, only wp_{fail} is updated while wp_{pass} remains $false$. The UpdateWP function is invoked with parameters $(s, fail, false)$, where $false$ indicates the path completed without pruning. The function first initializes the local variable $\hat{wp} = true$, serving as the base case for the backward computation. Starting from the last event in the path (at branch 11), the backward propagation begins. For the assume statement at branch 11, the local variable is conjoined with the branching condition: $\hat{wp} = true \wedge \neg(c < 3000)$, which simplifies to $c \geq 3000$. Then $wp_{fail}[l_{11}]$ is updated by disjuncting with the local variable: $wp_{fail}[l_{11}] = false \vee (c \geq 3000) = c \geq 3000$. The propagation continues backward through the remaining branches. At branch 9, wp_{fail} becomes $b \geq 5000 \wedge c \geq 3000$, and at branch 7, it becomes $a \geq 5000 \wedge b \geq 5000 \wedge c \geq 3000$.

Path#2 forks from Path#1 at the third if statement, taking the *then* branch at branch 10. At this branch point, the execution reaches with $sp = a \geq 5000 \wedge b \geq 5000 \wedge c < 3000$. We apply the SAT check from Equation 1 with the current wp values: $wp_{pass} = false$ and $wp_{fail} = c \geq 3000$:

$$\begin{aligned} & SAT((a \geq 5000 \wedge b \geq 5000 \wedge c < 3000) \wedge \neg(false \vee c \geq 3000)) \\ \Leftrightarrow\ & SAT((a \geq 5000 \wedge b \geq 5000 \wedge c < 3000) \wedge \neg(c \geq 3000)) \\ \Leftrightarrow\ & SAT(a \geq 5000 \wedge b \geq 5000 \wedge c < 3000). \end{aligned}$$

This formula is SAT, so the pruning condition would not happen and Path#2 continues execution to completion. Path#2 successfully passes the assertion. The path constraint $a \geq 5000 \wedge b \geq 5000 \wedge c < 3000$ is sent to the model counter, yielding 0.75×10^{11} passing inputs. After Path#2 completes, wp_{pass} is updated through backward propagation from the passing state. The UpdateWP function is invoked with parameters $(s, pass, false)$. Since the path completed without pruning, the local

variable is still initialized as $\hat{wp} = true$. Starting from the last event at branch 10, the local variable becomes $\hat{wp} = true \wedge (c < 3000) = c < 3000$, and $wp_{pass}[l_{10}]$ is updated to $false \vee (c < 3000) = c < 3000$. Continuing backward, at branch 9, wp_{pass} becomes $b \geq 5000 \wedge c < 3000$, and at branch 7, it becomes $a \geq 5000 \wedge b \geq 5000 \wedge c < 3000$.

In the DFS traversal order, Path#3 is forked at branch 8 from Path#1, taking the *then* branch. At branch 8, the pruning condition is evaluated with $sp = a \geq 5000 \wedge b < 5000$ and $wp_{pass} \vee wp_{fail} = b \geq 5000$, which is:

$$UNSAT((a \geq 5000 \wedge b < 5000) \wedge \neg(b \geq 5000)) \Leftrightarrow\ UNSAT(a \geq 5000 \wedge b < 5000).$$

The formula is SAT, so the execution continues until reaching branch 10, where the pruning condition is evaluated again with $sp = a \geq 5000 \wedge b < 5000 \wedge c < 3000$ and $wp_{pass} \vee wp_{fail} = c < 3000 \vee c \geq 3000$:

$$\begin{aligned} &UNSAT((a \geq 5000 \wedge b < 5000 \wedge c < 3000) \wedge \neg(c < 3000 \vee c \geq 3000)) \\ \Leftrightarrow\ &UNSAT(a \geq 5000 \wedge b < 5000 \wedge c < 3000 \wedge false). \end{aligned}$$

Now, the formula is UNSAT, and the execution stops at branch 10, where Path#3 is partially explored and Path#4 is completely pruned. With the pruned path, two path constraints are generated for the contributions of passing and failing inputs from the pruned path by conjoining sp with wp_{pass} and wp_{fail}, respectively:

$$\begin{aligned} sp \wedge wp_{pass}[l_{10}] &= (a \geq 5000 \wedge b < 5000) \wedge c < 3000 \\ sp \wedge wp_{fail}[l_{10}] &= (a \geq 5000 \wedge b < 5000) \wedge c \geq 3000 \end{aligned}$$

The two formulas yield 0.75×10^{11} passing inputs and 1.75×10^{11} failing inputs, respectively.

After the pruning, both wp_{pass} and wp_{fail} are updated by invoking UpdateWP twice with parameters $(s, pass, true)$ and $(s, fail, true)$, respectively. Since the path was pruned at branch 10, the local variable is initialized using the existing wp at the pruning location. For updating wp_{pass}, the function sets $\hat{wp} = wp_{pass}[l_{10}] = c < 3000$. The backward propagation then extends this value to earlier locations: at branch 9, wp_{pass} becomes $b < 5000 \wedge c < 3000$, and at branch 7, it becomes $a \geq 5000 \wedge b < 5000 \wedge c < 3000$. Similarly, for updating wp_{fail}, the function sets $\hat{wp} = wp_{fail}[l_{10}] = c \geq 3000$ and propagates backward.

Path#4 is completed skipped because of the pruning happens in Path#3, and Path#5 is the last one to be executed in our method. Path#5 is forked at branch 6 from Path#1, taking the *then* branch. The pruning happens at branch 9, where $sp = a < 5000$ and $wp_{pass} \vee wp_{fail} = true$:

$$UNSAT((a < 5000) \wedge \neg(true)) \Leftrightarrow\ UNSAT(a < 5000 \wedge false).$$

The formula is UNSAT, so the execution stops at branch 9, resulting in partial exploration of Path#5 and complete pruning of Paths#6, 7, and 8. For the pruned paths, two path constraints are generated in the same way as for Path#3:

$$\begin{aligned} sp \wedge wp_{pass}[l_9] &= (a < 5000 \wedge b \geq 5000) \wedge c < 3000 \\ sp \wedge wp_{fail}[l_9] &= (a < 5000 \wedge b \geq 5000) \wedge c \geq 3000. \end{aligned}$$

These two formulas yield 1.5×10^{11} passing inputs and 3.5×10^{11} failing inputs, respectively.

By aggregating the results from the 4 explored paths and 6 generated path constraints, our method obtain a final result as

$$\begin{aligned} |\mathcal{P}| &= 0.75 \times 10^{11} + 0.75 \times 10^{11} + 1.5 \times 10^{11} = 3.0 \times 10^{11}, \\ |\mathcal{F}| &= 1.75 \times 10^{11} + 1.75 \times 10^{11} + 3.5 \times 10^{11} = 7.0 \times 10^{11}, \end{aligned}$$

which are identical to the baseline method that explores all 8 paths without pruning. The result empirically demonstrates the correctness of our WP-based pruning with decomposition technique.

4.6 Soundness Proof

We now prove the soundness of our WP-based pruning for quantitative symbolic execution. We define the soundness property in the context of quantitative symbolic execution.

Definition 4.4 (Quantitative Soundness). Let $|\mathcal{P}|_{base}$ and $|\mathcal{F}|_{base}$ be the quantification results from the baseline symbolic execution, and let $|\mathcal{P}|_{wp}$ and $|\mathcal{F}|_{wp}$ be the results from our WP-based approach. The WP-based pruning is *sound* if and only if $|\mathcal{P}|_{wp} = |\mathcal{P}|_{base}$ and $|\mathcal{F}|_{wp} = |\mathcal{F}|_{base}$.

By prior work [15, 42, 43], the *weakest precondition* based path pruning is sound in terms of the path coverage, i.e., it does not miss unknown path suffixes when pruning a path.

Theorem 4.5 (Soundness of WP-based Pruning [15, 42, 43]). *Let $wp[l]$ be the weakest precondition at program location l, summarizing all previously explored path suffixes starting from l. If the pruning condition $\models sp \rightarrow wp[l]$ holds for an incoming state $s = \langle l, sp, mem\rangle$, then for every input x satisfying sp, continuing execution from l will only traverse path suffixes already in Π^{l}_{post}. That is, no new path suffix exists from l under the current state.*

By Theorem 4.5, the existing WP-based pruning can safely prune redundant paths without affecting soundness. However, Theorem 4.5 alone does not guarantee the correctness of quantification, because it can not distinguish between inputs that lead to passing versus failing the assertion for pruned paths. We next prove that the decomposition of wp into wp_{pass} and wp_{fail} correctly preserves this distinction.

Lemma 4.6 (Equivalence of Decomposed Weakest Precondition). *The decomposition of wp into wp_{pass} and wp_{fail} does not affect the soundness of WP-based pruning established in Theorem 4.5. Specifically, the following two properties hold:*

(1) ***Pruning condition equivalence.*** *The pruning condition $\models sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$ in Algorithm 4 is equivalent to $\models sp \rightarrow wp[l]$ in the non-decomposed setting;*

(2) ***Computation equivalence.*** *Using UpdateWP to update wp_{pass} and wp_{fail} separately in Algorithm 5 computes the same results as updating wp in the non-decomposed setting.*

Proof. **Property 1.** By Definition 4.2, $wp[l] = wp_{pass}[l] \vee wp_{fail}[l]$. Therefore, the pruning condition remains logically equivalent when replacing $wp[l]$ with $wp_{pass}[l] \vee wp_{fail}[l]$. The pruning condition is thus equivalent in the *qualitative* WP-based pruning.

