A Survey of Symbolic Execution and Its Hybrid Extensions in Real-World Applications

Paper review for CS 477 Formal Software Development Methods course project

Erkai Yu
erkaiyu2@illinois.edu

***Abstract*—Symbolic execution is a powerful program analysis technique used for automated test generation, bug finding, and vulnerability detection. While the idea dates back to the 1970s, recent developments have led to practical applications, particularly when symbolic execution is combined with other techniques like fuzzing and dynamic analysis. This paper reviews the foundational works in symbolic execution and explores state-of-the-art hybrid strategies, discussing their strengths, limitations, and real-world impact.**

***Index Terms*—symbolic execution, fuzzing**

I. Introduction

Symbolic execution has emerged as a cornerstone in the field of program analysis, enabling systematic reasoning about program behavior by treating program inputs as symbolic variables rather than concrete values. This abstraction allows a symbolic executor to explore multiple execution paths simultaneously, making it a compelling tool for detecting bugs, generating test cases, and uncovering security vulnerabilities. Since its inception in the late 1970s [1], [2], symbolic execution has evolved significantly—from a theoretical concept into a practical technique deployed in modern software engineering workflows.

Despite its theoretical appeal, early implementations of symbolic execution faced significant scalability challenges due to path explosion, constraint solving overhead, and limited support for complex language features or real-world programs. However, the last two decades have witnessed a resurgence of interest, driven by advances in constraint solvers, improvements in program instrumentation, and the rise of hybrid analysis approaches that combine symbolic execution with techniques such as fuzzing, concrete execution, and static analysis.

Today, symbolic execution plays a vital role in both academic research and industry. Tools like KLEE [3], SAGE [11], and angr [12] have been successfully applied to large-scale software systems, demonstrating its effectiveness in uncovering subtle bugs and security flaws. Furthermore, the integration of symbolic execution into CI pipelines and automated vulnerability discovery systems reflects its growing real-world relevance.

This paper surveys the development of symbolic execution from its classical formulations to its integration in state-of-the-art hybrid analysis tools. The paper mainly focuses on analyzing the fundamental problems faced by symbolic execution and explaining how some modern integration of symbolic execution with concrete execution techniques handles these problems.

A. Problems with Symbolic Execution

The core idea of symbolic execution is to represent certain variables in a program symbolically instead of assigning them concrete values. By combining this with constraint solving at branch points, symbolic execution can efficiently identify valid inputs that follow specific execution paths.

Although symbolic execution is theoretically efficient, its application to real-world software testing encounters several challenges. According to a survey [10] on symbolic execution, these challenges can be categorized into the following four types.

1) Memory: Symbolic execution struggles with accurately modeling memory when programs manipulate pointers, arrays, or complex data structures. This challenge arises not only from symbolic data values but also from symbolic memory addresses, making precise tracking and reasoning difficult.

2) Environment: Interactions with external code, such as library or system calls, can introduce side effects that influence program behavior. Symbolically modeling all possible outcomes of these interactions is often impractical, leading to incomplete or inaccurate analysis.

3) State space explosion: Symbolic execution faces a combinatorial explosion of execution paths, especially in the presence of constructs like loops and conditionals. This exponential growth in possible states makes it infeasible to explore all paths within reasonable time or computational resources.

4) Constraint solving: While modern SMT solvers can handle many complex constraints, symbolic execution is hindered by the difficulty of solving certain classes of constraints, such as those involving non-linear arithmetic or complex logic, which can severely impact performance and completeness.

B. Paper Structure

This paper presents a review on papers related to applying symbolic execution to real-world application testing. It focuses on how these papers attempt to solve the four critical problems with symbolic execution.

Concolic execution is an extension of symbolic execution, which merges concrete and symbolic execution techniques. Section II reviews some of the early work in this area. A more

recent development involves integrating concolic execution with fuzzing, which is named hybrid-fuzing, is explored in detail in Section III. Finally, Section IV summarizes the discussed techniques and outlines the specific problems each one addresses. Section V presents some future directions related to this topic.

