

Empc: Effective Path Prioritization for Symbolic Execution with Path Cover

Shuangjie Yao, Dongdong She
 Hong Kong University of Science and Technology
 {syaoap, dongdong}@cse.ust.hk

Abstract—Symbolic execution is a powerful program analysis technique that can formally reason the correctness of program behaviors and detect software bugs. It can systematically explore the execution paths of the tested program. But it suffers from an inherent limitation: path explosion. Path explosion occurs when symbolic execution encounters an overwhelming number (exponential to the program size) of paths that need to be symbolically reasoned. It severely impacts the scalability and performance of symbolic execution.

To tackle this problem, previous works leverage various heuristics to prioritize paths for symbolic execution. They rank the exponential number of paths using static rules or heuristics and explore the paths with the highest rank. However, in practice, these works often fail to generalize to diverse programs.

In this work, we propose a novel and effective path prioritization technique with path cover, named *Empc*. Our key insight is that not all paths need to be symbolically reasoned. Unlike traditional path prioritization, our approach leverages a small subset of paths as a minimum path cover (MPC) that can cover all code regions of the tested programs. To encourage diversity in path prioritization, we compute multiple MPCs. We then guide the search for symbolic execution on the small number of paths inside multiple MPCs rather than the exponential number of paths.

We implement our technique *Empc* based on KLEE. We conduct a comprehensive evaluation of *Empc* to investigate its performance in code coverage, bug findings, and runtime overhead. The evaluation shows that *Empc* can cover 19.6% more basic blocks than KLEE’s best search strategy and 24.4% more lines compared to the state-of-the-art work *cgs*. *Empc* also finds 24 more security violations than KLEE’s best search strategy. Meanwhile, *Empc* can significantly reduce the memory usage of KLEE by up to 93.5% and reduce the number of symbolic states by up to 88.6%.

1. Introduction

Symbolic execution [9,15,22,40,46,47] is a powerful program analysis technique that has been widely used in software testing [34,49,62,63,69,70,80], formal verification [26,27,38,72], and automated reasoning [29,42,50,67]. Unlike conventional software testing approaches that execute the program with concrete input values, symbolic execution treats inputs as symbolic variables, allowing it to explore multiple execution paths simultaneously [9,22]. This method systematically generates inputs that drive the program to

various states, helping to uncover hidden bugs and security vulnerabilities by checking against a set of correctness specifications [9,53,66]. By using symbolic values rather than actual data, it provides a comprehensive analysis of the program’s behavior, enabling testers and developers to verify the correctness and robustness of complex software systems. As such, symbolic execution plays a crucial role in improving software reliability and security, making it an essential topic of study not only in academic research but also in industrial practice such as Microsoft [35], IBM [8], NASA [59,61], Baidu [54] and so on.

Bottleneck. One of the main bottlenecks of symbolic execution is the path explosion [7,9,22]. This is caused by the characteristic that a symbolic execution engine forks off the state at every branch of the program. Hence, each conditional statement in the program can potentially double the number of paths, leading to an exponential number of paths to explore in symbolic execution. The path explosion not only incurs prohibitive computation costs on CPU and memory but also severely limits the scalability of symbolic execution. In practice, symbolic execution often fails to reason large real-world programs.

Existing work. Modern symbolic execution engines prioritize promising paths using various search strategies to tackle the path explosion problem [9]. KLEE [20], one of the most popular symbolic execution engines, incorporates multiple search strategies including bfs (breadth-first search), rps (random-path search) and nurs (non-uniform random search). md2u [18] leverages the control-flow graph to steer the search towards the closest uncovered branches. sgs (subpath-guided search) [55] prioritizes the least explored subpaths. Kapus et al. propose an approach aimed at exploring pending paths already known to be feasible [45]. Ferry [81] uses the dependence of the focused variable to guide the search. A recent work *cgs* (concrete-constraint guided search) [71] favors uncovered branches with concrete variable assignments. Although these static heuristics improve the search efficiency of symbolic execution, they cannot be generalized to diverse programs.

***Empc*.** In this work, we introduce a novel approach called *Empc* to mitigate the path explosion problem using path cover. Unlike prior works that rank the exponential number of paths following some static heuristics, we leverage a small subset of paths as a minimum path cover (MPC) such that it can cover all the code regions of the program using the minimum number of paths. We further compute multiple

MPCs to ensure diverse choices in path prioritization. We then search over the small subset of paths rather than the entire exponential number of paths. The key insight of our work is that not all paths need to be symbolically solved if we want to cover all the code regions of the program. We model the path prioritization in symbolic execution as an MPC problem [10] in the graph theory domain and guide the search of symbolic execution over path cover.

However, there are two practical challenges when applying our path-cover-based search in symbolic execution. Firstly, the inter-procedural control-flow graphs (iCFG) of programs are complex graphs with many cycles such as caller-callee cycles and loop cycles, while the existing MPC algorithm can only work on directed acyclic graphs (DAG). Hence, some approximation algorithms to eliminate different types of cycles in iCFG is needed. Second, some paths in our small set of path cover can be infeasible during symbolic execution. It happens when some path constraints are proved to be infeasible by the SMT solver in the symbolic execution engine.

We propose *Empc* to solve these challenges. Our approach includes two main modules: path prioritization via multiple MPCs and infeasible path handling. We first propose some approximation algorithms for the original MPC algorithm, performing graph transformation on the vanilla iCFG to obtain many acyclic iCFG subgraphs. The goal of such an iCFG graph transformation is to enable trackable MPC computation. Otherwise, it will be an NP-hard problem to compute MPCs on a graph with cycles. We then compute multiple MPCs on the transformed iCFG to increase the path diversity. And the small number of paths in multiple MPCs can cover all nodes on the iCFG. At run-time of symbolic execution, *Empc* compares and finds a solvable path in multiple MPCs to execute. When *Empc* encounters an infeasible path that the SMT solver cannot solve, the infeasible path handling module will be invoked and it will discover a new path using the program dependence information.

Evaluation. We implement *Empc* as a searcher module on top of KLEE. To investigate the performance of *Empc*, we conduct a comprehensive experiment on 12 real-world programs. Our evaluation shows that *Empc* increases the basic block coverage by 19.6% compared to KLEE’s best search strategy and the line coverage by 24.4% compared to the state-of-the-art work cgs [18] in arithmetic mean on these real-world programs. Meanwhile, *Empc* significantly reduces KLEE’s memory cost by up to 93.5% and further cuts the number of execution states in the symbolic execution engine by up to 88.6%. Our evaluation shows that the runtime overhead of *Empc* is minimal, with an average of 12% on 12 programs. In the end, we show that *Empc* finds 24 more security violations than KLEE’s best search strategy.

Contributions. Our main contributions are as follows.

- We model path prioritization in symbolic execution as a classic path cover problem in graph theory domain.
- We propose a novel search strategy for symbolic exe-

cution by searching over the small path cover instead of the exponential number of all possible paths.

- We implement our technique as a prototype *Empc* on top of KLEE and open source our tool at Github (<https://github.com/joshuay2022/empc>) to foster further research in this domain.
- We perform a comprehensive evaluation of *Empc* to investigate its performance in code coverage and bug finding. Our result shows that *Empc* can achieve 19.6% more basic block coverage and 24.4% more line coverage over the state-of-the-art search strategies.

2. Path Prioritization as Path Cover Problem

In this section, we first introduce some background knowledge for the MPC problem, then we formulate the path prioritization in symbolic execution as a path cover problem.

2.1. Minimum Path Cover

Given a graph $G = (E, V)$, a path cover for all vertices V is defined as a set of paths $P_C = \{p_1, p_2, \dots, p_k\}$ such that for each $v_i \in V$, there is at least one path p_j covering node v_i [10,60]. For a given graph, there could exist multiple path covers. An MPC P_m is a path cover whose size is minimum among all possible path covers [10,60]. Computing MPC on general graphs is an NP-hard problem. However, researchers propose multiple algorithms to solve it in polynomial time on directed acyclic graphs (DAG) [31,32,60]. In this work, we use Ntafos’s maximum matching method [60] to compute the MPC. We show this algorithm in Algorithm 1.

Algorithm 1 Compute One Minimum Path Cover

Input: A directed acyclic graph $G(V, E)$
--

```

1:  $min\_path\_cover \leftarrow \{\}$ 
2:  $n \leftarrow |V|, m \leftarrow |E|$ 
3:  $V_b \leftarrow \{x_1, x_2, \dots, x_n\} \cup \{y_1, y_2, \dots, y_n\}$ 
4:  $E \leftarrow \{(x_i, y_i) | 1 \leq i \leq n, v_i \text{ reaches } v_j \text{ in } G\}$ 
5: A bipartite graph  $G_b \leftarrow (V_b, E_b)$ 
6: A maximum matching  $M_m \leftarrow \text{Hopcroft\_Karp\_Algorithm}(G_b)$ 
    $M_m \leftarrow \{(x_{i1}, y_{j1}), (x_{i2}, y_{j2}), \dots, (x_{ic}, y_{jc})\}$ 
7: for  $(x_{ik}, y_{jk}) \in M_m$  do
8:   if  $(v_{ik}, v_{jk}) \notin E$  then
9:      $p_{sub} \leftarrow v_{ik}(v_{ik}, v'_{ik}) \dots v_{jk}$  ▷ construct a subpath
10:   else
11:      $p_{sub} \leftarrow v_{ik}(v_{ik}, v_{jk})v_{jk}$  ▷ construct a subpath directly
12:   for TRUE do
13:     find a path  $p_i$  from  $min\_path\_cover$  with a same end-vertex
   as  $p_{sub}$ 
14:     if  $p_i$  doesn't exist then
15:        $min\_path\_cover \leftarrow min\_path\_cover \cup \{p_{sub}\}$ 
16:       break
17:     else
18:       merge  $p_i$  and  $p_{sub}$  into  $p'_i$ 
19:        $min\_path\_cover \leftarrow min\_path\_cover \setminus \{p_i\}$ 
20:        $min\_path\_cover \leftarrow min\_path\_cover \cup \{p'_i\}$ 
21:        $p_{sub} \leftarrow p'_i$ 
22: return  $min\_path\_cover$ 

```

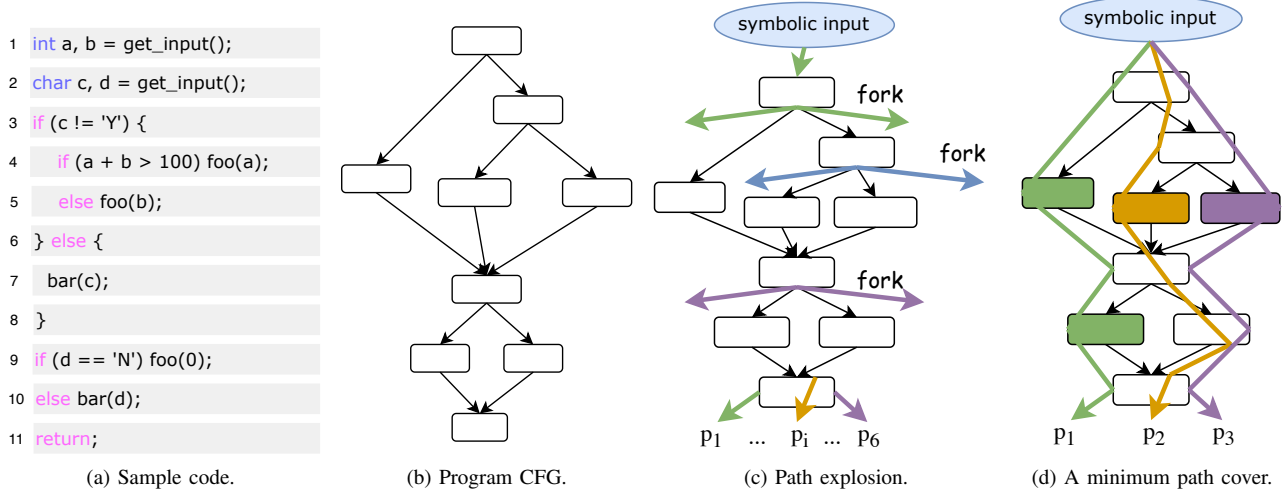


Figure 1: This figure shows a motivating example including sample code, its CFG, path explosion problem and our minimum path cover method. Our method only uses 3 paths to cover all basic blocks of this program with 3 branches while the number of paths $|P| = 6$ if there is a path explosion problem.