Property 2. The weakest precondition is updated in two cases: when a path completes execution and when a path is pruned. We show the equivalence for both.

Case 1: a path completes execution with outcome status. The UpdateWP function is called with $pruned = false$, initializing $\hat{wp} = true$. Let ψ_l denote the formula computed by backward propagation from $\hat{wp} = true$ through the execution trace to each location l. In the non-decomposed setting, this updates a unified wp at each location: $wp[l]' = wp[l] \vee \psi_l$. In the decomposed setting, the function selects either wp_{pass} or wp_{fail} based on $status$ and applies identical backward computation with the same initial value and the same event sequence. Suppose $status = pass$ (the $fail$ case is symmetric). Then $wp_{pass}[l]' = wp_{pass}[l] \vee \psi_l$ and $wp_{fail}[l]' = wp_{fail}[l]$. Therefore, $wp_{pass}[l]' \vee wp_{fail}[l]' = (wp_{pass}[l] \vee \psi_l) \vee wp_{fail}[l]$. By Definition 4.2, $wp[l] = wp_{pass}[l] \vee wp_{fail}[l]$, so $wp_{pass}[l]' \vee wp_{fail}[l]' = wp[l] \vee \psi_l = wp[l]'$. Hence the two forms of weakest preconditions are logically equivalent at every location along the path.

Case 2: a path is pruned at location l. **Step 1: Equivalence of initial values.** For the pruned path, UpdateWP is called once in the non-decomposed setting, starting from $\hat{wp} = wp[l]$, and

twice in the decomposed setting, starting from $\hat{wp} = wp_{pass}[l]$ and $\hat{wp} = wp_{fail}[l]$ respectively. By the invariant established in Case 1, $wp[l] = wp_{pass}[l] \vee wp_{fail}[l]$ holds at the pruning location l. Therefore, the initial values of the two settings are logically equivalent before the backward WP computation begins.

Step 2: Distributivity of backward computation. We show that the backward WP computation has the distributivity property, i.e., it distributes over disjunction. For branching instructions **assume**(c), the backward computation conjoins the branch condition: $\hat{wp} \mapsto \hat{wp} \wedge c$. Considering $\phi_1 \vee \phi_2$ as the logical formulas of wp_{pass} and wp_{fail}, we have $(\phi_1 \vee \phi_2) \wedge c \equiv (\phi_1 \wedge c) \vee (\phi_2 \wedge c)$. For assignment instructions $v := expr$, the backward computation performs substitution: $\hat{wp} \mapsto \hat{wp}[v/expr]$. Substitution also distributes over disjunction, so $(\phi_1 \vee \phi_2)[v/expr] \equiv \phi_1[v/expr] \vee \phi_2[v/expr]$. Since the WP computation only involves these two types of instructions, and each step of the backward computation distributes over disjunction, the entire backward computation has the distributivity property.

Step 3: Preservation of equivalence. Let f denote the transfer function that represents how the backward WP computation processes each instruction, i.e., $f(\hat{wp}) = \hat{wp} \wedge c$ for **assume**(c) and $f(\hat{wp}) = \hat{wp}[v/expr]$ for $v := expr$. For any prior location l' along the path, let $F = f_n \circ f_{n-1} \circ \cdots \circ f_1$ denote the consecutive application of transfer functions from the pruning location l to l', where each f_i corresponds to one instruction along the backward traversal. Since each f_i distributes over disjunction (Step 2), their composition F also distributes over disjunction. Combined with the equivalence of initial values (Step 1), we obtain at each prior location l':

$$F(\hat{wp}) = F(wp_{pass}[l] \vee wp_{fail}[l]) = F(wp_{pass}[l]) \vee F(wp_{fail}[l]) = wp_{pass}[l'] \vee wp_{fail}[l'].$$

Therefore, we demonstrate the equivalence of the WP computation for both the decomposed and non-decomposed settings.

□

Lemma 4.7 (Correctness of Quantification via Conjunction). *Let $s = \langle l, sp, mem \rangle$ be a pruned state satisfying the pruning condition $\models sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$. Then the conjunctions $sp \wedge wp_{pass}[l]$ and $sp \wedge wp_{fail}[l]$ precisely partition the inputs satisfying sp by their assertion outcome. For every input $x \models sp \wedge wp_{pass}[l]$, continuing execution from l leads to a passing outcome, and for every input $x \models sp \wedge wp_{fail}[l]$, continuing execution leads to a failing outcome. Consequently, $|sp \wedge wp_{pass}[l]| + |sp \wedge wp_{fail}[l]| = |sp \wedge wp[l]|$.*

Proof. By Theorem 4.5, the pruning condition ensures that every input satisfying sp only traverses path suffixes already in Π^l_{post}. By Definition 4.2, $wp_{pass}[l]$ characterizes exactly the inputs whose path suffixes from l lead to passing outcomes, and $wp_{fail}[l]$ characterizes those leading to failing outcomes. Therefore, for any $x \models sp \wedge wp_{pass}[l]$, the path suffix taken by x from l is already explored and leads to a passing outcome; symmetrically, any $x \models sp \wedge wp_{fail}[l]$ leads to a failing outcome. □

We now combine the above results to establish the main soundness theorem.

Theorem 4.8 (Quantitative Soundness). *The WP-based pruning in Algorithm 4 is sound the sense of Definition 4.4: $|\mathcal{P}|_{wp} = |\mathcal{P}|_{base}$ and $|\mathcal{F}|_{wp} = |\mathcal{F}|_{base}$.*

Proof. In both the baseline and the WP-based approach, fully explored paths contribute identically to $|\mathcal{P}|$ or $|\mathcal{F}|$, since they produce the same path constraint and obtain the same quantification result.

Furthermore, consider a path pruned at location l with state $s = \langle l, sp, mem \rangle$. By Theorem 4.5, the pruning condition $\models sp \rightarrow (wp_{pass}[l] \vee wp_{fail}[l])$ guarantees that every input satisfying

sp would only follow path suffixes already explored in Π^{l}_{post}. In the baseline, these inputs would be enumerated across individual path suffixes from l, each classified as passing or failing. By Lemma 4.6, the weakest preconditions with decomposed setting are computed equivalently to the baseline. By Lemma 4.7, the conjunctions $sp \wedge wp_{pass}[l]$ and $sp \wedge wp_{fail}[l]$ form disjoint partitions of the inputs satisfying sp, which precisely correspond to the number of passing and failing inputs that the baseline would count by enumerating all path suffixes from l. Therefore, the model counts $|sp \wedge wp_{pass}[l]|$ and $|sp \wedge wp_{fail}[l]|$ exactly match the counts of passing and failing inputs for continuing executing a pruned path starting from l in the baseline. Since this holds for every pruned path, we conclude that $|\mathcal{P}|_{wp} = |\mathcal{P}|_{base}$ and $|\mathcal{F}|_{wp} = |\mathcal{F}|_{base}$. □

5 Incremental Model Counting

In this section, we present our incremental model counting technique that reuses SAT models across path constraints to accelerate model counting.

5.1 Motivation

Model counting is one of the major bottlenecks in quantitative symbolic execution. As described in Algorithm 1, each path constraint produced by symbolic execution is associated with a model counting call. Although our WP-based symbolic execution introduced in Section 4 can effectively reduce the number of generated path constraints by pruning the redundancy, the remaining path constraints are still considerable, especially for larger programs. Moreover, model counting itself is computationally expensive. For the state-of-the-art sample-based model counting [6, 7, 33, 34, 41], each model counting process typically involves multiple SAT solver calls to enumerate solutions within cells partitioned by hash functions. Accordingly, our quantitative program analysis procedure would generate a large number of SAT models in total.

An intuitive idea to speed up model counting is to directly reuse the SAT models obtained from one path constraint for the next. However, this is generally not feasible. Due to the nature of symbolic execution, the path constraints generated from different paths in the program are typically disjoint, as they correspond to different choices at the branching points. For example, in the running example (Table 1), the path constraints for Path#1 and Path#2 are $a \geq 5000 \wedge b \geq 500 \wedge c \geq 3000$ and $a \geq 5000 \wedge b \geq 500 \wedge c < 3000$, respectively, which are mutually exclusive because the two paths fork at line 10 in Fig. 3.

Even if such direct reuse were feasible, making model counting itself incremental presents a separate challenge. In a high-level view, the approximate model counting can be viewed as a search procedure for the appropriate number of hash constraints m, where m XOR constraints partition the solution space of ϕ into 2^m cells from which the total model count is estimated. Any attempt to make model counting incremental must preserve the theoretical guarantees and avoid violating the efficient logarithmic search for m. Therefore, implementing incremental model counting is challenging due to the sophisticated and delicate internal logic of the approximate model counting algorithm.

Nevertheless, we observe that SAT models from consecutive path constraints often agree on a large portion of the literal assignments. Formally, let ϕ and ϕ' be the logical formulas obtained from two consecutive path constraints during symbolic execution. By the nature of symbolic execution, the two path constraints typically share a large common prefix and only differ at and after the branching point where the paths fork. We can express this relationship as

$$\phi = \phi_{pre} \wedge \phi_{suf}, \quad \phi' = \phi_{pre} \wedge \phi'_{suf},$$

where ϕ_{pre} encodes the shared prefix constraints and ϕ_{suf}, ϕ'_{suf} encode the differing suffix parts. For example, Path#1 and Path#2 in the running example shown in Table 1 share the prefix $a \geq$

$5000 \wedge b \geq 5000$ and differ only on the last branch condition ($c \geq 3000$ versus $c < 3000$). Since the shared prefix ϕ_{pre} often dominates the formula, a SAT model of ϕ could satisfy most clauses in ϕ' except for a few clauses in the suffix ϕ'_{suf}.