II. Earlier Works on Concolic Execution

Concolic execution (short for concrete + symbolic execution) is a program analysis technique that systematically explores program paths by combining concrete execution with symbolic reasoning. In this hybrid approach, the program is executed with concrete inputs while simultaneously tracking symbolic expressions that represent the behavior of the program over all possible inputs. This enables the generation of new inputs to explore alternative execution paths by negating symbolic constraints. Figure 1 illustrates the idea of concolic execution.

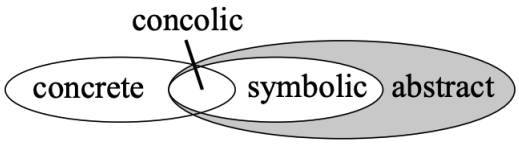

Fig. 1. Concrete and abstract execution machine models [10].

Notable tools and foundational work in this area include DART [5], CUTE [4], EXE [6], and KLEE [3].

A. DART

DART [5] mainly focused on automatically testing software. In order to realize automate unit testing of software, three main techniques were presented:

1) Automated extraction of the interface of a program with its external environment using static source-code parsing.
2) Automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in.
3) Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.

The major strength of DART is that it can perform completely automated tests on any program that compiles. DART is able to dynamically gather knowledge about the execution of the program in a directed search.

Starting with a random input, a DART-instrumented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths. Such way of depth-first searching helps identifying specific execution paths and reduces the chance of path explosion.

For memory modeling, DART represents symbolic variables with their memory addresses, these variables are initialized as NULL or with a random value. For interaction with the environment, DART deems as foreign interfaces all the external variables and functions referenced in a C program along with the arguments for a top-level function. External functions are simulated by nondeterministically returning any value of their specified return type.

B. CUTE

CUTE [4] focuses on resolving the issue with symbolic execution on complex data structures. CUTE uses memory graph as inputs. This approach represents inputs for the unit test using a logical input map that represents all inputs, including memory graphs, as a collection of scalar symbolic variables and then to build constraints on these inputs by symbolically executing the code under test.

An important contribution of CUTE is the idea of separating pointer constraints from integer constraints, which is motivated by the fact that pointer constraints are much more difficult to represent than integer constraints.

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int
f(int v) {
  return 2*v + 1;
}