2.2. Problem Formulation

We formulate path prioritization in symbolic execution as a classic path cover problem. We denote the iCFG of the tested program as $G = (E, V)$. Since Algorithm 1 is only applied to DAGs, we propose some approximation algorithms including graph transformation in Section 4.1.1 to compute MPCs. There are a total exponential number of paths to be symbolically reasoned on the graph G , denoted as $P_G = \{p_1, p_2, \dots, p_k\}$. We then compute an MPC $P_m = \{p_i, p_j, \dots, p_n\}$ such that all vertices in V can be covered by the n path in P_m and the size of P_m is much lower than P_G , that is, $|P_m| \ll |P_G|$. We then guide symbolic execution to search on the path cover P_m .

3. Overview

3.1. Motivating Example

In this section, we give a motivating example for our work. Figure 1 is a sample program and its CFG is shown in Figure 1b. This sample program shows a common case in real-world programs. Normally, the common symbolic execution engine forks off a state at each branch, so there will be 6 paths in this program, as shown in Figure 1c. This is because the first two forks only yield 3 subpaths and all 3 subpaths then merge at the last fork, and finally yield a total of $3 \times 2 = 6$ paths. If we are aimed at maximizing code coverage, it is obvious that at least 5 complete paths are needed to cover all basic blocks in this program in the worst case. However, to cover all basic blocks in CFG, our MPC method uses only 3 complete paths p_1, p_2 and p_3 as shown in Figure 1d. Because we have a predefined path cover, we only need to select the forked state that matches the path in the MPC at each branch. Ultimately, we use a

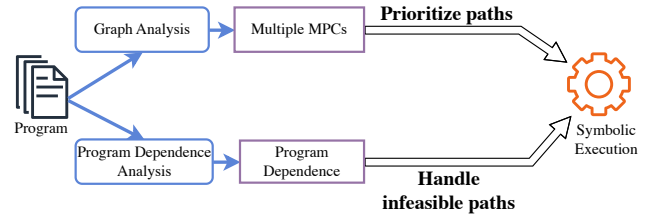


Figure 2: Workflow of *Empec*.

small subset of paths to cover all basic blocks by ignoring most states.

3.2. Workflow

Figure 2 shows the complete workflow of our symbolic execution framework. *Empec* first performs a graph analysis on the program to generate multiple MPCs. Then, it uses multiple MPCs to prioritize a subset of paths to guide symbolic execution at run-time. Moreover, *Empec* leverages a simple program dependence analysis to capture the program dependence between a branch and its dependent basic blocks. When symbolic execution encounters an infeasible path after searching multiple MPCs, *Empec* will find a new state to continue execution using the dependence information.

4. Methodology

In this section, we introduce *Empec* in detail. *Empec* consists of two components: path prioritization via multiple MPCs and infeasible path handling.

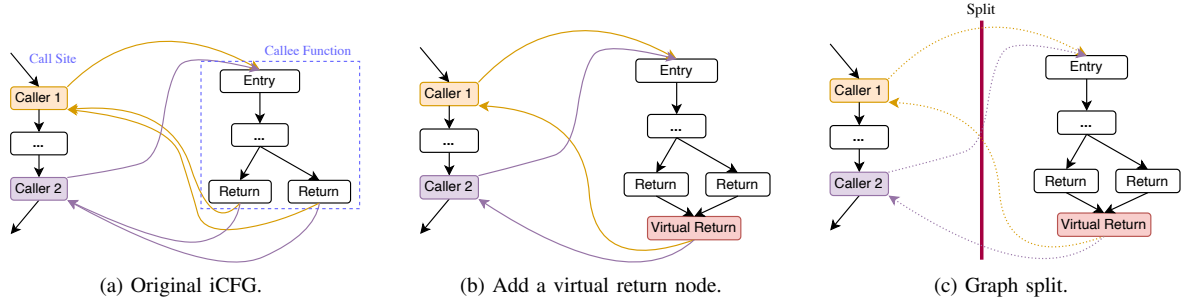


Figure 3: Transform an iCFG with caller-callee cycles into two subgraphs. The original iCFG contains two callers and the edges in yellow and purple are calling and returning edges in Figure 3a. We transform the iCFG by adding a virtual return node in red in Figure 3b. We remove the calling and returning edges for graph split and mark them as dotted lines in Figure 3c. The right subgraph is a one-entry-one-exit subgraph.

4.1. Path Prioritization via Multiple MPCs

We explain in detail how to compute multiple MPCs in iCFG and prioritize the paths for symbolic execution. Firstly, we transform the vanilla iCFG with cycles into many acyclic iCFG subgraphs. For each acyclic iCFG subgraph, we enumerate all MPCs using the maximum matching method in graph theory. Multiple MPCs can provide diverse path selection in the path prioritization phase of symbolic execution. In the end, we guide the symbolic execution to search a subset of paths instead of all possible paths with the help of multiple MPCs.

4.1.1. MPC Computation in Transformed iCFG. We start with the computation of a single MPC in an iCFG of a program. Computing an MPC in an iCFG is challenging because the iCFG contains cycles but computing an MPC in such a cyclic graph is an NP-hard problem [60]. In graph theory, there are only polynomial run-time algorithms in the directed acyclic graph (DAG) as shown in Algorithm 1. Therefore, we try to find a solution that closely approximates the MPC in iCFG via approximation algorithms, which are typical solutions to NP-hard problems [79]. Generally, our approximation algorithms transform the cyclic iCFG into DAGs and then compute MPC in the transformed iCFG. Note that we *only* transform the iCFG in the graph analysis step. The path exploration including path prioritization at run-time will not be influenced by our graph transformation.

The iCFG includes two types of cycles: **caller-callee cycles** and **loop cycles**. Figure 3a provides a simple example of a caller-callee cycle. This cycle begins at the node Entry in the callee function, traverses the entire callee function, and then returns to the node Caller 1. Subsequently, Caller 1 transitions to Caller 2, which then returns to Entry in the callee function. Thus, a caller-callee cycle typically occurs when a callee function is invoked by multiple callers. Additionally, as illustrated in Figure 4a, a **loop cycle** starts at the Loop Header, traverses through the loop body, and returns to the Loop Header via a back edge.

Both types of cycles—caller-callee and loop cycles—comprise the following components, as defined by

Empc. A **cycle body** is a set of vertices and edges that form the core structure of the cycle. **Cycle edges** are back edges that point to an ancestor of the current node, creating the cycle. **Connecting edges** are edges that connect the cycle body to other parts of the graph or within the cycle itself. As shown in Figure 3a, in a **caller-callee cycle**, the cycle body corresponds to the function itself, which is enclosed in the blue dashed box. The connecting edges include the calling and returning edges of the first caller, which are highlighted as yellow edges. The cycle edges are the calling and returning edges of the second caller, highlighted as purple edges. Similarly, as shown in Figure 4a, in a **loop cycle**, the cycle body refers to the loop body, excluding the back edges, and is enclosed in the blue dashed box. The connecting edges consist of entering and exiting edges, highlighted as yellow edges. The cycle edges are the back edges, highlighted as purple edges.

Based on this analysis, we approximate the computation of the MPC in an iCFG through the following two steps. Firstly, we transform the iCFG into an acyclic graph by removing cycles edges in the graph analysis phase, which precedes path prioritization, because this operation enables the computation of MPCs using Algorithm 1. Thus, the transformed iCFG includes the cycle body but excludes the cycle edges, ensuring that the vertices within the cycle body appear at most once on a path. Consequently, our approximation simplifies the computation on the original directed cyclic graph by reducing it to a directed acyclic graph, effectively ignoring the cycle edges. Secondly, we analyze each cycle body independently from the transformed iCFG as a transformed subgraph. We split the transformed graph by removing the connecting edges. This step is important because the cycles need to be reintroduced after computation to ensure program completeness. The subgraph contains subpaths of the paths in the iCFG, allowing us to compute the MPC within the transformed subgraph. During path prioritization, *Empc* can still account for repeated executions of the cycle body by considering a new path from the MPC of the cycle body each time it repeats. Finally, we propose new theorems and approaches to make the computation of

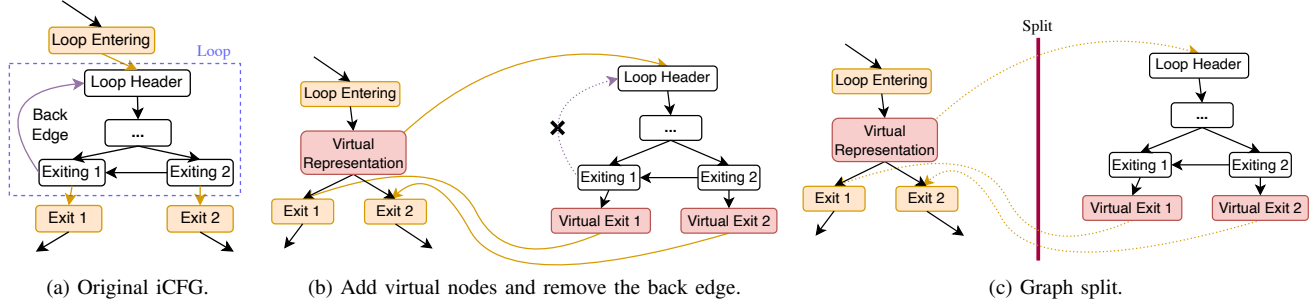


Figure 4: Transform the iCFG with loop cycles into two subgraphs without cycles. The original iCFG contains a loop with a loop body and a back edge in Figure 4a. The iCFG is transformed by removing back edges and adding several virtual nodes in red in Figure 4b. The loop body is represented by a virtual node in the transformed iCFG and the virtual exit nodes match up with the exit nodes respectively. The loop entering and exiting edges are removed for graph split and marked as dotted lines in Figure 4c and the right subgraph is a loop subgraph.

MPCs in each subgraph provably correct. Based on these analyses, we propose detailed approaches for transforming each of the two types of cycles described above.

Transform Caller-Callee Cycles. Based on the several steps described above, *Empc* first adds a virtual node as the successor of all return nodes in the function body (see Figure 3b), and then splits the graph into two parts, as shown in Figure 3c. The left part represents the transformed graph, while the right part is a subgraph with one entry and one exit node. We classify such subgraphs as one-entry-one-exit graphs. As the name suggests, each one-entry-one-exit graph $G_{sub}(V_{sub}, E_{sub})$ has exactly one entry vertex v_{ss} and one exit vertex v_{st} . Its formal definition is provided in Definition 1. The right part of Figure 3c illustrates a one-entry-one-exit subgraph. These subgraphs can be generated using Algorithm 2. For function calls, the entry block and exit block of a function can be easily identified, enabling the generation of a one-entry-one-exit subgraph via Algorithm 2.

Definition 1 (One-Entry-One-Exit Subgraph). A subgraph $G_{sub}(V_{sub}, E_{sub})$ is a one-entry-one-exit subgraph of $G(V, E)$ iff. i) G_{sub} is an induced subgraph of G ; ii) G_{sub} has two vertices v_{ss} and v_{st} where in-degree of v_{ss} is 0 and out-degree of v_{st} is 0 in G_{sub} ; iii) for each vertex $v_i \in V_{sub}$ with $v_i \neq v_{st}$, the successor vertices $V_i^s \subset V_{sub}$; iv) for each vertex $v_i \in V_{sub}$ with $v_i \neq v_{ss}$, the predecessor vertices $V_i^p \subset V_{sub}$.

