Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Enhancing Symbolic Execution of Programs for Interprocedural Control Flow Path Feasibility Analysis

Hovhannes Movsisyan
Russian-Armenian University
Hripsime Hovhannisyan
Russian-Armenian University
Tigran Avagyan
Russian-Armenian University
Hayk Aslanyan
hayk.aslanyan@rau.am
Russian-Armenian University

Research Article

Keywords: path-sensitive analysis, directed symbolic execution, memory model, path feasibility, static analysis, interprocedural control flow graph

Posted Date: July 8th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10066367/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Enhancing Symbolic Execution of Programs for Interprocedural Control Flow Path Feasibility Analysis

Hovhannes Movsisyan[1], Hripsime Hovhannisyan[1], Tigran Avagyan[1], Hayk Aslanyan[1*]

[1*]Center of Advanced Software Technologies, Russian-Armenian University, Yerevan, 0051, Armenia.

*Corresponding author(s). E-mail(s): hayk.aslanyan@rau.am;

Abstract

This article presents a method for verifying the feasibility of program interprocedural control flow paths using directed symbolic execution. Because symbolic execution is a resource-demanding process, the method directs the execution by eliminating redundant program branches that are not related to the path being verified. To make symbolic execution applicable to path feasibility analysis, we present automatic symbolization approaches specifically supporting variables and allocated memory locations with unknown values, entry-point function parameters, dynamically linked functions' return values and pointer arguments, as well as pointer-referenced memory locations. To mitigate path explosion in loops, the method employs a loop-bounding strategy to limit the iteration count without terminating symbolic execution. For evaluation, we extracted execution paths from real-world projects, such as GNU Coreutils, and used them as a test suite. The proposed method achieved 95.3% recall on this dataset. Additionally, integration of the method into the static analysis tool MLH increased the tool's precision on the Juliet test suite from 56% to 100% while preserving the original recall rate. Comparative testing with the KLEEF symbolic execution engine on the Juliet test suite demonstrated the superior robustness of the proposed approach. In particular, when applied as a post-processing step for two static analyzers, Infer and Clang Static Analyzer, our method maintained or improved their baseline performance, whereas KLEEF reduced the corresponding F1-scores to 0 and 0.058, respectively.

Keywords: path-sensitive analysis, directed symbolic execution, memory model, path feasibility, static analysis, interprocedural control flow graph

1 Introduction

Ensuring the security and reliability of software systems is a fundamental challenge in modern software engineering. As the systems grow in complexity and size, there is an urgent need for automated software analysis techniques. Static analysis addresses this challenge by analyzing program code without executing it. While these methods are scalable, they often produce false positives. One of the typical representations employed by static analyzers is the control flow graph (CFG). In a CFG, nodes represent basic blocks and edges represent possible control transfers between them. When analysis extends across function boundaries — tracking how calls and returns connect individual procedure CFGs into a unified whole — the result is an interprocedural CFG (ICFG). This structure enables static analyzers to explore possible execution paths through a program.

To balance code analysis performance and the level of false positives, static analyzers must determine which paths in ICFGs are actually executable at runtime. The problem of determining whether a specific path in a program's control flow graph is executable is known as path feasibility analysis.

For instance, consider the source code example in Figure 1, and its CFG presented in Figure 2. A static analyzer might report a control flow path (BB1→BB2→BB4→BB5) that reaches the `vulnerable_function` call.

```
1  int main()
2  {
3      int x = rand();
4      int y = rand();
5
6      if (x <= y) {
7          x = y + 1;
8      } else if (x > y) {
9          x = y - 1;
10     }
11
12     if (x == y) {
13         vulnerable_function();
14     }
15
16     return 0;
17 }
```

Fig. 1 An infeasible path example.

However, the reported path is infeasible because the code's semantics prevent its execution. To filter out such false-positive results, additional path feasibility analysis needs to be performed.

We propose a symbolic-execution-based (King, 1976) method to determine the feasibility of any path in a given set of interprocedural control flow graph (ICFG) paths

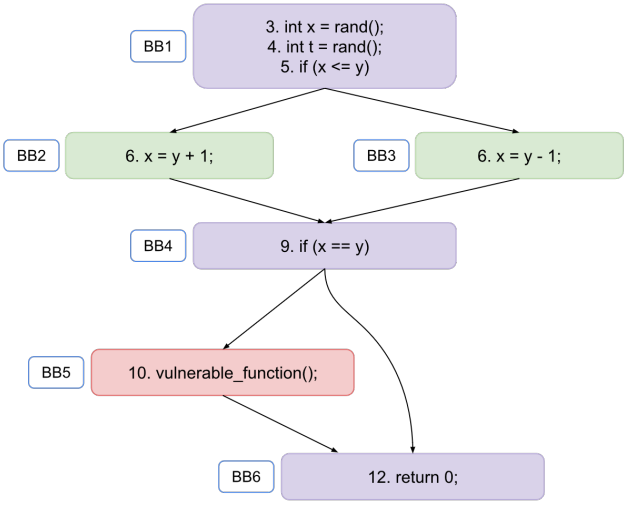


Fig. 2 CFG of the function from Figure 1.

extracted from C and C++ programs. Symbolic execution is a program analysis technique that interprets a program's execution using symbolic values rather than concrete inputs. During the interpretation, symbolic execution constructs path constraints that are solved using satisfiability modulo theories (SMT) (De Moura and Bjørner, 2008) solvers. While symbolic execution is a powerful technique, it is computationally expensive. Since it attempts to explore all possible program paths, the required resources increase rapidly, often leading to path explosion. To address these challenges for path feasibility analysis, we propose a method that directs symbolic execution, focusing only on the target path and eliminating exploration of unrelated program branches.

