Symbolic Execution in Practice: A Survey of Applications in Vulnerability, Malware, Firmware, and Protocol Analysis

Joshua Bailey (jbailey4@umbc.edu)
Charles Nicholas (nicholas@umbc.edu)
University of Maryland, Baltimore County

August 12, 2025

1 Introduction

Program testing is an essential aspect of software development. Testing not only helps to verify a program's capabilities, it also uncovers potential vulnerabilities that adversaries could exploit, and offers an opportunity to verify the correctness of an implementation. Traditionally, program verification relied on rigorous formal methods to prove the correctness of a program; while testing, it could be said, used a more practical, but less exhaustive approach where inputs were carefully chosen and run against the program. The outputs were manually inspected and compared against expected outcomes [1]. Although formal methods provide strong guarantees about program behavior, its often impractical against large scale software. Simpler testing strategies, while fast, may fail to detect deep, subtle bugs. Because of the complexity of modern software, a middle ground that provides both thoroughness and practicality became necessary.

In the 70s, researchers proposed such a middle ground in *symbolic execution* [1]–[4]. At the time, the approach was limited by the capability of theorem provers. However, the advent and advancement of satisfiability modulo theories (SMT) solvers [5], [6] have since made symbolic execution practical. The core idea is to analyze a program with symbolic variables that represent all possible inputs instead of concrete ones. This allows a symbolic execution engine to systematically explore multiple execution paths, offering greater code coverage and more precise bug finding.

This paper presents a comprehensive survey of how symbolic execution aids in the analysis of large and complex software systems. Rather than simply cataloging tools and techniques, we synthesize the surveyed literature to reveal a unifying theme: strategies to manage the complexity of symbolic execution is critical to practicality. We introduce a framework that categorizes these techniques into broad categories: scope reduction, hybrid analysis, and a taxonomy of guidance heuristics. We use this framework to analyze applications across diverse domains. The low-level mechanics of search strategies have been previously surveyed [7], our work provides a higher-level analysis of how these strategies are applied in practice in domains such as vulnerability research, program verification, emulation and firmware analysis, and obfuscated and malicious code analysis. For an introduction to many of these domains, we refer the reader to a modern software engineering textbook such as [8].

By examining representative tools and relevant research, we aim to help researchers and practitioners understand when and why symbolic execution is beneficial, and how they can select or adapt tools for their specific scenarios. In doing so, we highlight current strengths and identify areas requiring further development.

The remainder of this paper is organized as follows. Section 2 provides an overview of symbolic execution, detailing the core challenges and common mitigations in the field. Section 3 presents a landscape of popular symbolic execution engines, examining their high-level design, strengths and weaknesses. Section 4 introduces a taxonomy of guidance heuristics commonly employed in symbolic execution research and surveys how symbolic execution has been applied in various application domains. Section 5 outlines potential research directions. Section 6 offers concluding thoughts.

2 Symbolic Execution Fundamentals

Symbolic execution is a program analysis technique that explores executable paths by treating inputs as symbolic variables. For example, instead of running the program with specific inputs like `x = 4`, symbolic execution uses symbolic values, like `x =` λ, to represent a range of possible inputs, generating expressions that capture the behavior of the program along different paths.

As the symbolic execution engine traverses a program, it builds up *path conditions* which are logical expressions representing the constraints on inputs required to follow a particular execution path. In addition, the symbolic

execution engine may also maintain a store of symbolic memory which maps variables to symbolic expressions or values. In order to determine whether an execution path is feasible, the engine leverages a *constraint solver* (e.g., SMT solvers like Z3 or CVC4) which act as the brains of the operation. Path conditions are fed to the constraint solver and, if a feasible path exists, it will generate concrete inputs that trigger specific behaviors or bugs.

Symbolic execution allows for:

- **Comprehensive Path Exploration**: By considering multiple execution paths, symbolic execution can uncover edge cases and hidden bugs that might be missed by traditional testing. For example, in a program that computes the square root of a real number, symbolic execution should introduce conditions where the input is negative, zero, or positive, whereas traditional testing might only check positive inputs.
- **Automated Test Case Generation**: Symbolic execution can generate inputs that achieve high code coverage, aiding in thorough software testing.
- **Vulnerability Detection**: Symbolic execution can identify security vulnerabilities by exploring paths that lead to unsafe states or violate security properties.

Despite its effectiveness and versatility, symbolic execution is not without its challenges. Limitations such as path explosion, constraint solving, and scalability must be carefully evaluated to determine its feasibility for a given project.

2.1 Common Challenges in Symbolic Execution

As noted above, symbolic execution is a powerful technique program analysis technique, but it is not without its difficulties. The core challenges include path explosion, constraint solving difficulties, and environment modeling. In this section, we will describe these challenges, and others, in greater detail. Later, in Section 2.2 we discuss ongoing research and advances aimed at addressing these issues.

2.1.1 Path Explosion

Symbolic execution's strength lies in its ability to explore all possible executions paths of a program simultaneously. While this property allows for

thorough code coverage, it's this very strength that's also one its greatest weaknesses. The sheer number of paths to analyze can make analysis increasingly resource intensive. This phenomenon is known as path or state space explosion. As the number of paths increases, the symbolic execution engine has to manage memory, CPU time, and solver queries, ultimately limiting the scalability of the analysis. The following examples, while trivial, illustrate how quickly this complexity can arise.

A common cause of path explosion are *loops with symbolic conditions*. Consider the code snippet in Listing 1:

Listing 1: A loop with a symbolic condition

```
void loop_example(int n) {
    for (int i = 0; i < n; i++) {
        // Some code dependent on 'i'
    }
}
```

In Listing 1, if `n` is symbolic, the loop's iteration count is effectively unbounded. The symbolic execution engine must consider all possible values of `n`, leading to a potentially infinite number of paths. To mitigate infinite exploration, symbolic execution engines typically impose loop unrolling limits [9]. Alternatively, the variable could be concretized or constrained. However, such heuristics risk missing bugs that only manifest at larger iteration counts. Techniques like loop summarization [10] or state merging [11] can help reduce the number of paths while preserving a meaningfully high level of coverage.

Nested conditionals are another major contributor to path explosion. Listing 2 shows how even a small number of symbolic conditionals can cause exponential growth of the state space:

Listing 2: Path explosion caused by nested conditional statements

```
void conditional_example(int a, int b) {
    if (a < 0) {
        if (b > 0) {
            // Path 1
        } else {
            // Path 2
        }
    } else {
        if (b > 0) {
            // Path 3
```

```
        } else {
            // Path 4
        }
    }
}
```

For two symbolic variables `a` and `b`, there are four distinct paths. Adding a third symbolic variable `c` would create eight paths. Because symbolic execution must analyze the true and false branch every time a conditional expression is encountered. Consequently, the symbolic execution engine increases the number of paths by a factor of two, leading to an exponential growth of up to 2^n paths.

Certain design decisions, like *input-dependent branching*, exacerbate the path explosion problem. Consider the code in Listing 3 that independently checks multiple input bytes. Here, the variable `input` is treated as a symbolic array where each element represents a symbolic variable. While structurally different from nested conditionals, this pattern produces the same exponential effect since each `if` statement also introduces two branches. This branching pattern appears frequently in device drivers, file parsers, and embedded systems.

Listing 3: Path explosion caused by input-dependent branching

```
void input_branching(char *input) {
    if (input[0] == 'A') {
        // case where first character is 'A'
    }
    if (input[1] == 'B') {
        // case where second character is 'B'
    }
    if (input[2] == 'C') {
        // case where third character is 'C'
    }
    // ... additional conditions
}
```

In each of these examples, the engine must carefully track path conditions and program state which can become unwieldy and consume substantial resources. Techniques such as path pruning and heuristic search strategies can tame path explosion, though each comes with their own limitations, discussed in Section 2.2.

2.1.2 Constraint Solving Difficulties

Constraint satisfaction problems are applicable in a wide array of applications such as software and hardware verification, type-checking, test-case generation scheduling, planning, to name a few [12]. Boolean satisfiability (SAT) is a well-known constraint satisfaction problem that determines if a formula over Boolean variables can be made true. SAT solvers, while powerful, are limiting in that they are not able to reason about complex logic that might be encountered in modern programming languages. Satisfiability Modulo Theories (SMT) generalize the SAT problem to make them more amenable to such complex paradigms such as bit-vectors, arrays, and linear integer and real arithmetic, and non-linear arithmetic. Consider the simple function `check` 4:

Listing 4: An illustration of path condition generation

```
void check(int x) {
    int y = x * 2;
    if (y == 12) {
        // Interesting path
    }
}
```

If we want to determine whether the interesting path is reachable, the symbolic execution engine would generate the following path condition `(x_sym * 2) == 12`, where `x_sym` is a symbolic variable. This formula is sent to the SMT solver where it's task is to determine if a value exists that makes this formula true. In this example, the SMT solver would determine this path is satisfiable and provide the concrete example, `x = 6`.