int
testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
  return 0;
}
```

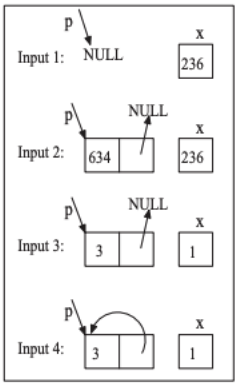

Fig. 2. Example C code and inputs that CUTE generates for testing the function testme [4].

As an example shown in Figure 2, CUTE first nonrandomly generates NULL for p and randomly generates 236 for x, respectively. Incrementally, CUTE assigns concrete values to replace the symbolic variables by solving the path constraints.

CUTE shares many design choices with DART. For example, both attempt to initialize nondeterministic variables with NULL or random values. For path searching, they both use depth-first search. Moreover, the constraint solver used by both concolic execution engines is lp_solve.

C. EXE

EXE [6] proposes many optimization on symbolic execution, aiming for a higher code coverage during tests.

EXE models each symbolic data block as an array of 8-bit bitvectors. The key advantage of using bitvectors is that, much like the C memory blocks they represent, they are effectively untyped. This untyped nature makes it straightforward to express constraints involving the same memory in multiple ways. Each memory read introduces constraints based on the read's static type (e.g., int, unsigned, etc.), but these types are not maintained beyond that point.

Aside from the optimization on fast array constraints, EXE also features the following optimizations:

1) Constraint caching: The result of satisfiability queries are cached and managed by a server process for multiple EXE processes.
2) Constraint independence optimization: The most important optimization by EXE. Treat constraints with disjoint set of operands independently so that the cost of solving satisfiability reduces and more cache hits become possible.
3) Search heuristics: By default, EXE uses depth-first search when forking a branch. To overcome the problem with search stuck in a loop, EXE proposes a heuristic using a mixture of best-first and depth-first search, which significantly reduces the cost of search time.

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p - 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

Fig. 3. A contrived, but complete C program (simple.c) that generates five test cases when run under EXE [6].

Figure 3 shows an example of a C program used by EXE to generate test cases. A limitation of EXE is that it focuses on solving constraints on integers and doesn't generate it on all real-world data types.

D. KLEE

KLEE [3] leverages lessons learned from EXE. Specifically, KLEE presents several optimizations on multiple aspects, and a better support for emulating the external environment.

One notable contribution by KLEE is its use of LLVM IR for constraint generation, making use of LLVM IR makes it much easier to generate constraints and perform symbolic execution on the granularity of basic blocks.

Another notable innovation is that KLEE uses about 2500 lines of C code to implement simple models that understands the semantics of the desired action of roughly 40 system calls. This enables KLEE to redirect system calls to these models such that it has a better support of the environment modeling.

Besides, there are some other optimizations performed by KLEE.

1) Compact state representation: KLEE tracks all memory objects, it can implement copy-on-write at the object level rather than page granulaity, which significantly reduces the memory usage.
2) Query optimization: KLEE performs a series of simplification on expressions before sending to constraint solver, including constraint set simplification, implied value concretization, constraint independence, and counter-example cache.
3) State scheduling: Two searching heuristics are used by KLEE when selecting the state to run. Firstly, random path selection enables KLEE to reach uncovered code with higher possibilities, and avoids state explosion caused by tight loop containing a symbolic condition. Secondly, coverage-optimized search uses heuristics to compute a weight for each state and then randomly selects a state according to these weights, making it more likely to explore new coverages.