Additionally, to be practical for static analyzers, path feasibility analysis must be fully automated and satisfy several key requirements:

- **Automated Memory Symbolization:** When the symbolic executor encounters reads from an uninitialized memory location, it should automatically symbolize the unknown values.
- **Memory Model:** Since a static analyzer may identify a bug originating from an arbitrary function, the sizes and connectivity of the involved data structures are not always known. Therefore, the symbolic execution engine must employ a memory model that supports symbolic pointers, enabling the analysis of arbitrary pointer parameters, complex data structures, and dynamic memory allocation.
- **External Function Behavior:** To ensure comprehensive exploration of all possible execution paths, the symbolic executor must model the side effects of calls to external functions (i.e., functions whose bodies are unavailable to the analysis).

- **Loop Bounding:** To prevent the symbolic execution from becoming trapped in long or infinite loops, a robust loop-bounding strategy must be integrated. This ensures that the engine can limit loop exploration and continue analyzing subsequent parts of the target path.

The main contributions of this work are as follows:

1. A directed symbolic execution algorithm for determining the feasibility of any ICFG path in a specified set.
2. Fully automatic symbolization strategies for variables with unknown values, external function return values, and pointer arguments.
3. A memory model supporting symbolic pointer reasoning.
4. A loop-bounding algorithm that limits iterations without terminating symbolic execution.
5. An empirical evaluation on real-world software and the Juliet test suite (NIST, 2026).

The remainder of this paper is organized as follows. Section 2 presents the path feasibility analysis algorithm. In Section 3, we describe approaches for the automatic symbolization. Section 4 presents the memory model. Section 5 describes an algorithm for limiting loop iteration counts without interrupting the symbolic execution. Section 6 provides implementation details. Section 7 is dedicated to related work. Section 8 provides the developed testing system and evaluation results. Section 9 presents the conclusion and future work.

2 Interprocedural path feasibility analysis

This section presents an algorithm, based on directed symbolic execution, for determining whether at least one path in a specified set of ICFG paths is feasible. All paths are assumed to share a common entry block and a common target block. Unlike standard symbolic execution, which attempts to explore all reachable execution paths, the proposed method directs execution to a predefined set of basic blocks corresponding to the set of paths. This algorithm is particularly useful for verifying program properties that may involve multiple valid control flow variations within a specific execution scope.

Algorithm description:
Input:

1. First basic block of all paths' entry-point function (B_{entry}).
2. Last basic block of all paths (B_{target}).
3. The set of basic blocks representing the allowed execution scope (S_{allowed}).
4. A set of basic blocks that represents the specific milestones that must be reached to consider the verification successful:

$$S_{\text{req}} = \{B_{\text{entry}}, B_1, B_2, \ldots, B_{\text{target}}\} \subseteq S_{\text{allowed}}. \tag{1}$$

5. Loop iteration bound $C_{\max}$.

Output: A boolean value indicating whether there is a feasible path that starts from B_{entry}, remains entirely within S_{allowed}, covers the set S_{req}, and reaches B_{target}.

1. **Preprocessing:** If the reported path begins at a basic block other than the entry block of the corresponding function, S_{allowed} is extended to include all basic blocks that belong to the CFG paths originating from the function's entry basic block and reaching B_{entry}. After, B_{entry} is replaced by the entry basic block of the function.
2. **Initialization:** Symbolic execution is initialized at B_{entry}. Entry-point function parameters and external variables are made symbolic.
3. **Execution Loop:** The engine symbolically executes the program instructions.
   3.1. **Automatic Symbolization:** Uninitialized local variables, memory locations, and environment-dependent values are made symbolic.
   3.2. **Basic Block Marking:** If the executed basic block belongs to S_{req}, it is marked.
   3.3. **Pruning:** At every branching point, for each successor basic block $B_{\text{successor}}$, exploration of the corresponding branch is pruned if $B_{\text{successor}} \notin S_{\text{allowed}}$.
   3.4. **External Consistency:** Calls to external functions trigger a re-symbolization of errno, return values, and memory locations referenced by pointer arguments to account for side effects when the function body is unavailable.
   3.5. **Loop Iteration Limit:** If a loop header executes more than $C_{\max}$ times, the engine forces the current symbolic state to exit the loop (detailed in Section 5).
4. **Final Verification:**
   4.1. When B_{target} is reached, the algorithm verifies whether all basic blocks in S_{req} are marked. If so, the path is reported as feasible.
   4.2. If the symbolic execution terminates without reaching B_{target}, or without executing all elements in S_{req}, the algorithm reports the path as infeasible.

The preprocessing step is required to handle cases where a static analyzer reports a path that begins at an arbitrary basic block rather than at the function entry. The pruning step eliminates exploration of regions outside the specified scope, thereby improving scalability.

2.1 Single path analysis

This subsection discusses the special case of determining the feasibility of a single ICFG path.

Path: Let P denote a sequence of basic blocks in the program's ICFG:

$$P = \langle B_{\text{entry}}, B_1, B_2, \ldots, B_{\text{target}} \rangle. \tag{2}$$

Where:

- B_{entry} is the first basic block of the path's entry-point function.
- B_{target} is the final basic block in the sequence.

In this case, we apply the algorithm where

$$S_{\text{allowed}} = S_{\text{req}} = \{B \mid B \in P\}. \tag{3}$$

Although P is defined as a sequence of basic blocks, the pruning and final verification steps use set-based matching rather than exact sequence matching. This is necessary to support the loop-bounding approach, which may force execution to exit a loop before all repeated occurrences of loop blocks in P have been traversed. Consequently, the algorithm verifies whether all basic blocks in P were executed and whether the target block remains reachable, rather than requiring exact reproduction of the original path.

3 Automatic symbolization approaches

For path feasibility analysis, the symbolic executor must be able to explore all potential paths reaching the target endpoint. To enable this, we propose a set of automated symbolization strategies that require no manual configuration. Specifically, our approach automatically symbolizes entry-point function parameters, uninitialized local variables and memory locations, external global variables, return values and pointer arguments of external functions, and pointer-referenced memory locations.

```
1 int foo(int a)
2 {
3     if (a < 10)
4         return 1;
5     else if (a > 10)
6         return 0;
7     else
8         return -1;
9 }
```

Fig. 3 Entry-point function example.