Symbolic execution heavily relies on constraint solvers to determine path feasibility. By only analyzing feasible paths, symbolic execution can accurately explore multiple execution paths and reason about program behaviors and variable properties. Early symbolic execution work by King [1] demonstrated that constraint solving is fundamental since symbolic execution requires evaluating path conditions and determining satisfiability of symbolic expressions. However, King noted that the theorem proving capabilities available at the time made symbolic execution impossible even for modest programming languages. At that time, efficient theorem solvers capable of handling the complex constraints generated during symbolic execution were not available.However, modern SMT solvers like STP [13] and Z3 [5] have enabled practical analysis by efficiently handling complex constraints over arithmetic, bit-vectors, arrays, and other operations. Despite these

advancements, constraint solving remains a bottleneck. Certain classes of constraints, such as non-linear arithmetic constraints [14], remain difficult to solve, and the presence of path explosion and complex theories can limit scalability. Even powerful solvers such as Z3 cannot fully mitigate these challenges, requiring researchers and practitioners to employ heuristics, approximations, or domain-specific optimizations.

2.1.3 Environmental Interaction Modeling

Any useful software application does not operate in a vacuum. Even modest application must interact with their environment via I/O operations, network communication, interfacing with hardware peripherals or some other external component. The challenge for symbolic execution engines is determining how to symbolically represent these interactions. As a result, the symbolic execution engine must simulate file systems, protocols, and devices with sufficient fidelity to avoid unsound assumptions or missing crucial paths. As mentioned in Section 2.1.4, concurrency can exacerbate environmental modeling complexity, as timing-dependent interactions across multiple threads interacting with external systems increase the number of scenarios to consider. Domain-specific knowledge, auxiliary models, or hybrid symbolic-concrete execution strategies are often required. However, these solutions may limit general applicability or require extensive manual effort.

Understanding these core challenges: path explosion, floating-point handling, concurrency, constraint solving difficulties, obfuscated code, scalability, and environmental modeling, provides a foundation for evaluating symbolic execution tools and the research efforts which aim to overcome these hurdles. In Section 2.2, we explore ongoing efforts and innovative techniques designed to address these persistent issues.

2.1.4 Other Challenges

Beyond the core foundational issues, there are more specialized issues that hamper the technique's effectiveness.

Floating-Point Arithmetic Handling Floating-point arithmetic is a critical part of modern computing, specifically in scientific computing where accuracy is paramount. However, real numbers are infinite so representing them in a finite binary format pose a real challenge in computer systems. While the IEEE 754 [15] standard provides a uniform representation

of floating-point values, it does not eliminate the fundamental rounding errors and precision loss that can make program behavior difficult to reason about [16].

For symbolic execution, handling floating-point arithmetic is especially challenging. The issues primarily stem from constraint solvers having difficulty solving formulas with floating-point values. Many solvers handle floating-point constraints using approximations, potentially leading to incomplete or inaccurate analyses [17].

As a result, floating-point computations remain an open problem area, with ongoing research trying to provide sound and efficient handling [17], [18]. For a comprehensive survey of the different techniques being developed to address these challenges, we refer the reader to work by Zhang et al. [19].

Multi-threading and Concurrency Multi-threaded programs enable software to make efficient use of CPU resources. Threaded applications are able to handle multiple streams of input concurrently, perform complex computations in parallel, and distribute work among various compute resources. While threads offer significant performance gains, it also introduces substantial challenges in debugging and analysis due to the inherent complexity and non-determinism of concurrent execution.

The non-determinism arises because the order in which threads execute is not strictly defined. Consequently, certain bugs are difficult to reproduce since the same sequence of operations might not occur consistently across multiple runs. Furthermore, multiple threads accessing the same shared resource without proper synchronization can lead to race conditions causing unpredictable and subtle-to-detect bugs. Another consequence of threads is the exacerbation of the path explosion problem. Moreover, the shared state must be accurately modeled across all thread, significantly increasing the complexity of the analysis. As discussed in Section 2.1.3, concurrency can also magnify the challenges of modeling external environments. Timing and ordering constraints introduced by multiple threads interacting with external resources or devices can greatly complicate environmental modeling. This forces the symbolic execution engine to consider more nuanced scenarios.

Approaches to mitigating concurrency-related challenges, such as program sequentialization [20], can simplify thread interleaving but introduce scalability issues, risk losing concurrency semantics, and rely on simplifying assumptions that may overlook critical issues.

Handling Obfuscated and Self-Modifying Code Code obfuscation is a technique that transforms a program into a form that is more difficult to understand. Obfuscation techniques such as self-modifying code complicate analysis by altering control flow and data structures in ways that hinder the engine's assumptions and simplifications. Self-modifying code forces a symbolic execution engine to adapt dynamically to changing code and maintain soundness despite transformations intended to confuse analysis. To this end, researchers have developed specialized techniques to counter specific types of obfuscation. For instance, some approaches combine symbolic execution with taint analysis to reverse virtualized code [21], while others use novel methods like backward-bounded dynamic symbolic execution (DSE) [22] to defeat opaque predicates. Despite these advances, such solutions are often tailored to particular obfuscation schemes, and creating a robust, general-purpose deobfuscation framework remains a significant and open research challenge.

Scalability It is often desirable to analyze software at scale whether it be analyzing one project with millions of lines or analyzing multiple projects efficiently. Applying symbolic execution to large-scale, real-world software systems involves addressing thousands if not millions of lines of code, external libraries, complex data types, and intricate state spaces. The core challenges presented earlier in this section contribute to the challenge of making symbolic execution scalable. Managing resource consumption, which may include efficient path pruning, leveraging parallelization, and distributed analysis, is essential to making symbolic execution practical at scale. Ultimately, achieving scalable analysis with symbolic execution requires mitigating one or more of the challenges presented here. We will discuss various scalability-improving techniques in Section 2.2.

2.2 Addressing Core Symbolic Execution Challenges

While Section 2.1 identified several ongoing difficulties in applying symbolic execution to real-world software, most research efforts have focused on the most pressing and foundational problems, particularly path explosion and constraint solving difficulties. By making path exploration more tractable, these approaches also improve scalability, reduce server load, and indirectly aid in handling other challenges like environmental modeling and complexity in multi-threaded scenarios. In symbolic execution, every path analyzed by the engine must maintain a consistent view of the environment (e.g., registers, file systems, etc.). Reducing the sheer number of states to track,

whether it be by pruning or merging paths, reduces the number of environmental states the engine must manage.

In this section, we highlight notable strategies and techniques that have proven effective in mitigating path explosion and, to a lesser extent, improving constraint handling. Although not every challenge from Section 2.1 is directly addressed here, many of the solutions are broad in scope, offering partial relief or foundational improvements that can be adapted or extended to tackle additional issues.

2.2.1 Techniques for Managing Path Explosion and Scalability

State Merging and Pruning. As highlighted in Section 2.1.1, path explosion often occurs because each symbolic branch creates new states. However, not all branches lead to unique states. Therefore similar states can be merged and redundant paths eliminated drastically reducing the number of paths to explore [11]. Likewise, certain paths may not yield promising results. These "low-value" paths can be pruned using heuristics to focus analysis on the most promising execution paths [23]. Bounded exploration [24], which limits the depth or number of paths explored, further controls resource usage. While these techniques do not solve path explosion entirely, they make symbolic execution more feasible for larger and more complex programs.

Concolic Execution As we've noted, constraint solving is computationally expensive and remains the primary bottleneck in symbolic execution [25]. To address this, researchers have employed a method that combines concrete execution with symbolic execution resulting in what is called *concolic* execution. The idea is to utilize concrete runs of the program under test to drive the symbolic execution engine in the right direction. As we will see in Section 4 guiding symbolic execution is a popular mechanism to reduce path explosion. Because determining path feasibility is not required (due to the concrete run providing inputs), it reduces the load on the constraint solver which continues to be the largest bottleneck for practical symbolic execution on large programs. In addition, because it is intractable to keep track of the entire environment and effects from a given function call, blending concrete execution with symbolic analysis guides path exploration on the basis of actual execution traces.

Heuristic Search Strategies. Because symbolic execution aims to cover all paths, providing guidance to focus its exploration is critical. As a result,

many researchers employ heuristics to prioritize certain paths over others. For example, SAVIOR [26] (detailed in Section 4.2) uses a bug-driven approach. First, it identifies code regions that trigger a undefined behavior address sanitizer alerts. Then symbolic execution is focused on paths that are more likely to lead to vulnerable code. This selective exploration can rapidly surface interesting program behaviors without exhaustively enumerating all paths, thus mitigating path explosion and accelerating bug discovery.

2.2.2 Parallelization

Distributing symbolic execution workloads across multiple processors or machines allows engines to handle more states in parallel. While parallelization does not inherently eliminate path explosion, it leverages additional computational resources to process a larger number of paths simultaneously, improving scalability and practical applicability. One example of this is the Cloud9 [27] platform which executes symbolic execution work on hardware clusters.

2.2.3 Hybrid Approaches

As noted earlier, fuzzing alone often yields limited code coverage. In contrast, symbolic execution excels in deeper analysis, but there are severe performance penalties. A promising approach combines both techniques: using a fuzzer for rapid exploration and symbolic execution for precision. The idea is to aid the fuzzer when it becomes "stuck" (i.e., fails to explore new paths), symbolic execution can generate inputs to expand coverage. This synergy can reduce the overall complexity that the symbolic engine faces, indirectly helping with path explosion and constraint-solving challenges. In Section 4 there are a few examples of systems which show how fuzzing and symbolic execution can be utilized together. Although hybrid approaches do not directly address all challenges, they improve the engine's efficiency and effectiveness in practice by leveraging symbolic execution when necessary.