To ensure the provable correctness of the MPC computation in each one-entry-one-exit subgraph, we propose Theorem 1 and provide its proof in Appendix A. After conceptually splicing the MPCs of the subgraphs at the merged vertex, the path cover for the transformed iCFG, excluding cycle edges, remains minimum. Regarding the maximum k value, it can be determined through path cover enumeration, as described in Section 4.1.2. At run-time, a function call may appear multiple times along a single path. For each function call, *Empc* always considers a new path in the MPC of the corresponding function subgraph to guide

Algorithm 2 Generate a One-Entry-One-Exit Subgraph

Input:	A transformed iCFG $G(V, E)$ without cycle edges Entry and exit vertices v_{ss}, v_{st} of subgraph
Output:	A subgraph $G_{sub}(V_{sub}, E_{sub})$ The transformed graph $G'(V', E')$ from G without V_{sub} and E_{sub}

- 1: $V_{sub} \leftarrow \{v_{ss}, v_{st}\}$
- 2: **for** $v_k \in V, v_{ss}$ reaches v_k, v_k reaches v_{st} **do**
- 3: Add v_k to V_{sub}
- 4: $G_{sub}(V_{sub}, E_{sub}) \leftarrow \text{get_induced_graph}(G, V_{sub})$
- 5: $V'_{sub} \leftarrow V_{sub} \setminus \{v_{ss}, v_{st}\}$
- 6: $V' \leftarrow V \setminus V'_{sub}$
- 7: $E' \leftarrow E \setminus E_{sub}$
- 8: Merge v_{ss} and v_{st} into v_{sst} in V'
- 9: **return** $G_{sub}(V_{sub}, E_{sub}), G'(V', E')$

path selection.

Theorem 1. For a directed acyclic graph $G(V, E)$, $G_{sub}(V_{sub}, E_{sub})$ is a one-entry-one-exit subgraph of G . $G'(V', E')$ is a transformed graph from $G(V, E)$ computed by Algorithm 2 and v_{sst} is the merged virtual vertex. P_m^{sub} is an MPC of G_{sub} ; P_m is an MPC of G . P'_m is an MPC of G' that satisfies: i) there are k paths going through v_{sst} in P'_m ; ii) k is the maximum among all MPCs in G' . We have $|P_m| = |P'_m| - k + \max(|P_m^{sub}|, k)$.

See Appendix A for proof.

Transform Loop Cycles. We adopt the definition of loops in LLVM [52], with some additional terminology illustrated in Figure 4a [5]. According to this definition, a loop has a single header but may contain multiple exiting nodes within its body. Consequently, the one-entry-one-exit subgraph used for function calls cannot be applied to loops, necessitating the definition of a different type of subgraph. In addition to the common loops defined by LLVM, developers occasionally write extraordinary loops that have more than one header (entry) node. We do not handle these extraordinary loops for two reasons. First, extraordinary loops cannot be easily transformed into a subgraph suitable for computing MPCs and splicing at a merged vertex. Second, these loops are extremely rare in real-world programs, accounting for

only 0.3% of the benchmark programs, as shown in Section 6.5.

The second type of subgraphs introduced in this section is called a loop subgraph, as shown in Figure 4. Each loop contains at least one back edge. To represent the loop body (i.e., the cycle body), we remove the back edge and introduce a virtual representation as a merged vertex, as illustrated in Figure 4b. The transformed iCFG is then split into two parts, as shown in Figure 4c, where the right part represents a loop subgraph. A loop subgraph can be formally defined as a directed acyclic subgraph $G_{sub}(V_{sub}, E_{sub})$ in Definition 2. The logic for generating a loop subgraph is described in Algorithm 3. Loop information, *loop_info*, can be obtained using loop analysis provided by compilers such as LLVM.

Definition 2 (Loop Subgraph). $G_{sub}(V_{sub}, E_{sub})$ is a loop subgraph of $G(V, E)$ iff. i) G_{sub} is an induced subgraph of G ; ii) G_{sub} has a vertex v_{ss} and a set of vertices $V_{st} = \{v_{st_1}, \dots, v_{st_k}\}$ where in-degree of v_{ss} is 0 and out-degree of $v_{st_i} \in V_{st}$ is 0 in G_{sub} ; iii) for each vertex $v_i \in V_{sub}$ with $v_i \notin V_{st}$, the successor vertices $V_i^s \subset V_{sub}$; iv) for each vertex $v_i \in V_{sub}$ with $v_i \neq v_{ss}$, the predecessor vertices $V_i^p \subset V_{sub}$.

Algorithm 3 Generate a Loop Subgraph

Input:	A transformed iCFG $G(V, E)$ without cycle edges Loop info <i>loop_info</i> analyzed by the compiler
Output:	A loop subgraph $G_{sub}(V_{sub}, E_{sub})$ A transformed graph $G'(V', E')$ from G without V_{sub} and E_{sub}

```

1:  $v_{ss} \leftarrow \text{get\_header}(\text{loop\_info})$ 
2:  $V_{st} \leftarrow \text{get\_exit\_vertices}(\text{loop\_info})$ 
3:  $E_{st} \leftarrow \text{get\_exiting\_edges}(\text{loop\_info})$ 
4:  $V_{sub} \leftarrow \text{get\_vertices}(\text{loop\_info})$ 
5:  $E_{sub} \leftarrow \text{get\_edges}(\text{loop\_info})$ 
6: Replace  $v_{ss}$  with a virtual vertex  $v_{sst}$  in  $G$ 
7:  $V' \leftarrow V \setminus V_{sub}$ 
8:  $E' \leftarrow E \setminus E_{sub}$ 
9:  $V_{sub} \leftarrow V_{sub} \cup V_{st}$  ▷ Include virtual exit vertices
10:  $E_{sub} \leftarrow E_{sub} \cup E_{st}$  ▷ Include virtual exiting edges
11: for  $v_{st_i} \in V_{st}$  do
12:   Add  $(v_{sst}, v_{st_i})$  to  $E'$ 
13: return  $G_{sub}(V_{sub}, E_{sub})$ ,  $G'(V', E')$ 

```

We present Theorem 2 and provide its proof in Appendix B to establish the correctness of the computation within each loop subgraph. Similar to function calls, at run-time, a loop may appear multiple times along a single path. For each loop, *Empc* steers path selection toward the next cycle of the loop subgraph rather than breaking out of the loop. Within each loop, *Empc* considers a new path in the MPC of the loop subgraph to guide path selection until no matching path remains in the MPC. At that point, *Empc* selects a state that breaks out of the loop.

Theorem 2. For a directed acyclic graph $G(V, E)$, $G_{sub}(V_{sub}, E_{sub})$ is a loop subgraph of G . $G'(V', E')$ is a transformed graph from (V, E) computed by Algorithm 3; v_{sst} is the replaced virtual vertex; V_{st} is a group of exit vertices. P_m^{sub} is an MPC of G_{sub} ; P_m is an MPC of G . P'_m

is an MPC of G' that satisfies: i) $P'_{sst} \subset P'_m$ is a group of paths in which each path p_{sst_i} goes through edge (v_{sst}, v_{st_i}) with $v_{st_i} \in V_{st}$; ii) $k = |P'_{sst}|$ is the maximum among all MPCs in G' . We have $|P_m| = |P'_m| - k + \max(|P_m^{sub}|, k)$.

See proof in Appendix B.

4.1.2. Multiple MPCs to Cover Diverse Paths. We now extend the computation of a single MPC to the computation of multiple MPCs. Conceptually, an MPC in a directed graph is not unique, meaning that a graph may have multiple distinct MPCs. Algorithm 1 outlines an approach to compute a single MPC, which is derived from a maximum matching. The generation of this matching depends on the starting vertex used in the maximum matching algorithm, making the computation of an MPC inherently non-deterministic among all possible MPCs in the graph. However, run-time path selection is influenced by various factors, including data flow and path feasibility determined by SMT solvers. As a result, it is not possible to predict which MPC aligns most closely with the actual path selection in the symbolic execution engine.

To handle scenarios where paths are fixed and uniquely determined without alternatives, enumerating all MPCs is an effective approach, as it ensures that every possible path cover is considered. Algorithm 1 employs a maximum matching method to compute an MPC using the Hopcroft-Karp algorithm [39]. However, the maximum matching generated by the Hopcroft-Karp algorithm is non-deterministic, as the algorithm begins its iterations from a random vertex. Consequently, the MPC generated by Algorithm 1 is also non-deterministic. Theorem 3 establishes that for each MPC in G , there exists a corresponding maximum matching in G_b . Therefore, the problem of enumerating MPCs in G can be reduced to the problem of enumerating maximum matchings in G_b .

Theorem 3. For a directed acyclic graph $G(V, E)$, an MPC P_m of G and a transformed bipartite graph $G_b(V_b, E_b)$ of G introduced in Algorithm 1, there must be a maximum matching M_m in G_b that can be converted to P_m via Algorithm 1.

See Appendix C for proof.

Enumerating maximum matchings in a bipartite graph is a well-studied problem in graph theory [74,75]. We adopt Takeaki's method [75] to enumerate all maximum matchings in a bipartite graph. This method leverages the property that the symmetric difference between two maximum matchings consists of cycles and paths of even length. It generates all maximum matchings by exchanging edges in existing matchings and iteratively applying this process. The time complexity of this algorithm is $O(|E_b| |V_b|^{\frac{1}{2}} + |V_b| |max_matching_group|)$ [75]. To optimize this computation process, we reduce the time complexity through two approaches. First, we leverage the one-entry-one-exit subgraphs introduced in Definition 1 and Theorem 1. By transforming a large, complex graph into smaller subgraphs, where each subgraph is a one-entry-one-exit graph, we en-

able independent analysis of each subgraph using Takeaki’s method and Algorithm 1 to generate multiple MPCs. Second, for highly complex subgraphs, we impose a limit on the number of MPCs generated to achieve an approximation. In practice, most subgraphs are small, containing only a few vertices and edges, which makes this optimization highly effective.

4.1.3. Path Prioritization at Run-time. After transforming the iCFG and computing MPCs for each subgraph, we do not merge these MPCs into a single set for the entire iCFG, as this operation is computationally expensive. Instead, we consider the MPCs in each subgraph independently during the path prioritization phase at run-time.

At run-time of symbolic execution, *Empc* maintains a group of MPCs for each subgraph. This group initially contains all enumerated MPCs generated during the pre-processing stage. The symbolic execution engine forks an execution state into two (or more) states at each branch. Each state represents a subpath from the program entry to the current basic block. After SMT solvers verify the feasibility of the two states, *Empc* compares the subpaths of the states with all paths in the MPCs in the group for their respective subgraphs. In practice, *Empc* only needs to compare subpaths in the subgraphs described in Section 4.1.1 and Section 4.1.2. *Empc* then selects the state with the matched subpath to continue execution. Meanwhile, *Empc* removes those MPCs that do not contain a matching path for the current subpath, but ensures that at least one MPC remains in the group. If none of the MPCs provides a matching path, *Empc* handles infeasible paths as described in Section 4.2. Thus, on the one hand, the multiple MPCs provide a small subset of paths and prioritize these paths to guide path selection at run-time; on the other hand, run-time path feasibility influences the group of MPCs, dynamically refining the guidance provided by the MPCs.

4.2. Handle Infeasible Paths

In this section, we introduce another component of *Empc* designed to handle infeasible paths. The MPCs in *Empc* provide predefined paths to guide run-time path selection. However, despite enumerating MPCs, these paths are derived solely from iCFG and do not incorporate data-flow analysis. As a result, there is no guarantee that every run-time path will have a corresponding path in the MPC group. This is because the prefix of a path in the MPC group may become infeasible according to SMT solvers at run-time.

