Journal of Software Engineering and Applications, 2026, 19(3), 55-72
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

SymPcNSGA-Testing: A Hybrid Approach to Mitigate Path Explosion in Software Programs

Xaveria Djam Youh Kimbi, Sandra Maïla Modjeu Sipewa, Levinne Clemence Djeumeni Djombissie, Flavie Davila Ndemafo Nkenang

Department of Computer Science, Faculty of Science, University of Yaoundé 1, Yaoundé, Cameroon
Email: xaviera.kimbi@facsciences-uy1.cm, sipewasandra@gmail.com

How to cite this paper: Djam Youh Kimbi, X., Modjeu Sipewa, S.M., Djeumeni Djombissie, L.C. and Ndemafo Nkenang, F.D. (2026) SymPcNSGA-Testing: A Hybrid Approach to Mitigate Path Explosion in Software Programs. *Journal of Software Engineering and Applications*, **19**, 55-72.
https://doi.org/10.4236/jsea.2026.193004

Received: August 27, 2025
Accepted: March 13, 2026
Published: March 16, 2026

Abstract

The path explosion problem poses a significant barrier in the domain of software testing, making it nearly impossible to exhaustively explore all execution paths in large or complex software. Despite the extensive use of symbolic execution in the literature, the issue of path explosion remains largely unresolved. To address this limitation, we propose SymPcNSGA-Testing (Symbolic execution, Path Clustering, and Non-dominated Sorting Genetic Algorithm-II Testing), a hybrid methodology combining symbolic execution, path clustering, and multi-objective optimization using NSGA-II (Non-dominated Sorting Genetic Algorithm-II). Our approach aims to select a reduced yet representative set of execution paths that ensures maximum branch coverage while minimizing redundancy. SymPcNSGA-Testing operates in three stages: 1) symbolic execution with KLEE to explore the path space, 2) clustering of paths using the Ktest-cluster algorithm to group similar execution behaviors, and 3) multi-objective path optimization using NSGA-II, which selects representative paths that balance high branch coverage and minimal path set size. We evaluated our methodology on ten open-source programs from the Coreutils 9.5 package and compared it against several KLEE exploration strategies (DFS, BFS, NURS, Merging, and Covering-New). The results demonstrate that SymPcNSGA-Testing outperforms some of KLEE's strategies by achieving high branch coverage while effectively mitigating the path explosion problem. This study highlights the benefits of combining symbolic analysis and evolutionary multi-objective optimization to enhance test efficiency in complex software systems.

Keywords

Multi-Objective Optimization, NSGA-II, Path Clustering, Path Explosion, Symbolic Execution

1. Introduction

The path explosion problem poses a significant barrier in the domain of software testing, making it nearly impossible to exhaustively explore all execution paths in large or complex software. Many techniques have been used, such as Symbolic Execution, which explores multiple execution paths by interpreting program inputs as symbolic variables and resolving constraints using SMT solvers. This technique has been adopted in several widely used tools, such as KLEE, due to its ability to systematically and automatically uncover hard-to-find execution paths [1]. However, as software systems grow in complexity, so does the number of potential execution paths, which makes exhaustive exploration increasingly impractical.

One of the main obstacles encountered in symbolic execution is the path explosion phenomenon, in which the number of possible execution paths grows exponentially as the complexity of the program increases. This explosion renders full exploration infeasible, even for relatively small programs, and limits the scalability and effectiveness of symbolic analysis. As a result, some tools find it difficult to achieve high branch coverage efficiently, both in terms of time and computational resources. This leads us to a critical research question: How can we efficiently reduce the number of paths to be analyzed while preserving maximum branch coverage?

Many strategies have been proposed to mitigate path explosion. Tools such as KLEE implement search heuristics like Depth-First Search (DFS), Breadth-First Search (BFS), and various NURS strategies to guide path exploration. Other techniques include state merging, where similar program states are combined to reduce redundancy [2], and state pruning, which discards states deemed less promising [2]. Meanwhile, Search-Based Software Testing (SBST) introduces metaheuristics like genetic algorithms to explore input domains and maximize coverage [3]. More recent approaches include machine learning-guided exploration [4] and concolic testing [2].

While these techniques provide valuable insights, they each have significant limitations. Heuristic-based search strategies may suffer from local optima, failing to explore diverse paths. State merging and pruning risk losing critical execution behaviors. Search-Based Software Testing techniques, like genetic algorithms, although powerful in searching input spaces, often lack structural awareness of symbolic paths and do not exploit similarities among them. Furthermore, few existing works systematically integrate path similarity reduction prior to optimization, resulting in redundancy and suboptimal selection of test paths.

To address these shortcomings, we introduce SymPcNSGA-Testing (Symbolic execution, Path clustering and NSGA-II Testing), a novel hybrid methodology for mitigating path explosion in symbolic execution. Our approach consists of three phases:

1) Symbolic execution using KLEE to generate a diverse set of execution paths;

2) Clustering of symbolic paths using the Ktest-cluster algorithm to group structurally similar paths;

3) Multi-objective optimization via NSGA-II to select a minimal, representative subset of paths that ensures maximum branch coverage.

The key research questions we explore are:

- **RQ1**: How can the combination of symbolic execution, a path clustering strategy, and multi-objective evolutionary algorithms improve the efficiency of software testing in the face of path explosion?
- **RQ2**: How effective is SymPcNSGA-Testing in achieving maximum branch coverage while mitigating path explosion?

2. Related Work

The path explosion problem is one of the most critical challenges in symbolic execution and search-based software testing. A wide range of techniques has been proposed to address it, including parallelization, heuristics, clustering, and evolutionary multi-objective algorithms. This section reviews existing approaches and highlights their contributions and limitations in relation to our proposed SymPcNSGA-Testing methodology.