While such a SAT model of ϕ can not be directly reused as a valid solution for ϕ', it can still provide useful guidance for the Conflict-Driven Clause Learning (CDCL) search [24] of the SAT solver when solving ϕ'. In particular, when searching SAT models for ϕ', we expect to improve the efficiency by a heuristic that biases the literal assignments toward the values in the SAT model of ϕ, which is likely closer to a valid solution for ϕ' than a random assignment, and thus would finally turn into a speedup for the overall model counting procedure. We explain how we implement this idea in our incremental model counting procedure through the later sections.

5.2 The INC-MC Procedure

Algorithm 6 presents our incremental model counting procedure INC-MC. If replace the highlighted INC-COUNTER function with an off-the-shelf model counter, such as ApproxMC, the procedure becomes the baseline method described in Section 2.3. The INC-MC procedure first converts a path constraint *pcon* in the Bitvector (BV) theory into a propositional CNF formula ϕ through bit-blasting, and then passes ϕ to the INC-COUNTER function along with ϵ (tolerance) and δ (confidence) to obtain an approximate model count. The INC-COUNTER function is our incremental extension of the ApproxMC algorithm proposed in [7], where the highlighted steps are our incremental heuristics that reuse SAT models across different path constraints. As a remainder of this section, we first introduce the overall workflow of the existing sample-based approximate model counting procedure in Section 5.3, and then present our incremental technique in Section 5.4.

Input: *pcon*, a path constraint
Output: n, an approximate model count

1 **Procedure** *INC-MC(pcon)*:
2 $\phi \leftarrow$ BITBLAST(*pcon*)
3 $n \leftarrow$ INC-COUNTER(ϕ, ϵ, δ)
4 **return** n
5 **Function** *INC-COUNTER*(ϕ, ϵ, δ):
6 $pivot \leftarrow 2\lceil e^{3/2}(1 + 1/\epsilon)^2 \rceil$
7 $T \leftarrow 17\lceil \log_2(3/\delta) \rceil$
8 **for** $i \leftarrow 1$ **to** T **do**
9 Select m and choose random h^m
10 $M \leftarrow$ LOOKUPMODEL(h^m)
11 **if** $M \neq \emptyset$ **then**
12 SETPOLARITY(M)
13 $c \leftarrow$ BSAT($\phi \wedge h^m = \alpha^m, pivot$)
14 STOREMODELS(h^m)
15 $n_i \leftarrow c \cdot 2^m$
16 **return** median($n_1, \ldots, n_T$)

Algorithm 6: The incremental model counting procedure.[4]

5.3 The Baseline Model Counting Procedure

We first introduce the principal ideas of the existing sample-based approximate model counting procedure as the foundation for our incremental technique.

The core idea of ApproxMC [7] is to use random XOR-based hash functions to uniformly partition the solution space of a CNF formula ϕ into cells, and then estimate the total model count from the number of solutions within the sampled cells. Concretely, the model counter constructs m independent random XOR constraints over the projected variable set S (denoted as h^m) to partition the solution space of ϕ into 2^m roughly equal-sized cells. Each XOR constraint is in the form $x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_k} = \alpha$, where x_{i_j} are literals randomly selected from S and $\alpha \in \{0, 1\}$ is a random bit. After a cell is fixed, a SAT solver is invoked to enumerate all satisfying assignments of $\phi \wedge h^m = \alpha^m$ by iteratively finding a SAT model and adding a banning clause to exclude it, until the formula becomes UNSAT. Finally, the total number of SAT models can be estimated as $c \cdot 2^m$,

[4] This is a simplified version of the ApproxMC algorithm proposed in [7]. We omit many details of the original algorithm and present a skeleton to highlight our incremental contributions. We refer readers to the original paper for full details, including but not limited to the theoretical guarantees, and the logarithmic search for value of m.

where c is the number of enumerated SAT models in the selected cell. The value of m is adaptively selected by a logarithmic search procedure, ensuring that each cell contains a minimum number of solutions and thus satisfies the tolerance requirement ϵ. To ensure the PAC guarantee, the counting process is repeated over T rounds with independently chosen random hash functions. The final estimate is the median of all results. [6].

5.4 The Incremental Model Counting Procedure

We introduce the INC-Counter function in Algorithm 6, which is our incremental extension of the state-of-the-art model counter, ApproxMC [6, 34, 41].

Our incremental technique operates at the SAT solver level. ApproxMC leverages CryptoMiniSat [36] as its built-in SAT solver to enumerate all SAT models of $\phi \wedge h^m = \alpha^m$. CDCL [24] is the core algorithm behind most modern SAT solvers, including CryptoMiniSat. It extends the classic Davis-Putnam-Logemann-Loveland (DPLL) algorithm [8] with clause learning and non-chronological backtracking. A CDCL solver operates in the following major phases: (1) *Decision*: the solver picks an unassigned literal and assigns it a truth value (polarity); (2) *Boolean Constraint Propagation* (BCP): unit propagation is applied to deduce further literal assignments forced by the current partial assignment; (3) *Conflict Analysis*: if BCP produces a conflict, the solver analyzes the implication graph and derives learned clauses from the conflict; (4) *Backtracking*: the solver backtracks to an earlier decision level determined by the learned clause and resumes the search from step (1). This loop repeats until either a satisfying assignment is found or the formula is proved UNSAT. There are multiple heuristics used in the CDCL search, such as the Variable State Independent Decaying Sum (VSIDS) heuristic for literal selection [26], and various strategies to determine the polarity of the selected literal, such as random assignment or saved polarity.

In our quantitative symbolic execution procedure, the WP-SE procedure generates a sequence of path constraints $\phi_0, \phi_1, \ldots, \phi_n$, each of which is passed to the model counter. In the baseline method, each path constraint ϕ_i is handled independently, and the model counting is conducted from scratch. However, as discussed in Section 5.1, that consecutive path constraints often share a large common prefix due to the nature of symbolic execution. To avoid the cold start of each individual model counting, we propose to leverage this observation and reuse the SAT models saved from the previous path constraint ϕ_{i-1} to incrementally guide the model counting for ϕ_i. Specifically, we target the *saved polarity* strategy in the decision phase of the CDCL search in the SAT solver. When the SAT solver is invoked by the model counter internally, we bias the solver by initializing the polarity map with a previous SAT model. Since the previous SAT model may already satisfy many clauses from the shared prefix ϕ_{pre}, the polarity map provides partially correct assignments. We expect that doing so would guide the CDCL search toward a more likely correct assignment for ϕ_i, which would reduce the total number of time for the decision procedures.

We now describe how this idea is realized in Algorithm 6. When enumerating the solutions of $\phi_{i-1} \wedge h^m = \alpha^m$, we store the generated SAT models and index them by the hash function h^m used in that round, using the StoreModels function. When a new path constraint ϕ_i arrives and the model counter selects a hash function h^m for a counting round, we look up whether any SAT models from a previous run were stored under the same h^m by the LookupModel function. If a matching model M is found, we use it to set the polarity map of the SAT solver before the enumeration begins. During the decision phase of the CDCL search, the selected literal will be assigned a truth value according to the saved polarity map. Note that we only reuse models stored under the same hash function when enumerating the solutions of $\phi_i \wedge h^m = \alpha^m$. The reason is because, the hash function h^m introduces additional XOR constraints to the original formula ϕ_i, which are used to ban certain solutions that are outside the current sampled cell. If the SAT models obtained from a different hash function $h^{m'}$ were reused, they may be directly banned by the hash function h^m, meaning that the

solution is beyond the selected cell, and has already been ruled out. Therefore, reusing such models would not provide speedup for the SAT solver. In contrast, it may even cause slowdown, since it breaks the search heuristics of the SAT solver and leads the solver to a less efficient search path.

The core idea of our incremental model counting design is to find a non-intrusive way without violating the efficient search for m but still effectively speed up the model counting. Instead of interfering with how m is adaptively selected, we let the model counting proceed with its original logic and only reuse SAT models from the previous path constraint when the hash function matches. Since the main logic of the sampling search is not changed, our incremental technique preserves all theoretical guarantees of the original model counting procedure. In contrast, we speed up the SAT solving when enumerating the solutions of a partitioned cell, which is the major computational overhead of this counting procedure. In a word, our incremental technique is designed to be non-intrusive that can speed up model counting without affecting the theoretical guarantees.

6 Experiments

6.1 Summary of Implementation Details

We have implemented *qklee*, a tool for efficient quantitative symbolic execution that builds upon three major components: the symbolic execution tool KLEE [5], the state-of-the-art SMT solver Z3 [10], and the Probably Approximately Correct (PAC) style model counting solver ApproxMC [6, 33, 34, 41]. Our tool first complies the input C programs into the Intermediate Representation (IR) using LLVM compiler [22] and then feeds the IR to KLEE for symbolic execution. On top of KLEE, we implemented our *weakest precondition* based pruning algorithm for quantitative symbolic execution, which not only can prune the redundant paths, but also preserve the soundness of the quantitative analysis. With the path constraints generated by KLEE, our tool further conducts incremental model counting. We employ CSB [32] as a front end to convert the path constraints written in Bitvector Theory (BV-Theory) to CNF formulas via bit-blasting. The CNF formulas are further input to an enhanced ApproxMC solver for incremental model counting with SAT model reuse from the previous runs.