We approach the infeasible path problem from the perspective of conditional branches. A conditional branch contains a condition, which is a logical expression such as $x > y$. By altering the results of the branch condition, we can modify the constraints of a subpath, potentially changing its feasibility. The outcome of a conditional branch depends on the values assigned to its associated variables. These variables must have been defined in preceding basic blocks, which means that they depend on prior basic blocks and branches. Therefore, the key to identifying a path prefix

that potentially reaches this branch lies in analyzing the dependence information of these variables.

Definition 3 (Data Dependence). Suppose that a variable var_i is used in the condition c_i of a branch br_i . Data dependence is a map from br_i to a basic block node n_j in CFG. We say that br_i has data dependence on n_j iff. n_j defines var_i such that i) there is a path p from n_j to br_i , and ii) there is no any other node n_k on p that defines var_i .

Definition 4 (Potential Dependence). Suppose a variable var_i is used in the condition c_i of a branch br_i . The potential dependence is a map from br_i to a branch br_j . br_i is potentially dependent on br_j iff. i) there is a path p from br_j to br_i in which var_i is not defined; ii) there is another path p' from br_j to br_i where var_i is defined.

Definition 3 and Definition 4 introduce two classic types of program dependence that have been studied for decades [6,36,64,77]. **Data dependence**, derived from the def-use chain in program analysis, represents the relationship between a branch condition and a basic block that may influence this condition, thereby affecting branch choices. **Potential dependence** reveals that a previous branch can influence the definition of a variable, which, in turn, can affect the outcome of the current branch. We leverage data dependence and potential dependence to identify the preceding basic blocks and branches that influence a given branch. We trace the operand variables involved in def-use chains and also track the parameters and returns of function calls. Additionally, we perform basic pointer analysis by ignoring nested or multilevel pointers, as complete pointer analysis is computationally expensive.

At run-time of symbolic execution, when *Empc* invokes the infeasible path handling mechanism, it identifies the last unvisited basic block on the infeasible path and its corresponding branch, denoted as $br_unvisited$. It then performs a backward search on the iCFG starting from $br_unvisited$ until it discovers an ancestor basic block, $bb_ancestor$, that has a program dependence on $br_unvisited$. *Empc* locates symbolic execution states that reach $bb_ancestor$ and redirects symbolic execution to these states. As described in Section 4.1.3, *Empc* ignores execution states with mismatched subpaths in the MPC group during execution. However, during the current search for handling infeasible paths, *Empc* reconsiders these previously ignored states. If such a state is reconsidered, *Empc* temporarily disregards the current mismatched subpaths to allow the execution of this state to continue.

5. Implementation

Empc comprises two main components, all implemented in C++ using the LLVM [52] API (version 13.0.0). The first component is responsible for generating MPCs. We implement all algorithms mentioned in Section 4.1.1 from scratch, including the Hopcroft-Karp algorithm [39]. The second component, for program dependence analysis, is built on the def-use chains provided by LLVM. Our implementation

TABLE 1: The details of 12 benchmark programs. Each program is the latest version upon the evaluation. The number of basic blocks and code lines are calculated via KLEE internal coverage.

Project	Program	Version	Category	Binary Size	# Basic Blocks	# Code Lines
GNU bc	bc	1.07.1	Calculator	169KB	2430	3754
GNU ncurses	tic	6.5	Text	606KB	8972	9764
GNU make	make	4.4.1	Text	562KB	9450	11086
GNU bison	bison	3.8.2	Text	1531KB	22096	22690
GNU binutils	readelf	commit eb7892c4	Binary	2793KB	38397	47939
GNU binutils	strip-new	commit eb7892c4	Binary	5417KB	57257	76380
NASM	nasm	2.16.02	Binary	2507KB	16733	22369
libtiff	tiffinfo	4.6.0	Image	887KB	13490	16335
JasPer	jasper	4.2.4	Image	1041KB	14079	19146
Little CMS	transicc	2.16	Image	929KB	12242	17800
FLVMeta	flvmeta	1.2.2	Video	361KB	5122	6468
curl	curl	8.10.1	Network	3458KB	39413	46682

analyzes instructions one by one using a breadth-first search approach. However, we do not handle indirect calls due to the high cost associated with performing pointer analysis. Nonetheless, our dependence analysis remains efficient and operates on a small scale.

We now describe how *Empc* is integrated into one of the most widely used symbolic execution engines, KLEE [20] (version 3.1). We incorporate several C++ header and source files into KLEE’s core modules. During the symbolic execution process, KLEE first loads a program’s bytecode file and parses it into LLVM IR. *Empc* uses the LLVM IR to perform graph analysis and dependence analysis without modifying the IR. Once these analyses are complete, KLEE continues with the symbolic execution of the program. Path prioritization and infeasible path handling are embedded into KLEE’s state update and state selection functions, as these functions serve as interfaces for new searchers. In Section 6.4, we evaluate the overhead introduced by *Empc* to the symbolic execution process.

6. Evaluation

We perform extensive evaluation of *Empc* aimed at answering the following research questions.

- 1) **RQ1 (Code coverage and resource usage):** Can *Empc* improve code coverage and meanwhile reduce the number of paths and memory usage?
- 2) **RQ2 (Finding security violations):** Can *Empc* find more security violations?
- 3) **RQ3 (Runtime overhead):** What is the runtime overhead of *Empc*?
- 4) **RQ4 (Design choice):** How do different components of *Empc* contribute to its performance?

6.1. Experiment Setup

Baseline search strategies. We consider all 11 search strategies in KLEE [2,20], including bfs, dfs, random-state, random-path, *nurs:rp*, *nurs:depth*, *nurs:covnew*, *nurs:md2u*, *nurs:icnt*, *nurs:cpicnt*, and *nurs:qc*. However, we exclude *nurs:depth* and *nurs:icnt* for the following reasons: *nurs:depth* is similar to *nurs:rp*, as both are guided by depth,

but *nurs:rp* consistently performs better than *nurs:depth*. Similarly, *nurs:icnt* is excluded for analogous reasons. In addition to the built-in search strategies in KLEE, we compare *Empc* with representative related works from recent years, as summarized in the following.

- **sgs.** Subpath-guided search (sgs) [55] prioritizes the selection of states whose subpaths have been explored the least frequently. In their implementation and evaluation, they executed four independent instances of the searcher with subpath lengths of 1, 2, 4, and 8. Each instance was allocated a quarter of the total time limit. We adopt their evaluation method as one of our baseline search strategies.
- **LEARCH.** LEARCH [37] is a novel learning-based strategy designed to effectively select promising states for symbolic execution, addressing the path explosion problem. It leverages existing heuristics for training data generation and feature extraction. LEARCH provides pre-trained feedforward network models [1], which we use directly in our evaluation.
- **cgs.** Concrete-constraint-guided search (cgs) [71] introduces a symbolic execution strategy guided by concrete constraints, aimed at covering more concrete branches and thereby improving overall code coverage in symbolic execution. cgs leverages data dependence to prioritize states that are likely to traverse partially covered concrete branches. We adopt their approach in our evaluation.

The three methods described above address the path explosion problem from different perspectives. *sgs* utilizes control-flow guidance and analyzes paths in CFGs. LEARCH represents the use of machine learning methods in symbolic execution. *cgs* is a recent approach that focuses on heuristics derived from real-world programs and employs data-flow analysis in symbolic execution. These search strategies make our baselines more comprehensive and extensive. However, these strategies are implemented on older versions of KLEE and LLVM. For example, LEARCH is built on KLEE 2.1 and LLVM 6.0, while *cgs* is based on KLEE 2.3 and LLVM 11.1.0. In contrast, *Empc* is implemented using KLEE 3.1 and LLVM 13.0. To address the issue of mismatched tool versions, we directly ported the source code of *sgs* into our

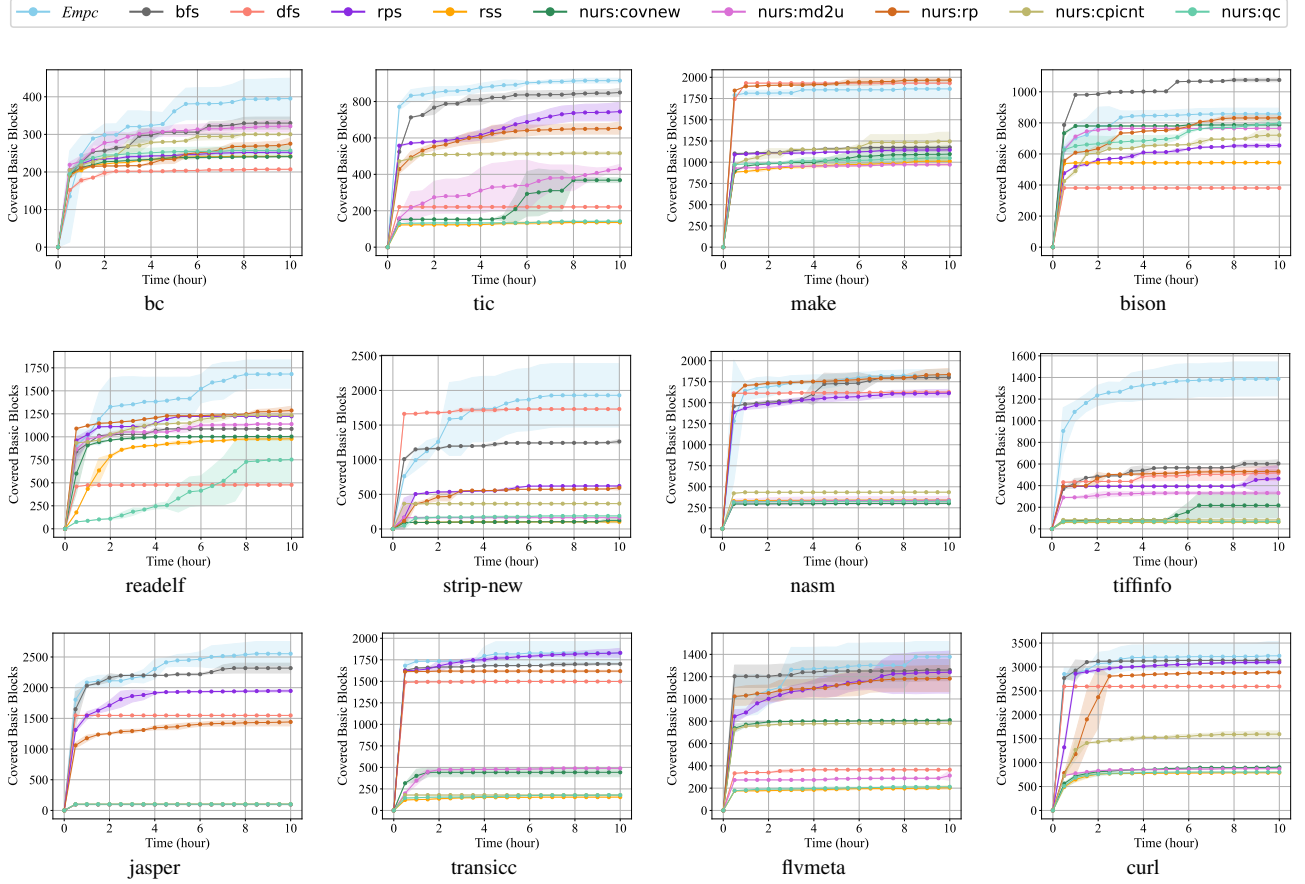


Figure 5: The arithmetic mean internal coverage of basic blocks of *Empc* and all KLEE baseline search strategies running for 10 hours on 12 benchmark programs and one standard deviation error bars over 10 runs. The basic block coverage is computed via KLEE internal coverage.