Symbolic execution has made significant strides in becoming a viable and practical option for analyzing large-scale software systems, and a wide array of tools exist to support this endeavor. In the next section, we highlight only a few of these tools in an effort to provide a broad overview of the symbolic execution frameworks. Rather than directly comparing the tools against one another, we focus on their diverse applications in tackling challenges such as vulnerability research, program verification, analysis of obfuscated and malicious code, emulation and firmware analysis, and protocol inference.

Following Section 3, we highlight how symbolic execution has been used in various application domains.

3 Symbolic Execution Tools Landscape

Symbolic execution has matured into a practical technique, supported by a diverse landscape of tools. While not an exhaustive list, this curated set showcases tools chosen for their influence, current relevance, and varied capabilities. These tools differ in their operational approach: some operate directly on binaries, while others require source code to translate into LLVM bitcode [28] and operate on that. These tools also differ in their primary focus, supported architectures, and performance characteristics and strategies for mitigating core challenges like path explosion. Understanding these distinctions is crucial for selecting the appropriate tool for a given analysis task.

Symbolic execution tools strive to balance factors such as speed, ease of implementation, and architecture independence. Tools like KLEE [29] and SymCC [30] closely integrate with source-level representations (typically LLVM bitcode), enabling comprehensive analysis integrated early in the development cycle. Conversely, tools such as angr and Triton adopt binary-only approaches, lifting machine instructions to an intermediate representation (IR) for analysis. This lifting is useful in scenarios where source code is not available, such as malware analysis or closed-source vulnerability analysis. Furthermore, frameworks like S^2E [31] combine symbolic and concrete execution to handle entire system stacks, including operating systems and device drivers, while hybrid approaches, exemplified by Driller, fuse symbolic execution with fuzzing to leverage the strengths of both techniques.

These diverse approaches can be broadly categorized based on their primary input analysis strategy: source-based analysis, binary-level analysis, and hybrid analysis (often combining symbolic execution with fuzzing or concrete execution). We will now discuss prominent examples within each category, outlining their methodologies, strengths, weaknesses, and typical application domains.

3.1 Source-Based Analysis

Source-based tools typically operate on intermediate representations such as LLVM bitcode, requiring access to the original source code or a compi-

lable form. The most prominent tools in this category are KLEE [29] and SymCC [30].

KLEE Developed in 2008 as an evolution of the earlier EXE tool [32], KLEE remains one of the most widely used symbolic execution frameworks, cited in numerous publications. KLEE's primary goal is high-coverage automated test generation for C/C++ programs. Key technical contributions include an efficient state representation using object-level copy-on-write memory to minimize state duplication overhead, and optimizations for constraint solving. Specifically, KLEE employs constraint independence (reducing solver queries), significantly reducing the burden on the underlying SMT solver (originally STP, now also supporting Z3). To use KLEE, programs must be compiled to LLVM bitcode. Developers can annotate their code to mark specific inputs as symbolic and add assumptions (e.g., constraining input values). KLEE's reliance on source/bitcode makes it particularly well-suited for integration into software development workflows for thorough testing and bug finding.

SymCC Introduced in 2020, SymCC [30] represents a distinct approach inspired by performance analysis [33] showing significant overhead from IR interpretation. Instead of interpreting bitcode symbolically like KLEE, SymCC compiles symbolic execution capabilities directly into the binary using a custom compiler pass built on LLVM. While SymCC still requires LLVM bitcode as input, the resulting instrumented binary performs symbolic tracking natively during execution. When run, SymCC's instrumentation interacts with a symbolic backend (runtime library) to explore new paths and generate inputs that increase code coverage. This compile-time approach allows, in theory, for recompiling libraries like libc to achieve comprehensive environment modeling. SymCC also handles calls to uninstrumented code gracefully by treating their effects concretely. The developers provide specialized code symbolic models for common C library functions (e.g., memset and memcpy). While initially focused on C/C++, the authors suggest the approach could be extended to other languages supported by LLVM.

KLEE and SymCC, while both source-based, reflect different design priorities originating from how they evaluated their tools. KLEE focused on whether test cases generated by KLEE improved code-coverage while SymCC focused on raw speed and correctness. The creators of KLEE primarily emphasized achieving high code coverage demonstrating effectiveness

in finding deep bugs in complex software, with constraint solving optimizations aimed at making this feasible. In contrast, the SymCC developers focused heavily on execution performance, comparing runtime overhead of their direct compilation approach against traditional interpretation methods, alongside code coverage metrics. This difference highlights a key trade-off: KLEE's architecture facilitates deep program state analysis and complex environment modeling, while SymCC prioritizes minimizing the performance impact of symbolic instrumentation.

3.2 Binary-Based Analysis

On the other end of the spectrum there are engines which do not require access to the source code. Due to the lack of semantic information, binary-based tools typically utilize a "lifter" to transform the machine code up to an intermediate representation (IR). The IR used varies, but the results allow for symbolic analysis on programs where source code is not available. The most prominent tools in this category are S^2E, `angr`, Triton, and BINSEC/SE.

S^2E Introduced in 2011, S^2E (Selective Symbolic Execution Engine) [31] was designed as a general platform for in-vivo multi-path analysis of complex software systems, scaling even to full OS stacks like Windows [31]. It addresses the significant challenge of analyzing software behavior within its real environment (libraries, kernel, drivers) without resorting to potentially inaccurate or labor-intensive abstract models. S^2E achieves this by combining virtualization (using QEMU [34]), dynamic binary translation (interpreting x86 machine code directly), and symbolic execution (leveraging KLEE).

The primary contribution of S^2E is the idea of selective symbolic execution. Instead of executing the entire system symbolically, which is often infeasible due to path explosion, S^2E allows analysts to precisely target specific code regions for symbolic exploration while executing the rest of the system concretely. It manages seamless, automatic, bi-directional transitions between concrete and symbolic modes, ensuring that interactions with the real symbolic state space and focuses analysis effort where needed. S^2E is modular in structure, featuring path selectors (to guide exploration) and path analyzers (plugins that observe or check properties along explored paths). The authors provide several plugins with its release, and they claim that developing new analyzers is trivial. They corroborate this by developing three analyzers using their system, DDT^+, a tool for testing closed-source

Windows device drivers, REV^+, a tool for reverse engineering binary Windows device drivers, and PROFs, a performance profiler and debugger.

Overall, the strength of S^2E lies in enabling complex, system-wide analyses directly on binaries within their native environment.

Triton Introduced in 2015, Triton [35] is presented as a dynamic binary analysis framework built upon Intel Pin [36]. The authors' primary motivation was to provide a modular binary analysis framework that is well suited to handle obfuscation in modern binaries.

Triton offers Python bindings and integrates several key components: dynamic taint analysis ("Hue Engine"), a symbolic execution engine, a snapshot engine for replaying/backtracking execution paths, x86-64 instruction semantics translated to the SMT2-LIB standard format, and an interface to the Z3 SMT solver. These components allow external Python tools built on Triton to perform tasks like symbolic fuzzing, trace analysis, and runtime analysis for vulnerability research. The authors acknowledge path explosion as a general challenge and describe the standard dynamic symbolic execution to negate branch conditions from previous runs. However, they do not explicitly discuss any heuristics to mitigate path explosion beyond the standard dynamic symbolic execution (DSE) workflow.

For constraint solving, Triton performs "backward reconstruction" to build solvable formulas from symbolic expressions. It also provides robust mechanisms to manage constraint complexity through SMT simplification passes. The framework offers two primary approaches to this simplification: an analyst can register custom simplification callbacks based on user-defined rules, or they can convert Triton's internal expressions into a Z3-compatible format to leverage the solver's simplification engine. The authors do not provide a formal evaluation, but rather demonstrate Triton's capabilities through illustrative examples and descriptions of specific runtime analyses implemented using the framework.

angr Next to KLEE, `angr` [37], developed in 2016, is the most influential binary analysis framework. The developers of `angr` aimed to overcome challenges associated with the reproducibility and comparability of previous research prototypes. Key design goals included cross-architecture support, cross-platform compatibility, support for different analyses, and usability via a Python interface.

In order to achieve cross-architecture support, `angr` utilizes Valgrind's VEX IR [38], although `angr`'s modular design allows for other IRs. Cross-

platform binary loading, including dependencies, is handled by the CLE (CLE Loads Every) module. Program state, including registers, memory, filesystem, etc., is managed by the SimState object within the SimuVEX module, which uses a plugin system to support different memory models and track environment interactions. Just like KLEE, `angr` has a rich ecosystem of tools built with `angr` including `angrop` [39] for automating the construction of ROP chains, `patchrex` [40] for patching binaries, and `rex` [41] for automated exploit generation.

The `angr` system incorporates several strategies to address common symbolic execution challenges. To address path explosion, `angr` implements advanced techniques like Veritesting [42] for state merging and support path prioritization strategies. For constraint solving they implement optimizations like splitting constraints into independent sets or using Value-Set Analysis (VSA) [43] to approximate solutions quickly. Finally, to model the environment, `angr` utilizes "SimProcedures" which are Python functions that model their effect on the SimState.