Aside from the four papers reviewed above, SAGE [11] and S^{2}E [7] are also representative research efforts on concolic execution.

III. Hybrid Fuzzing: Fuzzing + Concolic Execution

A more recent trend of concolic execution research is to integrate it with fuzzing techniques. In this direction, two insightful papers, QSYM [8] and SymCC [9] are reviewed.

At a high level, every implementation of symbolic execution can be illustrated as Figure 4. In details, QSYM and SymCC takes two different approaches to implement the constraint generation process.

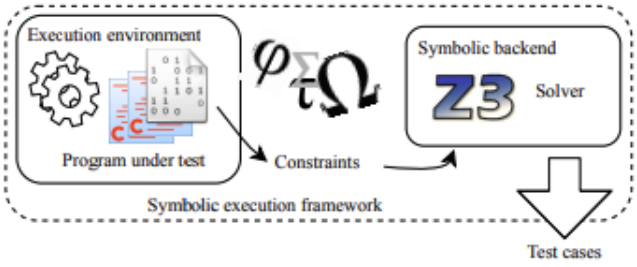

Fig. 4. The building blocks of symbolic execution. The entire system may be encapsulated in a component that handles forking and scheduling [9].

A. QSYM

QSYM [8] introduces a tool combining fuzzing with concolic execution. The main idea behind QSYM is to tightly integrate the symbolic emulation with the native execution using dynamic binary tranlation, making it possible to implement more fine-grained, so faster, instruction-level symbolic emulation.

At a high level, QSYM design can be illustrated by Figure 5 (IR-less), while a more widely adopted design of concolic execution engine can be illsutrated by Figure 6 (IR-based). A major difference between QSYM and those more widely adopted concolic engines like KLEE is that QSYM eliminates the IR layer in between source code and constraint generation.

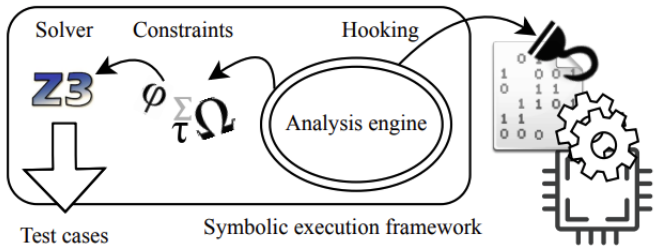

Fig. 5. IR-less symbolic execution attaches to the machine code executing on the CPU and instruments it at run time [9].

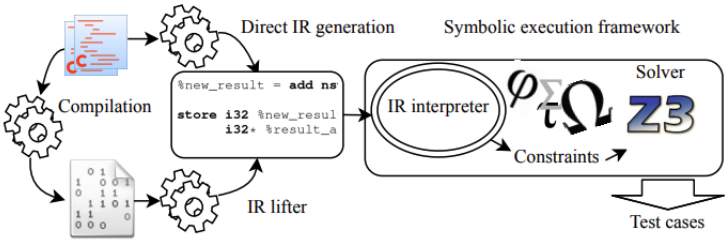

Fig. 6. IR-based symbolic execution interprets IR and interacts with the symbolic backend at the same time [9].

Specifically, QSYM outlines three main performance issues faced by conventional concolic executors used for hybrid fuzzers, and proposes their solutions to each of them.

1) Slow symbolic emulation: Adopting IR makes emulator implementation easy, but it introduces additiona overhead. Meanwhile, IR blocks further optimization or granularity of symbolic execution, most concolic execu tors treat basic blocks as the lowest possible level o performing concrete or symbolic execution, introducing some redundant overhead on instructions that don't neec symbolic execution.
   Solution: Remove the IR translation layer, perform constraint generation on machine code directly. Instead of symbolic execution on basic block level, perform symbolic execution on instruction level with the help of taint analysis.
2) Ineffective snapshot: Snapshot is a way of implementing state forking in concolic execution. When integrated with fuzzing, there are two notable issue with snapshots: 1. fuzzing input does not share a common branch since fuzzer may frequently go explore other paths whenever a path stuck, this makes it impossible to reuse snapshots. 2. snapshots cannot reflect external status such as interactions with file systems.
   Solution: Remove the snapshot mechanism and perform re-execution when taking different branches. This approach sacrifices time for space, but turns out to be efficient enough for real-world application.
3) Slow and inflexible sound analysis: While sound analysis is a critical concept in symbolic execution, it in fact introduces never-ending analysis for complex logic, and could over-constraint a path. As an example shown by Figure 7, if attempts to solve all constraints at line 2 for soundness, the fuzzer may miss the interesting code starting from line 7.
   Solution: Collect an incomplete set of constraints for efficiency and solve only a portion of constraints if a path is overly-constrained. This introduces the risk of generating concrete inputs that don't really cover a execution path, but it is acceptable under the setting of fuzzing, since fuzzer treats these inputs as just regular unsuccessful fuzzing attempts.