KLEE engine, as its implementation is relatively simple and the codebase is small. On the other hand, *LEARCH* and *cgs* involve extensive instrumentation of KLEE’s source code, making it difficult to port them directly. Fortunately, both provide source code and pre-configured environments that we can utilize. As a result, we build and maintain three separate environments separately for *Empc*, *LEARCH* and *cgs*.

Benchmark programs. We use 12 real-world open-source programs to evaluate *Empc*. These programs are widely used in fuzzing and symbolic execution techniques [14,19,37,45,56,58,71], and are applied in various domains such as text editing, binary operations, image and video processing, and networks. Table 1 provides details about these benchmark programs. To ensure *Empc* is applicable to current real-world programs, we use the latest versions of these benchmarks. Since *LEARCH* and *cgs* must be evaluated in their respective environments, we ensure that the benchmark programs are configured and built with consistent parameters across all environments. KLEE provides symbolic environment settings for each executed program, where it symbolizes program arguments as well as certain input and

output files based on these symbolic environment configurations. We configure these symbolic environments for our benchmarks as detailed in Table 6 in Appendix D. We define symbolic arguments based on prior works [37,45,71] and usage information provided by the programs themselves.

Environment setup. We conduct all evaluations on a 64-bit machine equipped with 128 Intel Xeon (Cascade Lake) Platinum 8269CY CPUs and 3072 GB of RAM. Each KLEE instance is restricted to a single CPU core and a maximum of 32 GB of RAM. We run up to 95 KLEE instances simultaneously.

6.2. RQ1: Code Coverage and Resource Usage

In this section, we evaluate *Empc* by comparing its code coverage and resource usage against the baseline search strategies mentioned earlier. Each KLEE instance is assigned a memory limit of 32 GB. This limit, which is significantly higher than the default of 4 GB, allows us to measure the resource usage, including memory consumption, for all search strategies. *LEARCH* and *cgs* are executed in their respective environments based on older versions of KLEE,

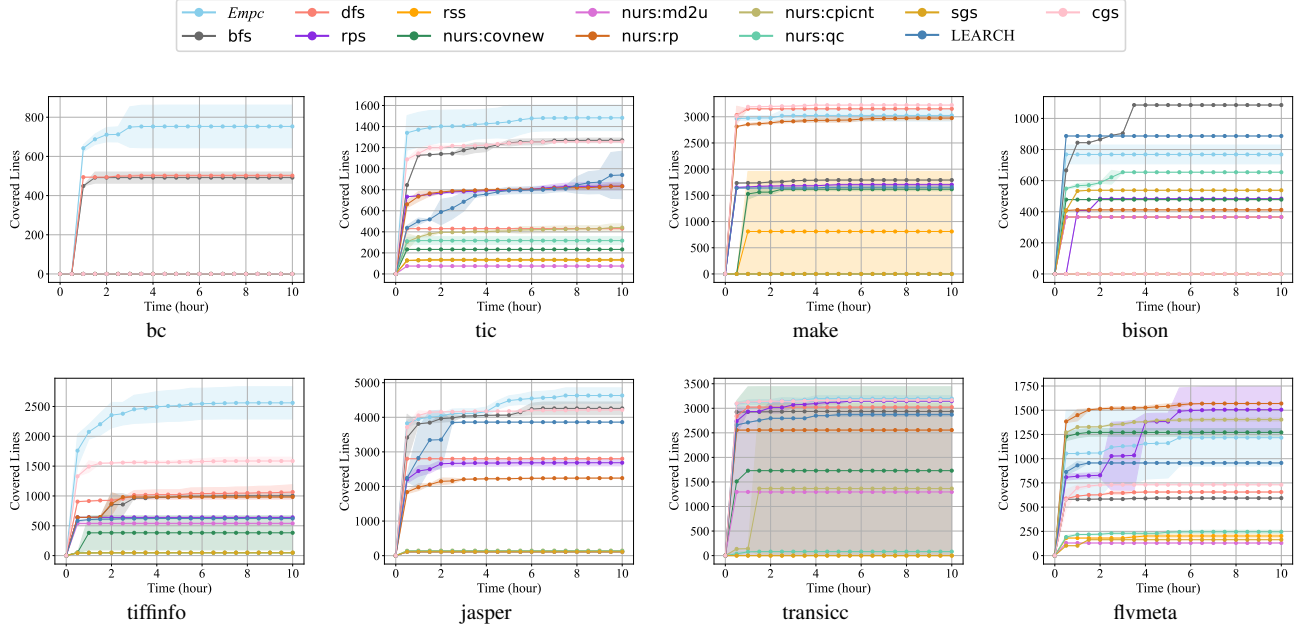


Figure 6: The arithmetic mean external coverage of code lines of *Empc*, KLEE search strategies and 3 prior works running for 10 hours on 8 benchmark programs and one standard deviation error bars over 10 runs. The line coverage is computed via replaying test inputs using gcov.

where some programs in our benchmarks (using the latest versions) cannot run successfully. These programs include readelf, strip-new, nasm, and curl. To address this compatibility issue, we divide the benchmarks and baselines into two groups: **Group A**: All benchmark programs with KLEE’s baseline search strategies; **Group B**: A subset of 8 benchmark programs compatible with all baseline search strategies. We repeat each experiment 10 times, with each KLEE instance running for 10 hours. Finally, we compute the mean and standard deviation to analyze the results.

6.2.1. Code Coverage. KLEE merges the program bitcode with the bitcode of external libraries to produce a final complete bitcode before symbolic execution. In our evaluations, we collect coverage only for the source code in the program binary, excluding external libraries (e.g. C standard library). This is because external libraries are shared across programs, and their coverage statistics are not meaningful in this context. There are two options [21] for measuring code coverage for programs symbolically executed on KLEE. The first is **internal coverage**, which is reported directly by KLEE. A statement is considered covered if it has been symbolically executed. The second is **external coverage**, which is calculated by replaying the test inputs generated by KLEE on a native version of the program instrumented with gcov [3] or llvm-cov [4]. We use both metrics to evaluate the code coverage of *Empc* and the baselines. We report internal coverage for basic blocks and external coverage for lines of code. This distinction aligns with *Empc*’s focus on basic blocks in CFGs, while replaying via gcov typically reports line coverage. However, due to the fact that LEARCH and

cgs are based on different versions of LLVM and KLEE, it is not possible to completely eliminate the inherent impact of version differences when reporting internal coverage. As a result, we report internal coverage of basic blocks for Group A experiments and external coverage of code lines for Group B experiments.

Figure 5 presents the experimental results for experiment Group A. Among the 12 benchmark programs, it is clear that *Empc* emerges as the best-performing search strategy in 10 of them. It also shows significant improvements in internal coverage for several programs, such as bc, readelf, and tiffinfo. For bison, *Empc* ranks second, trailing bfs by 200 basic blocks. In the case of make, *Empc* ranks third, closely matching the coverage achieved by nurs:rp and dfs. Overall, *Empc* demonstrates consistently strong performance across the 12 benchmark programs, in contrast to the highly variable results exhibited by KLEE’s baseline search strategies. The standard deviations in results for some programs are relatively high. This is because *Empc* explores fewer paths, which can lead to encountering complex path constraints more frequently. Some of these constraints are difficult for the SMT solver to resolve, occasionally resulting in a negative impact on *Empc*’s performance. Besides, our infeasible path handling includes some random choices for dependent states. Despite these challenges, *Empc* generally outperforms KLEE’s baselines in covering basic blocks. In total, *Empc* covers 19853 basic blocks, whereas the best baseline search strategy, bfs, covers 16602 basic blocks. This represents an overall improvement of 19.6% compared to KLEE’s baselines.

The experimental results for experiment Group B are

TABLE 2: The size of memory (in bytes) held in the heap by a KLEE instance is measured at the 3rd, 7th, and 10th hours of execution. This evaluation is conducted only on KLEE baseline search strategies. Reduction*: We exclude comparisons with dfs due to its specificity. We calculate the proportion of states reduced by *Empc* relative to the strategy with the minimum number of states. Non-negative reductions are represented by green cells. The corresponding memory usage of *Empc* is shown in blue cells, while the memory usage of the compared strategy is shown in red cells.

Program	Time	Reduction*	Empc	bfs	dfs*	rss	rps	nurs				
								rp	covnew	md2u	cpicnt	qc
bc	3h	-38.2%	94M	89M	71M	74M	80M	68M	87M	31.3G	427M	19.2G
	7h	-17.9%	99M	126M	75M	105M	101M	84M	164M	31.3G	1.36G	31.3G
	10h	-10.2%	108M	184M	79M	147M	113M	98M	237M	31.3G	2.37G	31.3G
tic	3h	44.9%	14.6G	31.3G	107M	31.3G	31.3G	31.3G	26.5G	31.3G	31.3G	31.3G
	7h	16.3%	26.2G	31.3G	108M	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G
	10h	0%	31.3G	31.3G	106M	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G
make	3h	68.7%	727M	31.3G	143M	31.3G	31.3G	31.3G	31.3G	31.3G	2.32G	31.3G
	7h	84.9%	1.11G	31.3G	146M	31.3G	31.3G	31.3G	31.3G	31.3G	7.35G	31.3G
	10h	87.6%	1.19G	31.3G	148M	31.3G	31.3G	31.3G	31.3G	31.3G	9.61G	31.3G
bison	3h	-68.7%	361M	274M	188M	282M	246M	240M	327M	328M	214M	302M
	7h	-61.5%	378M	354M	190M	349M	283M	278M	396M	387M	234M	449M
	10h	-51.2%	381M	385M	192M	400M	301M	305M	435M	437M	252M	529M
readelf	3h	-26.4%	1.77G	1.40G	420M	3.52G	3.09G	2.11G	3.77G	2.38G	1.75G	9.02G
	7h	-127%	4.11G	1.81G	604M	5.43G	4.56G	2.94G	7.38G	3.01G	3.43G	31.3G
	10h	-204%	6.18G	2.03G	718M	8.54G	4.57	4.27G	10.0G	3.60G	4.35G	31.3G
strip-new	3h	-203%	1.32G	651M	744M	31.3G	436M	461M	31.3G	31.3G	31.3G	461M
	7h	-210%	1.43G	841M	750M	31.3G	461M	525M	31.3G	31.3G	31.3G	470M
	10h	-201%	1.43G	1.01G	761M	31.3G	475M	571M	31.3G	31.3G	31.3G	476M
nasm	3h	72.8%	1.95G	31.3G	644M	11.8G	31.3G	31.3G	11.1G	31.3G	8.00G	7.17G
	7h	69.2%	3.60G	31.3G	1.06G	18.5G	31.3G	31.3G	17.5G	31.3G	13.6G	11.7G
	10h	69.3%	4.23G	31.3G	1.41G	22.2G	31.3G	31.3G	20.8G	31.3G	17.1G	13.8G
tiffinfo	3h	38.8%	1.09G	2.22G	934M	31.3G	1.78G	2.75G	31.3G	31.3G	31.3G	31.3G
	7h	32.2%	1.94G	4.08G	1.60G	31.3G	2.86G	4.11G	31.3G	31.3G	31.3G	31.3G
	10h	32.5%	2.35G	4.60G	2.06G	31.3G	3.48G	5.27G	31.3G	31.3G	31.3G	31.3G
jasper	3h	86.0%	449M	12.8G	587M	5.49G	31.3G	27.1G	4.97G	3.20G	7.84G	4.48G
	7h	80.3%	1.01G	22.3G	1.13G	8.77G	31.3G	31.3G	7.95G	5.12G	12.5G	7.41G
	10h	80.3%	1.22G	24.6G	1.51G	10.1G	31.3G	31.3G	9.54G	6.15G	15.0G	8.56G
transicc	3h	64.9%	6.10G	21.9G	831M	31.3G	31.3G	31.3G	31.3G	17.4G	31.3G	31.3G
	7h	56.2%	13.7G	31.3G	1.00G	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G
	10h	45.0%	17.2G	31.3G	1.14G	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G	31.3G
flvmeta	3h	88.5%	377M	3.42G	560M	31.3G	3.27G	4.34G	12.6G	31.3G	15.4G	31.3G
	7h	90.5%	489M	10.8G	911M	31.3G	5.15G	7.01G	17.2G	31.3G	25.7G	31.3G
	10h	93.5%	492M	11.7G	1.16G	31.3G	7.57G	7.97G	19.8G	31.3G	31.3G	31.3G
curl	3h	-85.2%	678M	496M	301M	1.51G	368M	366M	31.3G	26.1G	5.22G	31.3G
	7h	-38.8%	680M	578M	339M	3.24G	490M	508M	31.3G	29.7G	31.3G	31.3G
	10h	-15.0%	681M	647M	365M	5.97G	592M	641M	31.3G	28.8G	31.3G	31.3G

largely consistent with those of Group A, as shown in Figure 6. *Empc* achieves the best performance in 5 out of the 8 benchmark programs across all baseline search strategies, including prior works. In the case of flvmeta, *Empc* does not achieve the highest line coverage compared to its internal coverage. This is because certain paths that cover new lines are no longer executed when they do not align with the paths in the MPCs. Overall, *Empc* covers 17634 lines, while the best baseline search strategy, cgs, covers 14173 lines across the 8 benchmark programs. This represents an overall improvement of 24.4% compared to all baseline search strategies.

