Symbolic Execution and Applications

Christoph Thiede

TODO: Abstract

1. Introduction

Program analysis covers a wide range of techniques and programming tools for generating insights about the structure and the behavior of a software system. We can distinguish between *static* and *dynamic* analysis tools: static analysis tools examine a program by its source code to derive particular information (e.g., dependency analysis, type flow analysis) or detect occurrences of relevant patterns (e.g., spell checkers, linters). Dynamic analysis tools, on the other hand, examine a program by observing the behavior of the running application (e.g., tests, benchmarks, or assertions).

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

Symbolic execution addresses this issue by automatically uncovering the execution paths of a program and performing analyses on them while requiring no or fewer specifications from programmers. The fundamental operating principle of symbolic execution is to execute a program with *symbolic variables* instead of concrete values for its inputs and to fork the execution at each encountered conditional expression (e.g., an `if` statement) whose behavior depends on the concrete assignment of the symbolic variables.

In the last four decades, programmers and researchers have employed symbolic execution for different purposes such as vulnerability analysis of programs, program verification, unit testing, and various program understanding tasks including model generation and reverse engineering.

In this report, we give a summary of several implementations and applications of symbolic execution. In section 2, we describe the fundamental operating principle of symbolic execution. In section 3, we provide an overview of challenges for symbolic execution and existing solution approaches. We present different 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 different symbolic execution tools. We classify some alternative approaches and literature in section 7 and finally give a conclusion in section 8.

2. Approach

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

Symbolic execution (also abbreviated to *symbex* or *SymEx*) attempts to systematically uncover all execution paths of a program. 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" [9].

To this end, the *symbolic execution engine* or the *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.
- As 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* for the assignment of its symbolic variable in the form of a Boolean first-order expression.

Together, the symbolic memory and the constraint set of an execution path constitute its *symbolic state* (see fig. 1 for the symbolic execution of the `foo()` function in listing 1). The entirety of execution paths is named 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.

A crucial component of each symbolic execution engine is an *SMT solver* that decides whether a symbolic branch condition can be fulfilled and generates a concrete solution for a constraint set. 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 datatypes. Popular theories describe arithmetics, bitwise operations/integers with overflow semantics, or strings.

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

```
1  void foo(int a, int b, int c) {
2      int x = 0, y = 0, z = 0;
3      if (a) {
4          x = -2;
5      }
6      if (b < 0) {
7          if (!a && c) {
8              y = 1;
9          }
10         z = 2;
11     }
12 }
```

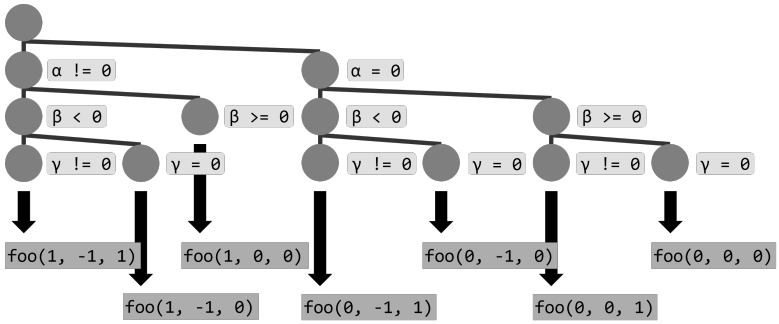

Figure 1: The symbolic execution tree and concrete input sets for each path after analyzing the `foo()` method from listing 1. The symbolic variables α, β, γ represent the assigned values for the input variables `a`, `b`, and `c`.

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 give an overview of common implementational challenges and solution strategies. Listing 2 gives examples for each challenge. A more detailed description of modern implementation strategies is given in appendix 1.

Path explosion, space explosion. For common programming constructs, the total number of execution paths may 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, causing an infinite size of the execution tree. In reality, this results in an impractical performance even for symbolic execution of small programs [10, 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]. On the hardware level, all I/O operations are blackboxes, including accesses to file systems, hardware-accelerated computation, and network communication [9, 4]. To symbolic executors that operate on the source code or intermediate representations of programs, also precompiled library binaries without sources represent a blackbox [34]. Importantly, event-driven applications that are built around a framework following the inversion of control principle [25] can only be symbolically executed in the context of the framework implementation which is often complex (see path explosion) or unknown [34]. Finally, concurrent programs typically rely on a blackbox scheduler that is part of the OS or the VM [55, sec. 7].

