Program Path Feasibility Through Symbolic Execution

Hovhannes Movsisyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
e-mail: hovhannes.movsisyan@rau.am

Hripsime Hovhannisyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
e-mail: hripsime.hovhannisyan@rau.am

Hayk Aslanyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
e-mail: hayk.aslanyan@rau.am

***Abstract*—This article introduces a method for checking program control flow path feasibility through symbolic execution. We present an automatic symbolization approaches that target these important parts: variables with unknown values, function arguments and return values, and pointer-referenced memories. The proposed techniques make symbolic execution more practical for software testing and verification. We also introduce an algorithm for program control flow path verification that utilizes the mentioned enhancements. As the symbolic execution is a resource-demanding process, the path verification algorithm eliminates redundant program branches. This feature is integrated into a static analysis tool for more accurate analysis.**

***Keywords*—Symbolic execution, automatic symbolization, program path feasibility.**

I. Introduction

In the modern world, where software systems play a crucial role, ensuring software security and reliability is essential. As the systems grow in complexity and size, there is an urgent need for automated software analysis tools. One of these techniques is symbolic execution. Symbolic execution [1] is a program analysis technique that interprets a program's execution using symbolic values instead of concrete inputs. It explores multiple execution paths simultaneously by treating input variables as symbolic values. This approach can help identify potential bugs, vulnerabilities, and edge cases that traditional testing methods might miss. Symbolic execution is useful for automated test generation, program verification, and vulnerability detection in software systems. However, its practical application is often limited by scalability issues, path explosion, environment modeling, and difficulties in handling complex program structures.

This paper focuses on source code symbolic execution for C and C++ languages. Here are some of the most well-known symbolic execution tools for the C and C++ languages.

EXE [2] is a symbolic execution tool for the C language designed for comprehensive testing of complex software. EXE models memory with bit-level accuracy. This is needed because code often treats memory as untyped bytes and observes a memory location in multiple ways. EXE's performance is based on its ability to resolve constraints rapidly. This speed is achieved through various optimizations. EXE utilizes its custom-built constraint solver, STP [3]. Additionally, several higher-level optimizations are employed, including caching mechanisms and the removal of irrelevant constraints.

KLEE [4] [5] is a redesign of EXE, designed for C and C++ languages, built on top of the LLVM compiler [6] infrastructure. It interprets a program's LLVM IR. Like EXE, KLEE models memory with bit-level accuracy and employs a variety of constraint-solving optimizations. One of the key improvements of KLEE over EXE is its ability to store a much larger number of concurrent states. Another important improvement is its enhanced ability to handle interactions with the outside environment— for example, with data read from the file system or over the network—by providing models designed to explore all possible interactions with the outside world. KLEE also uses different constraint solvers, STP and Z3 [7].

Despite KLEE being a state-of-the-art tool for symbolic execution, it suffers from major limitations:

- **Need for manual specification of which inputs should be treated as symbolic**: According to the C/C++ language standard, if memory is uninitialized, its value in most cases is undefined. In other words, such memory can contain any value. For example, non-static local variables, uninitialized dynamic memory, etc.. KLEE has no mechanism for automatic symbolization in such cases.
- **The entry function must be main or have no arguments:** Sometimes, there is a need to start the symbolic execution from a specific function. For example, when a library function must be executed symbolically.
- **Non-effective handling of symbolic pointers:** If there is a symbolic pointer in a program, KLEE tries to match the symbolic pointer with the existing objects of the program. This approach has several disadvantages. In real-life programs, a symbolic pointer can match numerous objects, creating new execution states, increasing the execution time and memory of symbolic execution. Also, the creation of new states can lead to a path explosion.
- **Dynamic behavior:** During the processing of the called function, KLEE checks whether the called function exists in the current module. If it does not exist, there are three available options:
  - None - No external function calls are permitted.

https://doi.org/10.51408/csit2025_44

- Concrete - Only external calls with concrete arguments are permitted.
- All - All external function calls are permitted.

To summarize this part, we can say that if there is an external call, KLEE cannot continue the symbolic execution or will execute the called external function dynamically, which leads to the dynamic behavior of the execution. It is a major limitation for large-scale project analysis.

This work aims to address the mentioned challenges by developing advanced methods and algorithms that can significantly increase the effectiveness of symbolic execution in real-world software analysis scenarios. All proposed improvements are designed with the ultimate goal of enabling fully automatic path feasibility analysis.

The rest of this paper is organized as follows. In Sections II and III, we describe the automatic symbolization of variables with unknown values, function arguments, and return values. Section IV shows the automatic symbolization of pointer-referenced memory. Section V describes the path feasibility algorithm through symbolic execution. Section VI provides the results. Section VII presents the conclusion.

II. Automatic Symbolization Of Variables With Unknown Values

As was mentioned above, in C/C++ programs in various scenarios, if memory remains uninitialized, it can contain any value. To explore as many execution paths as possible, it is reasonable to create symbolic objects for such cases.