BINSEC/SE BINSEC/SE [44], introduced in 2016, is a dynamic symbolic execution toolkit for binary-level security analysis, particularly reverse engineering tasks like malware analysis and vulnerability research. It is based on the open-source BINSEC [45] platform, which provides a formal, modular environment for binary analysis, featuring disassembly, intermediate representation translation, simulation, and static analysis capabilities. BINSEC/SE leverages the platform's front-end for translating x86 binaries into the Dynamic Bit-vector Automata (DBA) [46] IR. The BINSEC/SE architecture is notable for its modularity, comprising of a PIN-based tracer (PINSEC), a core DSE engine with advanced constraint optimizations, and a flexible path selection module inspired by OSMOE [47]. The initial release of BINSEC/SE only supported x86. However, since its inception, it has evolved to support a broader range of architectures including 64-bit architectures and ARM.

The authors evaluate BINSEC/SE through two reverse engineering case studies:

- Solving a Flare-On challenge crackme by using DSE to find the correct input bytes satisfying path conditions, demonstrating the ability to handle iterative constraints and interact with the tracer to force specific paths.
- Performing malware exploration on 11 samples from the VX Heaven dataset[48], where BINSEC/SE automatically discovered 43 new be-

haviors by systematically negating branch conditions found in initial traces.

3.3 Hybrid Analysis

Driller Driller [49] is a hybrid vulnerability analysis tool designed to find deep bugs in binary applications by combining fuzzing with concolic execution. Driller aims to address the weaknesses inherent in fuzzing and concolic execution by playing off the strengths of both techniques. Driller attempts to mitigate path explosion by offloading most of the path exploration task to its fuzzing engine, using concolic execution only to satisfy complex checks in the application [49].

Driller employs a feedback loop between a fuzzer (American Fuzzy Lop (AFL) [50]) and `angr` as its concolic execution engine. The process begins with AFL fuzzing the application. Once AFL fails to find new interesting states, Driller invokes the concolic execution engine which solves for a specific input and passes it back to the fuzzing component.

For the most part, Driller tackles path explosion by relying on the fuzzer for broad path discovery, and invoking the concolic engine only selectively. Likewise, it mitigates the cost of constraint solving by ensuring the solver is only used to generate the specific inputs necessary to bypass checks that are blocking the fuzzer. Because Driller is built on top of `angr` and AFL QEMU, it relies on those components for modeling the environment and therefore suffers the same drawbacks.

Driller was evaluated on the DARPA CGC dataset[51] containing 126 binaries. The authors compared it against standalone AFL and pure symbolic execution (`angr` with Veritesting). Driller did offer some improvements over AFL, identifying crashes in 77 binaries compared to 68 and 16 in AFL and `angr` respectively.

3.4 Selecting the Right Tool

While the tools described above offer powerful capabilities, they differ significantly in their underlying methodologies, target scope, and approaches to core challenges. Understanding these distinctions is crucial for researchers to select the appropriate tool for their specific analysis goals and constraints. In a development environment, KLEE and SymCC would be the obvious choices given access to source code. Tables 1 and 2 illustrate the characteristics and use-cases for each of the tools reviewed.

These selected tools represent a broad spectrum of symbolic execution

Figure 1: Comparative Overview of Symbolic Execution Tools, Part 1 of 2

Tool Name Reference	**Primary Category**	**Input Required**	**Core Technique(s)**	**Key Features**
KLEE [29]	Source-Based	LLVM Bitcode	Symbolic Execution Object-level COW Constraint Independence Counterexample Caching	High-coverage test generation, Bug finding Environment modeling
SymCC [30]	Source-Based	LLVM Bitcode	Compile-time SE Instrumentation, Concolic	High performance (avoids IR interp.) Flexible solver integration
`angr` [37]	Binary-Based	Binary (Multi-Arch)	DSE, VSA, CFG recovery, Veritesting, Path prioritization	Python framework Flexible, Extensible Arch-agnostic Large community
Triton [35]	Binary-Based	Binary (Multi-Arch)	Concolic Dynamic Binary Analysis Taint Tracking SMT Conversion	Python scripting Runtime tracing Taint analysis focus
S^2E [31]	Full-System Binary-Based	Binary x86 Full System Image	Selective Symbolic Execution Virtualization (QEMU) DBT and Relaxed Consistency Models	Full system analysis (incl. kernel/drivers) In-vivo analysis No env. models needed Consistency trade-offs
BINSEC/SE [44]	Binary-Based	Binary (Multi-Arch)	DSE	Modular, Configurable
Driller [49]	Hybrid Binary-Based	Binary	Fuzzing (AFL) + Selective Concolic Execution (`angr`)	Combines fuzzing breadth + SE depth Finds deep bugs

Figure 2: Comparative Overview of Symbolic Execution Tools, Part 2 of 2

Tool Name Reference	**Primary Use Case(s)**	**Constraint Solver(s)**	**Limitation(s) Trade-offs**
KLEE [29]	Software Testing Bug Detection Source/Bitcode available	STP, Z3, etc.	Requires source/bitcode Path explosion
SymCC [30]	Efficient test gen. Integration into build process	Z3 (default), others	Requires source/bitcode Newer tool/ecosystem
`angr` [37]	Binary Analysis, VR, Malware Analysis, RE, CTFs	Z3 (default), Claripy	Can be slow due to Path explosion Env. modeling complexity
Triton [35]	Obfuscated/Malware analysis RE, Dynamic analysis	Z3 (via SMT2-LIB)	Relies on Intel Pin (mostly x86) Input determinism assumption
S^2E [31]	Firmware analysis Driver testing/RE OS-level Perf. analysis	KLEE solvers (STP, Z3)	Complex setup Performance overhead State explosion
BINSEC/SE [44]	Malware exploration Reverse Engineering	(Z3, CVC4, Boolector)	Relies on PIN traces
Driller [49]	Vulnerability Research (deep bugs)	`angr` solvers (Z3)	Complexity of coordinating fuzzer/SE Relies on underlying tools

approaches, from source-based test generation (KLEE, SymCC) to binary-focused analysis (`angr`, Triton, S^2E, Driller). Their differing methodologies, integrations with constraint solvers, and architectures highlight the flexibility and adaptability of symbolic execution to various analysis challenges. Subsequent sections will reference these tools as examples when discussing how symbolic execution addresses specific problems in areas like vulnerability research, firmware analysis, and obfuscated code analysis.

4 Applications and Guidance Strategies in Symbolic Execution

This section surveys the practical application of symbolic execution across several challenging application domains. The tools presented in Section 3 have diverse design and goals. As a result, they are better suited towards specific tasks. For instance, source-based tools like KLEE are ideal in a development environment, but ill-suited for analyzing malware, which is typically obfuscated and source code is not available.

A unifying theme emerges from the surveyed literature: the practical success of symbolic execution hinges on the ability to intelligently its inherent complexity. Pure, unguided exploration is often intractable for any non-trivial program, a fact established in Section 2.1. Consequently, the practical applications detailed in this section are the result of applying various strategies to make the analysis possible. These strategies can be broadly categorized into:

- Scope Reduction, which limits the analysis to smaller manageable units of code.
- Guidance Heuristics, which intelligently steer the search to prioritize more promise execution paths.

Recently, it has become common to utilize hybrid analysis to implement the strategies above. Hybrid analysis combines symbolic execution with some another analysis technique (often fuzzing) to complement their strengths. This hybrid architecture is not a separate category of complexity management, but rather a powerful pattern for achieving scope reduction, by applying symbolic execution on targeted pieces of code, or implementing guidance heuristics.

Out of the categories outlines above, guidance heuristics represent a diverse and powerful set of techniques. Table 1 presents a taxonomy of

common guidance strategies that will serve as an analytical framework for the remainder of this section.

Guidance Strategy	**Guiding Principle**	**Example Paper(s)**	**Target Domain**
Bug-Driven / Vulnerability-Oriented	Prioritize paths that are statistically or structurally more likely to contain known bug patterns (e.g., near sanitizer alerts or type-unsafe operations).	Saviour, UAFDetect, Vital	General Vulnerability Research
Specification-Guided	Explore paths to check for conformance with, or violations of, a formal protocol specification (e.g., an RFC).	Wen et al., Asadian et al.	Protocol Conformance Testing
Model-Guided	Learn a state machine or model of the system's behavior first, then use that model to guide exploration towards interesting or uncovered states.	MACE, PISE	Protocol Inference & Analysis
Goal-Directed	Focus exploration exclusively on finding a path to a specific target state or program location, pruning all other paths.	Jetset	Firmware Re-hosting, Targeted Analysis
Invalidity-Guided	Learn correct system behavior by analyzing and reasoning about inputs that cause the system to enter an invalid state (e.g., a crash or stall).	uEmu	Firmware Peripheral Modeling

Table 1: Guidance Strategies in Software Analysis

4.1 Program Verification

A key goal in software development is verifying correctness. For simple systems, this is straightforward, however, as software complexity increases, so does the task of validating correctness across all possible inputs. Symbolic execution enables formal verification by exhaustively exploring pro-

gram paths and checking properties like safety, liveness, or adherence to protocols. The works surveyed here provide examples in all three broad categories outlined Table 1.