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

Symbolic pointers. If a pointer has a symbolic address value, naive symbolic execution engines cannot efficiently reason about the result of references to this pointer [4, sec. 3, 55, 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 invocations on an object of a symbolic class or prototype [4, sec. 3].

Advanced symbolic execution engines encounter this challenge by storing conditional expressions or symbolic pointers in the memory or by restricting the resolution of symbolic pointers

Listing 2: A selection of functions 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;
}
```

while sacrificing completeness (appendix A.5).

Solver limitations. Constraint solving is an NP-complete problem for which no generic efficient solution algorithm is known [10, sec. 4.2, 55, sec. 3]. In practice, this makes it hard 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 computing, optimized algorithms, caches, and other heuristics (appendix A.6).

4. Applications

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

4.1. Program Analysis

Through symbolic execution, programmers can automatically search large parts of their code bases for behavioral certain conditions or patterns. For instance, they can find all statements that violate an invariant, or they can detect possibly unreachable code [38, sec. 5]. It is also possible to infer new invariants from the execution tree [16].

A highly popular application of symbolic execution is *vulnerability detection* that allows programmers to scan programs for edge cases and circumstances causing security violations. In this context, several definitions of security violations can be used: in the simplest form, symbolic execution tools can uncover all paths that lead to an irregular abort or crash of a program [8, 18, 12]. In advanced scenarios, tools install an agent that tries to exploit potential attack vectors such as remote code execution, or they even detect suspicious behavior patterns such as accesses to uninitialized memory [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) [8, 12, 2].

4.2. Testing

Analogously to exploit generation, tools can also employ symbolic execution to *generate inputs* or *crash tests* for all covered execution paths of a program [44, 1]. Many tools generate code that contains only the arrange and act logic of a test but leave it to programmers to add assertions [44]; however, some approaches incorporate infered invariants to generate assertions [16, 37].

```
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 program crashes, generated by the symbolic execution engine KLEE [8].

Symbolic testing frameworks go beyond unspecific test generation by allowing programmers to write tests that operate on symbolic variables rather than concrete values [52, 20] (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*, e.g., for detecting breaking changes in a interface [43].

Listing 3: A symbolic unit test written in the parametrized unit testing framework Pex for C# [52]. The test asserts that for each pair of non-empty version strings, the method `Versions.Compare()` returns either `-1`, `0`, or `1`.

```
1  [PexMethod]
2  public int CompareTest(string version1, string version2)
3  {
4      PexAssume.IsNotNullOrEmpty(version1);
5      PexAssume.IsNotNullOrEmpty(version2);
6
7      int result = Versions.Compare(version1, version2);
8
9      PexAssert.IsTrue(new[] { -1, 0, 1 }.Contains(result));
10
11     return result;
12 }
```

A similar style of testing is *contract programming* where programmers specify pre- and postconditions for individual functions or methods that can be validated by a symbolic execution engine (see fig. 3) [44].