6.2.2. Resource Usage. In addition to measuring code coverage, we evaluate resource usage, including execution states and memory consumption, to demonstrate that our high-level approach is effective. We conduct experiments on both Group A and Group B. For Group A, memory usage can be easily obtained during execution using KLEE’s memory manager, which allows us to track heap memory usage

at any point in time. However, for Group B, comparing memory usage is not feasible because LEARCH and cgs are based on older versions of KLEE with different internal implementation details. Under these circumstances, a comparison of memory usage would not be reliable. To address this limitation, we instead compare the number of execution states for Group B, as the definition of states remains consistent across different versions of KLEE.

Table 2 presents the memory usage of *Empc* and KLEE baselines across 12 benchmark programs at the 3rd, 7th, and 10th hours of execution. We do not explicitly compare *Empc* with dfs, as it is inherently optimized to minimize the number of states at the graph level. Due to its depth-first nature, dfs maintains a small state pool, resulting in minimal memory consumption. However, this does not lead to higher code coverage for the majority of programs. Excluding dfs, *Empc* achieves the lowest memory usage in 7 out of the 12 benchmark programs, reducing memory consumption by up to 93.5% on flvmeta. Compared to other search strategies,

```

361 #define sum_get_unaligned_32(r, p) \
362 do { \
363     unsigned int val; \
364     memcpy(&val, (p), 4); \
365     r += val; \
366 } while(0); \
367 ... \
368 unsigned int \
369 jhash(unsigned const char *k, int length) \
370 { \
371     unsigned int a, b, c; \
372     ... \
395 if (length > 4) \
396 { \
397     sum_get_unaligned_32(b, k); //report error \
398     length -= 4; \
399     k += 4; \
400 } \
...

```

```

/* In file include/cms2.h */
102 typedef double cmsFloat64Number;
...
105 #if (USHRT_MAX == 65535U)
106 typedef unsigned short cmsUInt16Number;
107 #elif (UINT_MAX == 65535U)
108 typedef unsigned int cmsUInt16Number;
...
/* In file src/cms2_internal.h */
88 #define cmsINLINE static inline
...
182 cmsINLINE cmsUInt16Number
_cmsQuickFloorWord(cmsFloat64Number d)
183 {
184     return (cmsUInt16Number)
_cmsQuickFloor(d - 32767.0) + 32767U;
// report error
// implicit unsigned integer truncation
185 }

```

(a) Unsigned integer overflow at src/hash.c:397 in make

(b) Implicit unsigned integer truncation at src/cms2_internal.h:184 in transicc

Figure 7: The two UBSan violations discovered by *Empc* based on KLEE UBSan support.

Empc effectively reduces memory usage. Furthermore, we report the number of execution states in Table 7. The results are consistent: *Empc* maintains a small number of states in 5 out of the 8 benchmark programs and reduces the number of execution states by up to 88.6% on jasper.

Result 1: *Empc* increases basic block coverage by 19.6% compared to KLEE search strategies and line coverage by 24.4% compared to KLEE search strategies and 3 prior works. Moreover, *Empc* reduces memory usage by up to 93.5% compared to KLEE search strategies and the number of execution states by up to 88.6% compared to KLEE search strategies and 3 prior works.

6.3. RQ2: Finding Security Violations

We evaluate the ability of all search strategies to discover security violations using two metrics. The first metric leverages KLEE’s runtime Undefined Behavior Sanitizer (UBSan) support, and the second metric involves replaying test inputs on programs instrumented with sanitizers. nasm cannot be successfully compiled with UBSan, so it is excluded from the experiments in this section, leaving 11 benchmark programs for evaluation.

For the first metric, KLEE provides runtime support to detect certain UBSan bugs through its built-in functionality. However, since this support is limited, *Empc* identifies only two UBSan bugs, as shown in Figure 7, which is consistent with the results of KLEE’s baseline search strategies, as they also detect the same two UBSan bugs.

Based on the second metric, we replay the test cases generated by all search strategies on programs instrumented with UBSan and Address Sanitizer (ASan) to identify additional security violations. This experiment is conducted

```

/* In file scan.l */
276 yylval.s_value = strcpyof(yytext);
...
/* In file util.c */
42 char *strcpyof(const char *str){
...
47 temp = bc_malloc(strlen(str)+1);
...
647 void *bc_malloc(size_t size){
650 void *ptr;
652 ptr = (void *) malloc(size);
// direct leak of memory allocated here
...

```

```

/* In file jas_fix.h */
101 typedef int_least64_t jas_fix_t;
...
/* In file jas_seq.h */
106 typedef jas_fix_t jas_sequent_t;
...
/* In file pgx_dec.c */
524 static jas_sequent_t pgx_wordpoint(
524     uint_fast32_t v, int prec, bool sgnd){
528     jas_sequent_t ret = (sgnd && (v & (1
528     <<(prec - 1)))) ? (v - (1 << prec)) : v;
// multiple undefined behaviors
...

```

(a) Memory leak in bc

(b) Multiple undefined behaviors in jasper

Figure 8: One ASan violation and one UBSan violation discovered by *Empc* by replaying test inputs.

on 8 benchmark programs (i.e., the 8 programs compatible with LEARCH and cgs, as introduced in Section 6.2). The total number of security violations discovered by all search strategies on these 8 programs is reported in Figure 9. *Empc* detects a total of 70 security violations, outperforming the second-best baseline, bfs, by 24 violations. Furthermore, *Empc* uniquely identifies 15 new security violations that are not detected by any of the baseline search strategies. These results demonstrate that *Empc* is more effective at detecting security violations compared to baseline search strategies. This is because *Empc* prioritizes certain paths and maximizes code coverage within a limited time frame.

We illustrate four examples of security violations found by *Empc* in Figure 7 and Figure 8. In Figure 7a, the variable *r* adds *val* in a loop, resulting in an unsigned integer overflow. In Figure 7b, the return value is truncated from 32 bits to 16 bits. These two UBSan violations are detected by *Empc* using KLEE’s runtime UBSan support. Figure 8a demonstrates a memory leak: *yylval* copies text using *malloc* but does not release the allocated memory from the heap. Lastly, Figure 8b highlights a common violation that involves integer conversion. The variable is implicitly converted from type *int* to type *unsigned long*, altering its value. Additionally, the variable is left-shifted by 31 places, causing an overflow as the result cannot be represented within the *int*.

Although UBSan and ASan violations, such as memory leaks, may not cause a program to crash, they can indicate potential issues or unexpected behavior in deeper parts of the programs. Since our experiments are conducted on the latest versions of real-world programs, we have reported the 70 discovered security violations to the respective developers and received confirmation of the existence of 16 ASan and UBSan violations.

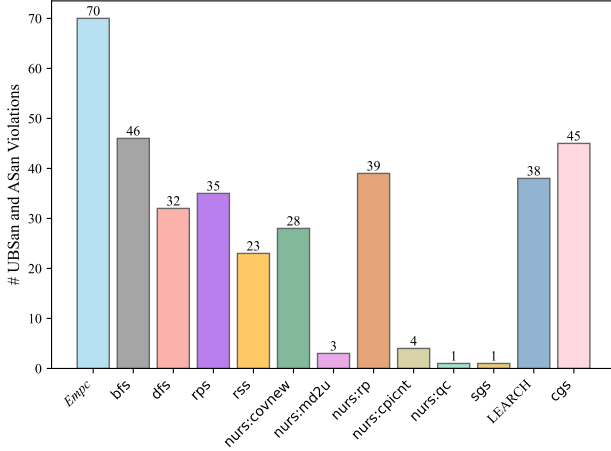


Figure 9: The total number of UBSan and ASan violations discovered by *Empc* and all baseline strategies via replay on 8 benchmark programs. This experiment is repeated 5 times and all security violations are collected.

Result 2: *Empc* is able to find more security violations than other search strategies on the 8 benchmark programs. It outperforms the second-best baseline search strategy by 24 violations.

6.4. RQ3: Runtime Overhead

In this experiment, we evaluate the overhead introduced by *Empc* to the symbolic execution engine KLEE. *Empc* comprises two analysis components performed prior to symbolic execution, and we report the time taken by each component as well as the overall analysis time in Table 3. The experiment is repeated 10 times, and the arithmetic mean is calculated. The results show that the analysis time for 10 out of 12 programs is limited to just a few dozen seconds. Even for more complex programs, such as *strip-new*, the analysis duration does not exceed 400 seconds. Consequently, the overall overhead introduced by *Empc* accounts for at most 1% of the total execution time. This overhead is negligible compared to the total run time of the programs, indicating that the analysis performed by *Empc* has minimal impact on performance.

Since runtime path prioritization is integrated into the state selection and update methods in KLEE, we measure the runtime overhead associated with these operations. We run *Empc* and *nurs:rp* on 12 benchmark programs for 10 hours, recording the total time spent on state selection and update during execution. This experiment is repeated 10 times, and the arithmetic mean is calculated. The comparison results are presented in Table 4. For 8 out of 12 programs, the total time spent on state selection remains less than 400 seconds, which is negligible compared to the overall run time of these programs. On average, *Empc* introduces a 12% overhead for engine maintenance across all programs. However, for more complex programs, such as *transicc* and

TABLE 3: The analysis performance overhead of *Empc* on all benchmark programs. It contains the time of graph analysis, dependence analysis and the overall analysis. The experiment is repeated 5 times and the mean is reported.

Program	Graph Analy.	Dep. Analy.	Overall
bc	561ms	262ms	823ms
tic	11.9s	1.5s	13.4s
make	11.2s	1.5s	12.7s
bison	18.2s	15.8s	34.0s
readelf	46.9s	28.9s	75.8s
strip-new	178s	200s	378s
nasm	15.0s	14.1s	29.1s
tiffinfo	39.3s	4.6s	43.9s
jasper	25.3s	2.3s	27.6s
transicc	12.3s	3.0s	15.3s
flvmeta	12.6s	1.1s	13.7s
curl	57.6s	81.8s	139.4s

TABLE 4: The runtime performance overhead of *Empc* and *nurs:rp* on benchmark programs, including the time of state selection and state update handled by the searchers. The experiment is repeated 5 times and the mean is reported.