As noted earlier scope reduction is one strategy to enable scalable symbolic execution. For example under-constrained symbolic execution [52] which analyzes individual functions in isolation. Ramos and Engler [53] apply this technique in UC-KLEE, a framework that bypasses the costly analysis from a program's `main` function to test individual functions in isolation. Hybrid analysis is another powerful option which combine the analysis prowess of symbolic execution with a faster technique. Map2Check [54], for example, adopts this approach by integrating fuzzing to quickly find shallow bugs with symbolic execution used for deeper path exploration. Finally, many approaches rely on guidance heuristics. In the specialized domain of hardware verification, SEIF provides a clear example of a model-guided solution. It addresses information flow in RTL Verilog designs by building a static signal connectivity graph, which then acts as a guide for the symbolic execution engine [55]. SEIF prunes paths that are logically or architecturally impossible.

These approaches, while powerful, highlight persistent challenges. Techniques like under-constrained symbolic execution can suffer from false positives due to missing context or unresolved external calls, while hybrid methods still struggle with complex data structure modeling. Domain-specific approaches like SEIF are effective but may require bounding analysis depth, potentially missing subtle bugs. Future work in this area will likely focus on automatically inferring function preconditions and invariants to reduce false positives and more tightly integrating diverse analysis techniques to improve both speed and precision.

4.2 Vulnerability Research

Vulnerability research is a critical area of study that interests security practitioners, white hat hackers, and developers alike. Identifying vulnerabilities in large, complex software systems is challenging due to code size, execution complexity, and diverse behaviors. Additionally, it can be difficult to verify that a vulnerability identified in an isolated function is truly reachable from an external entry point.

The strength of symbolic execution lies in its ability to reason about a program's internal behavior and automatically generate concrete inputs that trigger specific bugs. This greatly simplifies the task of creating proof-of-concept exploits, streamlining the verification and reproduction of vul-

nerabilities. However, given the vast search space of modern software, a dominant theme in this area is the use of bug-driven guidance to better explore the program. This strategy focuses the symbolic execution engine on paths that are more likely to contain bugs.

This section surveys various approaches that tailor this bug-driven strategy to specific bug classes, discussing their strengths and weaknesses.

Memory Corruption Vulnerabilities: Memory corruption vulnerabilities represent the majority of vulnerabilities discovered in software. Despite the software community's comprehensive understanding of various memory corruption vulnerability classes (e.g., buffer overflows, use-after-free (UAF) vulnerabilities), these flaws remain challenging to detect reliably. Their triggers often depend on intricate program states and specific input sequences, making them challenging to detect. Symbolic execution provides an approach to reliably detect these code violations.

Many tools focusing on bug finding leverage a hybrid analysis architecture while implementing a bug-driven guidance heuristic. SAVIOR [26] represents such an example. It uses UbSan [56] to label suspicious code regions, guiding the fuzzer toward potentially vulnerable paths, and then uses selective symbolic execution to validate errors and generate concrete test cases. UAFDetect [57] uses a bug-driven guidance heuristic approach to constrain symbolic execution and operates in two phases. First, it statically identifies potential UAF code regions. Second, it employs symbolic execution to perform dynamic typestate analysis [58] to track the lifecycle of pointers (e.g., Init, Allocated, Deallocated). UAFDetect prunes paths that are unlikely to cause UAF violations. Vital [59] also utilized a bug-driven approach guiding symbolic execution towards paths containing a high number type-unsafe pointers. The key insight here is that paths with a greater number of type-unsafe pointers are more likely to contain vulnerabilities. VITAL uses Monte Carlo Tree Search (MCTS) to proactively guide KLEE toward vulnerability-rich paths, prioritizing paths in the symbolic execution tree that contain larger numbers of unsafe pointers.

To tackle binary-only targets, researchers often turn to angr. UbSym [60] employs scope reduction introducing a unit-based methodology that addresses scalability challenges differently from the source-code approaches. Instead of applying symbolic execution to whole programs, UbSym breaks programs into smaller units and performs symbolic analysis on individual units that are statically identified as potentially containing vulnerabilities based on defined memory corruption vulnerability specifications. When vul-

nerabilities are detected at the unit level, UbSym utilizes machine learning techniques, specifically the TAR3 learning algorithm [61], a machine learning heuristic for feature ranking, to predict which paths will trigger program-level vulnerabilities.

Collectively, these studies demonstrate a clear trend: guided symbolic execution is essential for efficiently targeting memory corruption vulnerabilities. The strategies for guiding symbolic execution vary significantly based on both the vulnerability class being targeted and the analysis scope. SAVIOR, UAFDetect, and Vital all utilize a two-pass system. First, features are discovered through static analysis, then guiding symbolic execution along paths deemed interesting. In contrast, UbSym focuses on structural properties, decomposing the program and using ML predictions based on unit-level behavior. Furthermore, these approaches differ in scope and tooling. SAVIOR, UAFDetect, and Vital operate at the whole-program level and leverage KLEE as their symbolic execution engine. This makes them more suited for analysis when source code is available. UbSym's use of `angr` and unit-level decomposition showcases an approach for tackling binary-only targets and managing scalability by analyzing smaller code chunks. These complementary approaches highlight the diversity of strategies available for addressing the path explosion problem in symbolic execution while maintaining focus on vulnerability detection.

Concurrency Vulnerabilities: Analyzing multi-threaded programs is a challenge for even human programmers. In symbolic execution, a key challenge in analyzing multi-threaded programs is managing the dual nondeterminism of both symbolic inputs and thread scheduling, both of which exacerbate the path explosion problem. Within concurrency analysis, research has tackled both the correctness of thread interleaving and the raw scalability of the analysis. Research in this space shift the challenge from "which of these millions of paths is most interesting" to "how do we manage the explosion of thread interleavings and shared state?" The surveyed works generally fall into one of two categories: Managing Interleaving Explosion and Providing Foundational Support and Scalability.

To address interleaving explosions, Schemmel et al., [62] integrate symbolic execution into a partial-order reduction [63] (POR). They developed a KLEE-based prototype that constructs an unfolding of the program that captures concurrent behaviors while merging equivalent execution prefixes. To prune redundant paths, they employ cutoff events, detecting when a state has already been reached via a shorter path, and restrict scheduling

decisions to synchronization primitives (mutex, condition variables), also reporting data races as errors. While this technique has proved effective at identifying previously unknown vulnerabilities in Memcached [64], it focuses on reducing interleaving explosion rather than providing comprehensive threading models. As a result, it may still face scalability issues with very large programs. To address the separate challenge of scalability, researchers have explored distributing the symbolic workload itself. Bucur et al., [27] parallelize symbolic execution across a commodity cluster using a dynamic load-balancing scheme. The Cloud9 system, built on top of KLEE, distributes states among multiple nodes and provides a comprehensive symbolic model of the POSIX interface, including robust support for threads, networking, and I/O.

To provide better foundational support for analyzing multi-threaded programs, two works provide prominent examples. Notable efforts from Vishnyakov et al.'s Symbolic DynamoRIO (Sydr) [65], which leverages the Triton symbolic engine to analyze multi-threaded binaries. Sydr manages separate symbolic states for each thread and handles context switches but, as the authors note, it currently does not influence thread scheduling, a feature planned for future work. Similarly, Niskov et al. [66] introduced enhancements to the S^2E platform. Their contributions include enabling S^2E to support multiple virtual cores and implementing a plugin that functions as a data race checker for multi-threaded programs. This race checker's design is inspired by the $DJIT^+$ algorithm [67], aiming to detect concurrency defects within the symbolic execution framework. These developments signify important strides in extending the capabilities of binary symbolic execution to the complex domain of multi-threaded software analysis.

The primary limitations in this area remain performance and modeling complexity. Even with techniques like POR, memory usage can become prohibitive for long-running programs, and not all tools can yet influence thread scheduling to explore specific interleaving. Future improvements will likely focus on more advanced heuristics for path pruning and richer modeling of both synchronization primitives and external library calls to increase precision and scalability.

Specialized Vulnerability Domains: The previous paragraphs covered research in well-known application domains. However, symbolic execution has applications in a wide variety of specialized areas. While we cannot discuss all of the different domains, we highlight research we found most interesting. Specifically, we examine how symbolic execution has been applied

to specialized domains, focusing on smart contracts and microarchitecural side-channel attacks.

In the smart contract space, we highlight two prominent studies, both taking a learning-based approach. He et al. [68] employ a hybrid approach and introduce Imitation Learning Fuzzer (ILF) where a fuzzer learns to imitate the behavior of a symbolic execution 'expert' [1], aiming to combine the strengths of both techniques. The core idea is for the fuzzer to learn an effective fuzzing policy for generating input sequences by observing a symbolic execution 'expert'. Because symbolic execution of entire smart contracts is expensive even for small contracts, the expert symbolically executes a individual transactions and concretizes inputs to optimize coverage. The symbolic execution engine generates inputs leading to maximum coverage or indicates when no input would improve coverage. This enables the fuzzer to achieve better coverage than existing smart contract fuzzers. ILF's primary goal is to detect vulnerabilities in Ethereum [69] smart contracts [2] using specific detectors for each vulnerability class. MythrilQL [70] is an example of bug-driven guidance. It leverages Q-learning as a mechanism to guide the symbolic execution engine's path selection towards vulnerable paths. Additionally, they implement an incentive based path pruning strategy to eliminate redundant or useless paths. Finally, for constraint solving efficiency, they predict solution time to determine whether a constraints should be passed to the solver.