Other than testing frameworks, contracts are typically attached directly to the code under test, and many frameworks feature declarative specifications that are less suitable for imperative arrangements or disjoint invariants.

```
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 [44].

Symbolic testing tools can support programmers in improving the code coverage of their systems and provide them with faster feedback to discover bugs while reducing the required cost for writing tests. However, programmers take the usual overhead of writing tests as they need to 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 explicate them when deciding for the edge cases that their application should handle. If programmers have to deal with generated tests, typical concerns of code generation may arise, including readability and maintainability of generated code and concretized values (see appendix A.6). Finally, performance and theories of symbolic executors are limited, causing long wait times and incomplete coverage for complex programs (see section 3).

4.3. Reverse Engineering

Another category of symbolic execution tools support 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 method by constructing a table of input arguments, return values, and optionally user-specified expressions from the symbolic execution paths (see fig. 4) [23, 52]. Programmers can benefit from this to understand methods and explore their edge cases without studying the implementation.

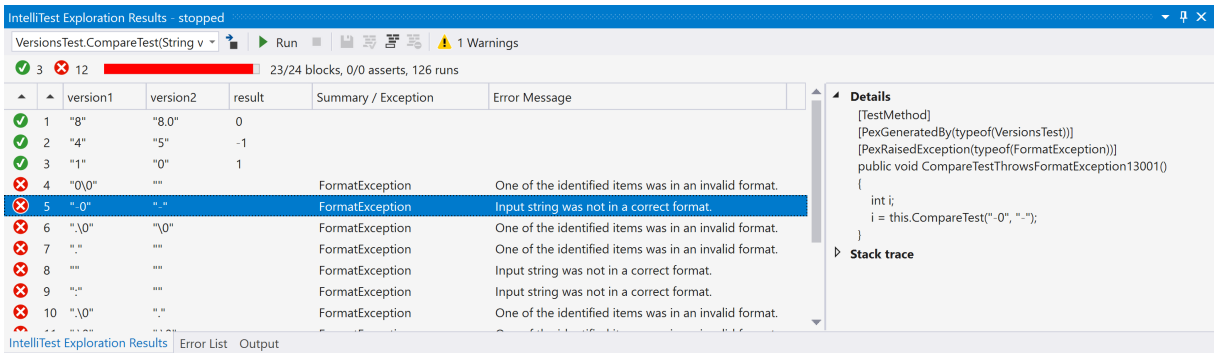

Figure 4: Screenshot of the IntelliTest explorer in Microsoft Visual Studio summarizing the behavior of the method `Versions.Compare()` [23]. Each result represents a different equivalence class of inputs (arguments) and outputs (return value or raised exception) that are caused by the same code path. 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 debugging*: programmers can debug a program without providing concrete entrypoints (see fig. 5) [22, 51, 24]. 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 may help programmers to evade irrelevant or distracting context, but they also may experience symbolic expressions in the memory of the program as too abstract and sophisticated. The extent of the symbolic execution tree may be overwhelming, and its structure may be confusing as it mismatches common control flow patterns such as loops. Thus, the value for symbolic debugging tools is likely maximal for codebases with a high complexity and a poor readability; for instance, several disassembling tools support symbolic debugging [24].

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

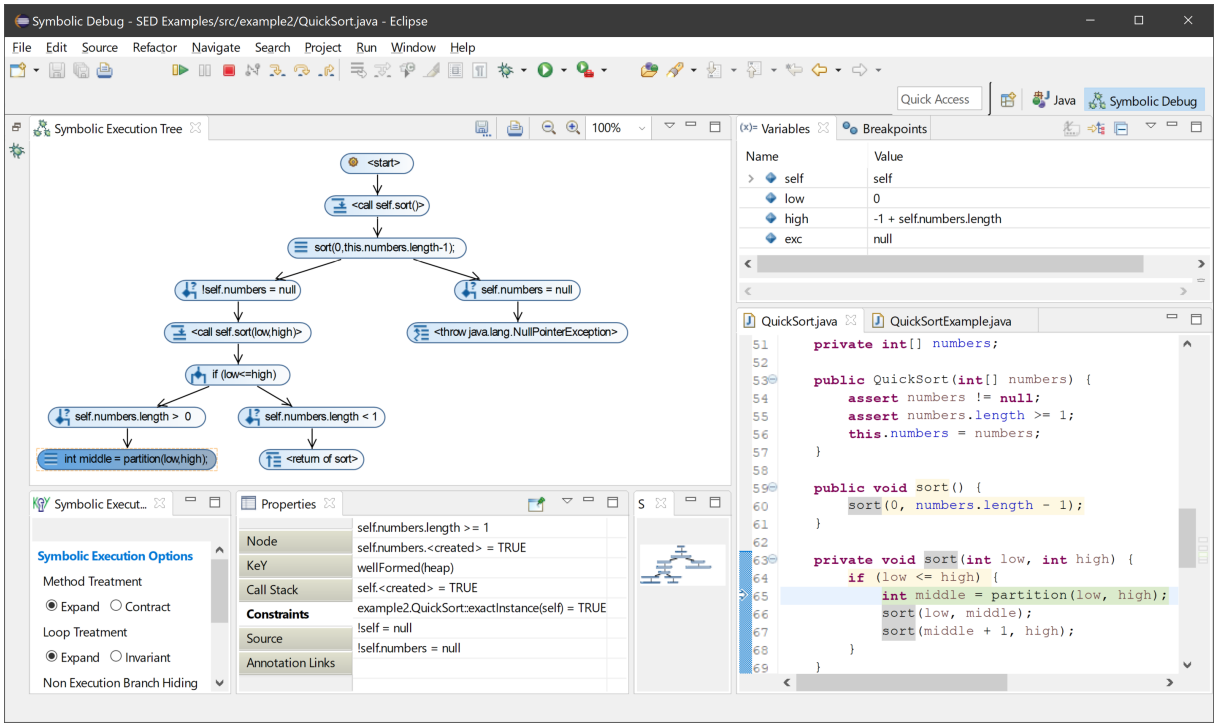

Figure 5: Screenshot of the *Symbolic Execution Debugger* (SED) while exploring the execution tree of a `QuickSort` implementation for Java in Eclipse KLEE [22, 51]. By selecting a node in the tree, programmers can inspect its symbolic state in the remaining panes and advance the execution of the code path stepwise.

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 an above-average popularity (at least 200 stars). The largest group of symbolic execution tools based on this criteria are frameworks and libraries that enable programmers to perform a dynamic analysis of their systems on a low or medium abstraction level. For instance, many tools offer APIs for running particular instructions or methods symbolically and inspecting the resulting execution tree [48, 17, 42]. A large number of tools provides means for semi-automated vulnerability detection and exploit generation [36, 26]. Most tools operate on binaries; however, some solutions target applications for Java, JavaScript, or smart contracts that are run in the Ethereum VM [35, 36]. Some tools also use symbolic execution to detect suspicious behavioral patterns such as the propagation of null pointers or object layouts and accesses that impede particular JIT optimizations [47], to verify automated optimizations [30], or to assist programmers with the exploration of disassemblied code [24].

To our knowledge, few solutions are widespread that assist programmers with writing tests [44, 20]. An exception is IntelliTest which has been integrated in Microsoft Visual Studio since 2015 and thus is available to a potentially large number of users [23].[1]

5.2. Impact for Vulnerability Detection

In the past two decades, symbolic execution tools have had a particular impact for detecting vulnerabilities (see section 4.1). Microsoft has used their in-house concolic execution engine SAGE to find more than 30% of all bugs in Windows 7 that were fixed prior to the release, has later established the tool as a standard component of their internal testing pipelines, and is continuously running it 24/7 on more than 200 machines to reduce the number of exploits in several products [19, 6].

