Symbolic Execution and Applications

Advanced Programming Tools Seminar 2022/2023

Christoph Thiede

Advisor: Patrick Rein

Hasso Plattner Institute, University of Potsdam

Symbolic execution is a dynamic program analysis technique that identifies and examines most execution paths of a program without requiring the programmer to specify inputs. We describe how symbolic execution works, give an overview of its uses and implications for vulnerability detection, testing, and program comprehension, and discuss its practicality and limitations.

1. Introduction

Program analysis encompasses a wide range of techniques and programming tools used to gain insight into the structure and behavior of software systems. We can distinguish between *static* and *dynamic* analysis tools: static analysis tools examine programs through their source code to derive certain information (e.g., dependency analysis, type flow analysis) or to detect occurrences of relevant patterns (e.g., spell checkers, linters). Dynamic analysis tools, on the other hand, examine programs by observing the behavior of running applications (e.g., tests, benchmarks, or assertions).

While static analysis tools typically offer higher performance, they often offer lower precision or selectivity than dynamic analysis. For example, static type flow analyzers cannot reason about metaprogramming constructs such as Python's `getattr` function or JavaScript's `Function.prototype.bind()` method, and static performance prediction tools cannot account for the bottlenecks of an individual hardware configuration. However, while dynamic analysis tools typically provide higher-quality results, they require *context* for running an application and reaching all its different states. Programmers can provide context by preparing a set of possible user interactions or by creating a comprehensive test suite, but this is a costly process, as these artifacts often do not yet exist or cover only a small portion of the code base.

Symbolic execution addresses this problem by automatically discovering and analyzing the execution paths of a program while requiring little or no specification on the part of programmers. The basic operating principle of symbolic execution is to run a program with *symbolic variables* instead of concrete values for its inputs and to fork the execution at each conditional expression (e.g., an `if` statement) whose behavior depends on the concrete assignment of the symbolic variables.

Over the past four decades, programmers and researchers have used symbolic execution for various purposes such as program verification, vulnerability analysis, unit testing, and various program understanding tasks including model generation and reverse engineering.

In this report, we provide an overview of several implementations and applications of symbolic execution.[1] In section 2, we describe the basic operating principle of symbolic execution. In section 3, we provide an overview of the challenges of symbolic execution and existing solution approaches.

[1] We make all artifacts of this work available on GitHub, including the slide deck for our talk and a set of reproducible application examples: https://github.com/LinqLover/symbolic-execution-survey

We present various use cases and tools for symbolic execution in section 4 and examine their impact on research and industry in section 5. In section 6, we discuss limitations and usage considerations for several symbolic execution tools. We classify some alternative approaches in section 7 and finally give a conclusion in section 8. Additionally, we provide details on different implementation strategies (appendix A) and further resources (appendix B).

2. Approach

In this section, we describe the general approach of symbolic execution and introduce key concepts.

Symbolic execution (also abbreviated *symbex* or *SymEx*) attempts to systematically uncover all execution paths of a program [11, 4]. It takes a program in an executable form (e.g., x86 binary, LLVM bitcode, or JVM bytecode) and produces a list of inputs that trigger different code paths in the program. Cadar describes it as a technique "to turn code 'inside out' so that instead of consuming inputs [it] becomes a generator of them" [10].

To this end, the *symbolic execution engine* (or *symbolic executor*) runs the program with its normal execution semantics but assigns a special *symbolic variable* to each input:

- As the program performs arithmetic or logic operations on symbolic variables, *symbolic expressions* are created that can be composed and stored in the *symbolic memory* (also *symbolic store*) of the program execution.
- When the program hits a branch instruction such as an `if` statement that depends on a symbolic condition, the executor decides the possible results of the depended expression (e.g., `true` or `false`). If the expression has multiple possible solutions, the entire execution is *forked* into multiple *execution paths* for each solution, and each fork is assigned a new *path constraint* on the assignment of its symbolic variables in the form of a first-order Boolean expression.

Together, the symbolic memory and the constraint set of an execution path define its *symbolic state* (see fig. 1 for the symbolic execution of the `foo()` function in listing 1) [11, 4]. All execution paths are contained in the *symbolic execution tree* of the program where each node represents a symbolic state and each edge represents a new path constraint. For each leaf, i.e., each completed execution path, the symbolic execution engine can generate a concrete set of input values from the constraint, and programmers can use these concrete inputs to reproduce the same execution path in a regular non-symbolic context [11, 9].

A critical component of any symbolic execution engine is a *satisfiability modulo theories solver* (*SMT solver*) which decides whether a symbolic branch condition can be satisfied and generates a concrete solution for a constraint set [4, 42]. SMT solvers are a special kind of *SAT solvers* that test the *satisfiability*

of a constraint set *modulo* (“within”) a set of *theories*. Theories are axiomatic systems for domain-specific algebras that enable the solver to reason not only about Boolean logic but also about predicates involving various data types. Common theories describe arithmetic, bitwise operations/integers with overflow semantics, or strings.

Listing 1: A simple function in C that can be executed symbolically.

```
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
}
```

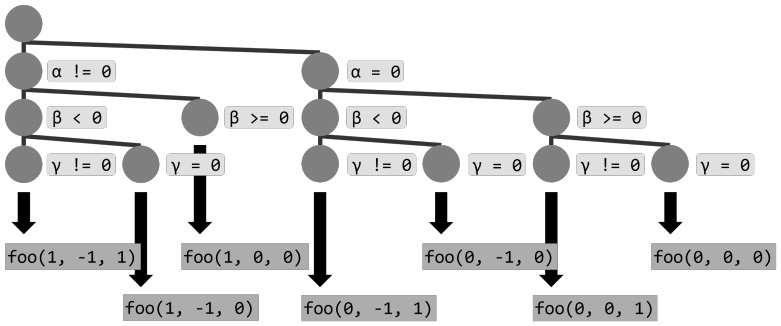

Figure 1: The symbolic execution tree and concrete input sets for each path after analyzing the `foo()` method from listing 1. The symbolic variables α, β, γ represent the assigned values for the input variables `a`, `b`, and `c`, respectively.

3. Technical Challenges

Despite the simple core concept of symbolic execution, several aspects of typical programs impede the exploration of all execution paths in a reasonable amount of time. In this section, we provide an overview of common implementation challenges and solution strategies. Listing 2 gives examples for each challenge. A more detailed description of modern implementation strategies is given in appendix A.