A. Automatic Symbolization of Uninitialized Local Variables

Consider the following code example.

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

Based on the C/C++ language standard non-static, uninitialized local variable's value is undefined, meaning it can hold any arbitrary value. Here, the variable "a" declared at line 2 is uninitialized and can contain any value. Therefore, there are three execution paths. But the original KLEE will explore only one of these paths because, in such cases, KLEE sets a fixed value in the variable "a".

Our approach is the following. We create a symbolic object for each "alloca" instruction [8]. This object remains symbolic till the end of the program or until initialization. As the variable "a" is symbolic, the modified KLEE will successfully find all three paths.

B. Automatic Symbolization of External Global Variables

In the following example, the variable "G" is defined and may be initialized in another module.

```
1.  extern int G;
2.  int main () {
3.      if (G == 10)
4.          return 0;
5.      else
6.          return 1;
7.  }
```

So there are two execution paths. The solution is the same as for local variables. We will make a symbolic object for the variable "G", and modified KLEE will find two paths.

Another interesting case is processing special global variables such as "errno". Every external function call can modify the variable "errno". Therefore, after every function call whose body is not available, the "errno" should be made symbolic. In the example below, the source code of the function "external_function" is not available, so it can change "errno".

```
1.  void external_function();
2.  int main () {
3.      external_function();
4.      if (errno != 0)
5.          return 1;
6.      return 0;
7.  }
```

After such external calls, the modified KLEE makes "errno" symbolic and can explore two execution paths. The same approach can be applied to other similar variables.

III. Automatic Symbolization of Function Arguments And Return Values

As was mentioned, in some situations, there is a need to start the symbolic execution from a specific function. Such a function is called "Entry Point".

A. Automatic Symbolization of Entry Point Function Arguments

KLEE allows the start of symbolic execution from the given function. On the other hand, KLEE supports only the processing of "main" function arguments. In other words, this feature works if the given entry point function is "main" or has no arguments.

The suggested approach is to make a symbolic object for every argument of the entry point function. If the current argument is a pointer, make a symbolic pointer (Section IV).

In the example below, there are three execution paths in the function "foo".

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

To explore all three execution paths, modified KLEE makes a symbolic object for the argument "a".

B. Automatic Symbolization of External Function Return Value

As was mentioned above, there can be a call to a function whose body is not available. E.g., a call to a function from a

dynamic library. In such cases, KLEE cannot continue the symbolic execution or will dynamically execute the called external function if possible. To prevent dynamic behavior and continue symbolic execution, modified KLEE ignores the called function and creates a symbolic object for the returned value. In the example below, modified KLEE will find two execution paths.

```
1.  int external_function();
2.  int main () {
3.      int a = external_function();
4.      if (a)
5.          return 1;
6.      return 0;
7.  }
```

C. *Automatic Symbolization of External Function Pointer Arguments*

If the pointer is passed to an external function, the pointer-referenced memory can be modified. So, for each pointer argument, modified KLEE will find the pointer-referenced memory and automatically make it symbolic.

```
1.  void external_function(int *ptr);
2.  int main () {
3.      int a = 10;
4.      int *ptr_1 = &a;
5.      int *ptr_2 = &a;
6.      external_function(ptr_1);
7.      if (a == 20)
8.          return 1;
9.      if(*ptr_2 == 30)
10.         return 2;
11.     return 0;
12. }
```

In the example above, modified KLEE will find all three execution paths, because after the call of "external_function", the memory under "ptr_1" will be symbolic. This means that execution can reach lines 8, 10, and 11.

IV. Automatic Symbolization of Pointer-Referenced Memory

Uninitialized pointers, like other uninitialized memories, are also made symbolic. However, implementing the dereference of a symbolic pointer is a challenging problem. When accessing a symbolic address, the original KLEE matches the symbolic address with the existing objects of the program. It can lead to the creation of numerous execution states and slow down the symbolic execution. Another approach is to allocate memory with a fixed size under a symbolic pointer [9]. This approach suffers from big arrays and arrays with symbolic sizes.

The key idea behind our method as follows. Create a single memory object under a symbolic address only when it is accessed for the first time.

For a symbolic pointer, modified KLEE does not create any memory until that memory is accessed. If the symbolic pointer accesses the memory, a symbolic object is created. The relationship between the symbolic pointer and the accessed memory is saved. If the same memory is accessed after, we use the existing memory that we have created previously.

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

For the code example above, modified KLEE at the beginning will create only a symbolic pointer for the argument "a". When execution reaches line 2, a symbolic executor creates a symbolic memory under "a[5]" and keeps mapping between the address "a+5" and the created object. Therefore, modified KLEE finds all 3 paths of the function.

V. Path Feasibility

Utilizing the implemented features, it is possible to check automatically the feasibility of control flow paths. A control flow path must contain the following information.

Entry point: A function name from which symbolic execution must start.

Endpoint: Function name and its basic block specifier, which specifies the end of the path.

Basic blocks: A set of functions with their basic blocks contained in the given path.