In research, the popular free open-source library GNU Core Utils (containing tools such as `cat`, `tee`, and `wc`) has established as a benchmark for evaluating the performance of symbolic execution engines. The entire library comprises 89 binaries with 72 kLOC (thousands lines of code) and the original test suite

[1] IntelliTest is available in the Enterprise edition of Visual Studio. More than 150,000 companies have subscribed to Visual Studio [15], and Microsoft provides free access to the Visual Studio Enterprise for students through the *Azure Education Hub* program [50]. However, the actual number of customers who work with C#/.NET Framework is unknown.

had a coverage of 67.6% LCOV (percent of lines covered) [8]. In 2008, KLEE detected 56 yet unknown bugs and crashes in the library within 1 hour per binary and increased the overall code coverage to 84.5% [8]. Later, other tools detected further bugs in the library while reducing the operational costs [33, 28]. In 2012, MAYHEM reached a coverage of 97.6% LCOV within 1 hour per binary for a subset of the library [12].

Researchers have also demonstrated the ability of vulnerability detection for other software systems: Inter alia, bugs and vulnerabilities were found in BusyBox [8], the Linux kernel [12], Minix [8], and Windows [12]. MERGEPOINT reached particularly remarkable results by discovering more than 11,000 bugs in the entire Debian kernel [3] (consisting of more than 33,000 binaries and 679 MSLOC (millions of source lines of code) [49]). This analysis took them 18 CPU-months but benefited from massive parallelization. They also estimated that the operational costs for renting virtual servers in a data center would correspond to $0.28 per discovered bug, showing that symbolic execution for security analysis can be financially worthwhile given the potentially high costs of zero-day exploits that are discovered in the public [19, 11, 39].

6. Discussion

7. Related Work

8. Conclusion

A. Implementation Strategies

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

A.1. Taxonomy of Executors

Symbolic executors can follow alternative design decisions that make different trade-offs between implementation costs, 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 every covered execution path is executed exactly once and the symbolic state is *cloned* at each symbolic

branch condition. On the other hand, *offline symbolic executors* do not fork the execution but instead re-run an execution path for each selected branch [4].

While online symbolic execution processes fewer instructions in total, it maintains many cloned states in the memory which can 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 effort 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 a memory boundary is reached [12]. At this 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: as any variables are possibly assigned symbolic expressions, the executor cannot pass them to a foreign system (e.g., when making a system call to the OS). *Dynamic symbolic execution* (DSE) resolves this limitation by combining symbolic execution with normal concrete execution which assigns concrete values to variables [10, 4].[2]

One form of DSE is *execution-generated testing* (EGT) which interleaves symbolic and concrete execution [9]. When a blackbox is called, 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 which executes a program both in concrete and symbolic style simultaneously [46, 18]. Concrete execution maintains a concrete state of all variables and is used to direct the control flow and call blackboxes. Symbolic execution, accompanies concrete execution to collect the symbolic constraints for the taken conditional branches. Once an execution path has been completed, further paths can be generated by negating single constraints from the collection and generating concrete input values for the next execution.

Thus, 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, removing the context-switch overhead of requesting the solver at every conditional instruction. If the solver is unable to find a solution for a constraint set, symbolic execution can omit some execution paths instead of inevitably aborting [10, sec. 3]. Practically, concolic execution is implemented by instrumenting the program for collecting constraints and running the instrumented program offline in the regular OS or VM [45, 40]. This also

[2] 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 [10] (instead of the taxonomy of Baldoni et al. who use both terms in the opposite way [4]).

improves performance and reduces implementation costs as a level of indirection for a separate interpreter is avoided.

Still, neither EGT nor concolic execution reveals conditional branches inside blackboxes. For improving the coverage of blackboxes, programmers can alternatively specify memory models (appendix A.5).

A.1.3. Static Symbolic Execution