Listing 2: A selection of functions in C that illustrate common challenges of symbolic execution.

```c
int count(int array[], int value) {
    int c = 0;
    // Path explosion!
    for (int i = 0; i < 100; i++) {
        if (array[i] == value) c++;
    }
    return c;
}

int product(int n) {
    int p = 1;
    // Infinite path explosion!
    for (int i = 1; i <= n; i++) {
        p *= i;
    }
    return p;
}

int read(char *fname) {
    // Blackbox!
    FILE *f = fopen(fname, "r");
    if (f == NULL) return -1;
    int x;
    // Blackbox!
    fscanf(f, "%d", &x);
    // Blackbox!
    fclose(f);
    return x;
}

int access(int index) {
    int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Symbolic pointer!
    if (array[index] == 5) return 1;
    return -1;
}

int check(char *password) {
    // Unsolvable constraint!
    if (superFastHash(password) == 0xdeadbeef) return 42;
    return 0;
}
```

Path explosion, space explosion. For common programming constructs, the total number of execution paths can be superpolynomial in the number of conditional statements. For example, a program that checks a symbolic condition n times within a loop or recursive function call can produce up to 2^n execution paths. If the break condition itself is symbolic, each check of the condition can generate additional execution paths, leading to an infinite size of the execution tree. In reality, this results in an impractical performance even for the symbolic execution of many small programs [11, 4].

Common solutions to this problem include restricting and prioritizing execution paths (appendix A.3.1), static symbolic execution (appendix A.1.3) and function summarization (appendix A.3.2), and selective symbolic execution (appendix A.2).

Environments, blackboxes. Traditional symbolic execution assumes a whitebox program where the symbolic executor has access to all instructions. However, practical programs often interact with blackboxes, including system calls to the operating system (OS) or primitive calls to the virtual machine (VM) [3, 4]. At the hardware level, all I/O operations are blackboxes, including file system accesses, network communication, and hardware-accelerated computation [10, 4]. For symbolic executors that operate on the source code or intermediate representations of programs, even precompiled library binaries without sources are a blackbox [40]. Importantly, event-driven applications that are built around a framework following the inversion of control principle [29] can only be executed symbolically in the context of the framework's implementation which is often complex (see path explosion) or unknown [40]. Finally, concurrent programs typically rely on a blackbox scheduler that is part of the OS or the VM [67, sec. 7].

This issue is typically addressed by using framework- or user-provided environment models or by lifting binaries to symbolically executable representations (see appendix A.4).

Symbolic pointers. When a pointer has a symbolic address value, naive symbolic execution engines cannot efficiently reason about the result of dereferencing that pointer [4, sec. 3, 67, sec. 6.3]. In higher-level programming constructs, this problem corresponds to array or dictionary accesses using a symbolic index or key, or to method lookups on an object of a symbolic class or prototype [4, sec. 3].

Advanced symbolic execution engines meet this challenge by storing conditional expressions or symbolic pointers in the memory or by restricting the resolution of symbolic pointers at the expense of completeness (appendix A.5).

Solver limitations. Constraint solving is an NP-complete problem for which no generic efficient solution algorithm is known [11, sec. 4.2, 67, sec. 3]. In practice, this makes it difficult to decide path conditions or generate inputs for constraints that contain nonlinear arithmetic expressions or one-way functions such as prime products or hashes [5].

SMT solvers employ strategies such as parallelized and hardware-accelerated computation, optimized algorithms, caches, and other heuristics (appendix A.6).

4. Applications

In the following, we describe three common use cases for symbolic execution: program analysis, testing, and reverse engineering.

4.1. Program Analysis

Through symbolic execution, programmers can automatically search large parts of a software system for certain behavioral conditions or patterns. For example, they can find all statements that violate an invariant, or they can detect potentially unreachable code [46, sec. 5]. It is also possible to infer new invariants from the symbolic execution tree [17].

A very popular application of symbolic execution is *bug detection* which allows programmers to scan programs for edge cases or even security vulnerabilities. In this context, several definitions of security violations can be used: in the simplest form, symbolic execution tools can find all paths that lead to an irregular abort or crash of a program [9, 21, 13]. In advanced scenarios, tools detect suspicious behavior patterns such as accesses to uninitialized memory, or they install an agent that attempts to exploit potential attack vectors such as remote code execution [2]. Some tools perform *exploit generation* by creating programs or sets of input values that programmers can use to reproduce the detected vulnerabilities (see fig. 2) [9, 13, 2].

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 2: Original exploits in the GNU Core Utils library that cause unexpected program crashes, generated by the symbolic execution engine KLEE [9].

4.2. Testing

Analogous to exploit generation, tools can also use symbolic execution to *generate inputs* or *crash tests* for all covered execution paths of a program [52, 1]. Many tools generate code that contains only the arrange and act logic of a test but leave it to programmers to add assertions [52]; however, some approaches incorporate inferred invariants to generate assertions [17, 44].

Symbolic testing frameworks go beyond undirected test generation by allowing programmers to write tests that operate on symbolic variables rather than concrete values [62, 23] (see listing 3). Instead of specifying actual example inputs and expected outputs, programmers declare preconditions (*assumptions*) on symbolic variables and postconditions (assertions) on the results from the system under test. The testing framework executes these tests symbolically and reports any inputs or input constraints that violate the symbolic assertions. Programmers can also use symbolic tests to *compare different implementations* by their behavior, e.g., for detecting breaking changes in an interface [51].

Listing 3: A symbolic unit test written in the parametrized unit testing framework Pex for C# [62]. The test asserts that for each pair of non-empty version strings, the method `Versions.Compare()` returns either `-1`, `0`, or `1`.

```
[PexMethod]
public int CompareTest(string version1, string version2)
{
    PexAssume.IsNotNullOrEmpty(version1);
    PexAssume.IsNotNullOrEmpty(version2);

    int result = Versions.Compare(version1, version2);

    PexAssert.IsTrue(new[] { -1, 0, 1 }.Contains(result));

    return result;
}
```

A similar style of testing is *contract programming* where programmers specify pre- and postconditions for individual functions that can be validated using symbolic execution (see fig. 3) [52]. Unlike testing frameworks, contracts are typically attached directly to the code under test, and many frameworks expect declarative specifications that are less suitable for imperative arrangements or disjoint invariants.

Symbolic testing tools can assist programmers in improving the code coverage of their systems and provide them with faster feedback to discover bugs while reducing the required cost of writing tests. However, programmers incur the usual overhead of writing tests because they must provide fakes, mocks, and stubs for external units. The need for formal pre- and postconditions further increases this overhead: typically, systems are based on many implicit assumptions, and programmers are forced to

```
class Node:
    def __init__(self, value: int, left: Optional['Node']=None, right:
    Optional['Node']=None):
        self.value = value
        self.left = left
        self.right = right

    def rotate_left(self):
        """
        pre: self.right is not None
        post: __old__.self.count() == __return__.count()
        false when calling rotate_left(Node(0, left = None, right =
        Node(0, left=None, right=Node(0)))) (which returns Node(0,
        left=None, right=Node(0))) CrossHair
        View Problem (Alt+F8)    No quick fixes available
        new_root = self.right
        self.right = new_root.left
        new_root.right = self
        return new_root
```

Figure 3: Screenshot of a method contract expressed as a Python docstring in Visual Studio Code. The symbolic execution tool CrossHair reports live feedback on the contract after saving the file and displays a violation of the specified postcondition [52].

explicate them when deciding which edge cases their application should handle. When programmers have to deal with generated tests, typical code generation concerns may arise, including readability and maintainability of generated code and concretized values (see appendix A.6). Finally, the performance and theories of symbolic executors are limited, resulting in long wait times and incomplete coverage for complex programs (see section 3).