In total, our implementation consists of approximately 4,500 lines of C++ code for WP-based pruning atop KLEE, and 500 lines of C++ code for incremental model counting built against ApproxMC. In addition, there are approximately 1,200 lines of Python code that combines the two components into a tool.

6.2 Benchmark Programs

To the best of our knowledge, there is no widely adopted benchmark suite specifically designed for *quantitative* symbolic execution with model counting. We therefore curated a benchmark dataset from multiple sources. Our benchmark set consists of three parts. First, we include the two motivating example programs used throughout the paper. Second, we include the widely used VerifyPIN test suite, which has been used in recent work on symbolic execution [13, 15] and program analysis [14, 19, 29, 37, 38]. Third, we adapt benchmarks from the *array-crafted* category of SV-COMP. We choose this category because it includes programs with integer inputs and assertion statements emebed to the end. Since processing unbounded loops in symbolic execution is challenging, we manually bound the number of loop iterations to a fixed number. In total, the dataset contains 52 C programs, including 2 paper examples, 8 VerifyPIN variants, and 42 adapted SV-COMP programs. We will release the benchmark dataset upon acceptance of the paper.

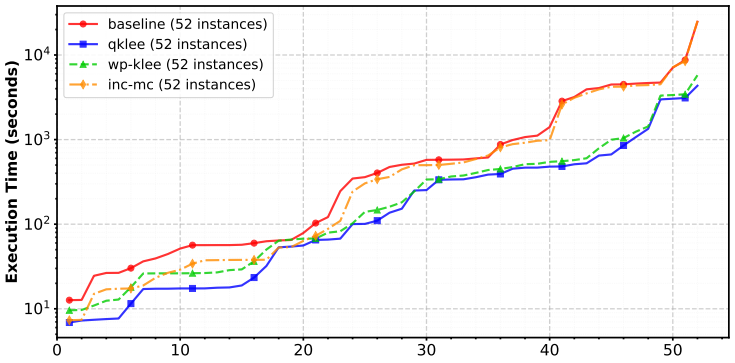


Fig. 5. Cactus plot of end-to-end execution time for *baseline*, *wp-klee*, *inc-mc*, and *qklee*. Each point corresponds to one benchmark program (sorted by runtime), with the x-axis showing the number of solved benchmarks and the y-axis showing the cumulative time in seconds; curves that extend further right and rise more slowly indicate better overall performance.

6.3 Evaluation Methods

We compare four methods in our experiments: *baseline*, *wp-klee*, *inc-mc*, and *qklee*. The *baseline* method uses standard symbolic execution (KLEE) without any pruning optimizations. The collected path constraints are then directly fed to the model counting pipepline, including bit-blasting by CSB and model counting by ApproxMC. The *wp-klee* method extends the baseline by incorporating our *weakest precondition* based pruning algorithm for quantitative symbolic execution, which reduces the number of paths explored during symbolic execution, while still using standard model counting. The *inc-mc* method uses standard KLEE symbolic execution without pruning, but employs our incremental model counting algorithm on top of ApproxMC that reuses SAT models across different path constraints. Finally, *qklee* is our complete method that combines both optimizations: WP-based pruning on KLEE and incremental model counting on ApproxMC.

The experiments are designed to answer the following two research questions:

- **RQ1:** Is *qklee* faster than the baseline method, and how much faster?
- **RQ2:** Is *qklee* sound in terms of the quantitative analysis results?

In addition, we also conduct an ablation study to investigate the contribution of each individual component in the proposed method. The ablation study is presented in Section 7 later in this paper.

Our evaluation experiments were conducted on a computer with a 4.7 GHz AMD R9 7900X CPU and 32 GB of RAM, running the Ubuntu 22.04 LTS operating system. In the remainder of this section, we will present the results for answering these two research questions.

6.4 Results for RQ1

To answer RQ1, we have compared the execution time of *qklee* against with the other three methods. Fig. 5 presents a cactus plot of end-to-end execution time for all four methods, where the red curve represents our proposed method *qklee*, the blue curve represents the baseline method *baseline*, and the green and yellow curves indicate the *wp-klee* and *inc-mc* methods, respectively. In the plot, the x-axis is the number of benchmarks solved, and the y-axis is the cumulative execution time in seconds in log scale. Note that the benchmark programs are sorted by the execution time for each method. Intuitively, the curve that stays lower and rises more slowly indicates that the method spends less total time to solve the same number of benchmarks.

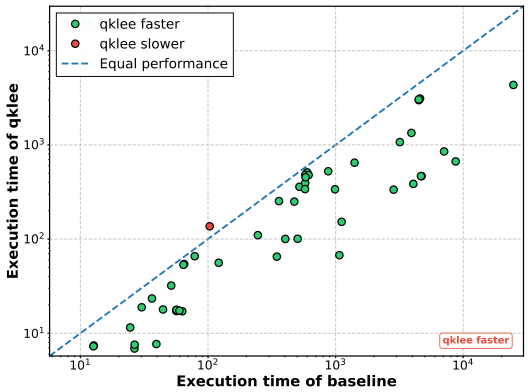


Fig. 6. Scatter plot of execution time between *qklee* and *baseline*. Each point corresponds to one benchmark program, with the x-axis showing the *qklee* execution time and the y-axis showing the *baseline* execution time. Green points indicate cases where *qklee* is faster than *baseline*, while red points indicate cases where *baseline* is faster.

Based on Fig. 5, we conclude that *qklee* consistently outperforms *baseline* across the benchmark suite. Across almost the entire x-axis, the *qklee* curve stays below the other three curves, indicating that it solves the same number of programs with less cumulative time. We also observe that *wp-klee* and *inc-mc* each improve over *baseline* on many programs, as their curves generally lie below the red curve. While the curve of *wp-klee* is relatively close to *qklee*, there are still certain winning cases that *qklee* is faster than *wp-klee*.

To further visualize per-program improvements, Fig. 6 shows a scatter plot comparing *qklee* and *baseline* on each benchmark program in log scale. In the scatter plot, the x-axis is the execution time of *qklee*, and the y-axis is the execution time of *baseline*, both in seconds. Green points indicate the winning cases where *qklee* runs faster than *baseline*, while red points indicate otherwise. All green points fall below the diagonal line, and the farther below, the greater the improvement. Based on the scatter plot, *qklee* is much more efficient on the quantitative analysis compared to *baseline* and outperforms the *baseline* method on most of the benchmark programs with only one exception.

In the evaluation experiments, *qklee* notably outperforms the *baseline* method on many methods. For example, *qklee* is about 13x and 10x faster than *baseline* on the benchmarks *xor5.c* and *mapsum3.c*, for which *baseline* takes more than 8,700 and 4,000 seconds to solve, while *qklee* only uses no more than 700 and 400 seconds to solve, respectively.

In a word, the experimental results show that *qklee* is significantly faster than *baseline* in terms of the end-to-end execution time, and achieves about 3.70x speedup over *baseline* across the benchmark suite.

6.5 Results for Answering RQ2

We compare the quantification results between *baseline* and *qklee* to ensure the soundness of the proposed method.

Fig. 7 presents scatter plots comparing the quantification results between *baseline* and *qklee* on all 52 benchmark programs. The left plot shows the estimated number of inputs that make the assertion pass, while the right plot shows the estimated number of inputs that cause the assertion to fail. In both plots, the x-axis represents the benchmark program name (sorted by program name), and the y-axis represents the quantification results (i.e., the estimated number of inputs that satisfy

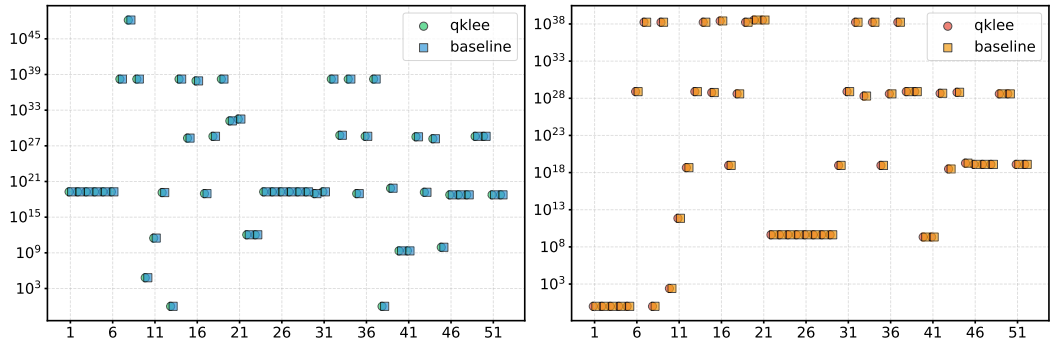


Fig. 7. Scatter plots comparing *baseline* and *qklee* on the estimated number of inputs that make the assertion pass (left) or fail (right). Each point is a benchmark program; the x-axis is the benchmark ID and the y-axis is the number of inputs that pass or fail the assertion. The points of the same pair that are closer to each other indicate better agreement.