2.1. Parallel and Distributed Symbolic Execution

Several works have focused on parallelizing symbolic execution to reduce exploration time. Staats and Păsăreanu [5] introduced Simple Static Partitioning, a method that partitions the symbolic execution tree using preconditions and assigns partitions to different processors. This method achieved significant speedups (up to 90×), but its static nature may leave relevant paths unexplored.

Cloud9 [6], proposed by Bucur *et al.*, tackles path explosion by distributing symbolic execution over clusters of commodity machines. Cloud9 offers linear scalability and integrates well with existing test suites. However, it does not reduce the number of paths explored, nor does it include any strategy to eliminate redundant or similar paths.

2.2. Heuristics and Search-Guided Execution

Heuristics have been introduced to guide symbolic execution toward unexplored or critical paths. van Vliet [7] extended the OOX verification engine with several heuristics, including minimal-distance-to-uncovered, random-path, and round-robin combinations. While they improve coverage in some cases, these heuristics do not eliminate redundant paths after execution.

Xiao *et al.* [8] addressed path explosion using heuristic learning, search strategies, and function abstraction to improve vulnerability discovery. Yet, the absence of empirical comparisons and a lack of path reduction mechanisms limit the scope of their contributions.

Zhang *et al.* [9] proposed MuSE (Multiplex Symbolic Execution), a technique that integrates constraint solving directly into symbolic path exploration, enabling concurrent exploration of multiple paths. MuSE achieved significant speedups across multiple solvers. Nonetheless, it does not reduce the total number of paths or apply

selection heuristics such as genetic algorithms.

Bessler *et al.* [10] introduced Metrinome, a tool that statically estimates the potential severity of path explosion based on structural complexity metrics (e.g., cyclomatic complexity, loop depth). Although useful for planning symbolic analysis, Metrinome does not propose any execution or optimization mechanism to reduce the number of generated paths.

2.3. Evolutionary and Feedback-Based Approaches

Genetic algorithms have been employed in several works to improve test generation. Gong *et al.* [11] proposed ATCG-PC, a multi-objective genetic algorithm where paths are grouped and optimized in parallel using subpopulations. While this improves coverage, no selection mechanism is applied to extract a minimal subset of relevant paths.

Zhu *et al.* [12] improved test generation by grouping paths based on similarity, identifying common prefix paths, and guiding symbolic execution with reduced constraints. This results in performance gains but still does not explicitly minimize the number of selected paths for testing.

Semujju *et al.* [13] introduced a feedback-driven approach that removes path groups from the search space if they show no improvement across generations. This mechanism prevents wasting search efforts on unreachable paths. However, this strategy may inadvertently drop critical paths if the dropout condition is too strict.

2.4. State Reduction Techniques

State merging and pruning are also explored to mitigate path explosion. Copeland [2] combined dynamic state merging and pruning in a KLEE prototype, outperforming the baseline on several Coreutils programs. However, merging can complicate constraint tracking and does not address test selection or redundancy reduction.

3. Motivating Example

Let consider the following **Figure 1**, which represents the C code of a program and its Control Flow Graph. It contains 101 independent conditions, each controlling the increment of a counter if a specific value is met. The critical state (ERROR) is reached only if all 100 conditions are met simultaneously. Each $if(inp[i] == i + 1)$ condition generates two possible branches: either the condition is true or it is false. Given that there are 101 independent conditions, the total number of execution paths is: $\boldsymbol{Number\ of\ paths} = 2^{101} \approx 2.53 \times 10^{30}$. The critical state (*ERROR*) is reached only if all conditions are met—in other words, only 1 path out of 2^{101} leads to the error. This makes the probability of reaching this path by chance almost zero and reinforces the importance of optimization or path selection methods. This image clearly highlights the problem of path explosion encountered during software testing.

```c
#include <stdio.h>

int main() {
    int inp[100];
    int cnt = 0;

    // Lecture des 100 entrées
    for (int i = 0; i < 100; i++) {
        scanf("%d", &inp[i]);
    }

    // 100 branches indépendantes
    if (inp[0] == 1) cnt++;
    if (inp[1] == 2) cnt++;
    if (inp[2] == 3) cnt++;
    if (inp[3] == 4) cnt++;
    if (inp[4] == 5) cnt++;
    if (inp[5] == 6) cnt++;
    if (inp[6] == 7) cnt++;
    if (inp[7] == 8) cnt++;
    if (inp[8] == 9) cnt++;
    if (inp[9] == 10) cnt++;
    // ...
    if (inp[99] == 100) cnt++

    // État critique
    if (cnt >= 100);

}
```

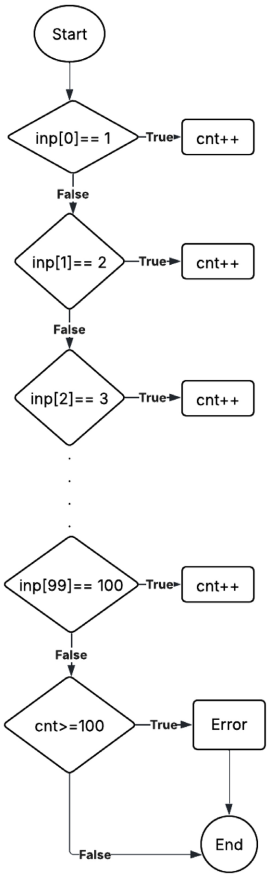

Figure 1. Motivating example of path explosion problem.

4. Problem Formulation

The path explosion problem in testing leads to a combinatorial explosion of execution paths, making exhaustive analysis intractable. The goal of SymPcNSGA-Testing is to mitigate this explosion while ensuring representative and diverse test coverage. To this end, we formulate the selection of execution paths as a multi-objective optimization problem addressed using NSGA-II.