4.3. Reverse Engineering

Another category of symbolic execution tools supports programmers in *reverse engineering* tasks by allowing them to explore the captured behavior of executed units through alternative representations.

Microsoft IntelliTest summarizes the behavior of a selected method by constructing a table of input arguments, return values, and optionally user-specified expressions from the symbolic execution paths (see fig. 4) [27, 62]. Programmers can benefit from this to understand methods and explore their edge cases without having to study the implementation.

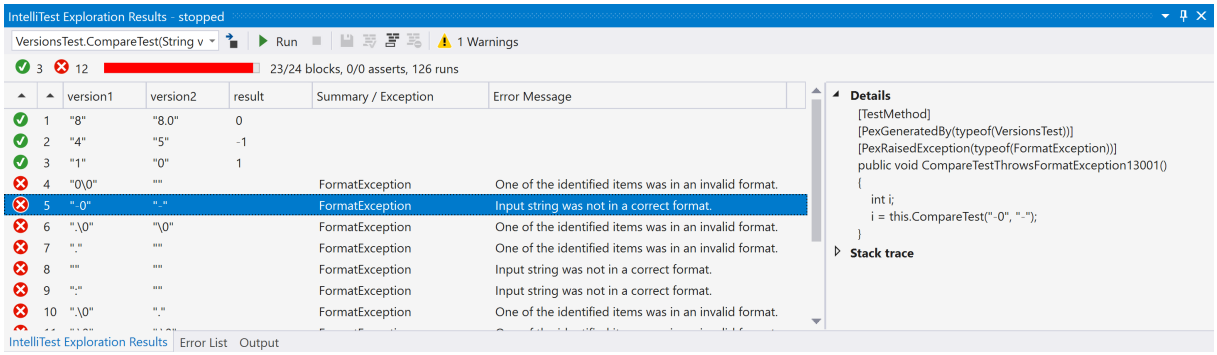

Figure 4: Screenshot of the IntelliTest explorer in Microsoft Visual Studio summarizing the behavior of the method `Versions.Compare()` [27]. Each result represents a different equivalence class of inputs (arguments) and outputs (return value or raised exception) that are caused by the same code path. On the right, the test code for reproducing the example (generated using Pex, see section 4.2) is shown.

Other tools use symbolic execution to aid program understanding through *symbolic execution debugging* where programmers can debug a program without providing concrete entrypoints (see fig. 5) [26, 60, 28]. All variables are initialized symbolically, and during debugging, programmers can interactively advance and explore a symbolic execution tree of the selected method. This style of debugging can help programmers evade irrelevant or distracting context, but they may also experience symbolic expressions in the inspected program state as too abstract and sophisticated. The size of the symbolic execution tree can be overwhelming, and its structure can be confusing because it mismatches common control flow graphs for patterns such as loops. Thus, the value of symbolic execution debugging

tools is likely maximal for codebases with high complexity and poor readability; for instance, several disassembly tools support symbolic execution debugging [28].

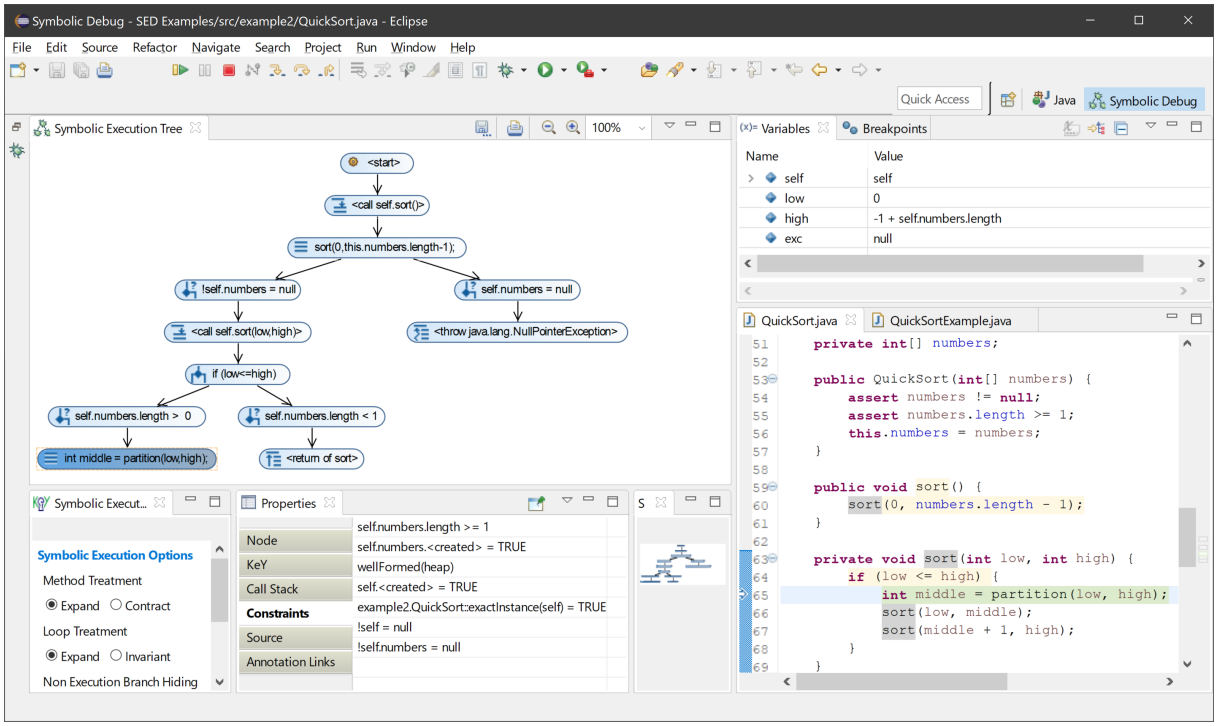

Figure 5: Screenshot of the *Symbolic Execution Debugger* (SED) for Eclipse while exploring the execution tree of a `QuickSort` implementation in Java [26, 60]. By selecting a node in the tree, programmers can inspect the associated symbolic state in the remaining panes and advance the execution of the code path step by step.

Like symbolic testing tools, symbolic reverse engineering tools suffer from the limited performance and coverage of execution engines, and programmers need to specify additional context to explore code that interacts with an external environment or relies on implicit assumptions.

5. Impact

In this section, we review the adoption of symbolic execution and its impact on particular software systems.