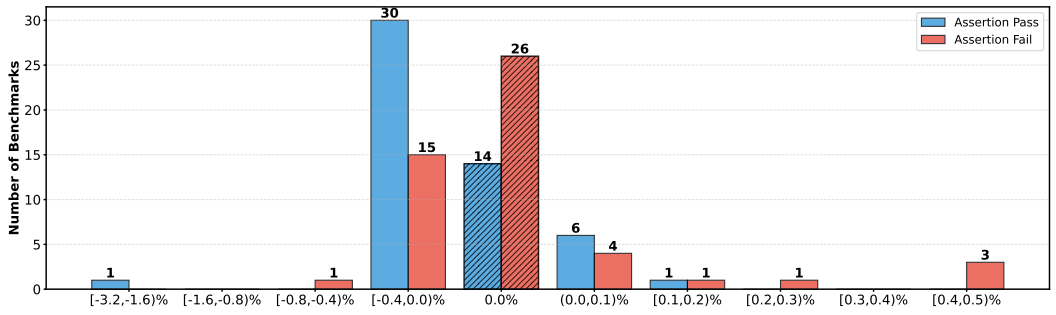


Fig. 8. The deviation rate Δ between analysis results of *baseline* and *qklee*. The blue and red bars represent the number of benchmarks that pass or fail the assertion, respectively, falling into the corresponding deviation rate range. The hatched bars indicate when there is no deviation.

the path constraints). Each pair of points with the same color and benchmark ID corresponds to the results from *baseline* (circle) and *qklee* (square) on the same program. Points that are closer to each other indicate better agreement between the two methods. As shown in the figure, all the points from *baseline* and *qklee* are nearly overlapping or very close to each other, indicating that *qklee* produces quantification results that are highly consistent with *baseline*.

To further investigate how the quantification results differ between *baseline* and *qklee*, Fig. 8 shows the distribution of the deviation rate between the analysis results of *baseline* and *qklee*. The deviation rate is calculated as

$$\Delta = \frac{qklee - baseline}{baseline} \times 100\%.$$

In the distribution plot, the x-axis represents the deviation rate range, and the y-axis shows the number of benchmarks falling into each range. The blue and red bars represent the benchmarks that pass or fail the assertion, respectively. The hatched bars at the center indicate benchmarks with no deviation between *baseline* and *qklee*. The distribution plot reveals that for most benchmarks, the quantification results of *qklee* are very close to the ground truth of *baseline*. Typically, the

deviation is within ±0.5%, and a significant number of programs have no deviation at all. Even for the worst case, the deviation is no more than 3.2%.

We further justify why the small deviation generally exists and is acceptable. The deviation comes from the inherent *approximate* nature of the model counting algorithm used in ApproxMC [6, 33, 34, 41]. Since ApproxMC relies on sampling-based techniques for approximate model counting, even when two sets of path constraints are logically equivalent (as ensured by our WP-based pruning), the sampled models from the two sets of path constraints may be different, thereby leading to the deviation. This inherent approximation is present in both *baseline* and *qklee*, and the deviations observed are well within the expected tolerance of approximate model counting.

In summary, the experimental results demonstrate that *qklee* maintains soundness in terms of quantitative analysis results, with deviations that are generally small and acceptable. The deviations generally come from the inherent approximation of the model counter rather than unsoundness in our optimizations.

7 Ablation Study

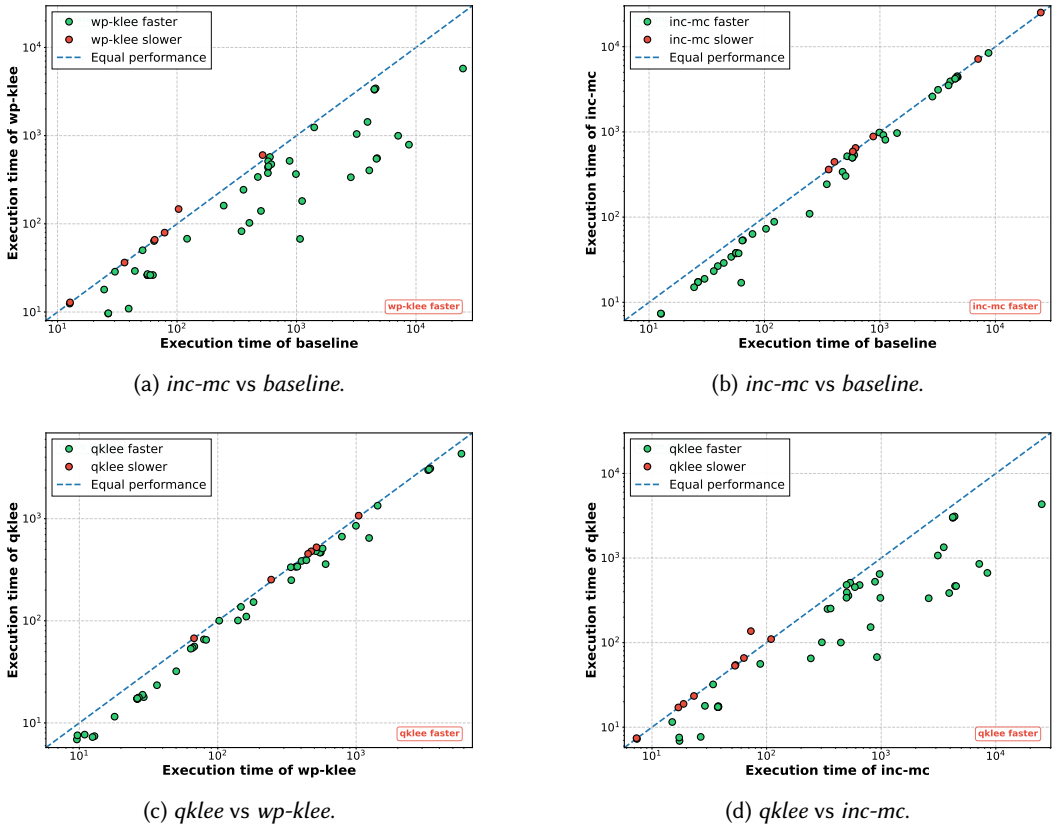
In this section, we conduct an ablation study to investigate the contribution of the proposed weakest precondition-based pruning and incremental model counting to the overall performance of the proposed method.

Evaluation Setup. We compare the execution time between *baseline* and *wp-klee*, and between *baseline* and *inc-mc* to understand the contribution of the individual components. The evalution is conducted by reusing benchmark programs with the same configuration from in Sec. 6. Additionally, we also compare the execution time between *qklee*, *wp-klee*, and *inc-mc* to demonstrate the synergy effect of the two components. The ablation results are presented in Fig. 9.

Baseline vs WP-based Pruning. We first demonstrate the effectiveness of the *weakest precondition* based path pruning. Fig. 9a shows the scatter plot of execution time between *wp-klee* and *baseline*. As shown in the figure, most of the data points are green and fall below the diagonal line (except very few red outliers), indicating that *wp-klee* outperforms *baseline* in most benchmark programs. This demonstrates that the *weakest precondition* based pruning effectively reduces the search space by eliminating the redundant paths, thereby reducing calls to the model counter and gaining speedup. On average, *wp-klee* achieves a 3.04× speedup over *baseline* across all benchmarks. The results confirm that the WP-based path pruning is highly effective for quantitative symbolic execution.

Baseline vs Incremental Model Counting. We then demonstrate the effectiveness of the incremental model counting. Fig. 9b shows the scatter plot of execution time between *inc-mc* and *baseline*. Similar to the WP-based pruning results, most data points are green and lie below the diagonal line, confirming that *inc-mc* consistently outperforms *baseline* across the majority of benchmarks. On average, *inc-mc* achieves a 34.2% improvement over *baseline* across all benchmarks. The results demonstrate that our proposed incremental model counting is an effective optimization for the usage of model counter in quantitative symbolic execution.

QKLEE vs Single Component. We finally demonstrate the synergy effect of the two optimization components. Fig. 9c and Fig. 9d show the scatter plots comparing *qklee* against *wp-klee* and *inc-mc*, respectively. As shown in both figures, *qklee* outperforms either individual component (*wp-klee* or *inc-mc*) alone for most benchmark programs, with the majority of data points colored green and positioned below the diagonal lines. This demonstrates that combining WP-based pruning and incremental model counting yields additional performance improvements beyond what each component achieves independently. On average, *qklee* achieves a 16.9% improvement over the best single component. While this improvement may appear modest compared to the gains from individual components, it is important to note that achieving additional speedup becomes



(a) *inc-mc* vs *baseline*.

(b) *inc-mc* vs *baseline*.

(c) *qklee* vs *wp-klee*.

(d) *qklee* vs *inc-mc*.

Fig. 9. Ablation scatter plots showing: (a-b) effectiveness of individual components against *baseline*, and (c-d) *qklee* against each component. Green points indicate the proposed method is faster, while red points indicate otherwise. The farther below the diagonal line, the greater the improvement.

increasingly challenging when each component has already delivered substantial contributions. In a word, the results validate the effectiveness of the combination of the two components.

8 Related Work

We review the related work in this section, including a vast amount of prior work on quantitative program analysis, the existing techniques for mitigating path explosion in symbolic execution while they were not designed for a quantitative setting, and state-of-the-art model counting techniques.