Static symbolic execution (SSE) tackles the issue of path explosion by deriving a static transformation of program parts rather than executing them [3, 4]. SSE employs *function summary* and *loop summary* techniques to translate a function or a group of instructions, resp., to an equivalent conditional symbolic expression (corresponding to a mathematical piecewise-defined function). For instance, *compositional symbolic execution* analyzes single program parts in isolation by treating any function calls as new symbolic expressions and translates them to pairs of preconditions and postconditions [55, sec. 5]. SSE reduces the cost for executing a program multiple times and deciding/negating 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 a function boundary, blackbox, or another limitation of SSE is hit. 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 instructions and execution paths of a program in reverse order [31, 4, sec. 2.3]. One application of BSE is post-mortem debugging [13]. To identify the possible callers of a method, a statically generated control-flow graph (CFG) can be used. As methods might 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 particular subsets or behaviors of the system. *Selective symbolic execution* includes several methods for selecting parts of the system for

analysis [14]. In its general form, programmers can specify an allow- or denylist of units (e.g., modules, classes, or methods) to be analyzed. All non-selected parts are treated as blackboxes: outside of the selected parts, the program can be executed concretely without generating new execution paths. However, this approach reduces the completeness of the analysis; for instance, not all possible return values or side effects from a call to a skipped function will be covered. *Chopped symbolic execution* can partially restore the lost completeness by employing static analysis of the skipped function's behavior [54].

Selective symbolic execution is well combinable with compositional symbolic execution where single units can be analyzed in isolation.

Shadow symbolic execution is another form of selective symbolic execution that selects parts for symbolic execution based on the changes to a software system against the latest run of the symbolic execution engine. In the context of continuous integration where tests and analyses are run for each new revision, this can significantly reduce the execution time or improve the coverage within a given time limit, resp. [27]

Using *preconditioned symbolic execution*, programmers can manually specify restrictions on inputs or program behavior [2, 4, sec. 5.5]. For instance, 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 regarding the static CFG of the system [31]. 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 [32].

A.3. Path Management

A major challenge of symbolic execution is path explosion where the number of execution paths may grow exponentially to the number of conditional branches or even infinitely. Different strategies exist to cope with the large size of the execution tree, and practical symbolic execution engines commonly combine multiple of them.

A.3.1. Path Selection

Path selection (also *branch prioritization* or *search strategies*) refers to the idea of maximizing the number of *relevant* execution paths within a given time limit [29, 4, sec. 2.2]. A criterion is defined for ordering execution paths, and the offline executor selects the next path based on this criterion.

Naive path selection strategies include *depth-first search* (DFS) and *breadth-first search* (BFS) of the execution tree [29, 41]. DFS is a prime example of search strategies that are not loop-safe: if the executor hits an infinite subtree (e.g., a loop or recursive call with a symbolic break condition), the search will never complete, and the remaining tree will not be explored further. 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 [29]. *Random search* combines the strengths of DFS and BFS by randomly selecting execution paths but is still not loop-safe [29].

Other strategies assign scores or weights to different execution paths. *Generational search* maximizes execution order for code coverage by computing the score of each new from the relative coverage increase of the previous (generating) path [18, 41, p. 24f.]. Execution paths can also be prioritized based on the mutation coverage of the generated tests or the coverage of the static CFG [41, p. 18ff.]. Some strategies use heuristics to predict the number of errors in subtrees based on certain instruction types or the historic density of defects in a unit (assuming that the defect density is not normally distributed amongst units but peaks in units written by different authors on different days) [12, 2, 41, p. 26]. Further heuristics try to detect repetitive loop iterations or recursive calls or use define *fitness functions* for symbolic variables to prioritize solutions for them that trigger certain branches [10, p. 9, 41, p. 27].