The example in Figure 3 illustrates the need for automatic symbolization. The control flow of `foo` contains three feasible execution paths, each determined by the value of the input parameter a: (1) $a < 10$, (2) $a > 10$, and (3) $a == 10$. To explore all three paths, the symbolic executor must treat a as a symbolic value rather than assigning it a concrete value. Automatic symbolization of entry-point function parameters enables the generation of the corresponding path constraints and ensures that all feasible paths dependent on the function inputs are systematically analyzed without requiring user intervention.

3.1 Automatic symbolization of entry-point function parameters

Static analyzers may report ICFG paths originating from arbitrary functions rather than only from program entry points. In such cases, symbolic execution must begin at the given entry-point function. The suggested approach is to symbolize every parameter of the entry-point function.

3.2 Automatic symbolization of uninitialized local variables and memory locations

According to the C and C++ standards, a non-static, uninitialized local variable's value is indeterminate, meaning it can hold any arbitrary value. Therefore, the engine must create a symbolic object for each uninitialized variable and allocated memory region whose content is not explicitly initialized (allocated by new, malloc, etc.). These symbolic objects remain unconstrained until they are overwritten by program assignments or restricted by accumulated path constraints.

Although reading an uninitialized object constitutes Undefined Behavior (UB), our model treats such values as symbolic for path feasibility analysis. Conservatively modeling all possible runtime values, rather than strictly adhering to the compiler's optimization assumptions. While this may introduce spurious feasible paths compared to a strict standard-compliant execution, it ensures that paths resulting from uninitialized memory usage are not missed.

3.3 Automatic symbolization of external global variables

External global variables may be initialized in separate modules that are not available to the analyzer. Consequently, the value of such a variable may be unknown at the beginning of symbolic execution. These variables are handled in the same way as uninitialized local variables.

While this approach is generally effective, there are special global variables such as errno (Linux, 2026a) and environ (Linux, 2026b) that require individual processing for more precise analysis.

3.3.1 Automatic symbolization of errno

Because the implementation of an external function is unavailable, its effect on errno is treated as unknown. Therefore, after every external function call, the errno should be made symbolic. This allows the symbolic executor to explore dependent execution paths.

3.3.2 Automatic symbolization of environ

Without symbolic environment variables, analysis can be incomplete because the target program's execution paths may depend on environment variables. To enable the symbolic executor to explore these paths, the environment variables must be considered symbolic. Treating environ as an ordinary global symbolic variable may cause non-termination or excessive state generation during calls to the setenv (Linux, 2026c) and getenv (Linux, 2026d) functions. The reason is that these functions iterate through the environ array, comparing strings. If the length of the symbolic variable is not fixed, the engine can never prove that it has reached the end of the array. Consequently, it will continue to create new execution states for every possible character position.

The developed algorithm automatically makes the environ global variable symbolic at the beginning of the symbolic execution by creating a fixed count of symbolic

environment variables with fixed lengths. The number of symbolic environment variables and their maximum lengths are configurable parameters. This approximation may miss behaviors that depend on a larger number of variables or longer strings than the configured bounds.

3.4 Automatic symbolization of external function return value

Programs may call external functions whose implementations are unavailable to the analysis, such as dynamic library functions. If such a call returns a value, the engine symbolizes the returned value.

3.5 Automatic symbolization of external function pointer arguments

If an external function has non-constant pointer or reference parameters, it may modify the memory reachable through those arguments. Thus, during processing an external call, the engine re-symbolizes the memory regions referenced by such pointer and reference arguments.

3.6 Function pointers and virtual functions

Symbolic function pointers demand another approach. A symbolic executor should determine what functions can be called using a function pointer. Our approach searches candidate functions by function signature and return value. If the engine finds a matching function, it forks the symbolic state for that function.

The solution takes into account virtual function calls. To resolve virtual function calls, the algorithm does the following:

1. **Build a virtual function registry:** Extract all virtual function pointers from the program's virtual tables.
2. **Identify caller's potential types:** During processing a virtual function call instruction, examine the first argument type. If it is a pointer to a class type (e.g., Base*), collect all classes that derive from Base, as the actual runtime type could be any of these derived classes.
3. **Generate call candidates:** For each derived class identified in step 2, look up its corresponding virtual table to find the function that occupies the same vtable slot as the one being called. Each candidate will have the same virtual function signature but will be the override implementation specific to that derived class (e.g., Base::f1(), Derived1::f1(), Derived2::f1()).
4. **Fork symbolic states:** Create a separate symbolic state for each candidate function, allowing the symbolic execution to explore all possible execution paths.

The algorithm is complete but not sound. It will find all possible candidates for a virtual function; however, the list of candidates may include non-virtual functions, which are redundant in this case. As a result, this approach makes the symbolic executor execute redundant paths.

4 Memory model

For a symbolic executor, modeling memory objects with statically known sizes is straightforward. For example, for a static array, it will allocate an object of a known size to model it. However, if an array is passed to a function, its size is not available. We aim to develop a flexible approach to handle such situations.

```
1 struct S1;
2 struct S2 {
3     struct S1* arr;
4 };
5
6 void foo(struct S2* arg)
7 {
8     ...
9 }
```

Fig. 4 Example of arrays with unknown size.

As an illustrative example, consider the code example in Figure 4. Assume that symbolic execution starts at `foo`. The parameter `arg` is represented as a symbolic pointer, but the size and structure of the memory region referenced by that pointer are unknown. Also, the number of objects of each `arg[i].arr` is unknown. Some approaches allocate fixed-size arrays under symbolic pointers or allocate an array at the first memory access again with a fixed size. These approaches are simple and easy to implement, but are limited to predefined array sizes. We introduce a more flexible approach.