Quantitative program analysis has been studied extensively, with techniques including but not limited to symbolic execution [3, 4, 18, 23, 40], static analysis [12, 25], and model checking [21]. One important application of quantitative program analysis is the analysis of side channels. For example, static analysis can be used to quantify the upper bound of information leakage through cache side channels [12], and timing side channels can also be measured through static analysis [25]. Our baseline method is directly inspired by *probabilistic symbolic execution* [18, 40], which uses a symbolic execution tool such as Symbolic PathFinder [27] to explore program paths and generate path constraints, and then uses an off-the-shelf model counter [1, 9] to compute the probability of each path. This approach has been extended to handle nondeterminism in multi-threaded software [23] and to analyze various side channels [3, 4]. However, none of these methods give a

try to mitigate the path explosion problem, especially it would get worse as we have discussed in Section 2. The path explosion problem along with the expensive cost of model counting makes this technique less practical for real-world programs.

Path explosion is a well-known problem in symbolic execution, and various pruning techniques have been proposed to mitigate it. These techniques can be broadly categorized into forward-style and backward-style approaches. State merging [20, 30] is a representative forward-style technique that merges equivalent states during execution to reduce the number of explored paths. WP-based pruning [15, 42, 43] is a representative backward-style technique. This technique [42, 43] computes the weakest precondition at each program location at the end of a path exploration, and use it to summarize all previously explored path suffixes. When an incoming path would indicate duplicated path suffixes for its remaining execution, it can be safely pruned without missing unknown program behaviors. The idea of WP-based pruning has also been applied to fault injection analysis [15], where the WP-based pruning is enhanced by a technique called fault saturation. Our work builds on this WP-based pruning and extends it to support quantitative analysis. Beyond these two styles, other pruning techniques have also been proposed. Chopped symbolic execution [39] focuses execution on program regions that are relevant to the target property by skipping irrelevant parts of the code. Statistical pruning [16] uses sampling to estimate path coverage and skip paths with low exploration value. However, all of these techniques were designed for non-quantitative settings, where the goal is to determine *whether* certain inputs can trigger a property violation, not *how many*. In these settings, pruning a path is safe merely depending on the reachability of the path, i.e., whether its execution would lead to known or unknown assertion outcomes. In contrast, our quantitative setting requires computing the number of inputs that result in a specific execution outcome, so discarding a path without accounting for its quantitative contribution would lead to unsound results.

Approximate model counting (ApproxMC) [6, 33, 34, 41] offers a practical solution to the #P-complete problem of model counting by leveraging XOR-based hash functions to uniformly partition the solution space and sample from it. This approach avoids exhaustive enumeration while providing theoretical guarantees on the tolerance and confidence of the counting results. CSB [32] is a model counter for constraints in the Bitvector (BV) theory, which converts BV constraints to CNF formulas via bit-blasting using STP [17] and then feeds the result to ApproxMC for counting. Our tool builds directly on this pipeline and extends the incremental techniques to ApproxMC after the bit-blasting phase. Beyond approximate counting, probabilistic exact model counters [31, 35] have also been developed, providing precise results at the cost of higher computational overhead. For constraints outside the BV theory or CNF, there are also model counting techniques targeting integer arithmetic [1, 9]. However, none of these techniques has such a unique incremental optimization as we proposed in this work for consecutive path constraints generated from symbolic execution.

9 Conclusion

We presented *qklee*, a tool for efficient quantitative program analysis that combines symbolic execution with model counting. Our approach makes two key contributions. First, we extended the *weakest precondition* (WP) based path pruning from qualitative to quantitative symbolic execution. By decomposing the weakest precondition into wp_{pass} and wp_{fail}, our method significantly prunes the redundant paths while preserving the correctness of the quantification results. Second, we proposed an incremental model counting technique that reuses SAT models from consecutive path constraints generated from the symbolic execution to bias the search heuristics of the SAT solver used in the model counting procedure and thus improve the efficiency. We evaluated *qklee* on wildly used benchmark programs and demonstrated that it achieves significant speedups over the baseline while maintaining consistent quantification results. The results show that *qklee* achieves

an average speedup of 3.70× over the baseline with up to 15.88× on individual programs with negligible differences in the quantification results. We also conducted an ablation study to investigate the contributions of each component of *qklee*. The results confirm that both the WP-based pruning and the incremental model counting can effectively improve the overall performance of the quantification procedure, and their combination yields additional gains beyond either component alone.

References

[1] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*. Springer, 255–272.

[2] D. Beyer and J. Strejček. 2025. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[3] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, 27–37. https://doi.org/10.1145/3213846.3213867

[4] Tevfik Bultan. 2018. Side-Channel Analysis via Symbolic Execution and Model Counting. *ACM SIGSOFT Softw. Eng. Notes* 43, 4 (2018), 55. https://doi.org/10.1145/3282517.3302416

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*.

[6] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls.. In *IJCAI*.

[7] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls.. In *IJCAI*, Vol. 16. 3569–3576.

[8] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.

[9] Jesús A De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. 2004. Effective lattice point counting in rational convex polytopes. *Journal of symbolic computation* 38, 4 (2004), 1273–1302.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[11] E. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall, NJ.

[12] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on information and system security (TISSEC)* 18, 1 (2015), 1–32.

[13] Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet. 2023. Adversarial Reachability for Program-level Security Analysis. In *European Symposium on Programming (ESOP)*.

[14] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security*.

[15] Yuzhou Fang, Chenyu Zhou, Jingbo Wang, and Chao Wang. 2026. Efficient Symbolic Execution of Software under Fault Attacks. *arXiv preprint arXiv:2503.15825* (2026).

[16] Antonio Filieri, Corina S. Pasareanu, Willem Visser, and Jaco Geldenhuys. 2014. Statistical symbolic execution with informed sampling. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 437–448. https://doi.org/10.1145/2635868.2635899

[17] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *International conference on computer aided verification*. Springer, 519–531.

[18] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*. 166–176.

[19] Guillaume Girol, Guilhem Lacombe, and Sébastien Bardin. 2024. Quantitative Robustness for Vulnerability Assessment. *Proc. ACM Program. Lang.* 8, PLDI (2024), 741–765. https://doi.org/10.1145/3656407

[20] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proc. ACM PLDI*.

[21] Marta Kwiatkowska, Gethin Norman, and David Parker. 2017. Probabilistic model checking: Advances and applications. *Formal System Verification: State-of the-Art and Future Trends* (2017), 73–121.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE International Symposium on Code Generation and Optimization*.

[23] Kasper Søe Luckow, Corina S. Pasareanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. 2014. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. ACM, 575–586. https://doi.org/10.1145/2642937.2643011

[24] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-driven clause learning SAT solvers. *Handbook of satisfiability* (2009), 131–153.

[25] Denis Mazzucato, Marco Campion, and Caterina Urban. 2024. Quantitative Static Timing Analysis. In *International Static Analysis Symposium (SAS)*.

[26] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*. 530–535.

[27] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 179–180.

[28] Quoc-Sang Phan and Pasquale Malacaria. 2014. Abstract model counting: a novel approach for quantification of information leaks. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 283–292.

[29] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. 2024. Inference of Robust Reachability Constraints. In *Proc. ACM POPL*.

[30] Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 842–853. https://doi.org/10.1145/2786805.2786830

[31] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter.. In *IJCAI*, Vol. 19. 1169–1176.

[32] Arijit Shaw and Kuldeep S Meel. 2024. CSB: A Counting and Sampling tool for Bit-vectors. (2024).

[33] Mate Soos, Stephan Gocht, and Kuldeep S Meel. 2020. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *CAV*.

[34] Mate Soos and Kuldeep S Meel. 2019. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *AAAI*.

[35] Mate Soos and Kuldeep S. Meel. 2025. Engineering an Efficient Probabilistic Exact Model Counter. In *CAV*. 72–91.

[36] Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 244–257.

[37] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. 2022. Exploration of Fault Effects on Formal RISC-V Microarchitecture Models. In *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*.

[38] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. 2023. μArchiFI: Formal Modeling and Verification Strategies for Microarchitetural Fault Injections. In *Formal Methods in Computer-Aided Design (FMCAD)*.

[39] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proc. ACM ICSE*. 350–360.

[40] Willem Visser and Corina S. Pasareanu. 2017. Probabilistic programming for Java using symbolic execution and model counting. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017*. ACM, 35:1–35:10. https://doi.org/10.1145/3129416.3129433

[41] Jiong Yang and Kuldeep S Meel. 2023. Rounding meets approximate model counting. *Formal Methods in System Design* (2023).

[42] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned Symbolic Execution. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[43] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2017. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Transactions on Software Engineering (TSE)* (2017).

A Notes and Drafts

The following categories of programs from SV-COMP appear to be the potential benchmarks.

- `xscp`, with 119 variants. These programs do not have loops but contain many branches.
- `array-cav19`, with 13 variants. Some of the programs contain unbounded loops, but we can fix them down with slightly modifications.
- `array-crafted`, with 43 variants. Similar to `array-cav19`.
- `other integer-based array directories`.
- `validation-crafted`, with 6 variants.
- `unsignedintegeroverflow-sas23`, some simple programs without branches but have integer overflow.
- `loops`, some examples with bounded loops.
- `Termination Crafted` and `Termination Crafted Lit`, from paper [25].
- `fuzzle-programs`, may appear to be a good candidate.
- `array-programs`, mainly contain bounded loops.
- In the introduction of SV-COMP, the meta directory 'Reach Safety' should be the most suitable one. See more details in https://sv-comp.sosy-lab.org/2025/benchmarks.php.

A.1 Quantitative WP-based Pruning.