4.1. Aim

Given a set of execution paths $P=\{p_1,p_2,\cdots,p_n\}$ partitioned into K clusters $C=\{C_1,C_2,\cdots,C_K\}$ obtained using a clustering algorithm, we aim to construct a subset $I\subseteq P$ (an individual) such that:

- At least one path is selected from each cluster: $\forall C_k\in C$, $\exists p\in I\cap C_k$.
- The selected subset I maximizes branch coverage across all paths it contains.
- The number of selected paths is minimized to reduce redundancy.

4.2. Decision Variables

Each individual $I=\{p_{i_1},p_{i_2},\cdots,p_{i_m}\}$ is a combination of execution paths selected from the original path set P. The size m of each individual is variable but must satisfy the coverage constraint for all clusters.

4.3. Objectives Functions

We define the problem as a bi-objective minimization problem:

Objective 1 (Maximize Coverage):

$$f_1(I) = \max_{p \in I} \left(\frac{\text{Number of covered branches in } p}{\text{Total number of branches}} \right) \tag{1}$$

Objective 2 (Minimize Path Set Size):

$$f_2(I) = |I| \tag{2}$$

Objective 1 aims to maximize branch coverage. Although NSGA-II is traditionally formulated for minimization problems, the algorithm is used here in a configuration that directly supports maximization objectives. Consequently, individuals exhibiting higher coverage values are systematically favored during the selection process.

4.4. Constraints

To ensure representativity across all clusters, we introduce the constraint:

$$\forall C_k \in C,\ \exists p \in I \cap C_k \tag{3}$$

This guarantees that each individual includes at least one path from every cluster, promoting diversity and representativity.

4.5. Optimization Strategy

The NSGA-II algorithm is applied to evolve a population of such individuals, balancing the trade-off between maximizing branch coverage and minimizing the number of selected paths. The non-dominated sorting and crowding distance mechanisms of NSGA-II ensure the preservation of diversity and the convergence toward a Pareto-optimal front of path subsets.

5. Methodology

In this section, we describe the proposed SymPcNSGA-Testing methodology, which aims to mitigate the path explosion problem while maximizing branch coverage during test phase. This combined methodology brings together symbolic execution, clustering of execution paths, and optimization using genetic algorithms in a structured and complementary sequence of steps. The entire process comprises three main phases, which can be seen in **Figure 2**.

5.1. Symbolic Execution with KLEE

The first step of the SymPcNSGA-Testing methodology is based on the use of symbolic execution using the KLEE 3.2 tool. KLEE 3.2 is an open-source symbolic execution engine that automatically generates test cases by exploring all possible execution paths of a program, based on symbolic inputs. During this phase, the program's inputs are rendered symbolic so that KLEE can follow all conditional branches encountered. At each conditional branch, KLEE generates a new path based on the accumulated constraints.

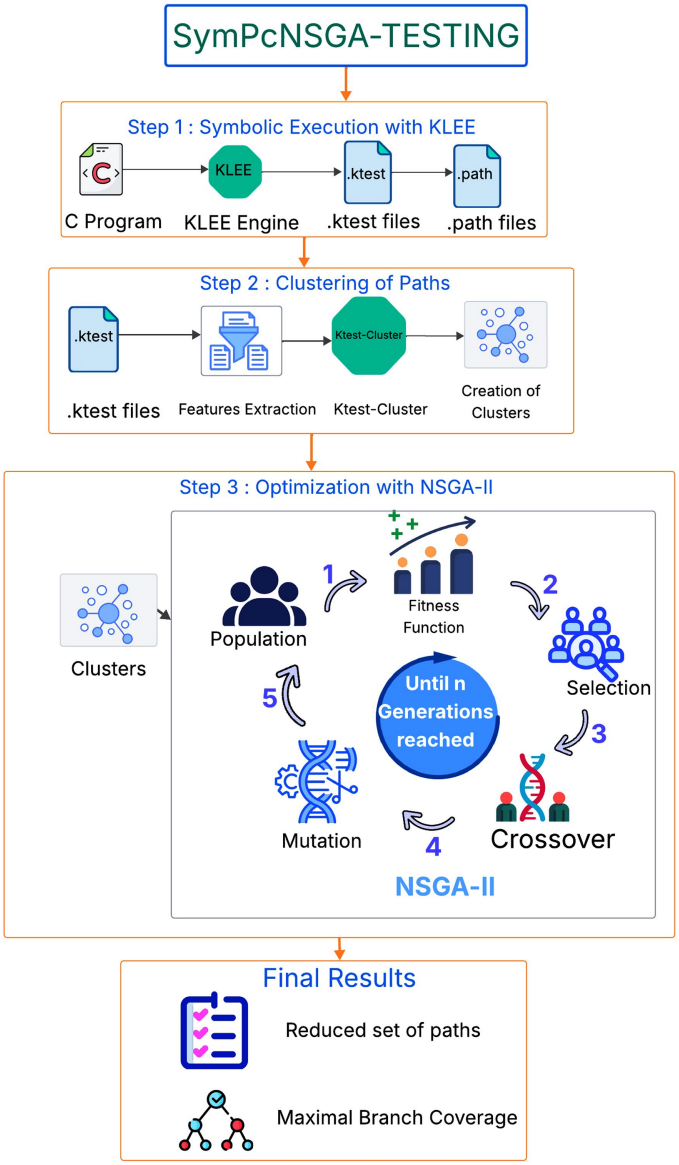

Figure 2. SymPcNSGA-testing architecture.

For each path explored, KLEE generates a .ktest file, which contains the concrete values of the inputs used to follow that specific path. It also generates a .path file representing the decision sequence taken through branches. These two types of generated files will be essential in the next steps of our methodology. The process can be described by the following algorithm (**Algorithm 1**).