The key of the proposed memory model is as follows: for a symbolic pointer of type `T*`, the algorithm does not create the memory object referenced by the pointer until that memory is accessed. When the address is accessed for the first time, a single memory object of type `T` is created, and the relationship between the symbolic pointer and the accessed memory is recorded. For subsequent dereferences for the same base pointer (the original symbolic pointer value from which subsequent addresses are derived using pointer arithmetic), the algorithm considers possible aliasing scenarios: (A) There is an old memory object with the same offset as the current dereference. In this case, the same memory object is reused. (B) There is an old memory object, the offset of which may differ from the current dereference offset. So, the pointer could either reference one of the previously allocated memory objects created from the same symbolic base or it could reference an entirely new memory location.

To explore these possibilities, the algorithm forks the symbolic state according to the equality of corresponding offsets. For each existing memory object allocated with some offset from that symbolic base, it creates a state that reuses that object. Additionally, it creates one more state representing the case where the pointer does not alias any currently allocated objects, requiring a new memory object.

```
1  int foo(int *arr)
2  {
3      if (arr[5] < 10)
4          return 1;
5      else if (arr[5] > 10)
6          return 0;
7      else
8          return -1;
9  }
```

Fig. 5 Simple symbolic pointer dereferencing example.

```
1  void foo(int i, int j, int k, char *arr)
2  {
3      char a = arr[i];
4      char b = arr[j];
5      char c = arr[k];
6  }
```

Fig. 6 Example for possible aliasing scenarios.

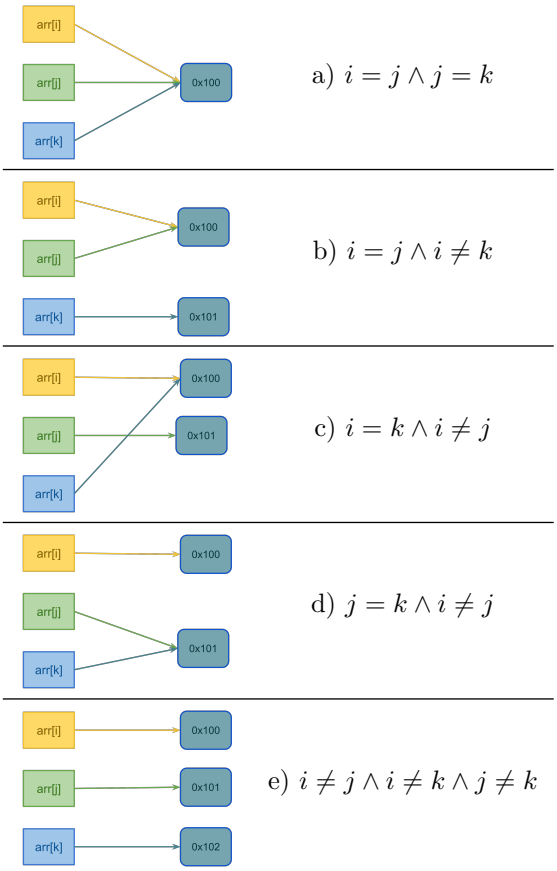


Fig. 7 Memory states.

This approach systematically explores feasible memory configurations while maintaining the relationship between symbolic pointers and their accessed memory locations. To mitigate potential state explosion caused by aliasing of symbolic pointers, the count of newly created states per pointer is user-configurable, allowing a trade-off between analysis precision and scalability.

For the code example in Figure 5, at the beginning, the engine will create only a symbolic pointer for the parameter `arr`. When execution reaches line 3, a symbolic executor creates a symbolic memory under `arr[5]` and records a mapping between the address `arr+5` and the created object. As a result, our approach enables the symbolic executor to explore all three feasible execution paths of the function.

To clarify how additional states are created based on previously allocated objects, consider the example in Figure 6. Since all the parameters are made symbolic, the symbolic executor will explore five execution paths depending on the program's memory state. The states are described in Figure 7.

This memory model supports symbolic exploration of various data structures, including linked lists, trees, and hash tables, without relying on fixed-size allocations.

5 Bounding loops iterations

Path explosion is one of the main challenges in symbolic execution, and loops are a major source of this problem. Especially, a loop whose termination condition depends on symbolic values may lead to non-terminating symbolic execution. Even loops with non-symbolic conditions in real-world projects can lead to exhaustive symbolic execution of the program. Therefore, bounding loop exploration is essential in symbolic execution. It is important to notice that after exiting the loop, the symbolic state should continue its execution. We suggest a fast-bounded loop termination solution. The high-level description of our algorithm is as follows: it takes as input a maximum loop iteration bound. If a symbolic state enters a loop more than the given threshold times, the algorithm redirects the symbolic execution of the state to the loop exit.

Let $C_{\max}$ denote the maximum number of times a symbolic state is allowed to enter a given loop. For every loop, the algorithm records its header and exit edges, as well as the iteration count. If a loop header executes more than $C_{\max}$ times, the algorithm redirects the symbolic state using only exit edges whose source basic block is executed by the current symbolic state at least once.

After redirecting the symbolic state, the engine symbolizes memory locations written during the execution of that symbolic state. After the termination, the algorithm resets the iteration count of the loop.

Restricting the forced exits to edges whose source basic blocks were already executed avoids making gaps in execution flow. Symbolization of written memory locations enables the engine to explore some feasible paths (e.g., paths requiring higher iteration counts) that could be missed because of termination.

This loop-bounding strategy introduces a deliberate trade-off between completeness and scalability. By forcing a loop exit after $C_{\max}$ iterations, the analyzer may explore paths that are logically unreachable under standard execution semantics (false

positives). However, in the context of validating static analysis reports, this trade-off is acceptable as it prioritizes the reachability of the target defect location over exhaustive exploration of all loop behaviors.