5.1. Adoption of Symbolic Execution Tools

As an example, we evaluate the adoption of symbolic execution tools in the open-source community by considering a selected sample of repositories on GitHub that use symbolic execution concepts, are actively maintained, and have above-average popularity (at least 200 stars). The largest group of symbolic execution tools matching these criteria are frameworks and libraries that enable programmers to perform a dynamic analysis of their systems at a low or medium level of abstraction. For example, many tools offer APIs for symbolic execution of particular instructions or methods and for inspection of the resulting execution tree [56, 18, 50]. A large number of tools provide means for semi-automated vulnerability detection and exploit generation [43, 30]. Most tools operate on binaries [18, 50, 30, 48, 35, 15]; however, some solutions target JavaScript applications [55] or smart contracts running in the Ethereum VM [41, 43]. Some tools also use symbolic execution to detect suspicious behavioral patterns such as null pointer propagation or object layouts and accesses that prevent particular JIT optimizations [55], to verify automated optimizations [35], or to help programmers explore disassembled code [28].

To our knowledge, few widely adopted symbolic execution tools assist programmers with writing tests [52, 23]. An exception is IntelliTest which has been integrated into Microsoft Visual Studio since 2015 and is thus available to a potentially large number of users [27].[2]

5.2. Impact on Vulnerability Analysis

Over the past two decades, symbolic execution tools have had a particular impact on vulnerability detection (see section 4.1). Microsoft used its in-house concolic execution engine SAGE to find more than 30% of all bugs in Windows 7 that they fixed before the release, has later established the tool as a standard part of its internal testing pipelines, and is continuously running it on more than 200 machines 24/7 to reduce the number of exploits in several products [22, 7].

In research, the popular open-source library GNU Core Utils (which includes tools such as `cat`, `tee`, and `wc`) has been established as a benchmark for evaluating the performance of symbolic execution engines. Originally, the entire library consisted of 89 binaries with 72 kLOC (thousands of lines of code), and the test suite had a coverage of 67.6% LCOV (percent of lines covered) [9]. In 2008, KLEE found 56 previously unknown bugs and crashes in the library within 1 hour per binary and generated crash tests that increased the total code coverage to 84.5% (see fig. 2) [9]. Later, other tools detected further bugs in the library while reducing operational costs [39, 32]. In 2012, Mayhem achieved a coverage of 97.6% LCOV in a subset of the library within 1 hour per binary [13].

[2] IntelliTest is available in the Enterprise edition of Microsoft Visual Studio. More than 150,000 organizations have subscribed to any commercial edition of Visual Studio [16], and Microsoft provides free access to Visual Studio Enterprise for students through the *Azure Education Hub* program [59]. However, the actual number of users working with C# and the .NET Framework is unknown.

Researchers have also demonstrated the potential of vulnerability detection in other software systems: Bugs and vulnerabilities have been found in BusyBox [9], the Linux kernel [13], Minix [9], and Windows [13], among others. MergePoint achieved particularly remarkable results by discovering more than 11,000 bugs in the entire Debian kernel [3] (which consists of more than 33,000 binaries and 650 MSLOC (millions of source lines of code) [58]). This analysis took them 18 CPU-months but they benefited from massive parallelization across multiple machines. They also estimated that the operational cost of renting virtual servers in a data center would be $0.28 per bug discovered, showing that symbolic execution for security analysis can be financially worthwhile given the potentially high cost of publicly disclosed zero-day exploits [22, 12, 47].

6. Discussion

Symbolic execution has proven useful for several applications such as bug detection, test generation, or program exploration. Programmers can improve the test coverage of their systems and gain faster insights into the behavior and edge cases of programs while reducing the effort and distraction of considering context. Still, symbolic execution has several limitations and costs: depending on the complexity of an application, the performance may be insufficient for interactive response times or even require hours to days and significant financial resources. The coverage of execution paths is limited and depends on the coupling of a software to blackboxes and environments, on the algebraic complexity of algorithms, and on the use of lookup constructs such as pointers or large dictionaries. In many cases, programmers are required to provide additional configuration to model blackboxes or to specify some context, and generated artifacts may suffer from poor readability or sustainability.

Thus, programmers must evaluate the suitability of symbolic execution tools for particular software projects by weighing these costs against the potential value of the tools. For example, for safety- or security-critical applications involving domains such as finance or user data, programmers may prioritize high quality and dependability of their software over a fast and inexpensive development process. The value of tools that aim to improve the programming experience and aid program understanding depends primarily on the implementation complexity of projects.

7. Related Tools

Besides symbolic execution, programmers can also consider alternative approaches and tools. *Static analysis tools* [24, 19, 33] and *fuzzing tools* [68, 20] typically provide significantly better performance but deliver lower precision and selectivity because they cannot systematically scan all execution paths. Nevertheless, these techniques are more popular and, for many languages and environments, may surpass symbolic execution in terms of the number and maturity of available solutions.

Generative AI software such as GitHub Copilot or ChatGPT forms another category of tools that can practically assist programmers in analyzing programs or generating tests, but their results often suffer from poor and unreliable quality especially for less popular domains, and programmers tend to experience the workflow as cumbersome and unsatisfactory [6, 65, 57].

8. Conclusion

Symbolic execution is a dynamic program analysis technique that systematically identifies and examines most of the execution paths of a program without any concrete inputs from programmers. It is used in several types of programming tools to assist programmers with tasks such as vulnerability detection, exploit generation, testing, and program comprehension, and it has been successfully adopted by the industry for both internal purposes and commercial products. Still, performance and completeness of symbolic execution are limited, and programmers face a potential overhead for specifying required context about a program's environment.

A. Implementation Strategies

In this appendix, we describe several modern implementation strategies and considerations that address existing challenges (section 3) and optimize symbolic execution.

A.1. Taxonomy of Executors

Symbolic executors can follow alternative design decisions that make different tradeoffs between implementation cost, runtime performance, and systemic limitations.

A.1.1. Online and Offline Symbolic Execution

Traditional symbolic execution as described in section 2 is classified as *online symbolic execution* where each covered execution path is executed exactly once and the symbolic state is *cloned* at each symbolic branch condition [4, sec. 2.4]. On the other hand, *offline symbolic executors* do not fork the execution but instead re-run an execution path for each selected branch [4, sec. 2.4].

While online symbolic execution processes fewer instructions in total, it maintains many cloned states in the memory which may be infeasible or require frequent swapping to slower storage [4]. However, online symbolic executors can reduce memory consumption by using copy-on-write data structures to share common symbolic states among multiple execution paths [4]. Offline symbolic execution does

not share any resources between multiple execution paths, reducing the implementation overhead for resource isolation (see also appendix A.4).

Hybrid symbolic executors combine both online and offline execution styles to benefit from the improved performance of online execution until they reach a memory boundary [13]. At that point, they switch to an offline execution strategy and re-execute further execution paths from the beginning.