For the given path, the path feasibility checker should answer whether it is feasible. To answer the question, symbolic execution should start from the beginning of the path (Entry point) and reach the end (Endpoint), visiting all basic blocks of the path. Here is the description of the algorithm.

1. Start symbolic execution from the beginning of the given path.
    a. Make entry function parameters symbolic.
    b. Make external global variables symbolic.
    c. Make uninitialized variables symbolic.
2. If the executed basic block is from the given path's basic blocks, mark it as executed.
3. If the branch instruction leads to a basic block outside of the given path, eliminate the execution of that basic block. Such eliminations will prevent the execution of redundant parts and improve scalability.
4. If the current instruction is the last instruction of the Endpoint of the given path and all basic blocks of the given path are visited, then stop the execution. It means that the given path exists. Otherwise, continue the execution.
5. The given path does not exist if symbolic execution is completed, but the endpoint is not reached, or not all the given basic blocks are executed.

Algorithm 1: Path Feasibility Algorithm.

Input: Path P as a sequence of basic blocks
Output: Boolean indicating if path P is feasible

```
1.  MakeSymbolics()
2.  ExecutedBlocks = EmptySet()
3.  while not SymbolicExecutionComplete() do
4.      Instruction = GetNextInstruction()
5.      ExecuteInstruction(Instruction)
6.      CurrentBlock = GetBasicBlock(Instruction)
```

```
7.    if CurrentBlock in P then
8.       ExecutedBlocks.Add(CurrentBlock)
9.    if IsLastInstructionOfBlock(Instruction) then
10.      NextBlock = GetNextBasicBlock()
11.      // Check if the next block is outside the path
12.      if NextBlock not in P then
13.         EliminateExecution(NextBlock)
14.         continue
15.   // Check if we reached the endpoint
16.   if IsEndpoint(CurrentBlock) and
17.      IsLastInstruction(Instruction) and
18.      AllBlocksExecuted(P, ExecutedBlocks) then
19.      return true
20. return false
```

This is the main algorithm for path feasibility checking. Modifications and improvements are possible based on control flow path specifications. For example, the modified version of this algorithm was integrated into MLH [10], a static analyzer for memory leak detection. Reported paths of detected memory leaks are verified or rejected. This step improved the results of the whole analysis. More detailed information is provided in section VI.

VI. Results

MLH with and without path verification was tested on the Juliet test suite [11]. Juliet contains more than 112,000 test cases, including about 868 memory leak test cases. The results of the evaluation are provided in Table 1. The results show that the path verification mechanism filters out all false positive results of the static analyzer.

Method	True Positives	True Negatives	False Positives	False Negatives
MLH	868	3940	666	0
MLH+Path Verification	868	4606	0	0

Table 1: Static analysis results with and without the path verification algorithm

VII. Conclusion

In this work, we have introduced several methods for enhancing symbolic execution to support fully automatic control-flow path feasibility analysis. By developing automatic symbolization techniques for variables with unknown values, function arguments, return values, and pointer-referenced memories, we have developed a path verification algorithm. The integration of this algorithm into a static analysis tool demonstrates its practical utility: experimental results show significant improvements in analysis accuracy.

Acknowledgment

The work was supported by the Science Committee of the Republic of Armenia, in the frames of the research project 24AA-1B021.

References

[1] J. C. King, "Symbolic execution and program testing," *Association for Computing Machinery*, vol. 19, no. July 1976, p. 385–394, 1976.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 12, pp. 1-38, 2008.

[3] V. Ganesh and D. L. Dill , "A decision procedure for bit-vectors and arrays," *Lecture Notes in Computer Science*, vol. 4590, pp. 519-531, 2007.

[4] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Association*, San Diego, 2008.

[5] "KLEE Symbolic Execution Engine," [Online]. Available: https://klee-se.org/. [Accessed 28 June 2025].

[6] "The LLVM Compiler Infrastructure," [Online]. Available: https://llvm.org/. [Accessed 28 June 2025].

[7] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69-77, 2011.

[8] "LLVM Language Reference Manual," [Online]. Available: https://llvm.org/docs/LangRef.html. [Accessed 28 June 2025].

[9] A. Misonizhnik, S. Morozov, Y. Kostyukov, V. Kalugin, A. Babushkin, D. Mordvinov and D. Ivanov, "KLEEF: Symbolic Execution Engine (Competition Contribution)," in *In Fundamental Approaches to Software Engineering: 27th International Conference*, Luxembourg, 2024.

[10] H. Aslanyan, H. Movsisyan, H. Hovhannisyan, Z. Gevorgyan, R. Mkoyan, A. Avetisyan and S. Sargsyan, "Combining Static Analysis With Directed Symbolic Execution for Scalable and Accurate Memory Leak Detection," *IEEE Access*, vol. 12, pp. 80128-80137, 2024.

[11] "Juliet test suite," [Online]. Available: https://samate.nist.gov/SARD/test-suites. [Accessed 28 June 2025].