An execution path in symbolic execution can be regarded as a concatenation of a pair of path prefix and suffix. For path prefix, it can be denoted by the strongest postcondition (sp) during the symbolic execution and the suffix is the weakest precondition (wp). Whether a path at a program location can be pruned or not, it is determined by the *validity* of

$$sp \rightarrow wp,$$

which is equivalent to

$$sp \wedge \neg wp$$

is *UNSAT*. In other words, if

$$sp \wedge \neg wp$$

is *SAT*, the validity of

$$sp \rightarrow wp,$$

does not hold, and the path should be not be pruned. For *klee*, sp actually refers to the reachable state. If the current instruction was executed, sp has already been proved to be *true*. To determine the satisfiability of

$$sp \wedge \neg wp,$$

we only need to check $\neg wp$.

However, for the quantitative analysis task, we treat wp as wp_{good} and wp_{bad} which are disjointed together, saying

$$wp = wp_{good} \vee wp_{bad}.$$

When a path is determined to be pruned, the number of violating an assertion N_{assert} and the number of not violating N_{normal} are

$$N_{normal} = count(wp_{good} \wedge sp)$$
$$N_{assert} = count(wp_{bad} \wedge sp).$$

A.2 No Common Models Shared between Path Conditions.

Let me use a simple example to illustrate the problem. During symbolic execution at a branching program location, SE will generate two different path constraints, for which they share a common path prefix, denoted as π_{pref}, and two orthogonal path suffixes, denoted as π_{suff} and $\neg\pi_{suff}$. In this case, the path constraints for these two paths can be expressed as:

$$pcond_1 = \pi_{pref} \wedge \pi_{suff}$$
$$pcond_2 = \pi_{pref} \wedge \neg\pi_{suff}.$$

We expect that the two path constraints may share some common models, denoted as SAT_{common}, which can satisfy both path. The relation between the two path constraints can be expressed as:

$$pcond_1 \wedge pcond_2 = \pi_{pref} \wedge \pi_{suff} \wedge \neg\pi_{suff}.$$

Apparently, $pcond_1 \wedge pcond_2$ is unsatisfiable, which indicates that there is no common model between the two path constraints. However, our assumption for accelerating the model counting is that we can reuse the known models to quickly find the SAT assignments, there are no common models between any of two path constraints generated by the symbolic execution.

A.3 Implementation details of CSB.

CSB is only a front end that integrates several solver techniques and uses them as a model counter for the BV Theory.

- *Solver STP* conducts Bit-blasting that converts the constraints in BV Theory to CNF.
- *ApproxMC* takes the input CNF constraints and conducts approximate model counting to obtain a number of SAT models.
- *Solver CryptoMiniSat*, the backend of ApproxMC. ApproxMC relies on CryptoMiniSat (cms) to resolve the satisfiability of a given CNF.

Note that in cms, it has implemented incremental approaches since its application scenarios heavily rely on them to accelerate the counting process. The reason why incremental SAT solving has been implemented in cms is explained as follows:

- Given a CNF ϕ, let's assume cms finds a SAT model denoted as M.
- To find the next SAT model, cms bans the previous SAT model and obtains $\phi' = \phi \wedge \neg M$.
- Note that ϕ' adds several extra clauses to ϕ, which is naturally suitable for *incremental SAT solving*.
- The solver cms would keep the learned clauses in solving ϕ and reuse them when resolving ϕ' to accelerate.

A.4 The Counting Approach.

We first clarify what we will *count* during the symbolic execution for the quantitative analysis. During the symbolic execution for each SymbolicState, qKLEE will collect the path conditions and know if there is an assertion violation happens at the end of the program execution.

- If there is, we will apply the model counter to the path conditions to know the number of the SAT models. We will also add the number to N_{bad}, indicating how many kinds of input can trigger the assertion failure.
- If there isn't, we do the same model counting for the path conditions and add the number to N_{good}, indicating the assertion passes. Based on such a process, we are continuing to use the model counter to the path conditions collected during the symbolic execution.

The above process describes how the baseline approach works for the quantitative task. In a word, the baseline approach will continue to apply model counters to the path conditions collected and add the number of SAT models corresponding to each path condition to N_{good} or N_{bad}.

A.5 The Incremental Approach.

We first need to understand if it is feasible to directly reuse the SAT model of the previous path constraints. My conclusion is, it is **infeasible** to directly reuse the SAT models from one path condition pc to another one pc'.

Let's denote path conditions pc and pc' are the ones forked at the same branch location, which means that they share the same path prefix and could be denoted as:

$$pc = prefix \wedge cond$$
$$pc' = prefix \wedge \neg\, cond.$$

While pc and pc' share the same $prefix$, they are still mutually exclusive. By such an observation, we notice that the path conditions from different paths are always mutually exclusive to each other, while they could share common prefixes. As a result, it is infeasible to directly use full SAT models from one path condition to another.

The incremental idea is record the SAT models for each literal in the path conditions, and then use them to initialize the polarity of each literal when they are assigned values during the CDCL search.

A.6 Raw Results for Experiments.

A.6.1 Incremental Model Counting. We now introduce the incremental model counting, without building analysis from scratch. Two incremental implementations in CSB:

- Hack to the phase injection: change the polarity of the variables that appear in the SAT models.
- During the sampling, even on the same projected variable set, the hash functions used may be different.
- Sometimes the hash functions will directly ban a SAT model, which will have no effect on improving the efficiency.
- Need to ensure the SAT models under the same hash functions will be passed to accelerate.

When doing incremental model counting, we need to ensure that,

- The previous SAT models should also be satisfiable to hash function h^m, otherwise the solver will learn conflict clauses on the hash function instead of the formula ϕ.
- When applying the previous SAT model to $\phi' \wedge h^m$, I want the source of SAT models, i.e., $\phi \wedge h^m$, to be closer to $\phi' \wedge h^m$, where there would be more opportunities for reducing the number of conflicts.
- Therefore, we thus have to select SAT models using the same hash functions as $\phi \wedge h^m$.

Current Design of CSB++

The current design of CSB++ is as follows.

For the input CNF ϕ, its SAT models M_{SAT} will be stored and indexed by the hash functions h^m. During the sampling process, appmc will count the number of SAT models over $\phi \wedge h^m$. For different rounds of sampling, the hash functions h^m differ a lot from each other. For example, initially the hash function may only contain one constraint (each constraint will approximately have $\frac{|S|}{2}$ XOR expressions, where $|S|$ is the number of literals in the projected variable set). With the sampling going forward, the hash functions may have more than one XOR constraint; typically, it can be 20-40, or other larger numbers.