Another compelling research direction utilizes symbolic execution to identify Spectre vulnerabilities [71]. These vulnerabilities differ from traditional memory safety bugs because they only manifest during speculative execution. That is when processors execute instructions based on predictions that may prove incorrect. Attacks against these vulnerabilities exploit side effects resident by these executions even if the processor reverts its state when predictions fail. The challenge for detection tools is modeling both regular execution paths and speculative (transient) paths simultaneously while managing exponential explosion of possible execution traces. This creates a unique state explosion problem, similar to the interleaving explosion in concurrent programs. To tackle this, Daniel et al.'s [72] BINSEC/HAUNTED utilize relational symbolic execution [73], [74] (RelSE) [3] to tackle the challenge of detecting Spectre vulnerabilities [71]. This al-

[1] ILF utilizes the VerX symbolic execution engine

[2] The specific vulnerabilities are Locking, Leaking, Suicidal, Block Dependency, Unhandled Exception, and Controlled Delegatecall

[3] Relational symbolic execution is a technique that analyzes pairs of execution traces simultaneously.

lows them to analyze both regular and transient execution paths in a single run, enabling efficient reasoning about side-channel leaks caused by speculative misprediction. Haunted RelSE represents transient executions alongside regular executions without explicitly exploring all speculative paths.

While these specialized approaches show great promise, their focus is also their limitation. Learning-based tools like ILF may struggle with smart contracts that have unique logic that is not present in the training data. Similarly, while BINSEC/HAUNTED scales better than explicit speculative exploration, it remains infeasible for very large binaries and is currently limited to detecting specific Spectre variants (PHTand STL). Future work could refine these specialized heuristics and extend them to cover a broader range of behaviors of attack classes.

4.3 Obfuscated and Malicious Code Analysis

Obfuscation is often used to hinder reverse engineering and make static analysis difficult by transforming code logic into more complex or misleading forms. Common techniques include control-flow flattening, encryption of constants, and self-modifying code, all of which complicate attempts to reason about program behavior. Malware authors routinely exploit such obfuscation to evade detection, creating an ongoing arms race where security researchers strive to uncover malicious intent and shared code patterns across evolving malware variants.

Symbolic execution offers a powerful countermeasure by systematically exploring paths within obfuscated binaries and reasoning about hidden logic. That is, symbolic execution can reveal malicious routines, identify hidden command structures, and generate meaningful inputs to drive deeper analysis. However, symbolic execution still faces significant hurdles in practice: large search spaces, runtime-based anti-analysis, and environment-specific dependencies can all limit its effectiveness. To overcome these hurdles, researchers have employed a variety of strategies, including goal-directed guidance to focus on malicious behaviors, scope reduction to make large-scale analysis feasible, hybrid analysis to combine complementary techniques, and specialized methods like invalidity-guided analysis to defeat specific obfuscation schemes.

Several approaches use symbolic execution to extract semantic features for malware classification and analysis. For instance, SEMA [75] utilize a goal-directed guidance strategy to build System Call Dependency Graphs (SCDGs) that serve as behavioral signatures. The goal is not necessarily a specific line of code, but rather to generate a behavioral signature. To al-

leviate path explosion, they implement a custom breadth-first search strategy [76] that prioritizes interesting paths for exploration. SEMA extends `angr` with specialized strategies to create representative signatures based on SCDGs to feed machine learning models for classification. Vouvoutsis et al. [77] take a different approach, applying scope reduction at the dataset level. They developed a detection pipeline that balances accuracy and automated detection of evasive malware. Their pipeline clusters malware using TLSH, performs symbolic execution on representatives from each cluster using `angr` to gather API call features, applies unification to the extract API features, and finally feeds these features to machine learning classifiers. Beyond classification, Botacin et al. [78] introduce Malverse, which uses a goal-directed approach to automatically identify execution paths hidden behind evasive conditions and automatically patch the binary. Malverse uses a Bayesian model as the heuristic to steer symbolic execution toward potentially malicious paths. Malverse uses symbolic execution to discover function inputs and returns that trigger malicious behaviors, the malware is then patched with values satisfying the conditions forcing the malware to execute its malicious behavior.

Symbolic execution has also been employed to automatically deobfuscate code. Salwan et al. [21] implement a unique hybrid approach that combines symbolic execution with taint analysis to automatically deobfuscate virtualized code. Their approach, built on Triton, uses taint analysis to isolate the pertinent instructions related to the original program logic, and then uses symbolic execution to reconstruct their behavior. Another common obfuscation technique is an opaque predicate. Bardin et al. [22] take an invalidity-guided approach. The authors introduce a technique, Backward-Bounded Dynamic Symbolic Execution (BB-DSE), to identify opaque predicates by proving their branches are unreachable. BB-DSE works backward from a suspicious region of code to determine if they can ever be satisfied allowing the obfuscation artifacts to be eliminated.

These approaches reveal several methods in which symbolic execution provides leverage against obfuscated and evasive malware. While powerful, these applications highlight the persistent challenges of scalability and environment modeling. Strategies to cope with these challenges include reducing the analysis scope via clustering (Vouvoutsis et al.), employing heuristics and targeted analysis (SEMA, MalVerse, BB-DSE), and focusing on specific, constrained problems (Salwan et al).

4.4 Emulation and Firmware Analysis

Embedded systems are becoming increasingly popular. Unfortunately, their increased popularity and widespread use has not resulted in better software hygiene when developing for embedded systems. Because these systems are often constrained in both compute power and storage, developers want to get the absolute most out of the hardware on these systems. As a result, security is often overlooked for the sake of performance.

Firmware analysis presents unique challenges due to its dependence on hardware for execution. Tools like QEMU[4] and Unicorn [5] allow firmware emulation, but replicating behavior of hardware peripherals often requires detailed hardware specifications that are not always available. The concept of firmware re-hosting was usually done in ad-hoc environments for smaller projects. However, the rise of the use of embedded devices has brought scrutiny to the field of firmware re-hosting. Fasano et al [79] described the challenges in re-hosting embedded devices. In order to properly emulate firmware, you often need a significant amount of information about the hardware (e.g., memory maps, peripheral register behaviors) which is not always readily available. Symbolic execution can bridge this gap by learning peripheral responses dynamically, reducing the reliance on hardware models. Research in the area of firmware emulation has diverged into two main camps: high-fidelity analysis using hardware-in-the-loop (HIL), and hardware-free analysis that infers peripheral behavior using strategies like hybrid analysis and various forms of guidance.

HIL approaches offer the best fidelity. By incorporating physical hardware, there is no need to speculate peripheral responses or firmware behavior. Avatar2 [80] stands out not as an analysis technique, but rather a mature orchestration platform. It addresses the interoperability problem between analysis tools by enabling coordination among emulators (QEMU, PANDA), debuggers (GDB, OpenOCD), symbolic execution engines (`angr`), and physical hardware. While `angr` was used in their original implementation, Avatar2's modular design allows many different configurations. Another HIL technique is CO3 [81] which utilizes a hybrid approach. CO3 brings hybrid (concolic) execution to resource-constrained MCUs by offloading the complex analysis to a powerful workstation while executing on the real device.

In contrast, several systems use symbolic execution to enable firmware analysis without physical hardware by modeling or inferring peripheral be-

[4] `https://www.qemu.org/`

[5] `https://github.com/unicorn-engine/unicorn`

havior. These approaches differ in their specific goals and methodologies. Despite a lack of hardware, a hybrid approach is still feasible through emulation. For instance, Fuzzware [82] takes a hybrid approach which uses `angr` to learn precise Memory Mapped Input Output (MMIO) models that guide fuzzing to more meaningful inputs. Specifically, Fuzzware uses `angr` to determine precise models of how the firmware reads from MMIO registers allowing Fuzzware to build abstract models providing meaningful choices to the fuzzer. For instance, if `angr` classifies a peripheral interaction as constant, the fuzzer consistently produces the same value for that peripheral access. uEmu [83] takes a different approach. It uses invalidity-guided symbolic execution with S^2E to infer correct register responses by learning from states that cause the firmware to crash or stall. The peripheral responses are stored in a knowledge base which can be used to subsequently run the firmware in an emulator for fuzzing. For targeted re-hosting scenarios, Jetset [84],implemented using `angr` and QEMU, uses goal-directed symbolic execution to find only the specific I/O constraints necessary to boost a firmware image to a target address. These methods demonstrate a trade-off between the high fidelity of HIL systems and the broader applicability and scalability of peripheral inference techniques.

Firmware analysis is challenging due to how tightly coupled the behavior is tied to specific hardware peripherals, which are often undocumented or hard to accurately emulate. While Avatar2 and CO3 aim for high fidelity by integrating hardware in their analysis, uEmu, Jetset, and Fuzzware take a different approach by utilizing symbolic execution to infer peripheral behavior. Still, the approaches to infer peripheral behavior differ significantly. uEmu uses invalidity guidance to infer correct register responses. Jetset uses goal-directed symbolic execution to find constraints just sufficient to reach a target state for rehosting. Finally, Fuzzware uses symbolic execution to learn precise, compact models of MMIO behavior to optimize fuzzing. Each approach has its own tradeoffs and goals. However, the key insight across all peripheral modeling approaches is that emulating sufficient peripheral behavior to enable firmware execution and testing does not require perfect accuracy.