6 Implementation

We implemented our method in the KLEE (Cadar et al., 2008; KLEE, 2026) (v3.1) symbolic execution engine, designed for C and C++ languages, built on top of the LLVM (LLVM, 2026a) compiler infrastructure. It interprets a program's LLVM intermediate representation (LLVM IR) (LLVM, 2026a). The tool models memory with bit-level accuracy and employs a variety of constraint-solving optimizations. KLEE utilizes various constraint solvers, such as STP (Ganesh and Dill, 2007) and Z3 (De Moura and Bjørner, 2008; Z3, 2026).

While KLEE is a state-of-the-art symbolic execution engine, it requires manual specification of symbolic variables. So, we extended KLEE using its API and LLVM's API (v16.0.4) to automatically create symbolic objects for several cases, such as uninitialized local variables and memory locations, external global variables, special global variables (errno and environ), entry-point function parameters, external function return values and memory locations referenced by a pointer argument of an external function.

6.1 Automatic symbolization of entry-point function parameters

KLEE supports starting symbolic execution from any specified function, but only handles parameters automatically for the main function. In other words, this feature works if the given entry-point function is main or has no parameters. We extended this feature, making all parameters of the entry-point function symbolic.

6.2 Automatic symbolization of external function return value and pointer arguments

When an external function is called, the original KLEE cannot continue symbolic execution unless it is possible to execute the external function dynamically, concretizing the arguments of the call. To prevent dynamic behavior and continue symbolic execution, our method abstracts the function call and creates a symbolic object for the returned value and memory locations referenced by non-constant pointer arguments of the external function call.

6.3 Symbolic function pointers and virtual functions

Original KLEE has limited support for symbolic function pointers and virtual functions. We extended this part by implementing the described algorithm. For symbolic function pointers, we iterate over all defined functions in the module and find exact matches. For virtual functions, we use LLVM's virtual table information. It keeps global virtual tables for each class. These classes contain function pointers to all candidate functions. Also, every derived class contains a pointer field to its base class.

Using this information, we find all possible candidates from the current module and create a new symbolic state for every matched function.

6.4 Automatic symbolization of pointer-referenced memory

When accessing a symbolic address, the original KLEE matches the symbolic address with the existing memory objects of the program. This may produce many symbolic states and significantly slow down the symbolic execution process. Our implementation bypasses matching against the existing objects. Instead, it creates a new symbolic object for the dereferenced symbolic pointer as described above. We implemented our model, combining it with KLEE's memory model.

6.5 Bounding loops iterations

LLVM does loop analysis and provides collected information via its API. We collect loops with headers and exit edges for each function, and apply our bounded loop termination algorithm during symbolic execution.

7 Related work

The problem of determining the feasibility of a specific control flow path is undecidable in the general case, yet it remains an important component of automated software analysis, testing, and verification. Research in this area is extensive and can be broadly categorized into symbolic execution, model checking (Clarke et al., 1986), abstract interpretation (Cousot and Cousot, 1977), and hybrid approaches.

7.1 Symbolic execution

This method treats program inputs as symbolic variables rather than concrete values, enabling the executor to explore multiple execution paths simultaneously. Uses a constraint solver to determine possible execution directions.

One of the fundamental limitations of symbolic execution is path explosion, in which the number of symbolic states grows exponentially with branching and loop nesting. Our approach mitigates this by directing the symbolic execution strictly to the target path or subgraph, thereby avoiding the cost of exhaustive execution.

7.2 Model checking

Model checking is an automated formal verification technique that constructs a finite model of a program and exhaustively checks if a desired property holds. Bounded model checking (Biere et al., 1999) (BMC) unrolls program loops a finite number of times and translates the resulting bounded program and the verification property into a single formula. This formula is then solved using a SAT (Sörensson and Eén, 2003; Moskewicz et al., 2001) or SMT solver to check the feasibility of all paths up to a certain length.

In contrast, while BMC is effective for finding counterexamples in bounded depths, it often may not scale to large software systems due to the complexity of the generated

formulas. Our method's loop-bounding strategy provides a similar benefit but focuses by directing execution toward specific target paths rather than encoding the entire program state.

7.3 Abstract interpretation

Abstract interpretation (Cousot and Cousot, 1977) analyzes programs by interpreting them over abstract domains rather than concrete values. It is based on lattice theory (Davey and Priestley, 2002) and computes fixpoints over program states. By merging states using abstraction operators, it often achieves better scalability than symbolic execution, at the cost of reduced precision.

Because it is not path-sensitive, abstract interpretation may over-approximate program behavior and produce spurious reports about feasible execution paths. In contrast, the proposed method focuses on path-sensitive symbolic execution and aims to improve precision at the level of individual control-flow paths, thereby reducing the probability of infeasible path reports.

7.4 Hybrid and specialized tools

Several tools have attempted to bridge the gap between scalability and precision by combining different techniques.

SATRACER (Li et al., 2021) utilizes a lightweight symbolic executor specifically designed to filter out infeasible paths reported by static analysis. However, it relies heavily on deep, specific information generated during the static analysis phase and has limited memory modeling capabilities. Notably, it does not support symbolic memory locations, as offsets are always concretized.

Symbiotic (Chalupa et al., 2021) combines instrumentation, program slicing, and symbolic execution with k-induction (Sheeran et al., 2000). The method relies on loop summarization yet lacks support for nested loops or inter-procedural function calls. Furthermore, regarding unbounded loops or infinite execution paths, program slicing alone is insufficient unless the slicing criterion explicitly excludes the non-terminating computation from the analysis scope.

Snugglebug (Chandra et al., 2009) uses directed symbolic execution to verify if a bug report is reproducible. Uses backward symbolic analysis to compute weakest preconditions along control-flow paths. By propagating constraints from the target statement back to the entry point, the tool determines the input conditions required to trigger the specific bug.

Especially proposing shortest-distance symbolic execution (SDSE), call-chain-backward symbolic execution (CCBSE), and a mixed strategy, Mix-CCBSE, that mixes CCBSE with a forward strategy.