Program	<i>Empc</i>	<i>nurs:rp</i>
bc	35ms	5.8ms
tic	190min	42.8s
make	122min	83.5s
bison	1.9s	36ms
readelf	270s	2.7s
strip-new	267s	721ms
nasm	283min	61.5s
tiffinfo	279s	5.7s
jasper	312s	34s
transicc	321min	6.5s
flvmeta	24.8s	9.3s
curl	421s	611ms

nasm, the time required for this procedure is longer. This increase in time consumption is attributed to *Empc*'s need to record, update, and manage all state operations during the selection and update process. Additionally, *Empc* performs more graph-level operations, further extending this phase. Despite the increased time required for these operations, it is important to emphasize that this does not adversely affect code coverage or memory usage.

Result 3: *Empc* adds at most 1% overhead to the overall analysis. *Empc* adds at most 1% overhead for symbolic execution engine maintenance on 8 out of 12 programs and adds 12% overhead in average for engine maintenance on all programs.

6.5. RQ4: Design Choice

Firstly, we provide an experimental analysis of extraordinary loops in the 12 benchmark programs. Extraordinary loops, as described in Section 4.1.1, are loops with multiple loop headers. Among the 12 benchmark programs, we identify 68 extraordinary loops out of a total of 21568 loops,

TABLE 5: The final internal coverage of basic blocks of *Empc* and the modified version without dependence analysis. The experiment is repeated 5 times and the mean is reported.

Program	<i>Empc</i>	Modified Version
bc	395	236
tic	915	860
make	1863	1820
bison	858	1100
readelf	1683	1570
strip-new	1930	1359
nasm	1830	1815
tiffinfo	1382	989
jasper	2553	2518
transicc	1828	1765
flvmeta	1378	928
curl	3233	3200

accounting for only 0.3% of all loops. This result indicates that extraordinary loops are rare in real-world programs.

Secondly, we investigate the contributions of different components of *Empc*. *Empc* comprises two main types of analysis: graph analysis, which is used for the computation of MPCs, and dependence analysis, which handles infeasible paths. Since graph analysis is the fundamental high-level concept of *Empc*, we begin by evaluating the performance impact of dependence analysis. To do this, we remove dependence analysis from the implementation and replace the method for handling infeasible paths with a dfs selection strategy. We then conduct experiments on 5 benchmark programs, repeating each experiment 5 times. Table 5 presents the comparison results between the original *Empc* and its modified version.

In our ablation experiments, integrating dependence analysis into *Empc* generally results in an increase in code coverage for 11 out of 12 benchmark programs, with an average increase of 16%. This demonstrates that program dependence analysis plays an important role in *Empc*, contributing to the observed code coverage improvements. However, the contribution of our MPC method is even more significant, as the increase introduced by program dependence analysis is only 16%. Overall, these results suggest that the primary driver of the code coverage improvement is the MPC method, which provides the most substantial contribution. In contrast, dependence analysis plays a secondary, yet supportive, role in enhancing *Empc*’s performance.

Result 4: Our results empirically support *Empc*’s design choices, demonstrating that the MPC method is the primary contributor, while program dependence plays a supportive role.

7. Limitations and Future Work

In this paper, we propose a novel approach that leverages minimum path covers to prioritize paths for symbolic execution, addressing the path explosion problem. To address the challenge of infeasible paths at run-time, we incorporate

program dependence to refine path prioritization. In our evaluation, we compare *Empc* only with KLEE-based search strategies because *Empc* is implemented on KLEE. In the future, we plan to extend *Empc* to other symbolic execution engines, such as angr [76], to evaluate its performance more broadly. Additionally, we plan to explore the application of *Empc* in the field of fuzzing, including using test inputs generated by *Empc* as fuzzing seeds to uncover more security vulnerabilities. We also plan to compare *Empc*’s effectiveness with existing fuzzing tools to assess its potential in improving software testing and vulnerability detection.

8. Related Work

Minimum path cover. Minimum path cover (MPC) is a classic problem in graph theory. Dilworth [31] and Fulkerson [32] proved it’s an NP-hard problem for general graphs but can be done in polynomial time for directed acyclic graphs. Many algorithms were proposed to compute MPC based on either maximum matching [32,60] or maximum flow [60]. MPC has been applied to various fields such as software testing and programming languages [11,12,48,78], distributed computing [44] and bioinformatics [65].

Symbolic execution. Symbolic execution is a powerful technique in software analysis that systematically explores the execution paths of a program by treating inputs as symbolic variables instead of concrete values [9,20,22,47,76]. This technique has enabled an increasing number of applications [16,26,27,29,38,42,50,67,72]. Research in this field has focused mainly on improving the scalability and efficiency of symbolic execution tools, particularly in addressing challenges such as path explosion, where the number of execution paths grows exponentially with the size of the program [9,22].

Concolic execution. Concolic execution combines the concrete and symbolic execution of the code under test to overcome the limitations of purely symbolic approaches [34,68,69]. By leveraging concrete inputs to guide exploration while using symbolic constraints to systematically cover program paths, many tools [23,24,28,34,41,57,69,80] have effectively detected vulnerabilities and generated comprehensive test suites. Although powerful, concolic execution still faces challenges with path explosion and complex constraints when analyzing large software systems.

Searching strategies tackling path explosion. Researchers have proposed various strategies to address the path explosion problem. One common approach involves using heuristics to prioritize and selectively explore paths that are more likely to uncover errors, thereby reducing the resource burden [13,20,55,81]. Another technique, known as state merging, combines similar execution paths to reduce the total number of paths [43,51]. Furthermore, integrating symbolic execution with other program analysis techniques and modern machine learning methods has proven effective in pruning infeasible paths early [17,25,73]. Additionally, advanced constraint-solving techniques aim to solve multiple constraints simultaneously, thereby reducing the burden

on the SMT solver [30,33,45]. Collectively, these methods contribute to mitigating the path explosion problem, making symbolic execution a more practical and scalable tool for software verification and security testing.

9. Conclusion

In this paper, we introduce a new approach *Empc* to tackle the path explosion problem in the symbolic execution technique based on minimum path cover (MPC). MPC provides an option for runtime path selection in order to use the least number of paths to maximize code coverage. This method not only increases code coverage but also reduces resource usage like memory usage. Moreover, we add program dependence analysis to handle some infeasible paths at run-time. We implement *Empc* on KLEE and show its effectiveness in increasing code coverage and reducing memory usage on 12 benchmark programs.

Acknowledgements

We sincerely appreciate the anonymous reviewers and our shepherd for their valuable feedback and guidance. We also thank Yuchong Xie and Qiao Zhang for their helpful comments and discussions that improved the paper.

References

- [1] Source code of Learch: a Learning-based Strategies for Path Exploration in Symbolic Execution. <https://github.com/eth-sri/learch.git>, 2022.
- [2] KLEE documentation. <https://klee-se.org/docs/>, 2023.
- [3] gcov — a test coverage program. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, 2024.
- [4] llvm-cov - emit coverage information. <https://llvm.org/docs/CommandGuide/llvm-cov.html>, 2024.
- [5] LLVM loop terminology (and canonical forms). <https://llvm.org/docs/LoopTerminology.html>, 2024.
- [6] Hiralal Agrawal, Joseph Robert Horgan, Edward W Krauser, and Saul A London. Incremental regression testing. In *1993 Conference on Software Maintenance*, pages 348–357. IEEE, 1993.
- [7] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings 14*, pages 367–381. Springer, 2008.
- [8] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, 2008.
- [9] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.
- [10] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.
- [11] Antonia Bertolino and Martina Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, 1994.
- [12] Antonia Bertolino and Martina Marré. How many paths are needed for branch testing? *Journal of Systems and Software*, 35(2):95–106, 1996.
- [13] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abughararh, and Lucas Bang. Metrinome: Path complexity predicts symbolic execution path explosion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 29–32. IEEE, 2021.
- [14] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.
- [15] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.
- [16] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 239–254, 2014.
- [17] Suhabe Bugrara and Dawson Engler. Redundant state detection for dynamic symbolic execution. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 199–211, 2013.
- [18] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.
- [19] Frank Busse, Martin Nowack, and Cristian Cadar. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 63–74, 2020.
- [20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
- [21] Cristian Cadar and Timotej Kopus. Measuring the coverage achieved by symbolic execution. <https://ccadar.blogspot.com/2020/07/measuring-coverage-achieved-by-symbolic.html>, 2020.
- [22] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
- [23] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1244–1254, 2018.
- [24] Sooyoung Cha and Hakjoo Oh. Concolic testing with adaptively changing search heuristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 235–245, 2019.
- [25] Sooyoung Cha and Hakjoo Oh. Making symbolic execution promising by learning aggressive state-pruning strategy. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 147–158, 2020.
- [26] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. Symcerts: Practical symbolic execution for exposing noncompliance in x. 509 certificate validation implementations. In *2017 IEEE symposium on security and privacy (SP)*, pages 503–520. IEEE, 2017.
- [27] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification. In *Network and Distributed Systems Security (NDSS) Symposium 2019*, 2019.

- [28] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. {SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2531–2548, 2022.
- [29] Emilio Coppa, Daniele Cono D’Elia, and Camil Demetrescu. Rethinking pointer reasoning in symbolic execution. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 613–618. IEEE, 2017.
- [30] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [31] Robert P Dilworth. A decomposition theorem for partially ordered sets. *Classic papers in combinatorics*, pages 139–144, 1987.
- [32] Delbert R Fulkerson. Note on dilworth’s decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4):701–702, 1956.
- [33] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 519–531. Springer, 2007.
- [34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.
- [35] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [36] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. *ACM SIGSOFT Software Engineering Notes*, 24(6):303–321, 1999.
- [37] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. Learning to explore paths for symbolic execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2526–2540, 2021.
- [38] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2245–2262, 2017.
- [39] John E Hopcroft and Richard M Karp. An $n^5/2$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.
- [40] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, (4):266–278, 1977.
- [41] Jie Hu, Yue Duan, and Heng Yin. Marco: A stochastic asynchronous concolic explorer. In *Proceedings of the 46th IEEE/ACM International conference on software engineering*, pages 1–12, 2024.
- [42] Yigong Hu, Gongqi Huang, and Peng Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 719–734, 2020.
- [43] Soha Hussein, Stephen McCamant, Elena Sherman, Vaibhav Sharma, and Mike Whalen. Structural test input generation for 3-address code coverage using path-merged symbolic execution. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 79–89. IEEE, 2023.
- [44] Selma Ikiz and Vijay K Garg. Efficient incremental optimal chain partition of distributed program traces. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS’06)*, pages 18–18. IEEE, 2006.
- [45] Timotej Kapus, Frank Busse, and Cristian Cadar. Pending constraints in symbolic execution for better exploration and seeding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 115–126, 2020.
- [46] James C King. A new approach to program testing. *ACM Sigplan Notices*, 10(6):228–233, 1975.
- [47] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [48] Mirosław Kowaluk, Andrzej Lingas, and Johannes Nowak. A path cover technique for lcas in dags. In *Scandinavian Workshop on Algorithm Theory*, pages 222–233. Springer, 2008.
- [49] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. Confetti: Amplifying concolic guidance for fuzzers. In *Proceedings of the 44th International Conference on Software Engineering*, pages 438–450, 2022.
- [50] Daniil Kuts. Towards symbolic pointers reasoning in dynamic symbolic execution. In *2021 Ivannikov Memorial Workshop (IVMEM)*, pages 42–49. IEEE, 2021.
- [51] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.
- [52] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [53] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *2013 International Conference on Availability, Reliability and Security*, pages 446–454. IEEE, 2013.
- [54] Peng Li, Rundong Zhou, Yaohui Chen, Yulong Zhang, and Tao (Lenx) Wei. Confuzzer: a sanitizer guided hybrid fuzzing framework leveraging greybox fuzzing and concolic execution. In *1st International KLEE Workshop on Symbolic Execution*, 2018.
- [55] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. *ACM SigPlan Notices*, 48(10):19–32, 2013.
- [56] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Security Symposium*, 2021.
- [57] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. Legion: Best-first concolic testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2020.
- [58] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery.
- [59] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
- [60] Simeon C. Ntafos and S. Louis Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, (5):520–529, 1979.
- [61] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180, 2010.