5.2. Path Clustering with Ktest-Cluster

The second step of our methodology aims to group execution paths into sets to reduce redundancy, facilitate selection, and limit the impact of combinatorial explosion. To this end, we propose KTest-Cluster (K, the number of cluster, that we will produce), an innovative clustering algorithm based not on symbolic paths or executed branches, but directly on the .ktest files produced by KLEE.

Algorithm 1. Symbolic execution with KLEE.

Input: C program P to be analyzed
Output: `.ktest` and `.path` files for each execution path P_i
Function ***SymbolicExecutionWithKLEE***(P)**:**
1: Compile P to LLVM bytecode:
`clang -emit-llvm -g -c` P `-o` P`.bc`;
2: Execute KLEE on the compiled bytecode:
`klee` P`.bc`;
3: **foreach** *execution path* P_i *generated by KLEE* **do**
4: Extract the symbolic constraints (path conditions) ;
5: Generate concrete inputs satisfying the constraints ;
6: Save the inputs in P_i`.ktest` file ;
7: Save the path trace in P_i`.path` file ;
8: Return the set of `.ktest` and `.path` files

Unlike traditional approaches that analyze paths from concrete values generated, KTest-Cluster leverages input values contained in .ktest files, which represent the test cases associated with each execution.

Step 1: Data Preparation

Each .ktest file contains the concrete inputs associated with a given path. These files are first processed to extract the input data in a structured manner using KLEE's ktest-tool. Each .ktest file is then transformed into a numerical vector, where each component represents an input variable.

Step 2: Vector Normalization

Before clustering, the vectors are normalized. The vectors are padded with zeros at the end so that they all have the same maximum length. This is to prevent certain large dimensions from overwhelming others in the distance calculation.

Step 3: Applying K-means with Euclidean Distance

The normalized vectors are then clustered using the K-means algorithm, using Euclidean distance as the similarity measure. In this work, K-means clustering with Euclidean distance was selected due to the numerical and fixed-length representation of execution paths, where each path is encoded as a vector of branch coverage features and execution characteristics. Euclidean distance provides an intuitive measure of similarity in this vector space by capturing the overall geometric proximity between paths in terms of covered branches and execution. K-means was chosen for its computational efficiency and scalability, which is particularly important given the large number of paths generated by symbolic execution.

Alternative distance metrics (e.g., cosine or Jaccard distance) and clustering algorithms (e.g., hierarchical or density-based clustering) could also be considered. However, these approaches either introduce higher computational costs or require additional parameters that are difficult to tune in large-scale symbolic execution settings. In this study, the choice of K (the number of clusters) is based on the Rule of thumb [14] technique in which $k = \sqrt{\frac{N}{2}}$ where N is the number of data points. In our case, N is the number of total path executions produced by Symbolic Execution. The result of this process is a partitioning of the paths into coherent groups of similar input behaviors. A systematic comparison of clustering techniques is left as future work (**Algorithm 2**).

Algorithm 2. Ktest-Cluster algorithm.

Input: Set of execution paths $\mathcal{P}$ represented by `.ktest` files,
number of clusters k
Output: Set of clusters $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ each containing a group of similar paths
Function ***Clustering***$(\mathcal{P}, k)$**:**
1: Initialize an empty list $\mathcal{D}$ to store feature vectors x ;
2: **foreach** *execution path* $P_j.ktest$ *in* $\mathcal{P}$ **do**
3: Extract data from the file associated with $P_j.ktest$ using `ktest-tool` ;
4: Convert the extracted data into a feature vector x ;
5: Add the vector x to $\mathcal{D}$;
6: Uniformize the vectors in $\mathcal{D}$ (pad with zeros) ;
7: Randomly select k vectors from $\mathcal{D}$ as initial centroids $C_1, C_2, \ldots, C_k$;
8: Set the iteration count `iter` to 0 ;
9: Initialize a variable `convergence_flag` to `False` ;
10:**while** `convergence_flag` *is* `False` ***and*** `iter` < 15 **do**
11: Initialize empty clusters $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$;
12: **foreach** *vector* x *in* $\mathcal{D}$ **do**
13: Compute the Euclidean distance between x and each centroid C_j ;
14: Assign x to the closest centroid C_j ;
15: Store the current cluster assignments ;
16: **foreach** *cluster* $\mathcal{C}_j$ **do**
17: Recompute centroid C_j as the mean of all vectors $x \in \mathcal{C}_j$;
18:Compare the new cluster assignments with the previous ones from the previous iteration;
19: **if** *the assignment has not changed* **then**
20: Set `convergence_flag` to `True` ;
21: Increment the iteration count ;
22: Return a mapping of each path $P_j.ktest \in \mathcal{P}$ to its assigned cluster label ;

5.3. Optimization with Non-Dominated Sorting Genetic Algorithm-II (NSGA-II)

The final phase of the *SymPcNSGA-Testing* methodology focuses on reducing path explosion through a multi-objective optimization using NSGA-II (Non-dominated Sorting Genetic Algorithm-II). This step seeks to balance two conflicting goals: 1) maximize branch coverage and 2) minimize the number of selected execution paths. Each individual in the population represents a subset of symbolic paths and is evaluated by a fitness pair $\left(f_1(I), f_2(I)\right)$ where:

- $f_1(I)$ denotes the *branch coverage ratio*;
- $f_2(I)$ denotes the *number of paths* in the subset.

Branch coverage was computed using KLEE's branch coverage instrumentation, where each conditional branch is represented as a binary value indicating whether it was exercised by at least one execution path, and coverage is reported at the source-level branch granularity as the percentage of covered branches over the total number of branches.