```
1 // @funcs.c:221 in file v5.6
2 if ((ms->flags & MAGIC_NO_CHECK_COMPRESS) == 0) {
3   m = file_zmagic(ms, &b, inname); // zlib decompress
4   ...
5 }
6
7 // other interesting code
```

Fig. 7. Collecting complete constraints for complicated routines such as file_zmagic() could prohibit finding new paths [8].

B. SymCC

SymCC [9] has a key contribution on introducing the benefit of integrating concolic execution by compilation rather than performing it by interpretation. At a high level, the design of SymCC can be illustrated by Figure 8.

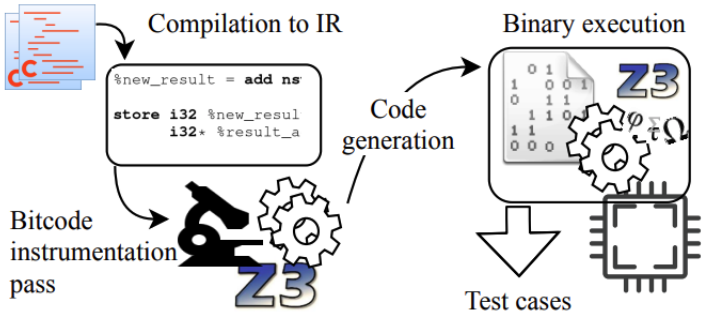

Fig. 8. Compilation-based approach compiles symbolic execution capabilities directly into the target program [8].

SymCC introduces compilation-based symbolic execution, which differs from both conventional IR-based and IR-less symbolic execution. To compile symbolic execution capabilities into a target program, SymCC implements the symbolic backend into a library that can be used by the target program, and it instruments the source code with calls to entry points of the library.

IV. Summary

Symbolic execution, while powerful, is hindered in practice by four critical challenges: memory modeling, environmental interaction, state space explosion, and constraint solving. Over the years, multiple tools have proposed targeted solutions to these problems. Table I summarizes how the six representative research works reviewed by this paper tackle each challenge.

TABLE I
SUMMARY OF HOW TOOLS ADDRESS THE FOUR MAIN CHALLENGES OF SYMBOLIC EXECUTION

Tool	Memory	Environment	State Space Explosion	Constraint Solving
DART	Symbolic variables mapped to memory addresses; initialized to NULL or random	Models external functions with nondeterministic return values	Uses dynamic test generation and depth-first path exploration	Uses lp_solve; handles simple predicates
CUTE	Uses memory graphs; separates pointer vs integer constraints	Same as DART	DFS for path exploration	lp_solve; pointer/integer split
EXE	Bitvector-based symbolic memory; fast memory reads/writes	Limited real-world modeling	DFS + best-first heuristics	Optimizations: caching, independence, reuse
KLEE	Compact memory tracking with copy-on-write	Models 40+ syscalls via C stubs	Coverage-optimized + random path search	Simplification, concretization, caching
QSYM	Instruction-level taint tracking; no IR	Re-execution avoids stale snapshots	Light path exploration; partial constraints accepted	Partial solving for efficiency
SYMCC	Compile-time instrumentation; symbolic library hooks	Tight system integration	Fast fork-free execution	Efficient constraint collection at compile time

V. Future Directions

While significant progress has been made in symbolic execution and its hybrid variants, several open challenges remain:

- **Scalable constraint solving**: Even state-of-the-art solvers struggle with complex path constraints, especially involving non-linear arithmetic or heap-based structures. Future work could investigate domain-specific solvers or approximations that retain soundness guarantees.
- **Real-world system modeling**: Symbolic executors still fall short in accurately emulating complex system interactions (e.g., file systems, networks). Building more comprehensive models or integrating virtualization may help bridge this gap.
- **Intelligent path selection**: Current heuristics are limited; integrating ML-guided search strategies could prioritize more semantically meaningful paths.
- **Hybrid fuzzing synergy**: Hybrid fuzzers like QSYM and SYMCC show promise, but integrating coverage feedback, path prioritization, and mutation strategies more deeply remains underexplored.
- **Parallel and distributed execution**: Path exploration can be embarrassingly parallel. Architectures that support distributed symbolic execution could scale better on modern hardware.

References

[1] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. “SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution.” In Proc. of Int. Conf. on Reliable Software. ACM, 234–245. https://doi.org/10.1145/800027.808445

[2] William E. Howden. 1977. “Symbolic Testing and the DISSECT Symbolic Evaluation System.” IEEE Trans. on Software Engineering (TSE) 3, 4 (1977), 266–278. https://doi.org/10.1109/TSE.1977.231144

[3] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs.” In Proc. 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI’08). USENIX Association, 209–224.

[4] Koushik Sen, Darko Marinov, and Gul Agha. 2005. “CUTE: A Concolic Unit Testing Engine for C.” In Proc. 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE’13). ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. “DART: Directed Automated Random Testing.” In Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI’05). 213–223. https://doi.org/10.1145/1065010.1065036

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. “EXE: Automatically Generating Inputs of Death.” In Proc. 13th ACM Conf. on Computer and Communications Security (CCS’06). ACM, 322–335. https://doi.org/10.1145/1180405.1180445

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. “The S2E Platform: Design, Implementation, and Applications.” ACM Trans. on Computer Systems (TOCS) 30, 1 (2012), 2:1–2:49. https://doi.org/10.1145/2110356.2110358

[8] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, Taesoo Kim. 2018. “QSYM: a practical concolic execution engine tailored for hybrid fuzzing.” SEC’18: Proceedings of the 27th USENIX Conference on Security Symposium. 745-761.

[9] Sebastian Poeplau, Aurélien Francillon. 2020. “Symbolic execution with SYMCC: don’t interpret, compile!” SEC’20: Proceedings of the 29th USENIX Conference on Security Symposium: 11, 181-198.

[10] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, Irene Finocchi. 2018. “A Survey of Symbolic Execution Techniques.” ACM Computing Surveys (CSUR), Volume 51, Issue 3: 50, 1-39 https://doi.org/10.1145/3182657

[11] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. “Automated Whitebox Fuzz Testing. In Proc. Network and Distributed System Security Symp.” (NDSS’08).”

[12] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis.” In IEEE Symp. on Security and Privacy (SP’16). 138–157. https://doi.org/10.1109/SP.2016.17