[62] Sebastian Poehlau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[63] Sebastian Poehlau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *NDSS 2021, Network and Distributed System Security Symposium*. Internet Society, 2021.

[64] Dawei Qi, Hoang DT Nguyen, and Abhik Roychoudhury. Path exploration based on symbolic output. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):1–41, 2013.

[65] Romeo Rizzi, Alexandru I Tomescu, and Veli Mäkinen. On the complexity of minimum path cover with subpath constraints for multi-assembly. In *BMC bioinformatics*, volume 15, pages 1–11. Springer, 2014.

[66] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. SymL: Guiding symbolic execution toward vulnerable states through pattern learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 456–468, 2021.

[67] Malte H Schwerhoff. *Advancing automated, permission-based program verification using symbolic execution*. PhD thesis, ETH Zurich, 2016.

[68] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[69] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[70] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[71] Yue Sun, Guowei Yang, Shichao Lv, Zhi Li, and Limin Sun. Concrete constraint guided symbolic execution. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[72] Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. Symbolic execution for randomized programs. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1583–1612, 2022.

[73] David Trabish, Andrea Mattavelli, Noam Rinetzy, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 350–360, 2018.

[74] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[75] Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *Algorithms and Computation: 8th International Symposium, ISAAC'97 Singapore, December 17–19, 1997 Proceedings*, 8, pages 92–101. Springer, 1997.

[76] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[77] Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. Dependence guided symbolic execution. *IEEE Transactions on Software Engineering*, 43(3):252–271, 2016.

[78] HS Wang, SR Hsu, and JC Lin. A generalized optimal path-selection model for structural program testing. *Journal of Systems and Software*, 10(1):55–63, 1989.

[79] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[80] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[81] Shunfan Zhou, Zheming Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: {State-Aware} symbolic execution for exploring {State-Dependent} program paths. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4365–4382, 2022.

Appendix A. Proof of Theorem 1

We first combine the MPCs in G' and G_{sub} to construct a path cover in G . If $k \geq |P_m^{sub}|$, we can easily expand $|P_m^{sub}|$ paths at v_{sst} in P'_m using the $|P_m^{sub}|$ paths in P_m^{sub} , and then the combined path cover P_{com} in G with $|P_{com}| = |P'_m|$; if $k < |P_m^{sub}|$, we can first expand k paths at v_{sst} in P'_m using the k paths in P_m^{sub} , and then add $|P_m^{sub}| - k$ paths in P_m^{sub} and expand them into complete paths in G , so we can get the combined path cover P_{com} in G with $|P_{com}| = |P'_m| + |P_m^{sub}| - k$. Then we just need to prove $|P_{com}| = |P_m|$.

Assume the opposite of what we want to prove, which is $|P_{com}| \neq |P_m|$. There must be $|P_{com}| > |P_m|$ since P_m is an MPC in G . Suppose there are j paths going through v_{ss} and v_{st} in P_m . There must be $j \geq |P_m^{sub}|$ since P_m^{sub} is an MPC in G_{sub} , so we can get $|P_m| \leq |P_m| - |P_m^{sub}| + j$. We remove the subgraph G_{sub} and merge it as a vertex v'_{sst} in the transformed graph G' . Let the paths in P_m going through v_{ss} and v_{st} go through the merged vertex v'_{sst} , then we can get a new path cover P'' in G' with $|P''| = |P_m|$. It's obvious that there are some identical paths in P'' due to the merged vertex v'_{sst} , and we can remove at least l paths to make no identical paths appear in P'' with $0 \leq l < j$. If we remove these l paths, we can get a new path cover P''' in G' with $|P'''| = |P''| - l = |P_m| - l$. If $k \geq |P_m^{sub}|$, we have $|P_{com}| = |P'_m|$, so we can get $|P'''| < |P'_m| - l < |P'_m|$, which contradicts the assertion that P'_m is an MPC in G' . If $k < |P_m^{sub}|$, we have $|P_{com}| = |P'_m| + |P_m^{sub}| - k$ and then we can get $|P'''| < |P'_m| + |P_m^{sub}| - k - l$. It's obvious that $j - l \leq k$ since k is the maximum, so we have $k + l \geq j \geq |P_m^{sub}|$. Finally, we get $|P'''| < |P'_m| + |P_m^{sub}| - k - l < |P'_m|$, which also contradicts the assertion of MPC. Thus, there must be $|P_{com}| = |P_m|$.

Appendix B. Proof of Theorem 2

Similar to the proof in Appendix A, we can easily expand $|P_m^{sub}|$ paths at v_{sst} and link vertices in V_{st} in P'_m using the $|P_m^{sub}|$ paths in P_m^{sub} . We still assume the opposite of what we want to prove, which is $|P_{com}| \neq |P_m|$. Thus, we conclude $|P_{com}| = |P_m|$.

Appendix C. Proof of Theorem 3

Before proving Theorem 3, we propose a lemma: for each MPC P_m , there must be at least one matching M in G_b that can be converted to P_m . This is because we can remove the disjoint vertices of these paths in P_m , and then

TABLE 6: The symbolic arguments in KLEE format for benchmark programs. These symbolic arguments are configured based on prior works and program-specific usage information.

Program	KLEE Symbolic Environment
bc	–sym-stdin 20
tic	–sym-args 0 2 8 A –sym-files 1 100
make	–n -f A –sym-files 1 40
bison	–sym-args 0 2 2 A –sym-files 1 100
readelf	–a A –sym-files 1 100
strip-new	–sym-args 0 2 8 A –sym-files 1 100
nasm	–sym-args 0 2 2 A –sym-files 1 100
tiffinfo	–sym-args 0 3 8 A –sym-files 1 300
jasper	–input A –output B –input-format –sym-arg 3 –output-format –sym-arg 3 –sym-args 0 3 15 –sym-files 2 300 –save-all-writes
transicc	–sym-args 0 2 4 A B –sym-files 2 200 –save-all-writes
flvmeta	–sym-arg 2 –sym-args 0 2 6 A –sym-files 1 300 –save-all-writes
curl	–sym-args 0 4 6 –sym-args 0 1 20

TABLE 7: The number of execution states held by a KLEE instance is recorded at the 3rd, 7th, and 10th hours of execution. This evaluation is conducted only on programs that are compatible with all baseline search strategies. Reduction*: We exclude comparisons with dfs due to its specificity. We calculate the proportion of states reduced by *Empc* relative to the strategy with the minimum number of states. Non-negative reductions are highlighted in green cells.

Program	Time	Reduction*	Empc	bfs	dfs*	rss	rps	nurs				sgs	LEARCH	cgs	
								rp	covnew	md2u	cpicnt	qc			
bc	3h	-15.9%	175	355	151	173	287	151	255	180K	2.6K	132K	453	492	220
	7h	34.4%	217	772	141	432	528	331	812	184K	9.6K	222K	1.2K	1.4K	1.4K
	10h	31.6%	310	1.4K	155	717	633	453	1.3K	187K	16K	221K	1.5K	2.0K	2.0K
tic	3h	-935%	238K	595K	499	1.3M	478K	569K	1.3M	1.7M	409K	1.3M	23K	82K	378K
	7h	-1153%	426K	552K	499	1.2M	418K	516K	1.6M	1.7M	360K	1.2M	34K	86K	504K
	10h	-1166%	519K	541K	495	1.2M	399K	505K	1.6M	1.7M	360K	1.2M	41K	85K	760K
make	3h	-32.4%	4.9K	575K	61	544K	576K	554K	564K	532K	39K	556K	13K	3.7K	111K
	7h	-79.2%	8.6K	565K	61	486K	565K	549K	529K	440K	127K	517K	21K	4.8K	148K
	10h	-87.8%	9.2K	560K	61	457K	562K	547K	530K	475K	167K	481K	26K	4.9K	175K
bison	3h	-2.1%	146	772	70	2.5K	512	475	3.1K	3.0K	143	2.6K	2.1K	497	335
	7h	14.0%	208	1.1K	94	3.3K	760	751	3.9K	3.7K	242	3.7K	2.7K	921	525
	10h	34.3%	222	1.3K	105	3.6K	860	882	4.3K	4.4K	338	4.5K	3.3K	1.3K	700
tiffinfo	3h	-44.7%	6.8K	53K	241	1.6M	44K	83K	2.1M	2.0M	1.3M	1.6M	31K	74K	4.7K
	7h	-60%	16K	96K	240	1.6M	70K	133K	2.1M	2.1M	1.4M	1.7M	48K	88K	10K
	10h	-53.8%	20K	105K	222	1.6M	85K	160K	2.1M	2.1M	1.4M	1.6M	58K	93K	13K
jasper	3h	88.6%	3.3K	632K	27	115K	1.4M	1.2M	101K	160K	156K	128K	29K	90K	350K
	7h	58.1%	18K	1.1M	27	182K	1.5M	1.5M	158K	259K	246K	208K	43K	90K	914K
	10h	62%	19K	1.2M	27	218K	1.5M	1.5M	191K	313K	296K	221K	50K	93K	1.2M
transicc	3h	62.1%	7.2K	57K	1.5K	1.6M	832K	328K	1.3M	852K	1.6M	1.6M	19K	53K	38K
	7h	40%	15K	67K	1.7K	1.4M	1.1M	326K	1.3M	1.4M	1.5M	1.5M	25K	124K	73K
	10h	37.9%	18K	70K	1.9K	1.4M	1.1M	325K	1.3M	1.4M	1.5M	1.5M	29K	120K	82K
flvmeta	3h	78.1%	4.6K	192K	264	1.4M	104K	161K	677K	1.9M	441K	1.5M	21K	141K	2.9M
	7h	81.9%	5.8K	646K	272	1.4M	163K	264K	873K	1.9M	735K	1.4M	32K	62K	4.5M
	10h	83.7%	6.2K	673K	262	1.4M	199K	298K	975K	1.9M	897K	1.4M	38K	95K	4.9M

we get multiple simple paths or isolated vertices, which can be represented in a matching M . Then assume the opposite of what we want to prove, that is, there is no maximum matching that can be converted to P_m . According to the lemma, we have a matching M that can be converted to P_m . Ntaofs’s work [60] gives a theorem that provides a largest incomparable vertex set $|I_m| = |V| - |M_m|$. Thus, we cannot get a largest incomparable vertex set I_m via matchings, which contradicts the assertion of $|I_m| = |P_m|$. Therefore, there must be a maximum matching M_m corresponding to P_m .

Appendix D. Symbolic Environment for Benchmarks

The KLEE symbolic environment settings in our evaluations are listed in Table 6.

Appendix E. Evaluation of Execution States

The evaluation results of the number of execution states are listed in Table 7.

Appendix F.

Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

This paper introduces Empec, a symbolic execution approach to address path explosion by modeling path selection as a minimum path cover problem on the program's interprocedural control flow graph. When a path in the cover is deemed infeasible, Empec dynamically adjusts and selects alternate paths. Implemented using KLEE, Empec boosts coverage by about 20% (basic blocks) and 24.4% (source lines), while reducing memory and symbolic state usage by up to 93.5% and 88.6%, respectively.

F.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

F.3. Reasons for Acceptance

- 1) The Program Committee appreciated the use of Path Cover to push forward the state of path prioritization, as this provides a good theoretical grounding compared to current techniques, which tend to rely on heuristics.
- 2) The paper's commitment to open science and the release of a prototype was viewed as a significant contribution.