4.5 Protocol Inference and State Analysis

Effective communication requires protocols that establish an agreement between the communicating parties on how communication is to proceed [85]. While formal methods can verify protocol designs using tools such as CPSA [86], ensuring that a software implementation correctly adheres to its specifica-

tion is a distinct and critical challenge. Deviations from specifications can introduce vulnerabilities, potentially subverting the protocol's intended security guarantees. For example, implementation vulnerabilities such as the Heartbleed vulnerability [87], [88], the POODLE attack [89], and a recent OpenSSL Vulnerability [90] arise from deviations or errors in protocol implementations. This challenge is heightened when analyzing proprietary or malicious protocols (e.g., malware command and Control), where the specification is unknown and must be reverse engineered.

Symbolic execution is a powerful technique for this domain, but the stateful nature of protocol can quickly lead to path explosion. To manage this complexity, researchers primarily employ two powerful guidance strategies: model-guided analysis, where a state machine is learned from the implementation, and specification-guided analysis, which uses a formal specification, such as an RFC, to direct the search for bugs.

A dominant strategy is model-guided symbolic execution, where a state machine is first learned from the implementation and then used to steer analysis. L* [91] is the primary algorithm used to infer a protocol. MACE [92], PISE [93], and Wen et al. [94] employ this strategy. MACE iteratively infers a finite-state protocol model from the application's input/output behavior, which is then used to guide concolic execution using DART [95]. PISE uses symbolic execution with `angr` to guide message exchanges and pair it with an extended L* automata-learning algorithm. By instrumenting send/receive functions and systematically constraining message bytes, PISE discovers valid protocol states and message formats through incremental collision resolution and alphabet refinement. While we have primarily used our taxonomy to label research into one of the guidance strategies, we note that some systems may employ a blend of these strategies. Wen et al. combine both approaches: they start with an RFC-defined message format (spec guidance) to build an initial model, then use L* to infer a state machine (model-guided). With this, they are able to utilize L* to generate a finite-state machine which guides the symbolic execution engine, S^2E, to less traversed paths. Asadian et al. [96] follow a purely specification-driven approach. First, they translate the protocol's RFC message and state rules into logical assertions for KLEE. Then KLEE explores those assertions to uncover implementation paths that conflict with the spec. To demonstrate the effectiveness of their approach, the authors applied it to the Datagram Trannsport Layer Security (DTLS) protocol and successfully reproduced CVE-2014-0195. The primary drawback is manually extracting and formalizing an RFC's details is labor-intensive and potentially error-prone. Another interesting approach is from Sun et al. [97] who introduce

Spenny, which uses a "field coverage" metric to guide the analysis of proprietary Industrial Control System (ICS) protocols from firmware binaries. A fundamental challenge in this domain is handling cryptography; Vanhoef and Piessens [98] tackle this by simulating cryptographic primitives under the Dolev-Yao model [99], enabling analysis of security protocol implementations without getting stalled by complex SMT constraints.

As we observed in Section 4.2, guiding symbolic execution is crucial especially when dealing with stateful protocols. The guidance provided varies from inferred models, specifications, or domain-specific heuristics. As with all other challenges, path/state explosion remains a concern even with guidance. Furthermore, handling complex environment interaction (network responses, timing) complicates analysis.

5 Challenges and Future Directions

While symbolic execution has matured into a practical analysis technique, its full potential is constrained by foundational challenges that continue to drive research. As discussed in Section 2.1, issues such as path explosion, the computational cost of constraint solving, and the complexities of environment modeling remain significant hurdles. While much progress has been made, future work will likely focus not only on refining existing solutions but also on pioneering new approaches that integrate symbolic execution more deeply into the software life-cycle and apply it to increasingly sophisticated problem domains. This section outlines several promising research directions, moving from core technique enhancements to domain-specific applications.

5.1 Adapting Symbolic Execution to Real-Time Operating Systems

Throughout this survey, we have presented the plethora of applications in which symbolic execution has been used. The application of symbolic execution in analyzing real-time operating systems (RTOS) remains underrepresented. RTOSes are widely deployed in safety and mission-critical applications including automotive, aerospace, and medical industry. Given their key role in critical systems, ensuring their implementation are correct and secure is essential. RTOSes pose unique analysis challenges. because timing guarantees are fundamental to their correct operation Modeling this in a symbolic execution environment requires explicitly representing time with symbolic variables. This would enable the engine to reason about deadlines

and worst-case execution times (WCET). The challenge lies in determining which timing-related constraints should be modeled while balancing precision with practicality.

Furthermore, RTOSes typically leverage multitasking with priority-based scheduling, preemptions, and interrupts, all of which significantly hinder path exploration. As with the analysis of multi-threaded systems, symbolic execution must consider various ways tasks can interleave under preemption and time constraints. As a result, the path explosion problem is amplified. Partial-order reduction has proved effective at managing thread interleavings, however, it would like need to be adapted to fit this specific execution environment.

5.2 Automated Characterization of Evasive Triggers in Malware

Symbolic execution is often applied to extract features for classification, described in Section 4.3. However, a significant opportunity exists to move beyond classification and toward characterization, that is, understanding precisely how a malware specimen avoid detection. Symbolic execution could be used to automate this characterization to pinpoint the exact environmental conditions a malware sample checks. Instead of just bypassing a check, the research would focus on using the path constraints generated by the symbolic execution to produce a human-readable "trigger signature." For example, the analysis would not just find an input to bypass an anti-VM check; it would produce the very predicate the malware is solving for: (result_of_cpuid_vendor_string == "VMWareVMWare"). This research would address several open questions:

- Can symbolic constraints generated from analyzing anti-analysis code be automatically simplified into concise, semantic predicates representing the evasion check?
- How can this technique scale to handle complex, multi-stage checks or environmental interactions (e.g., checking for specific registry keys, file paths, or running processes)?
- Could the resulting database of trigger signatures be used to automatically classify evasion techniques across different malware families, providing a deeper understanding of the attacker's toolkit?

5.3 Analyzing Type-Safe Languages

The increasing adoption of type-safe languages like Rust and Go is shifting the focus of vulnerability research from detecting memory corruption bugs to identifying logic errors and concurrency issues. This shift necessitates an evolution in how symbolic execution is applied. While significant research by Schemmel et al. [62] and others has advanced the analysis of traditional thread-based concurrency in C programs, the structured concurrency models of modern languages like Rust and Go present new and distinct challenges. This future work would build upon foundational concepts like partial-order reduction but would require re-imagining them for these new paradigms.

For a language like Rust, a primary research thrust is the formal verification of unsafe blocks. Symbolic execution is an ideal tool to formally analyze these small, critical sections of code where the language's safety guarantees are manually suspended. Furthermore, with memory safety largely guaranteed in safe code, symbolic execution can focus on program correctness. This leads to questions such as: can symbolic execution find inputs that trigger denial-of-service conditions by causing a panic? Or can it verify higher-level semantic properties, such as "this function should never return an error if the input is positive"? While Rust's ownership model prevents data races at compile time, it does not prevent all concurrency bugs, such as deadlocks or logical race conditions. Symbolic execution could explore thread inter-leavings with Rust's concurrency primitives to find these higher-level bugs.

Similarly, a language like Go introduces its own unique modeling challenges, particularly concerning its garbage collector. Since the symbolic state of memory can be altered at any point by the garbage collector, this raises a critical research question: how can the effects of a garbage collector be soundly modeled within a symbolic execution engine without sacrificing performance, or can its effects be safely abstracted away?

5.4 LLM-Assisted Symbolic Execution

The rise of large language models (LLMs) presents an opportunity to blend these two technologies. While research in LLMs is still in its infancy, several promising directions have begun to emerge [100]–[103]. LLMs could be applied to address the primary challenges in symbolic execution: path explosion, constraint solving, and environment modeling.

- Path Explosion: To manage potentially infinite state spaces, LLMs can assist in both the selection of paths and pruning of states. A

hybrid system using an LLM can analyze a codebase to identify critical sections and guide symbolic execution towards those paths, avoiding those that are less relevant. This approach helps prioritize paths based on the user's goal and prune paths that do not align with those goals.

- Constraint Solving: LLMs can also be used to lessen the high computational cost of the constraint solving process. There are a few ways in which LLMs can be applied here. First, as a means of supporting new languages that are difficult to model. For example, utilizing an LLM to support the list data type in Python [100]. Alternatively, LLMs might be used as either a constraint solver for simpler queries, relieving the more expensive solver of that work, or as a means to simplify constraints before passing them onto a more robust solver.

- Environment Modeling: LLMs can provide great utility in environment modeling. Instead of requiring developers to manually write models for complex system calls or library functions, an LLM can be used to infer and predict the behavior of these external interactions [103]. In this scenario, the LLM can act as the environment with which the symbolic execution engine interacts, improving the fidelity of the analysis.

A primary concern regarding using LLMs in symbolic execution is managing the probabilistic nature of LLMs. In addition, working with LLMs could require more fine-tuned models specifically designed for the domain to ensure maximum fidelity and accuracy.