Benchmark Name	Baseline Method			Incremental Model Counting			WP-based KLEE			QKLEE			Speed Up
	SE Time	MC Time	Total Time	SE Time	MC Time	Total Time	SE Time	MC Time	Total Time	SE Time	MC Time	Total Time	
example.c	0.1585	23.36	24.45	0.1605	14.635	15.625	0.137	17.47	18.013	0.066	11.01	11.47	2.11×
crc8.c	39.4764	45.6819	403.4347	42.1951	49.8372	444.7388	5.8038	8.094	102.5076	5.696	7.4815	100.3184	4.02×
bAnd1.c	0.1127	25.855	26.5274	0.1049	16.7355	17.382	0.1099	9.1061	9.5985	0.1019	6.4029	6.8841	3.85×
bAnd2p.c	1.1803	43.3112	51.5083	1.212	26.0285	34.1524	3.1209	42.6216	50.1347	2.8801	24.7417	32.0629	1.61×
bAnd3.c	0.4461	41.5316	44.3957	0.4445	26.2557	28.979	0.3098	27.9407	29.2104	0.2968	16.6757	17.8793	2.48×
bAnd4.c	0.5065	36.1581	39.3915	0.5301	23.3345	26.6142	0.4671	9.2402	10.9268	0.4391	6.2397	7.6792	5.13×
bor1.c	0.1317	25.6891	26.5922	0.204	16.2857	17.2702	0.1448	8.7797	9.6852	0.2355	6.3579	7.5591	3.52×
bor2.c	0.9608	69.7834	78.7094	0.9572	54.5056	63.4114	3.2821	69.6234	79.4341	3.2427	56.1056	65.7607	1.20×
bor3.c	3.6802	4690.657	4721.653	3.5956	4378.678	4409.316	9.9639	537.1786	555.4829	10.2885	447.65	466.5055	10.12×
bor4.c	14.307	3080.479	3187.111	14.1886	3013.696	3119.99	310.5223	683.245	1043.196	301.2507	720.6242	1071.129	2.98×
bor5.c	3.4863	4637.559	4668.466	3.5625	4474.419	4505.685	10.043	527.598	545.93	10.2407	447.6256	466.1728	10.01×
mapavg1.c	5.0199	514.2955	520.482	5.5981	511.0689	517.5598	2.4257	598.9221	601.9933	2.5461	355.8147	359.0168	1.45×
mapavg2.c	739.2237	238.7526	988.8867	739.2378	233.5645	983.3727	115.4914	239.0131	366.4564	109.1632	218.5967	337.7633	2.93×
mapavg3.c	1082.683	1750.222	2848.982	931.962	1651.792	2597.789	21.6379	312.5968	337.3275	24.3318	307.2001	334.7567	8.51×
mapavg4.c	156.8446	1238.218	1406.689	164.472	793.3311	969.051	94.9216	1139.082	1240.639	98.6155	541.1695	647.348	2.17×
mapavg5.c	415.9615	24280.48	24730.58	397.3072	24748.43	25180.16	108.1381	5647.434	5765.971	129.7158	4185.068	4330.56	5.71×
mapsum1.c	0.1928	61.7762	62.8657	0.1022	16.3662	16.9865	0.1278	25.5757	26.2509	0.1355	16.44	17.1229	3.67×
mapsum2.c	0.3366	118.9941	121.3601	0.3813	85.6029	88.0902	0.4728	66.1632	67.724	0.4964	54.4127	56.0228	2.17×
mapsum3.c	1.6904	4048.575	4065.523	1.6015	3905.252	3921.503	3.278	397.8286	403.6789	3.198	379.5866	385.5987	10.54×
mapsum4.c	7.7892	809.9153	875.2053	7.6513	818.6517	883.6589	59.437	431.9784	517.2808	58.2708	440.4674	524.616	1.67×
mapsum5.c	3.3438	7048.356	7087.128	3.2715	7136.753	7174.882	19.5848	966.5651	996.5689	19.7967	821.5177	852.0671	8.32×
xor1.c	0.2056	29.2387	30.2374	0.1903	17.882	18.8572	0.1663	27.7345	28.6479	0.1678	17.9401	18.8707	1.60×
xor2.c	1.7801	236.0492	245.8363	1.2773	101.3987	109.4796	3.9079	151.3793	160.859	3.5817	101.1301	110.0593	2.23×
xor3.c	3.8005	1046.841	1070.817	3.7736	893.3345	918.186	2.6058	61.3332	67.4134	2.5587	61.3856	67.4422	15.88×
xor4.c	3.6997	459.7164	474.6146	3.348	325.5	340.1673	14.8334	316.8239	339.8087	14.8571	226.8425	249.9531	1.90×
xor5.c	13.7673	8656.12	8728.194	15.6503	8368.38	8445.836	39.5076	727.5304	789.268	39.3593	605.9564	667.4585	13.08×
zero_sum_1.c	0.1369	35.4496	36.3563	0.1362	22.3565	23.2457	0.2188	35.3278	36.382	0.2202	22.3638	23.4008	1.55×
zero_sum_2.c	1.659	4482.792	4494.961	1.5822	4210.9	4223.047	10.1032	3296.175	3313.327	10.1766	2968.994	2986.254	1.51×
zero_sum_3.c	0.2743	1095.835	1113.993	0.3343	790.394	808.5731	2.4125	176.5151	181.3931	2.7503	147.6514	152.4513	7.31×
zero_sum_4.c	0.4633	314.9639	346.4714	0.4163	210.7427	243.1592	3.9236	75.6171	82.4932	3.9391	58.3894	65.1185	5.32×
zero_sum_5.c	0.282	480.0998	504.0671	0.2763	279.4956	303.7849	2.0921	135.209	139.9964	2.0725	96.092	100.8104	5.00×
zero_sum_const_1.c	0.1318	64.515	65.0517	0.1243	53.015	53.5544	0.1641	65.2569	65.8389	0.1716	53.7445	54.3524	1.20×
zero_sum_const_2.c	0.1241	63.6661	64.2002	0.1281	52.4796	53.0382	0.3653	63.037	63.8388	0.3746	52.5121	53.3113	1.20×
zero_sum_const_3.c	0.4693	336.3711	359.8202	0.4116	339.1743	361.8336	70.7341	158.4823	243.2601	73.7117	164.8948	253.1024	1.42×
zero_sum_const_4.c	0.3279	592.7208	599.0761	0.3193	530.6753	536.8183	22.5891	544.2895	572.2204	22.4532	483.0895	510.867	1.17×
zero_sum_const_5.c	0.6468	3898.178	3928.539	0.6086	3484.689	3515.536	82.0682	1333.657	1430.124	82.1746	1243.73	1340.805	2.93×
zero_sum_const_m2.c	0.6093	99.9773	103.068	0.4631	70.2984	72.9143	1.29	143.7322	147.1012	1.2509	133.3938	136.727	0.75×
zero_sum_const_m3.c	0.4227	572.5973	579.0144	0.371	492.4352	498.7398	70.4177	436.5805	511.9845	87.9572	388.3112	481.2659	1.20×
zero_sum_const_m4.c	0.3023	571.0793	577.2179	0.3585	494.607	501.0232	66.5476	365.7208	436.7858	63.9436	324.0959	392.1666	1.47×
zero_sum_const_m5.c	0.3234	569.9734	575.9276	0.3677	491.9248	498.1467	61.3783	311.0401	376.1911	58.5152	277.2569	339.2675	1.70×
zero_sum_m2.c	1.7801	4578.7	4591.573	1.7502	4357.389	4370.187	29.1773	3396.018	3434.122	29.9246	3066.185	3105.254	1.48×
zero_sum_m3.c	1.999	4465.988	4479.031	1.8456	4201.085	4213.943	61.5305	3286.768	3362.189	60.2423	2967.601	3041.8	1.47×
zero_sum_m4.c	0.4492	608.0846	614.8709	0.4649	641.525	648.3565	74.1365	395.097	473.7857	89.7447	384.7728	479.0791	1.28×
zero_sum_m5.c	0.4298	575.8092	582.0399	0.4933	582.4846	589.0216	72.7789	372.9137	449.546	96.471	352.7918	453.063	1.28×
verifyPIN_0.c	0.0472	12.3236	12.7246	0.0445	6.9437	7.3297	0.0506	12.4607	12.8615	0.0493	7.0236	7.4176	1.72×
verifyPIN_1.c	0.0484	12.2421	12.6526	0.0441	7.0018	7.3937	0.0474	12.0677	12.453	0.0459	6.8835	7.2668	1.74×
verifyPIN_2.c	0.0475	56.0299	57.1204	0.0463	36.918	37.9953	0.0647	25.8321	26.4592	0.0662	16.7658	17.4208	3.28×
verifyPIN_3.c	0.0452	55.4819	56.5487	0.049	36.6803	37.7895	0.0649	25.615	26.2427	0.0625	16.6861	17.2782	3.27×
verifyPIN_4.c	0.05	55.5353	56.6216	0.0447	36.3677	37.4559	0.0677	25.5376	26.1521	0.0692	16.6343	17.2665	3.28×
verifyPIN_5.c	0.0606	55.4314	56.5476	0.0593	36.7848	37.8818	0.0857	25.7083	26.3452	0.0887	16.7729	17.402	3.25×
verifyPIN_6.c	0.0464	55.5049	56.6194	0.0478	36.8021	37.8633	0.0722	26.2813	26.9276	0.0634	17.1364	17.7504	3.19×
verifyPIN_7.c	0.0451	58.639	59.7084	0.0449	36.5444	37.6041	0.0942	25.572	26.2242	0.0946	16.7563	17.3991	3.43×

Recall our assumptions, we assume that the difference between ϕ and ϕ' is minor. But consider the hash functions added to the sampled CNF,

$$\Phi = \phi \wedge h^m$$

$$\Phi' = \phi' \wedge h^{m'}$$

h^m may only contain one XOR constraint, but $h^{m'}$ may contain tens of XOR; the difference would become huge. Therefore, when applying previous SAT models to ϕ', we first ensure that the current added hash functions h^m are the same as $h^{m'}$; otherwise, there will be no SAT models used. Note that the hash functions are only about the set of projected variables.

For the incremental part, we now do not make changes to the variable selection part, i.e., VSIDS, since the results show that enforcing the order of selected literals would only impose overhead without gains.

Instead, we focus on the *phase injection*. In other words, there are multiple strategies in *cryptominisat* to determine the **polarity** of a selected literal at the decision phase, i.e., determining if a literal should be assigned with True or False. To this end, we first force the solver to use the *saved_polarity* strategy only, which will refer to a polarity map when assigning the truth values. We

then set the polarity map of every literal in the projected variable set based on one of the previous SAT models.

Specifically, when CSB performs sampling on a specific $\Phi' = \phi' \wedge h^{m'}$,

- We first find a SAT model M from $\Phi' = \phi' \wedge h^{m'}$, where $h^{m'} = h^m$.
- If M is found, We then map the truth value of M to the polarity map.
- Otherwise, we do nothing.

The results of performance gains are impressive. Without optimizations, model counting for Φ' takes about 10 seconds. With optimizations, model counting for Φ' takes about 5.6 seconds.

A.7 Baseline (state of the art)

We first run KLEE to generate the Z3 formula for each symbolically executed path. Then, we translate the Z3 formula to a Boolean formula; Finally, we feed the Boolean formula to the model counting solver (CSB or an exact model counter).

- TO-DO Task#1: create a motivating example program (to be used in Section 2) The motivating example now runs very well on the baseline method and incremental Klee!
- TO-DO Task#2: create the set of benchmark programs (to be used in experimental evaluation)
  - **Option#1**: Use benchmark programs in papers on *quantitative program analysis*
  - https://dl.acm.org/doi/pdf/10.1145/2590296.2590328 (code snippets in this paper)
  - **Option#2**: Convert `assert()` in SV-COMP programs to quantification problems The category *array-crafted* is not suitable, since there will only be one path in the benchmark programs! *bitvecotr* also seems not to be a good candidate. The path space is relatively small, and there are not a lot of pruning opportunities.
  - how about the programs used for our CAV/FMCAD submission? We can try on them, but most of the benchmarks intend to have more loops than branches, it is hard to decide how effective our pruning will be.

Contents