A.3.2. Path Summarization

Path summarization strategies analyze single functions or loops statically to avoid path explosion. These strategies commonly 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 doubles [4, sec. 5.4, 55, sec. 4.2]. By storing pre- and postconditions, they can also detect *equivalent* execution paths that only differ in side effects or values that are irrelevant to the remaining control flow.

Path merging strategies avoid path multiplicities by combining similar execution paths [3, 4, sec. 5.6, 55, sec. 4.3]. The state of merged execution paths contains conditional constraints and conditional expressions in the symbolic store. To weigh up the reduced costs for execution against the additional pressure on the SMT solver, they employ several heuristics. Among others, these heuristics take into account the following instruction types and the static CFG of a program to predict the solver costs.

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 [7, 3].

A.4. Environment Models

To avoid blackboxes, programmers can provide models to emulate parts of the environment [4, sec. 4]. Similar to unit testing, they can configure fakes and stubs that are whiteboxes to the symbolic executor [21]. For instance, they can replace the real filesystem with a virtual filesystem or redirect all requests to a remote server to a stub. 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 approach to automate model generation.

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 single call targets that are exposed to the framework [34]. 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 [8] or by employing virtualization techniques to emulate critical instructions [14].

A.5. Memory Models

To properly handle symbolic pointers, symbolic execution engines have to consider all possible symbolic states that can result from dereferencing the pointer (or looking up an array element, resp.) [10, 4, sec. 3]. As this is prone to path explosion, one 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. Still, 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 boundaries to the address space [4, sec. 3]. For instance, symbolic execution engines can constraint symbolic pointers to all previously allocated addresses plus a canonical null pointer, or they may immediately report an error for an execution path that accesses unallocated space as this is deemed an unsafe practice. Alternatively, offline engines can randomly concretize symbolic pointers, allocate new space dynamically, and (optionally lazily [55, sec. 6]) initialize it with new symbolic variables or data structures. Similar to preconditioned symbolic execution, engines may also enable programmers to specify custom constraints for newly initialized data.

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

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

A.6. Constraint Solvers

Despite there is still no generic efficient algorithm known for the satisfiability problem, the performance of SMT solvers has significantly improved over the past decades. Causes for this include advances regarding the underlying theories and algorithms for reducing constraints, deferring computation, and reusing solutions. Solvers also benefit from increased hardware performance, parallelization, and acceleration using GPUs.

Another challenge is the generation of intuitive data during concretization. For instance, some solvers favor numbers with small absolutes over random integers, or they stick with alphanumerical characters for strings and maximize repetition [53, sec. 6.3].

A.7. Trends in Symbolic Execution

Today, major challenges to symbolic execution include path explosion (appendix A.3) and solver performance (appendix A.6). Several approaches employ the results of efficient static analysis to assist the symbolic execution engine with these challenges. Another open problem is the analysis of concurrent programs with an exponential number of possible execution orders [55, sec. 7].

References

[1] Elvira Albert et al. “Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-based Instance, and Actor-based Concurrency”. In: ed. by Marco Bernardo et al. Vol. 8483. Lecture Notes in Computer Science. 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 (Feb. 2014), 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: 51.3 (May 2018), 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] 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.

[7] 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.

[8] 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.

[9] 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. Aug. 2005, pp. 2–23. ISBN: 978-3-540-28195-5. DOI: 10.1007/11537328_2.

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

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