A.1.2. Dynamic Symbolic Execution

Traditional symbolic execution is unable to handle blackboxes: since any variable may be assigned a symbolic expression, the executor cannot pass it to a foreign system (e.g., when making a system call to the OS). *Dynamic symbolic execution* (DSE) overcomes this limitation by combining symbolic execution with normal concrete execution that assigns concrete values to variables [11, 4].[3]

One form of DSE is *execution-generated testing* (EGT) which interleaves symbolic and concrete execution [10]. When a blackbox is invoked, the executor concretizes any symbolic arguments before passing them to the blackbox by requesting a solution for the current constraint set from the SMT solver.

Concolic execution (a portmanteau of "concrete" and "symbolic") is another form of DSE that executes a program in both the concrete and symbolic styles simultaneously [54, 21]. Concrete execution maintains a concrete assignment of all variables and is used to direct the control flow and invoke blackboxes. Symbolic execution accompanies concrete execution to collect the symbolic constraints for the conditional branches taken. Once an execution path has been completed, the executor creates further paths by negating single constraints from the collection and generating concrete input values for their execution. As a consequence, the analysis can center around practically relevant execution paths: for example, the concolic execution of a file parser could start with a real sample file and then explore edge cases in the parser for degenerate files by negating constraints [22].

Concolic execution shifts the usage patterns of the SMT solver [4, sec. 2]: the solver is only requested once per execution path to generate constraints, eliminating the context-switching overhead of invoking the solver at each conditional instruction. If the solver is unable to find a solution for a constraint set, symbolic execution can skip some execution paths instead of inevitably aborting [11, sec. 3]. In practice, concolic execution is implemented by instrumenting the program to collect constraints and running the instrumented program offline in the regular OS or VM [53, 48]. This also improves performance and reduces implementation costs by avoiding the indirection of a separate interpreter.

[3] The terms "dynamic symbolic execution" and "concolic execution" are used inconsistently in the literature. In this report, we adhere to the taxonomy of Cadar et al. who use "dynamic symbolic execution" as an umbrella term for execution-generated testing and concolic execution [11] (instead of the taxonomy of Baldoni et al. who use both terms in the opposite way [4]).

However, neither EGT nor concolic execution uncovers conditional branches inside blackboxes. To improve the coverage of programs that interact with blackboxes, programmers can alternatively specify memory models (see appendix A.5).

A.1.3. Static Symbolic Execution

Static symbolic execution (SSE) addresses the path explosion problem by deriving a static transformation of program parts instead of executing them [3, 4]. SSE uses *function summary* and *loop summary* techniques to translate a function or a group of instructions, respectively, into an equivalent conditional symbolic expression (corresponding to a mathematical piecewise-defined function). For example, *compositional symbolic execution* analyzes individual program parts in isolation by treating each function call as a new symbolic expression and translates them into pairs of preconditions and postconditions [67, sec. 5]. SSE reduces the cost of executing a program multiple times and deciding or negating many constraint sets but increases the pressure on the SMT solver to handle conditional expressions; however, recent advances in solvers make this a worthwhile shift (see also appendix A.6) [3]. Two limitations of SSE are that it cannot handle blackboxes and that it cannot follow any control flow patterns beyond basic conditional or loop jumps [3].

Veritesting is another technique that combines the strengths of SSE and concolic execution: it runs a program in SSE mode as long as possible and falls back to concolic execution when it hits a function boundary, a blackbox, or some other limitation of SSE. From concolic mode, veritesting returns to SSE mode at the next supported instruction. Veritesting outperforms regular DSE by more than 70% [3].

A.1.4. Backward Symbolic Execution

Another approach to symbolic execution is *backward symbolic execution* (BSE) which explores the instructions and execution paths of a program in reverse order [37, 4, sec. 2.3]. One application of BSE is post-mortem debugging [14]. To identify the possible callers of a method, BSE uses a statically generated control-flow graph (CFG). Since methods can have multiple callers, BSE is also affected by the path explosion problem [4, sec. 2.3].

A.2. Selective Symbolic Execution

Complete symbolic execution of a software system can take hours up to months [3], but in many cases, programmers are only interested in the results of analyzing certain subsets or behaviors of the system. *Selective symbolic execution* includes several methods for selecting parts of the system for analysis [15]. In its general form, programmers can specify a whitelist or blacklist of units (e.g., modules, classes, or methods) to be analyzed. All unselected parts are treated as blackboxes: outside of the selected parts,

the program is executed concretely without generating new execution paths. However, this approach reduces the completeness of the analysis; for example, not all possible return values or side effects of a call to a skipped function will be covered. *Chopped symbolic execution* can partially restore the lost completeness by statically analyzing the behavior of skipped functions [64].

Selective symbolic execution is well combinable with compositional symbolic execution where individual units can be analyzed in isolation (see appendix A.1.3).

Shadow symbolic execution is another form of selective symbolic execution that selects parts for symbolic execution based on the changes to a software system since the previous run of the symbolic execution engine. In the context of continuous integration where tests and analysis are run on each new revision, this can significantly reduce the execution time or improve the coverage within a given time limit [31].

With *preconditioned symbolic execution*, programmers can manually specify constraints on inputs or program behavior [2, 4, sec. 5.5]. For example, they can limit the size of certain buffers to 100 bytes, disallow non-ASCII characters, or provide regular grammars for generated inputs.

Instead of binary filters, programmers can also specify a priority list of different program parts. In *directed symbolic execution* or *shortest-distance symbolic execution*, the selected program parts are analyzed first, followed by adjacent program parts based on their distance from the selected parts regarding the static CFG of the system [37]. Similarly, *lazy expansion* explores the call graph of a test method top-down by initially treating all function calls as blackboxes and descending into them later [38].

A.3. Path Management

A major challenge in symbolic execution is path explosion since the number of execution paths can grow exponentially with the number of conditional branches or even infinitely. Practical symbolic execution engines typically combine multiple strategies to deal with the large size of the execution tree.

A.3.1. Path Selection

Path selection (also *branch prioritization* or *search strategies*) refers to maximizing the number of *relevant* execution paths within a given time limit [34, 4, sec. 2.2]. A criterion is defined for ordering execution paths, and the offline executor selects the next path based on that criterion.

Naive path selection strategies include *depth-first search* (DFS) and *breadth-first search* (BFS) of the execution tree [34, 49]. DFS is a prime example of search strategies that are not loop-safe: if the executor encounters an infinite subtree (e.g., a loop or recursive call with a symbolic break condition), the search

will never complete, and the executor will never continue with the exploration the remaining tree. BFS, on the other hand, involves a maximum memory overhead for preserving incompletely explored parts of the execution tree and is unlikely to explore relevant deeply nested subtrees early [34]. *Random search* combines the strengths of DFS and BFS by randomly selecting execution paths but is still not loop-safe [34].