Ma et al. (2011) introduce several directed symbolic execution strategies designed to reach a specific target line in the code. Their work notably proposes shortest-distance symbolic execution (SDSE), call-chain-backward symbolic execution (CCBSE), and a mixed strategy, Mix-CCBSE, that mixes CCBSE with a forward strategy.

Trabish et al. (2018) introduce a semi-automated approach where users manually annotate 'uninteresting' code regions. By excluding these regions, the symbolic execution engine can bypass irrelevant computations and reach target locations more efficiently. However, this method relies heavily on manual user guidance and is primarily optimized for exploration speed rather than automated precision.

CPAchecker (Beyer and Keremoglu, 2011) addresses path feasibility by orchestrating the interaction between multiple analysis domains. Specifically, it combines explicit-value analysis with predicate abstraction to identify and prune execution paths that are determined to be infeasible.

KLEEF (Misonizhnik et al., 2024) is a specialized derivative of the KLEE symbolic execution engine, fine-tuned to analyze industrial-grade C and C++ codebases. It can be used in a trace reproduction mode to verify a static analyzer's reports in SARIF format. Uses a lazy approach for symbolic pointers, but unlike our method, it is limited by creating fixed-size arrays under symbolic pointers.

In contrast to the reviewed tools, the proposed method combines automated symbolization with a lazy memory model that supports symbolic pointers and nested structures, thus enabling various data structures to be modeled within path feasibility analysis. It operates without manual instrumentation or reliance on analysis-specific information produced by a particular static analysis tool.

8 Experimental evaluation

To assess the effectiveness and efficiency of the proposed method, we conducted a series of experiments. The evaluation focuses on three key aspects. First, we measure the scalability of the approach on complex and long ICFG paths extracted from real-world software. Second, we assess its impact on static analysis precision and recall by integrating it as a verification stage in the MLH (Aslanyan et al., 2024) static analysis tool. Third, we compare the robustness of the method against KLEEF using the Juliet benchmark suite and reports generated by the Clang Static Analyzer (CSA) (LLVM, 2026b) and Infer (Facebook, 2026) static analyzers.

Experiments were conducted on Ubuntu 22.04 using an 11th Gen Intel Core i7-11700 processor (8 cores, 16 threads) with 32 GB of RAM. All experiments were performed using KLEE's default SMT solver configuration.

8.1 Testing system

To evaluate the efficacy of the proposed path feasibility analysis, we developed a testing system designed to extract real-world execution paths, verify them using our path feasibility analysis tool, and quantify the improvement. The path generation and verification process consists of four stages:

1. **Bitcode Extraction:** We employed gllvm (gllvm, 2026) (v14.0.0) to compile the target project and extract the whole-program LLVM IR.
2. **Instrumentation:** A custom LLVM (v16.0.4) pass was developed to instrument the LLVM IR. This pass inserts a trace logging instrumentation call at the entry of every basic block, allowing us to log the sequence of executed blocks dynamically.

3. **Trace Collection:** The instrumented LLVM IR was compiled into executable binaries using Clang (LLVM, 2026c) (v16.0.4). We then ran the project's existing test suite. As the tests were executed, the instrumentation logged the exact sequence of basic blocks, providing us with a set of "ground truth" feasible paths.
4. **Verification:** Finally, these recorded paths and corresponding LLVM bitcode files were provided as input to our analysis tool to determine whether the tool could correctly validate them. A reported path is considered successfully verified if symbolic execution reaches the target location while satisfying all path constraints. Failure to verify a feasible path is treated as a false negative.

Since the execution traces were collected from real executions of the target programs, all extracted paths represent ground-truth feasible paths. Consequently, this dataset contains only positive samples. Thus, successful verification corresponds to a true positive, while failure to verify a recorded path corresponds to a false negative. Therefore, the evaluation primarily measures recall, whereas precision and F1-score are not applicable due to the absence of infeasible paths.

We benchmarked the tool using GNU Coreutils (GNU, 2026) (commit 24450e). For the experiments, we selected 234 unique ICFG paths. The shortest path contains 22 basic blocks; the longest contains 1218; the average path length is 362. The single path verification algorithm (Section II) was applied. Symbolic execution was limited to a maximum of 20 million executed instructions.

Analysis performance:

1. **Path verification success rate (recall):** The tool successfully verified 223 of the 234 paths (95.3%).
2. **Standard Execution:** 174 paths were verified without requiring loop bounding.
3. **Bounded Execution:** 49 paths required the loop bounding strategy, with the iteration count threshold set to 4.
4. **Failed Cases:** Only 11 paths could not be verified due to exceeding the instruction count limit or bounded loop termination.

Search strategy analysis: Experiments demonstrated that KLEE's Depth First Search (DFS) strategy yielded the best performance for the majority of cases, successfully verifying 221 of the 223 paths. However, 2 paths required a Breadth First Search (BFS) strategy. In these specific cases, the bounded loop termination algorithm encountered a loop with a symbolic condition that generated a large number of symbolic states. Under DFS, the state required to verify the path was pushed to the end of the queue, making it unreachable within reasonable time limits. BFS allowed the engine to explore the state space more evenly and find the target state. This finding highlights the importance of adaptive state selection strategies in path feasibility analysis.

Table 1 demonstrates some long and medium-sized paths' feasibility analysis details. Execution times varied significantly based on path complexity. While simple paths were verified in under a minute, complex paths involving heavy symbolic pointer arithmetic and path constraints required up to 30 minutes.

Table 1 Path feasibility analysis results.