NSGA-II applies non-dominated sorting to classify individuals into Pareto fronts. An individual I_1 is said to dominate another I_2 (denoted $I_1 \prec I_2$) if:

$$f_1(I_1) \geq f_1(I_2) \text{ and } f_2(I_1) \leq f_2(I_2),$$

with at least one strict inequality. This mechanism identifies optimal trade-offs between coverage and cost.

Evolutionary operators such as *crossover* and *mutation* are applied to ensure that each individual maintains at least one path per cluster, thus preserving structural diversity.

The algorithm iterates over a fixed number of generations, gradually improving the population. At the end of the process, the first Pareto front offers a set of non-dominated solutions representing various trade-offs. These allow testers to select configurations with high coverage, minimal redundancy, or a balanced compromise, depending on testing goals and resources (Algorithm 3).

Algorithm 3. SymPcNSGA-II optimization.

Input: Set of execution paths $\mathcal{P}$ with cluster labels, total number of generations G, population size N
Output: Pareto front of non-dominated individuals
Function ***NSGA2_Optimization***$(\mathcal{P}, G, N)$**:**
1: Initialize the population $\mathcal{P}_0$ with N individuals such that each individual contains at least one path from each cluster;
2: **foreach** *individual* $I \in \mathcal{P}_0$ **do**
3: Evaluate fitness values:
$f_1(I) = \max_{p \in I} \left(\frac{\text{Number of covered branches in } p}{\text{Total number of branches}} \right)$;
$f_2(I) = |I|$
4: **for** *generation* $g = 1$ **to** G **do**
5: Apply non-dominated sorting to current population $\mathcal{P}_{g-1}$, obtain fronts $F_1, F_2, \ldots$;
6: Compute crowding distance for each individual in each front;
7: Select mating pool from top fronts using binary tournament based on Pareto rank and crowding distance;
8: Select two parents (I_1, I_2) via binary tournament based on Pareto rank 9: Select a cut point $i \in [1, \min(|I_1|, |I_2|) - 1]$;
10: Apply single-point crossover to produce child individual:
$I_{\text{child}} = \text{EnsureValid}(I_1[:i] \cup I_2[i:])$;
11: Apply mutation to I_{child}:
a) Randomly select a cluster C_j;
b) Replace a path from C_j in I_{child} by another path from C_j;
c) Ensure representativity: $\forall C_i, \exists P_i \in I_{\text{child}} \cap C_i$;
12: Evaluate fitness:
$f_1(I_{\text{child}}) = \max_{p \in I_{\text{child}}} \left(\frac{\text{Covered branches in } p}{\text{Total number of branches}} \right)$;
$f_2(I_{\text{child}}) = |I_{\text{child}}|$;
13: Form new population $\mathcal{P}_g$ from parents and offspring;
14: Return final Pareto front $F_1 \subseteq \mathcal{I}_G$;

6. Experimental Setup

To evaluate the effectiveness of our proposed methodology, **SymPcNSGA-Testing**, we conducted a series of experiments on 10 real-world programs from the GNU Coreutils 9.5 suite. This section presents the experimental environment, datasets used, and parameter settings.

6.1. Environment Configuration

All experiments were conducted on a machine running Linux Mint 21 with the following hardware specifications:

- **CPU:** Intel® Core™ i5-11th generation @ 2.90 GHz (8 cores).
- **RAM:** 16 GB.
- **Storage:** HDD 1 TB.
- **KLEE Version:** 3.2 with LLVM 14.0.0, Clang 14.0.0, GCC 11.4.0.
- **Python Version:** 3.10.
- **Scikit-learn Version:** 1.2.0.
- **Clustering:** Implemented via Scikit-learn (K-means).
- **Genetic Algorithm:** Implemented in Python.

6.2. Programs under Test

To evaluate the **SymPcNSGA-Testing** methodology, we selected 10 diverse programs from GNU Coreutils 9.5: basename, cut, csplit, dirname, echo, expr, printf, sort, tr, uniq. These programs present a variety of input/output behaviors and logical complexities.

Each program was executed **20 times**, with a single execution taking approximately **2 hours**, totaling:

20 runs × 10 programs × 2 hours = 400 **hours of experimentation**

6.3. Methodology Parameters

This subsection presents the parameters used at each stage of the SymPcNSGA-Testing methodology. Each parameter is crucial for optimizing the performance of the genetic algorithm, clustering, and symbolic execution.

6.3.1. Symbolic Execution Parameters

The following Table 1 presents the different parameters that we have used for Symbolic Execution.

Table 1. Symbolic execution parameters.

Parameter	Description
--libc=uclibc	Uses the minimal uClibc library for standard C functions
--posix-runtime	Enables POSIX-like environment simulation for command-line arguments, stdin, and file I/O
--sym-arg N	Declares one symbolic argument of up to N characters

6.3.2. Clustering Parameter

The following Table 2 presents the description of the parameter ***k*** that is essential for clustering step.

Table 2. Clustering parameter.

Parameter	Description	Value Used
Number of clusters k	Specifies the number of groups formed from symbolic execution paths	$k = \sqrt{\frac{N}{2}}$, where N is the number of paths

6.3.3. NSGA-II Parameters

The following Table 3 presents the different parameters that we have used for running NSGA-II algorithm.

Table 3. NSGA-II parameters.

Parameter	Description	Value
Population size	Number of individuals per generation	50

Continued

Number of generations	Number of evolutionary iterations	100
Selection method	Based on Pareto ranking and diversity	Binary Tournament
Crossover type	Recombination of individuals	One-Point Crossover
Crossover rate	Probability of crossover per pair	100%
Mutation type	Replacement of a path with another from the same cluster	Intra-cluster replacement
Mutation rate	Probability of mutation per individual	10%
Termination criteria	Stopping condition	100 Generations
Individual size	Variable size; each individual includes at least one path per cluster	$\geq k$

The NSGA-II algorithm was configured with a population size of 50 individuals and executed for 100 generations. Each individual represents a variable-length combination of execution paths, with the constraint that it must be greater than and at least one path is selected from each cluster to ensure representativeness.