[12] 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, May 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.

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

[14] 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.

[15] *Companies using Microsoft Visual Studio and its marketshare*. Enlyft. URL: https://enlyft.com/tech/products/microsoft-visual-studio (visited on 03/03/2023).

[16] 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.

[17] Fabrice Desclaux. "Miasm: Framework de reverse engineering". In: *Actes du SSTIC. 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.

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

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

[20] Peter Goodman and Alex Groce. "DeepState: Symbolic Unit Testing for C and C++". In: *NDSS Workshop on Binary Analysis Research*. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/07/bar2018_9_Goodman_paper.pdf.

[21] Jonathan de Halleux and Nikolai Tillmann. "Moles: Tool-Assisted Environment Isolation with Closures". In: *Objects, Models, Components, Patterns: 48th International Conference, TOOLS 2010, Málaga, Spain, June 28–July 2, 2010. Proceedings 48*. TOOLS'10. Springer. Málaga, Spain: Springer-Verlag, 2010, pp. 253–270. ISBN: 3642139523. DOI: 10.5555/1894386.1894400.

[22] 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 (Oct. 2019), pp. 485–513. ISSN: 1433-2779. DOI: 10.1007/s10009-018-0490-9.

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

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

[25] Ralph E Johnson and Brian Foote. "Designing Reusable Classes". In: *Journal of object-oriented programming* 1.2 (1988), pp. 22–35. URL: http://www.laputan.org/drc.html.

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

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

[28] Volodymyr Kuznetsov et al. “Efficient State Merging in Symbolic Execution”. In: *SIGPLAN Not.* PLDI ’12 47.6 (June 2012), pp. 193–204. ISSN: 0362-1340. DOI: 10.1145/2345156.2254088.

[29] 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.

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

[31] Kin-Keung Ma et al. “Directed symbolic execution”. In: *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*. Springer. 2011, pp. 95–111.

[32] Rupak Majumdar and Koushik Sen. *Latest: Lazy dynamic test input generation*. Tech. rep. Technical Report UCB/EECS-2007-36, EECS Department, University of California …, 2007.

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

[34] Nariman Mirzaei et al. “Testing android apps through symbolic execution”. In: *ACM SIGSOFT Software Engineering Notes* 37.6 (2012), pp. 1–5.

[35] Mark Mossberg et al. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. DOI: 10.48550/ARXIV.1907.03890.

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

[37] ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. “Symlnfer: Inferring program invariants using symbolic states”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 804–814.

[38] 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 (2009), pp. 339–353.

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

[40] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SymCC: Don’t interpret, compile!” In: *Proceedings of the 29th USENIX Conference on Security Symposium*. 2020, pp. 181–198.

[41] 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: https://doi.org/10.1016/j.csi.2020.103444. URL: https://www.sciencedirect.com/science/article/pii/S0920548919300066.

[42] Florent Saudel and Jonathan Salwan. “Triton: A Dynamic Symbolic Execution Framework”. In: *Symposium sur la sécurité des technologies de l’information et des communications*. SSTIC. Rennes, France, June 2015, pp. 31–54.

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

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

[45] Koushik Sen. “Concolic testing”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 571–572.

[46] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A concolic unit testing engine for C”. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

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

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

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

[50] 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 03/03/2023).

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

[52] Nikolai Tillmann and Jonathan De Halleux. “Pex–white box test generation for .NET”. In: *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings 2*. Springer. 2008, pp. 134–153.

[53] Nikolai Tillmann, Jonathan 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*. 2014, pp. 385–396.

[54] David Trabish et al. “Chopped Symbolic Execution”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 350–360. DOI: 10.1145/3180155.3180251.

[55] Guowei Yang et al. “Advances in symbolic execution”. In: *Advances in Computers* 113 (2019), pp. 225–287.