Other strategies assign scores or weights to different execution paths. *Generational search* maximizes code coverage by calculating the score of each new path from the relative coverage increase of the previous (original) path [21, 49, p. 24f.]. Execution paths can also be prioritized based on the mutation coverage of the generated tests or the coverage of the static CFG [49, p. 18ff.]. Some strategies use heuristics to predict the number of defects in subtrees based on certain instruction types or the historical density of defects in a unit (assuming not a normal distribution of the defect density across units but peaks in units written by particular authors on particular days) [13, 2, 49, p. 26]. Other heuristics attempt to detect repetitive loop iterations or recursive calls or use *fitness functions* for symbolic variables to prioritize solutions for them that trigger certain branches [11, p. 9, 49, p. 27].

A.3.2. Path Summarization

Path summarization strategies analyze individual functions or loops statically to avoid path explosion. Many of these strategies overlap with techniques employed for SSE (appendix A.1.3).

A.3.3. Path Pruning and Merging

Path pruning or *path subsumption* strategies detect identical execution paths and eliminate duplicates [4, sec. 5.4, 67, sec. 4.2]. By storing pre- and postconditions, they can also detect *equivalent* execution paths that differ only in side effects or values irrelevant to the remaining control flow.

Path merging strategies avoid path multiplicities by combining similar execution paths [32, 3, 4, sec. 5.6, 67, sec. 4.3]. The state of merged execution paths contains conditional constraints and conditional expressions in the symbolic store. To balance the reduced execution cost against the additional load on the SMT solver, they employ several heuristics. Inter alia, these heuristics take into account the subsequent instruction types and the static CFG of a program to predict the solver cost.

A.3.4. Parallelization

Depending on the architecture of the software under investigation, the available input seeds (e.g., from a test suite), and the resulting shape of the execution trees, symbolic execution can be accelerated by distributing the execution tree to multiple processors or multiple machines [8, 3].

A.4. Environment Models

To avoid blackboxes, programmers can provide models to emulate parts of the environment [4, sec. 4]. Similarly to unit testing, they can configure *fakes* and *stubs* that are whiteboxes to the symbolic executor [25]. For example, they can replace the actual file system with a virtual file system or redirect all requests to a remote server to a stub object. Fakes can model uncertainties about the original environment by creating new symbolic variables, e.g., to represent the contents of a file or to decide whether a server is reachable. For large blackboxes, some tools offer approaches to automate model generation [4, sec. 4].

For event-driven applications that pass control to a framework, such as a GUI framework for web pages or mobile applications, another approach is the compositional analysis of individual call targets that the application exposes to the framework [40]. These *sub-call graphs* can then be composed based on regular grammars that describe possible call sequences.

Other approaches attempt to disclose blackboxes by lifting binary code to a higher-level representation such as LLVM bitcode that is compatible with the symbolic executor [9] or by employing virtualization techniques to emulate critical instructions [15].

A.5. Memory Models

To handle symbolic pointers properly, symbolic execution engines must consider all possible symbolic states that can result from dereferencing a pointer (or looking up an array element, respectively) [11, 4, sec. 3]. Since this is prone to path explosion, an alternative is to store a conditional expression in the symbolic memory that represents all possible results and can be handled from a single execution path (similar to path merging, see appendix A.3.3). For write operations, it is also possible to use the unresolved symbolic address as a key in the symbolic store. However, for read operations, this approach would frequently lead to an explosion of the symbolic store, given the typical address space of programs.

Other approaches set limits on the address space [4, sec. 3]. For instance, symbolic execution engines can restrict symbolic pointers to all previously allocated addresses plus a canonical null pointer, or they can immediately report an error for an execution path that accesses unallocated space as this is considered an unsafe practice. Alternatively, offline engines can randomly concretize symbolic pointers, dynamically allocate new space, and (optionally lazily [67, sec. 6]) initialize it with new symbolic variables or data structures. Similarly to preconditioned symbolic execution, engines can also allow programmers to specify custom constraints on newly initialized data.

Some offline symbolic executors skip constraints that contain symbolic pointers during path generation but omit a part of the execution tree [4, sec. 3]. Engines that follow a hybrid strategy concretize some

symbolic addresses only depending on the access type (read or write instructions), the number of possible results, and other heuristics.

A.6. Constraint Solvers

Although there is still no known generic efficient algorithm for the satisfiability problem, the performance of SMT solvers has improved significantly over the last decades. This is due to advances in the underlying theories and algorithms for constraint reduction, deferred computation, and reuse of solutions. Solvers also benefit from increased hardware performance, parallelization, and acceleration using GPUs [11, 45].

Another challenge lies in generating intuitive data during concretization. For example, some solvers favor small numbers over random integers, or they stick to alphanumeric characters for strings and maximize repetition [63, sec. 6.3].

A.7. Trends in Symbolic Execution

Today's major challenges in symbolic execution include further optimizations for the path explosion (appendix A.3) problem and solver performance (appendix A.6). Several approaches use the results of efficient static analysis to assist the symbolic execution engine with these challenges. Another open problem is the analysis of concurrent programs where the number of possible execution orders can be exponential in the number of instructions [67, sec. 7].

B. Further Reading

Here, we recommend a selection of additional resources for studying the field of symbolic execution in greater depth.

Online resources. [36] provides a collection of papers, educational materials, and implementations of symbolic execution techniques. [61] gives a more detailed overview of relevant papers. [66] summarizes the history of symbolic execution engines and solvers in the form of graphical timelines.

Surveys. [11] gives a first overview of symbolic execution approaches and implementations. [4] provides a comprehensive review of implementation techniques and considerations. [67] describes some present challenges and trends.

To address the path explosion problem, search strategies are a much-noticed solution approach (see appendix A.3.1). [34] and [49] survey this subfield of symbolic execution in detail.

References

[1] Elvira Albert et al. “Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-based Instance, and Actor-based Concurrency”. In: *Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Advanced Lectures 14*. Ed. by Marco Bernardo et al. Vol. 8483. Lecture Notes in Computer Science. Bertinoro, Italy: Springer, 2014, pp. 263–309. DOI: 10.1007/978-3-319-07317-0_7.

[2] Thanassis Avgerinos et al. “Automatic Exploit Generation”. In: *Communications of the ACM* 57.2 (2014-02), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/2560217.2560219.

[3] Thanassis Avgerinos et al. “Enhancing Symbolic Execution with Veritesting”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1083–1094. ISBN: 9781450327565. DOI: 10.1145/2568225.2568293.

[4] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018-05), pp. 1–39. ISSN: 0360-0300. DOI: 10.1145/3182657.

[5] Sebastian Banescu et al. “Code Obfuscation against Symbolic Execution Attacks”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 189–200. ISBN: 9781450347716. DOI: 10.1145/2991079.2991114.

[6] Shraddha Barke, Michael B. James, and Nadia Polikarpova. “Grounded Copilot: How Programmers Interact with Code-Generating Models”. Preprint. 2022. DOI: 10.48550/ARXIV.2206.15000.