One-point crossover operator was used with a probability of Pc = 100%, enabling the recombination of path subsets between individuals. Mutation was applied with probability Pm = 10%, where mutation consists of randomly adding, removing, or replacing a path within a cluster. These operators were selected to preserve cluster coverage while maintaining population diversity.

Parameter values were chosen empirically based on preliminary experiments and commonly adopted settings in SBST literature, providing a trade-off between convergence speed and solution diversity.

6.4. Comparative Tools

The following Table 4 lists the different KLEE configurations used for comparison with our proposed methodology. Each configuration aims to tackle the path explosion problem using various strategies.

Table 4. Comparative KLEE techniques.

No.	Technique	Purpose	KLEE Parameters
0	KLEE-BASE	Standard configuration with symbolic execution using POSIX system calls	--posix-runtime --libc=uclibc
1	Covering-New	Reduces redundant outputs by generating tests only for new code coverage	--only-output-states-covering-new
2	NURS-Depth	Explores longer execution paths first to reach deep branches	--search=nurs:depth

Continued

3	BFS + Merge	Combines breadth-first search with state merging and new coverage focus	--search=bfs – use-merge --only-output-states-covering-new
4	Random-Path	Introduces randomness in path selection for diversity	--search=random-path
5	DFS	Depth-first search explores long paths first, may ignore breadth	--search=dfs
6	State-Merge	Merges execution states to limit the number of paths explored	--use-merge
7	Covering-New + Merge	Enhances coverage while merging redundant states	--only-output-states-covering-new --use-merge
8	MD2U	Prioritizes paths closer to uncovered instructions to guide search	--search=nurs:md2u --only-output-states-covering-new
9	NURS CovNew + DFS	Combines depth-first and coverage-guided heuristics	--search=nurs:covnew --search=dfs

7. Results and Discussion

7.1. Answer to RQ1

The first research question (RQ1) aims to analyze the contribution of our SymPcNSGA-Testing methodology compared to different KLEE techniques in the context of generating execution paths. More specifically, we seek to evaluate whether SymPcNSGA-Testing better manages the combinatorial explosion of paths in the tested program.

The analysis of our experimental results, illustrated in Table 5, shows that **SymPcNSGA-Testing** systematically generates fewer paths than **KLEE-BASE** and its different advanced techniques. This reduction in the number of paths demonstrates the ability of our approach to control path explosion, a major challenge in symbolic execution.

Table 5. Number of paths generated by each KLEE input and by SymPcNSGA-Testing.

Programs	KLEE Input										Our Approach
	KLEE-BASE	A	B	C	D	E	F	G	H	I	SymPcNSGA-Testing
basename	110	27	34	23	27	29	30	30	30	33	**39**
csplit	5306	45	48	39	44	46	47	48	47	51	**64**
cut	1244	56	69	51	57	55	59	57	68	68	**62**
dirname	2314	22	34	22	22	21	23	23	24	34	66
echo	3363	7	13	9	10	10	11	11	11	31	**68**
expr	1375	25	37	23	25	25	25	25	27	35	**69**
printf	10593	133	129	96	115	109	129	129	120	133	**22**
sort	6186	75	77	61	64	59	86	85	92	87	**62**

Continued

tr	5286	36	42	31	36	40	39	39	42	44	**50**
uniq	3530	5	7	5	10	5	5	5	5	6	**68**

A: Covering-New; B: Covering-New + NURS-Depth; C: Covering-New + BFS + Merge; D: Covering-New + Random; E: Covering-New + DFS; F: Covering-New + Merge; G: Covering-New + NURS-CovNew + Merge; H: Covering-New + NURS-MD2U; I: Covering-New + NURS-CovNew + DFS.

Among the ten tested programs, **SymPcNSGA-Testing** produces fewer paths than KLEE-BASE in all cases:

- basename: 39 paths vs 110 (KLEE-BASE).
- csplit: 64 paths vs 5306 (KLEE-BASE).
- cut: 62 paths vs 1244 (KLEE-BASE).
- dirname: 66 paths vs 2314 (KLEE-BASE).
- echo: 68 paths vs 3363 (KLEE-BASE).
- expr: 69 paths vs 1375 (KLEE-BASE).
- printf: 22 paths vs 10593 (KLEE-BASE).
- sort: 60 paths vs 6186 (KLEE-BASE).
- tr: 50 paths vs 5286 (KLEE-BASE).
- uniq: 68 paths vs 3580 (KLEE-BASE).

Comparison with Advanced KLEE Strategies

In most cases, **SymPcNSGA-Testing** generates fewer paths than advanced KLEE techniques using specific strategies such as NURS, BFS, DFS, random-path, and use-merge.

For example:

- On expr, SymPcNSGA-Testing generates 69 paths while Covering-New + NURS-Depth produces 37 and Covering-New + DFS generates 35. SymPcNSGA-Testing is more efficient in reducing the number of paths.
- On sort, SymPcNSGA-Testing generates 60 paths, fewer than most techniques except Covering-New + DFS (59 paths).
- On all the programs, Sampans-Testing outperforms KLEE-BASE.

Case Study: basename—Drastic Reduction of Path Explosion

- **KLEE-BASE:** 110 paths.
- **Best advanced KLEE technique:** Covering-New + BFS + Merge = 23 paths.
- **SymPcNSGA-Testing**: 39 paths.

Interpretation: basename processes text inputs and includes multiple branches based on arguments, leading to significant path growth. SymPcNSGA-Testing eliminates redundant paths while maintaining sufficient coverage. This illustrates the effectiveness of clustering and genetic algorithms.