Blocks	Instructions	Time (s)	Queries	Strategy	Loop bound
1191	10328647	1270	68673	DFS	3
1155	10124940	1329	68713	DFS	3
1054	10680091	1852	68709	DFS	3
855	12532657	1739	92857	BFS	3
850	13663509	1732	251199	BFS	4
477	13286	23	2116	DFS	∞
308	12845	25	2267	DFS	∞

8.2 Integration with MLH static analysis tool

To evaluate the impact of our method on static analysis precision, we evaluated the MLH tool, both with and without our path feasibility analysis, using the Juliet Test Suite. The Juliet suite is a widely recognized benchmark in the software security community, specifically designed to evaluate the capability of static analysis tools to detect diverse software vulnerabilities. It contains over 112,000 test cases covering 118 distinct Common Weakness Enumeration (CWE) categories, offering a comprehensive dataset of both buggy and patched (fixed) code samples.

For this evaluation, we focused specifically on the CWE-401 (Memory Leak) subset, a category where static analyzers often struggle with high false positive rates due to the difficulty of tracking complex heap management across control flow paths.

For a static analyzer's reports evaluation, a true positive corresponds to a vulnerability report associated with a feasible execution path leading to the vulnerability location, whereas a false positive corresponds to an infeasible or safe reported path. Precision, recall, and F1-score are computed for static analysis results before and after applying the feasibility analysis.

MLH specifies each path using two sets of basic blocks: an allowed set and a required set. To verify feasibility, the execution must reach the designated endpoint by traversing strictly within the allowed set while visiting every block in the required set. A path is considered infeasible if any required basic block is not executed during symbolic exploration.

To enforce these constraints, we employed our path set feasibility algorithm (Section 2) by mapping the allowed basic blocks to S_{allowed}, and the required basic blocks to S_{req}.

Table 2 MLH results with and without path feasibility analysis.

Method	TP	TN	FP	FN	Precision	Recall	F1-score
MLH	868	3940	666	0	0.56	1	0.71
MLH + Path Feasibility	868	4606	0	0	1	1	1

The quantitative results of this evaluation are detailed in Table 2. The data demonstrates that integrating our path feasibility analysis eliminated false positives

generated by the MLH static analyzer on the evaluated Juliet CWE-401 subset. Furthermore, the analysis successfully verified all true positives, ensuring that no valid vulnerability reports were discarded. This confirms that the feasibility analysis algorithm effectively refines static analysis results by filtering infeasible error paths while preserving recall.

8.3 Comparison with KLEEF

To benchmark the robustness of our approach, we compared our path feasibility analysis against KLEEF (commit c460ace), a derivative of KLEE designed for industrial code analysis. Precision, recall, and F1-score were computed for the static analysis results as described in subsection 8.2.

We utilized the Juliet Test Suite to generate reports using two static analyzers: Infer and CSA. For the comparative dataset, we selected the CWE-415 (Double-Free) and CWE-416 (Use-After-Free) subsets. These categories were chosen specifically because KLEEF explicitly supports verification logic for Use-After-Free (for both Infer and CSA) and Double-Free (for CSA) reports.

To ensure a fair comparison, both engines were configured with identical resource limits: a timeout of 10 minutes per report and a memory limit of 2GB. These limits were selected because the Juliet test cases are relatively small and are expected to complete within modest computational resources, while still ensuring that the evaluation finishes in an acceptable time and memory. In practice, none of the evaluated test cases exhausted either limit. KLEEF was executed using its default search strategy and solver.

The results of the comparison are provided in Tables 3 and 4. While our method verified all true positive results of Infer, KLEEF classified 36 true positive reports as infeasible and crashed on 60 reports.

For CSA reports, KLEEF verified only 30 out of 780 true positive results and classified 196 reports as infeasible. It did not report anything on rest. Our method verified all 780 true positives and classified 473 false positive reports as feasible paths. The reason for verified false positives is that our method is designed for path feasibility analysis. When static analyzers generate a false positive report with a potentially feasible path, our algorithm validates it. In such cases, additional checks can be applied to enhance the metrics.

Failure analysis of KLEEF indicates that performance degradation is primarily due to its inability to process reports of a static analyzer, to handle the specific external function calls, and symbolic pointer arithmetic, leading to early termination.

Table 3 Infer results on CWE-416 (use-after-free).

Method	TP	FP	Precision	Recall	F1-score
Infer	96	0	1	1	1
Infer + KLEEF	0	0	0	0	0
Infer + Path Feasibility	96	0	1	1	1

Table 4 CSA results on CWE-416 (use-after-free) and CWE-415 (double-free).

Method	TP	FP	Precision	Recall	F1-score
CSA	780	475	0.621	1	0.766
CSA + KLEEF	30	0	1	0.03	0.058
CSA + Path Feasibility	780	473	0.622	1	0.767

In summary, KLEEF decreases the F1-Score of Infer results from 1 to 0 and the F1-Score of CSA from 0.766 to 0.058. Our method keeps the same F1-Score for Infer and slightly improves by 0.4% CSA results.

9 Conclusion and future work

This paper presents a method for checking interprocedural control flow path feasibility using directed symbolic execution. By introducing automatic symbolization approaches—covering uninitialized variables, external function returns, and pointer arithmetic—and implementing a bounded loop strategy, we effectively addressed the impact of the path explosion and resource constraints typical of symbolic execution.

Our evaluation demonstrates the practical effectiveness of this approach on both testing examples and real-world projects. These results indicate that the presented technique is efficient for eliminating false positives in static analysis.

Despite these advantages, the approach introduces a trade-off between completeness and scalability due to loop bounding and state-space constraints.

In future work, we aim to further improve scalability by parallelizing the analysis. Specifically, we plan to split large complex paths into smaller segments and analyze them simultaneously. Additionally, our experiments highlighted that standard state exploration strategies are not always optimal for targeted path verification. Therefore, we plan to develop and implement a specialized state selection strategy designed to prioritize and reach the target path's endpoint more efficiently.

Declarations

Acknowledgements. The authors thank Dr. Sevak Sargsyan for reviewing the manuscript and providing valuable comments and suggestions that improved the quality of this paper.