[7] Ella Bounimova, Patrice Godefroid, and David Molnar. “Billions and Billions of Constraints: Whitebox Fuzz Testing in Production”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 122–131. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/main-may10.pdf.

[8] Stefan Bucur et al. “Parallel Symbolic Execution for Automated Real-world Software Testing”. In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys ’11. Salzburg, Austria: Association for Computing Machinery, 2011, pp. 183–198. ISBN: 9781450306348. DOI: 10.1145/1966445.1966463.

[9] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. “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*. Vol. 8. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 209–224. DOI: 10.5555/1855741.1855756.

[10] Cristian Cadar and Dawson Engler. “Execution Generated Test Cases: How to Make Systems Code Crash Itself”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 2005-08, pp. 2–23. ISBN: 978-3-540-28195-5. DOI: 10.1007/11537328_2.

[11] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Commun. ACM* 56.2 (2013-02), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795.

[12] Michael del Castillo. *The DAO Attacked: Code Issue Leads to $60 Million Ether Theft*. 2016-06-17. URL: https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft/ (visited on 2023-01-23).

[13] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE. Los Alamitos, CA, USA: IEEE Computer Society, 2012-05, pp. 380–394. DOI: 10.1109/SP.2012.31.

[14] Ning Chen and Sunghun Kim. “STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution”. In: *IEEE Trans. Softw. Eng.* 41.2 (2015-02), pp. 198–220. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2363469.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 265–278. ISBN: 9781450302661. DOI: 10.1145/1950365.1950396.

[16] *Companies Using Microsoft Visual Studio and Its Marketshare*. Enlyft. URL: https://enlyft.com/tech/products/microsoft-visual-studio (visited on 2023-03-03).

[17] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. “DySy: Dynamic Symbolic Execution for Invariant Inference”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE ’08. IEEE. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 281–290. ISBN: 9781605580791. DOI: 10.1145/1368088.1368127.

[18] Fabrice Desclaux. “Miasm: Framework de reverse engineering”. French. In: *Actes du SSTIC* (2012). URL: https://www.sstic.org/media/SSTIC2012/SSTIC-actes/miasm_framework_de_reverse_engineering/SSTIC2012-Article-miasm_framework_de_reverse_engineering-desclaux_1.pdf.

[19] Pär Emanuelsson and Ulf Nilsson. “A Comparative Study of Industrial Static Analysis Tools”. In: *Electronic Notes in Theoretical Computer Science*. SSV 2008 217 (2008). Proceedings of the 3rd International Workshop on Systems Software Verification, pp. 5–21. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.06.039.

[20] Marcel Garus. “Fuzzing”. Advanced Programming Tools Seminar 2023, Software Architecture Group, Hasso Plattner Institute. 2023.

[21] Patrice Godefroid, Michael Y. Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. In: *NDSS*. Vol. 8. 2008-11, pp. 151–166. URL: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft.” In: *ACM Queue* 10.1 (2012-01), pp. 20–27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081.

[23] Peter Goodman and Alex Groce. “DeepState: Symbolic Unit Testing for C and C++”. In: *Proceedings of 2018 Workshop on Binary Analysis Research*. Internet Society, 2018-01. DOI: 10.14722/bar.2018.23009. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/07/bar2018_9_Goodman_paper.pdf.

[24] Anjana Gosain and Ganga Sharma. “Static Analysis: A Survey of Techniques and Tools”. In: *Intelligent Computing and Applications: Proceedings of the International Conference on ICA*. Ed. by Durbadal Mandal et al. New Delhi: Springer India, 2015, pp. 581–591. ISBN: 978-81-322-2268-2. DOI: 10.1007/978-81-322-2268-2_59.

[25] Jonathan de Halleux and Nikolai Tillmann. “Moles: Tool-Assisted Environment Isolation with Closures”. In: *Objects, Models, Components, Patterns: 48th International Conference*. TOOLS’10. Springer. Málaga, Spain: Springer-Verlag, 2010, pp. 253–270. ISBN: 3642139523. DOI: 10.5555/1894386.1894400.

[26] Martin Hentschel, Richard Bubel, and Reiner Hähnle. “The Symbolic Execution Debugger (SED): A Platform for Interactive Symbolic Execution, Debugging, Verification and More”. In: *International Journal on Software Tools for Technology Transfer* 21.5 (2019-10), pp. 485–513. ISSN: 1433-2779. DOI: 10.1007/s10009-018-0490-9.

[27] Gordon Hogenson, Genevieve Warren, Tim Sherer, et al. *Overview of Microsoft IntelliTest*. 2017–2022. URL: https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual (visited on 2023-01-23).

[28] Alberto Garcia Illera and Francisco Oca. *Ponce*. 2016-2022. URL: https://github.com/illera88/Ponce (visited on 2023-01-23).

[29] Ralph E. Johnson and Brian Foote. “Designing Reusable Classes”. In: *Journal of Object-Oriented Programming* 1.2 (1988-06), pp. 22–35. URL: http://www.laputan.org/drc.html.

[30] Cheick Keita et al. *OneFuzz*. Microsoft. 2020 – 2023. URL: https://github.com/microsoft/onefuzz (visited on 2023-03-03).

[31] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. “Shadow Symbolic Execution for Testing Software Patches”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (2018-09). ISSN: 1049-331X. DOI: 10.1145/3208952.

[32] Volodymyr Kuznetsov et al. “Efficient State Merging in Symbolic Execution”. In: *SIGPLAN Not.* PLDI ’12 47.6 (2012-06), pp. 193–204. DOI: 10.1145/2345156.2254088.

[33] Wenbin Li, Franck Le Gall, and Naum Spaseski. “A Survey on Model-based Testing Tools for Test Case Generation”. In: *Tools and Methods of Program Analysis: 4th International Conference, Revised Selected Papers 4*. Ed. by Vladimir Itsykson, Andre Scedrov, and Victor Zakharov. TMPA 2017. Springer. Moscow, Russia: Springer International Publishing, 2018, pp. 77–89. ISBN: 978-3-319-71734-0. DOI: 10.1007/978-3-319-71734-0_7.

[34] Yu Liu, Xu Zhou, and Wei-Wei Gong. “A Survey of Search Strategies in the Dynamic Symbolic Execution”. In: *ITM Web of Conferences*. Vol. 12. EDP Sciences. 2017, p. 03025. DOI: 10.1051/itmconf/20171203025.

[35] Nuno Lopes, Juneyoung Lee, et al. *Alive2*. AliveToolkit. 2018 – 2023. URL: https://github.com/AliveToolkit/alive2 (visited on 2023-03-03).

[36] Kasper Luckow et al. *Awesome Symbolic Execution*. 2017 – 2022. URL: https://github.com/ksluckow/awesome-symbolic-execution (visited on 2023-01-23).