Case Study: cut—Drastic Reduction of Path Explosion

- **KLEE-BASE:** 1244 paths.
- **Advanced KLEE strategies:** All above 60 paths.
- **SymPcNSGA-Testing:** 62 paths.

Interpretation: cut manipulates structured inputs and contains many conditions.

SymPcNSGA-Testing reduces the number of execution paths by over 95%, while maintaining branch representativity.

Case Study: printf—Exceptional Path Reduction

- **KLEE-BASE:** 10,593 paths.
- **Advanced KLEE options:** All greater than 100 paths.
- **SymPcNSGA-Testing:** 22 paths.

Interpretation: printf includes multiple formats and features, leading to exponential path explosion. Our approach reduces the number of paths by more than 99% while retaining the essential test cases. This confirms its suitability for highly combinatorial programs.

These results confirm that **SymPcNSGA-Testing** is a robust and effective approach for mitigating the path explosion problem in symbolic execution while maintaining sufficient test coverage.

7.2. Answer to RQ2

The second research question (RQ2) aims to evaluate whether reducing the number of paths tested compromises the quality of the generated tests and, consequently, the reliability of the tested software. The goal is to examine whether the SymPcNSGA-Testing methodology maintains sufficient test coverage while optimizing path exploration to limit combinatorial explosion.

Among the ten tested programs, **SymPcNSGA-Testing** achieves branch coverage rates comparable to those obtained with **KLEE-BASE** and its different techniques. Table 6 presents these results. For example:

Table 6. Comparison of branch coverage (C) generated by KLEE inputs and SymPcNSGA-Testing for each program.

Program	Metric	KLEE-BASE	A	B	C	D	E	F	G	H	I	SymPcNSGA-Testing
basename	C (%)	73.19	72.96	73.19	72.96	72.96	72.97	72.96	72.96	72.96	73.19	**73.19**
csplit	C (%)	84.70	81.03	75.04	74.12	79.40	78.12	80.21	80.28	80.82	79.76	**84.28**
cut	C (%)	85.12	79.27	78.76	78.52	79.85	78.90	78.82	78.86	83.61	78.80	85.07
dirname	C (%)	73.03	81.03	75.04	74.12	79.40	78.12	80.21	80.28	80.82	79.76	**73.00**
echo	C (%)	70.05	71.50	59.87	74.91	56.14	59.85	59.80	59.80	59.80	67.46	**69.40**
expr	C (%)	75.40	75.28	75.28	75.28	75.28	74.92	74.92	75.24	75.28	75.25	**75.28**
printf	C (%)	76.36	74.01	71.86	77.13	73.41	77.71	71.86	71.87	71.92	71.86	**73.00**
sort	C (%)	92.09	79.63	89.65	80.48	75.30	76.92	70.53	70.59	70.19	75.37	**86.03**
tr	C (%)	84.93	76.53	79.64	74.18	76.99	80.55	79.47	79.42	80.36	79.32	**84.44**
uniq	C (%)	72.56	72.02	72.22	72.60	72.43	72.13	72.02	72.02	72.16	72.12	**72.56**

A: Covering-New; B: Covering-New + NURS-Depth; C: Covering-New + BFS + Merge; D: Covering-New + Random; E: Covering-New + DFS; F: Covering-New + Merge; G: Covering-New + NURS-CovNew + Merge; H: Covering-New + NURS-MD2U; I: Covering-New + NURS-CovNew + DFS.

- **csplit:** 84.28% for SymPcNSGA-Testing compared to 84.70% for KLEE-BASE, which achieves the best coverage. The difference is only 0.42%.
- **cut:** 85.07% for SymPcNSGA-Testing compared to 85.12% for KLEE-BASE. The difference is only 0.05%.
- **dirname:** 73.00% for SymPcNSGA-Testing compared to 73.03% for KLEE-BASE and 81.03% for Covering-New, which achieves the best coverage. Differences of 0.03% and 8.03% respectively.
- **echo:** 69.40% for SymPcNSGA-Testing compared to 70.05% for KLEE-BASE and 71.50% for Covering-New. Differences of 0.65% and 2.1%.
- **printf:** 73% for SymPcNSGA-Testing compared to 76.36% for KLEE-BASE and 77.71% for Covering-New + DFS. Differences of 3.36% and 4.71%.
- **sort:** 86.03% for SymPcNSGA-Testing compared to 92.09% for KLEE-BASE, which achieves the best coverage.

These results show that although SymPcNSGA-Testing explores far fewer paths than KLEE, the difference in branch coverage remains minimal.

Specific Cases

Program *uniq*:

- KLEE-BASE explores 3530 paths and achieves 72.56% coverage (best).
- Covering-New + BFS + Merge explores 5 paths and achieves 72.60% coverage.
- SymPcNSGA-Testing retains 68 paths and achieves 72.56% coverage.

Interpretation:

- 98.07% reduction in the number of tested paths.
- Minimal drop in coverage (−0.01%).
- SymPcNSGA-Testing achieves the same coverage as KLEE-BASE despite drastically reducing the number of paths. This confirms the effectiveness of SymPcNSGA-Testing in generating fewer, but more representative paths.

Program *basename*:

- KLEE-BASE explores 110 paths with 73.19% coverage.
- Covering-New + NURS-Depth and Covering-New + NURS-CovNew + DFS both achieve 73.19% coverage with only 34 and 33 paths, respectively.
- SymPcNSGA-Testing retains 39 paths and achieves 73.19% coverage.

Interpretation:

- 65% reduction in tested paths.
- Identical coverage rate compared to other KLEE techniques.
- This demonstrates that SymPcNSGA-Testing not only reduces the number of paths but also maintains high test quality through equivalent branch coverage.

Program *expr*:

- KLEE-BASE explores 1375 paths with 75.40% coverage (best).
- SymPcNSGA-Testing retains 69 paths and achieves 75.28% coverage.