The authors used ChatGPT, Gemini, and Claude for grammar checking and language enhancement. All content was reviewed and edited by the authors, who take full responsibility for the final manuscript.

Data Availability. The benchmark datasets and open-source projects analysed during the current study are publicly available. The Juliet Test Suite is available in the GitHub repository at https://github.com/find-sec-bugs/juliet-test-suite. The GNU Coreutils project is available at https://www.gnu.org/software/coreutils/.

Funding. This research was supported by the Higher Education and Science Committee of MESCS RA (Research Project No. 24AA-1B021).

Author Contributions. H.M. contributed to the design of the study, drafted Sections 2, 6, 7, and 8.1, implemented the main algorithm, and participated in the analysis and interpretation of the results. H.H. contributed to the design of the study, drafted Sections 3 and 8.2, implemented the automatic symbolization strategies, and participated in the analysis and interpretation of the results. T.A. contributed to the design of the study, drafted Sections 4 and 8.3, implemented the memory model, and participated in the analysis and interpretation of the results. H.A. supervised the research, coordinated the study, drafted Sections 1, 5, and 9, reviewed and revised the manuscript critically for important intellectual content, and participated in the analysis and interpretation of the results. All authors contributed to the development of the methodology, discussed the results, reviewed and approved the final manuscript, and agree to be accountable for all aspects of the work.

Competing interests. The authors declare no competing interests.

References

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM, 19*(7), 385–394.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems*. Budapest.

NIST. (2026). *Juliet test suite*. Retrieved May 22, 2026, from https://github.com/find-sec-bugs/juliet-test-suite

Linux. (2026a). *errno(3) — Linux manual page*. Retrieved May 22, 2026, from https://man7.org/linux/man-pages/man3/errno.3.html

Linux. (2026b). *environ(7) — Linux manual page*. Retrieved May 22, 2026, from https://man7.org/linux/man-pages/man7/environ.7.html

Linux. (2026c). *setenv(3) — Linux manual page*. Retrieved May 22, 2026, from https://man7.org/linux/man-pages/man3/setenv.3.html

Linux. (2026d). *getenv(3) — Linux manual page*. Retrieved May 22, 2026, from https://man7.org/linux/man-pages/man3/getenv.3.html

Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on operating systems design and implementation*. San Diego.

KLEE. (2026). *KLEE symbolic execution engine*. Retrieved May 22, 2026, from https://klee-se.org

LLVM. (2026a). *The LLVM compiler infrastructure*. Retrieved May 22, 2026, from https://llvm.org

Ganesh, V., & Dill, D. L. (2007). A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on computer aided verification*. Berlin.

Z3. (2026). *Z3*. Retrieved May 22, 2026, from https://github.com/Z3Prover/z3

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems, 8*(2), 244–263.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on principles of programming languages*. Los Angeles.

Biere, A., Cimatti, A., & Clarke, E. M. (1999). Symbolic model checking without BDDs. In *Proceedings of the 5th international conference on tools and algorithms for construction and analysis of systems*. Berlin.

Sörensson, N., & Eén, N. (2003). An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. Berlin.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual design automation conference*. Las Vegas.

Davey, B. A., & Priestley, H. A. (2002). *Introduction to lattices and order* (2nd ed.). Cambridge University Press.

Li, G., Yuan, T., Lu, J., Li, L., Zhang, X., Song, X., & Zhang, K. (2021). Exposing vulnerable paths: Enhance static analysis with lightweight symbolic execution. In *28th Asia-Pacific Software Engineering Conference (APSEC)*. Taipei.

Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., & Strejček, J. (2021). Symbiotic 8: Beyond symbolic execution: (Competition contribution). In *Tools and algorithms for the construction and analysis of systems*. Luxembourg.

Sheeran, M., Singh, S., & Stålmarck, G. (2000). Checking safety properties using induction and a SAT-solver. In *Formal methods in computer-aided design*. Berlin.

Chandra, S., Fink, S. J., & Sridharan, M. (2009). Snugglebug: A powerful approach to weakest preconditions. *SIGPLAN Notices, 44*(6), 363–374.

Ma, K.-K., Khoo, Y. P., Foster, J. S., & Hicks, M. (2011). Directed symbolic execution. Springer Berlin Heidelberg.

Trabish, D., Mattavelli, A., Rinetzky, N., & Cadar, C. (2018). Chopped symbolic execution. In *Proceedings of the 40th international conference on software engineering*. Gothenburg.

Beyer, D., & Keremoglu, M. E. (2011). CPACHECKER: A tool for configurable software verification. In *Proceedings of the 23rd international conference on computer aided verification*. Snowbird.

Misonizhnik, A., Morozov, S., Kostyukov, Y., Kalugin, V., Babushkin, A., Mordvinov, D., & Ivanov, D. (2024). KLEEF: Symbolic execution engine (competition contribution). In *Fundamental approaches to software engineering: 27th international conference, FASE 2024, held as part of the European Joint Conferences on Theory and Practice of Software*. Luxembourg.

Aslanyan, H., Movsisyan, H., Hovhannisyan, H., Gevorgyan, Z., Mkoyan, R., Avetisyan, A., & Sargsyan, S. (2024). Combining static analysis with directed symbolic execution for scalable and accurate memory leak detection. *IEEE Access, 12*, 80128–80137.

LLVM. (2026b). *Clang-analyzer*. Retrieved May 22, 2026, from https://clang-analyzer.llvm.org/

Facebook. (2026). *Infer*. Retrieved May 22, 2026, from https://fbinfer.com/

gllvm. (2026). *Whole program LLVM in Go*. Retrieved May 22, 2026, from https://github.com/SRI-CSL/gllvm

LLVM. (2026c). *Clang*. Retrieved May 22, 2026, from https://clang.llvm.org/

GNU. (2026). *Coreutils – GNU core utilities*. Retrieved May 22, 2026, from https://www.gnu.org/software/coreutils/