[37] Kin-Keung Ma et al. “Directed Symbolic Execution”. In: *Proceedings of the 18th International Conference on Static Analysis*. SAS’11. Venice, Italy: Springer-Verlag, 2011, pp. 95–111. ISBN: 9783642237010. DOI: 10.5555/2041552.2041563.

[38] Rupak Majumdar and Koushik Sen. *Latest: Lazy Dynamic Test Input Generation*. Tech. rep. UCB/EECS-2007-36. EECS Department, University of California, 2007. URL: https://www.semanticscholar.org/paper/LATEST-%3A-Lazy-Dynamic-Test-Input-Generation-Majumdar-Sen/2b323db2dda6ec45f6f1a31acd62f5cfb506f51b.

[39] Paul Dan Marinescu and Cristian Cadar. “make test-zesti: A Symbolic Execution Solution for Improving Regression Testing”. In: *34th International Conference on Software Engineering*. ICSE 2012. IEEE. 2012, pp. 716–726. DOI: 10.1109/ICSE.2012.6227146.

[40] Nariman Mirzaei et al. “Testing Android Apps through Symbolic Execution”. In: *ACM SIGSOFT Software Engineering Notes* 37.6 (2012-11), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/2382756.2382798.

[41] Mark Mossberg et al. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. DOI: 10.1109/ASE.2019.00133.

[42] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[43] Bernhard Mueller, Nikhil Parasaram, Joran Honig, et al. *Mythril*. ConsenSys. 2017 – 2023. URL: https://github.com/ConsenSys/mythril (visited on 2023-03-03).

[44] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. “Symlnfer: Inferring Program Invariants Using Symbolic States”. In: *32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. IEEE. Los Alamitos, CA, USA: IEEE Computer Society, 2017-11, pp. 804–814. DOI: 10.1109/ASE.2017.8115691.

[45] Alessandro Dal Palù et al. “CUDSAT: SAT Solving on GPUs”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 27 (2015), pp. 293–316. DOI: 10.1080/0952813X.2014.954274.

[46] Corina S. Păsăreanu and Willem Visser. “A Survey of New Trends in Symbolic Execution for Software Testing and Analysis”. In: *International Journal on Software Tools for Technology Transfer* 11.4 (2009-10), pp. 339–353. ISSN: 1433-2787. DOI: 10.1007/s10009-009-0118-1.

[47] Nicole Perlroth. *The Untold History of America’s Zero-Day Market*. Wired. 2021-02-14. URL: https://www.wired.com/story/untold-history-americas-zero-day-market/ (visited on 2023-03-04).

[48] Sebastian Poeplau and Aurélien Francillon. “Symbolic Execution with SymCC: Don’t Interpret, Compile!” In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC’20. USA: USENIX Association, 2020, pp. 181–198. ISBN: 978-1-939133-17-5. DOI: 10.5555/3489212.3489223.

[49] Arash Sabbaghi and Mohammad Reza Keyvanpour. “A Systematic Review of Search Strategies in Dynamic Symbolic Execution”. In: *Computer Standards & Interfaces* 72 (2020), p. 103444. ISSN: 0920-5489. DOI: 10.1016/j.csi.2020.103444.

[50] Florent Saudel and Jonathan Salwan. “Triton: Framework d’exécution concolique et d’analyses en runtime [Triton: A Concolic Execution and Runtime Analysis Framework]”. French. In: *Symposium sur la sécurité des technologies de l’information et des communications*. SSTIC. Rennes, France, 2015-06, pp. 31–54. URL: https://github.com/JonathanSalwan/Triton/blob/master/publications / SSTIC2015 _ French _ Paper _ Triton _ Framework _ dexecution _ Concolique _ FSaudel_JSalwan.pdf.

[51] Phillip Schanely. *Contractual SemVer*. 2022. URL: https://github.com/pschanely/contractual-semver (visited on 2023-01-23).

[52] Phillip Schanely et al. *CrossHair Documentation*. 2019 – 2022. URL: https://crosshair.readthedocs.io/en/latest/ (visited on 2023-01-23).

[53] Koushik Sen. “Concolic Testing”. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 571–572. ISBN: 9781595938824. DOI: 10.1145/1321631.1321746.

[54] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *ACM SIGSOFT Software Engineering Notes*. ESEC/FSE-13 30.5 (2005-09), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1081706.1081750.

[55] Koushik Sen, Manu Sridharan, et al. *Jalangi2*. Samsung. 2014 – 2023. URL: https://github.com/Samsung/jalangi2 (visited on 2023-03-03).

[56] Yan Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. SP ’16. Los Alamitos, CA, USA: IEEE Computer Society, 2016-05, pp. 138–157. DOI: 10.1109/SP.2016.17.

[57] Dominik Sobania et al. “An Analysis of the Automatic Bug Fixing Performance of ChatGPT”. Preprint. 2023. DOI: 10.48550/ARXIV.2301.08653.

[58] *Statistics – Debian Sources*. URL: https://sources.debian.org/stats/ (visited on 2023-01-23).

[59] Lee Stott et al. *Frequently Asked Questions about the Education Hub*. Microsoft. URL: https://learn.microsoft.com/en-us/azure/education-hub/azure-dev-tools-teaching/program-faq (visited on 2023-03-03).

[60] *Symbolic Execution Debugger (SED)*. The KeY Project. URL: https://www.key-project.org/eclipse/sed (visited on 2023-01-23).

[61] *Symbolic Execution Papers*. 2021 – 2023. URL: https://github.com/XMUsuny/symbolic-execution-papers (visited on 2023-03-04).

[62] Nikolai Tillmann and Jonathan Peli de Halleux. “Pex – White Box Test Generation for .NET”. In: *Proceedings of Tests and Proofs: Second International Conference. Proceedings 2*. Vol. 4966. LNCS. Springer. Prato, Italy: Springer Verlag, 2008-04, pp. 134–153. URL: https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/.

[63] Nikolai Tillmann, Jonathan Peli de Halleux, and Tao Xie. “Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 385–396. DOI: 10.1145/2642937.2642941.

[64] David Trabish et al. “Chopped Symbolic Execution”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 350–360. ISBN: 9781450356381. DOI: 10.1145/3180155.3180251.

[65] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. “Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models”. In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA ’22. New Orleans, LA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3491101.3519665.

[66] Sergey Vartanov. *History of Symbolic Execution*. 2017 – 2021. URL: https://github.com/enzet/symbolic-execution (visited on 2023-03-04).

[67] Guowei Yang et al. “Advances in Symbolic Execution”. In: *Advances in Computers* 113 (2019). Ed. by Atif M. Memon, pp. 225–287. ISSN: 0065-2458. DOI: 10.1016/bs.adcom.2018.10.002.

[68] Andreas Zeller et al. *The Fuzzing Book*. CISPA+ Saarland University, 2019. URL: https://www.fuzzingbook.org/.