Interpretation:

- 94.98% reduction in the number of paths tested.
- Almost the same coverage rate than advanced KLEE techniques.

Program *tr*:

- KLEE-BASE explores 5286 paths and achieves 84.93% coverage (best).

- SymPcNSGA-Testing retains 50 paths and achieves 84.44% coverage.

Interpretation:

- 99.05% reduction in tested paths.
- Minimal coverage drop (−0.49%).
- SymPcNSGA-Testing achieves high coverage while significantly reducing the number of paths.

8. Conclusions

Our methodology demonstrated a significant reduction in the number of paths analyzed, thus limiting the combinatorial explosion inherent in symbolic execution. The integration of the genetic algorithm optimized path selection by favoring those with a high impact on branch coverage. Moreover, SymPcNSGA-Testing proved more scalable than traditional approaches, especially for programs where KLEE quickly reaches memory saturation.

The experimental evaluation was conducted on the GNU Coreutils benchmark, which is widely adopted in the symbolic execution and software testing literature, ensuring both reproducibility and comparability of the results. While the evaluation relies on this standard experimental framework, the proposed methodology remains generic and independent of the specific characteristics of the analyzed programs.

These results highlight the potential of SymPcNSGA-Testing as an effective solution to the path explosion problem in symbolic execution. By combining clustering and evolutionary optimization, this methodology enables better management of the exploration space while ensuring high branch coverage. These advancements offer promising prospects for improving structural testing, making symbolic execution applicable to larger-scale programs.

Although the SymPcNSGA-Testing methodology has effectively mitigated the path explosion problem, several directions for improvement and extension can be considered to further optimize its performance and broaden its applicability. These include the integration of machine learning techniques, resource optimization, parallel execution, and evaluation on industrial-scale codebases.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Cadar, C., Dunbar, D., Engler, D.R., *et al.* (2008) KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. 8*th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, 8-10 December 2008, 209-224.

[2] Copeland, P.T. (2014) Using State Merging and State Pruning to Address the Path Explosion Problem Faced by Symbolic Execution. Master's Thesis, Air Force Institute of Technology.

[3] Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., *et al.* (2011)

Symbolic Search-Based Testing. 2011 26*th IEEE/ACM International Conference on Automated Software Engineering* (*ASE* 2011), Lawrence, 6-10 November 2011, 53-62. https://doi.org/10.1109/ase.2011.6100119

[4] He, J., Sivanrupan, G., Tsankov, P. and Vechev, M. (2021) Learning to Explore Paths for Symbolic Execution. *Proceedings of the* 2021 *ACM SIGSAC Conference on Computer and Communications Security*, 15-19 November 2021, 2526-2540. https://doi.org/10.1145/3460120.3484813

[5] Staats, M. and Păsăreanu, C. (2010) Parallel Symbolic Execution for Structural Test Generation. *Proceedings of the* 19*th International Symposium on Software Testing and Analysis*, Trento, 12-16 July 2010, 183-194. https://doi.org/10.1145/1831708.1831732

[6] Bucur, S., Ureche, V., Zamfir, C. and Candea, G. (2011) Parallel Symbolic Execution for Automated Real-World Software Testing. *Proceedings of the Sixth Conference on Computer Systems*, Salzburg, 10-13 April 2011, 183-198. https://doi.org/10.1145/1966445.1966463

[7] van Vliet, D. (2023) A Heuristic Approach to the Path Explosion Problem for Complex Heap Programs. Master's Thesis, Utrecht University.

[8] Xiao, X., Zhang, X. and Li, X. (2010) New Approach to Path Explosion Problem of Symbolic Execution. 2010 *First International Conference on Pervasive Computing, Signal Processing and Applications*, Harbin, 17-19 September 2010, 301-304. https://doi.org/10.1109/pcspa.2010.80

[9] Zhang, Y., Chen, Z., Shuai, Z., Zhang, T., Li, K. and Wang, J. (2020) Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. *Proceedings of the* 35*th IEEE/ACM International Conference on Automated Software Engineering*, 21-25 December 2020, 846-857. https://doi.org/10.1145/3324884.3416645

[10] Bessler, G., Cordova, J., Cullen-Baratloo, S., Dissem, S., Lu, E., Devin, S., *et al.* (2021) Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. 2021 *IEEE/ACM* 43*rd International Conference on Software Engineering: Companion Proceedings* (*ICSE-Companion*), Madrid, 25-28 May 2021, 29-32. https://doi.org/10.1109/icse-companion52605.2021.00028

[11] Gong, D., Zhang, W. and Yao, X. (2011) Evolutionary Generation of Test Data for Many Paths Coverage Based on Grouping. *Journal of Systems and Software*, **84**, 2222-2233. https://doi.org/10.1016/j.jss.2011.06.028

[12] Zhu, Z.M., Xu, X. and Jiao, L. (2017) Improved Evolutionary Generation of Test Data for Multiple Paths in Search-Based Software Testing. 2017 *IEEE Congress on Evolutionary Computation* (*CEC*), San Sebastian, 5-8 June 2017, 612-620. https://doi.org/10.1109/cec.2017.7969367

[13] Semujju, S.D., Huang, H., Liu, F., Xiang, Y. and Hao, Z. (2023) Search-Based Software Test Data Generation for Path Coverage Based on a Feedback-Directed Mechanism. *Complex System Modeling and Simulation*, **3**, 12-31. https://doi.org/10.23919/csms.2022.0027

[14] Hassan, A., Shah, W., Husein, A., Talib, M.S., Mohammed, A.A.J. and Iskandar, M. (2019) Clustering Approach in Wireless Sensor Networks Based on K-Means: Limitations and Recommendations. *International Journal of Recent Technology and Engineering*, **7**, 119-126.