In addition to these directions, new application domains continue to emerge. The rise of IoT devices, automotive software, and even quantum computing platforms expands the frontier for symbolic execution, creating a need for multi-architecture, domain-specific solutions. While these challenges may not have immediate solutions, ongoing research seeks to adapt symbolic execution frameworks and constraint solvers to handle unconventional architectures and computation models.

Overall, the future of symbolic execution lies in making the technique more scalable, more versatile, and more accessible. This could be in the form of utilizing guidance strategies more effectively more robust environment modeling. By building on established suggestions from the literature and drawing on insights from research efforts in program analysis, automated reasoning, and domain-specific tool development, the community can continue to refine symbolic execution as a critical tool for understanding and securing complex software systems.

6 Conclusion

Symbolic execution continues to be a critical tool in various application domains. Our survey demonstrates that despite foundational issues such as path explosion, complex constraint solving, and accurate environment modeling (Section 2.1), researchers have employed sophisticated guidance strategies and heuristics to mitigate these issues and broaden the applicability of symbolic execution.

Combining fuzzing with symbolic execution has shown promise in addressing the weaknesses of both techniques. Fuzzing often can "stall" and fail to discover new and interesting code paths. Hybrid approaches, exemplified in tools like Driller [49] and guided techniques like SAVIOR [26], capitalize on fuzzing's speed for broad exploration while using symbolic execution's analytical power to bypass complex checks. Future work could focus on enabling richer feedback between the fuzzer and symbolic engine. This might involve leveraging detailed constraint metrics or incorporating lightweight static analyses to more intelligently guide the exploration process.

In vulnerability research, guided symbolic execution has proven effective for memory corruption issues. Researchers have utilized heuristics based on runtime events [26], [57], static properties [59], or structural decomposition [60]. A natural progression is to extend these guided techniques to detect other classes of bugs such as type confusion and logic errors at both the binary and source levels.

The challenges posed by analyzing malware and obfuscated code continue to be of great interest. We've seen symbolic execution be used as another method to extract features for malware classification. Future directions in this area include developing techniques to systematically explore malware binaries and identify hidden checks the malware performs before triggering the malware. This could be taken further to use constraint solving to identify the exact check the malware performs (e.g., "Is the result of CPUID vendor string 'VMWare?'"). This would provide an automated identification and characterization of these evasion triggers directly from the binary code which could be used to classify different evasion techniques based on the patterns observed in their symbolic constraints.

In firmware and protocol analysis, the need to balance fidelity with scalability remains a challenge. Symbolic execution has been utilized to in hardware-software co-execution frameworks [80], [81], for inferring peripheral behavior to enabled emulation or guided fuzzing [82]–[84], [104], and for protocol inference or conformance testing using learned models or spec-

ifications [92]–[94], [96], [97]. Future work could focus on improving the fidelity/scalability or achieving more complete automated inference of complex or encrypted protocols.

Overall, the future of symbolic execution likely lies in its intelligent integration with other techniques, fuzzing, static analysis, machine learning, and its specialization for challenging domains like firmware, protocols, and concurrent systems. These advances are essential as software systems grow ever more complex, ensuring that both vulnerabilities are detected early and that our defenses remain robust in the face of evolving threats.

References

[1] J. King, "Symbolic execution and program testing," in *Communications of the ACM Volume 19 Number 7*, 1976, pp. 385–394.

[2] W. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 266–278, 1977. DOI: 10.1109/TSE.1977.231144.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[4] J. King, "A new approach to program testing," in *Communications of the ACM Volume 19 Number 7*, 1975, pp. 228–233.

[5] L. D. Moura and N. Bjorner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software*, ACM, 2008, pp. 337–340. DOI: 10.1145/2345156.2254088.

[6] C. Barrett, C. L. Conway, M. Deters, *et al.*, "Cvc4," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 171–177.

[7] A. Sabbaghi and M. R. Keyvanpour, "A systematic review of search strategies in dynamic symbolic execution," *Computer Standards & Interfaces*, vol. 72, p. 103 444, 2020.

[8] P. Bourque and R. E. Fairley, Eds., *Guide to the Software Engineering Body of Knowledge, Version 3.0 (SWEBOK Guide V3.0)*. IEEE Computer Society, 2014, ISBN: 978-0769551661. [Online]. Available: http://www.swebok.org.

[9] J. Jaffar, J. A. Navas, and A. E. Santosa, "Unbounded symbolic execution for program verification," in *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2*, Springer, 2012, pp. 396–411.

[10] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 23–33.

[11] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2012, pp. 193–204. DOI: 10.1145/2345156.2254088.

[12] L. De Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, Springer, 2009, pp. 23–36.

[13] V. Ganesh and D. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, vol. 4590, Jan. 2007, pp. 519–531, ISBN: 978-3-540-73367-6. DOI: 10.1007/978-3-540-73368-3_52.

[14] E. Ábrahám and G. Kremer, "Smt solving for arithmetic theories: Theory and tool support," in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, IEEE, 2017, pp. 1–8.

[15] "Ieee standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985. DOI: 10.1109/IEEESTD.1985.82928.

[16] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM computing surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[17] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zahl, and K. Wehrle, "Floating-point symbolic execution: A case study in n-version programming," in *2017 32nd IEEE/ACM International Con-*

ference on Automated Software Engineering (ASE), IEEE, 2017, pp. 601–612.

[18] Z. Fu and Z. Su, "Achieving high coverage for floating-point code via unconstrained programming," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 306–319, 2017.

[19] G. Zhang, Z. Chen, and Z. Shuai, "Symbolic execution of floating-point programs: How far are we?" In *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, 2022, pp. 179–188. DOI: 10.1109/APSEC57359.2022.00030.

[20] A. Bakst, K. Gleissenthall, R. Kici, and R. Jhala, "Verifying distributed programs via canonical sequentialization," in *Proceedings of the ACM on Programming Languages, Volume 1 Issue OOPSLA*, ACM, 2017, pp. 1–27. DOI: 10.1145/3133934.

[21] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018.

[22] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded dse: Targeting infeasibility questions on obfuscated codes," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 633–651. DOI: 10.1109/SP.2017.36.

[23] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[24] J. Siddiqui and S. Khurshid, "Scaling symbolic execution using ranged analysis," in *Proceedings of the ACE International Conference on Object Oriented Programming Systems Languages and Applications*, ACM, 2012, pp. 523–536. DOI: 10.1145/2384616.2384654.

[25] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[26] Y. Chen, P. Li, J. Xu, *et al.*, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1580–1596.

[27] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 183–198.

[28] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[29] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[30] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" In *29th Usenix Security Symposium*, 2020, pp. 123–156.

[31] V. Chipounov, V. Kuznetsov, and G. 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*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 265–278, ISBN: 9781450302661. DOI: 10.1145/1950365.1950396. [Online]. Available: https://doi.org/10.1145/1950365.1950396.

[32] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. 1. R. Engler, "Exe: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ACM, 2006, pp. 322–335.

[33] S. Poeplau and A. Francillon, "Systematic comparison of symbolic execution systems: Intermediate representation and its generation," in *Annual Computer Security Applications Conference*, Dec. 2019, p. 14.

[34] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[35] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *In Symposium sur la securite des technologies d l'information et des communications, SSTIC, France, Rennes*, 2015, pp. 31–54.

[36] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, ser. WCAE '04, Munich, Germany: Association for Computing Machinery, 2004, 22–es, ISBN: 9781450347334. DOI: 10.1145/1275571.1275600. [Online]. Available: https://doi.org/10.1145/1275571.1275600.

[37] Y. Shoshitaishvili, R. Wang, C. Salls, *et al.*, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[38] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[39] salls. "angrop: Angr ROP gadget finder." GitHub repository, Accessed. (), [Online]. Available: https://github.com/salls/angrop (visited on 07/20/2025).

[40] angr Project Developers. "patcherex." GitHub repository, Accessed. (), [Online]. Available: https://github.com/angr/patcherex (visited on 07/20/2025).

[41] angr Project Developers. "rex: The angr Exploit Generator." GitHub repository, Accessed. (), [Online]. Available: https://github.com/angr/rex (visited on 07/20/2025).

[42] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 1083–1094, ISBN: 9781450327565. DOI: 10.1145/2568225.2568293. [Online]. Available: https://doi.org/10.1145/2568225.2568293.

[43] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[44] R. David, S. Bardin, T. D. Ta, *et al.*, "Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 653–656.

[45] A. Djoudi and S. Bardin, "Binsec: Binary code analysis with low-level regions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2015, pp. 212–217.

[46] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 165–170.

[47] S. Bardin and P. Herrmann, "Osmose: Automatic structural testing of executables," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 29–54, 2011.

[48] UCI Machine Learning Repository. "Malware static and dynamic features vxheaven and virus total dataset." Accessed. (2019), [Online]. Available: `https://archive.ics.uci.edu/dataset/541/malware+static+and+dynamic+features+vxheaven+and+virus+total`.

[49] N. Stephens, J. Grosen, C. Salls, *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," English (US), in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016*, ser. 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, Publisher Copyright: © 2016 Internet Society.; 23rd Annual Network and Distributed System Security Symposium, NDSS 2016 ; Conference date: 21-02-2016 Through 24-02-2016, The Internet Society, 2016. DOI: `10.14722/ndss.2016.23368`.

[50] M. Zalewski, *American Fuzzy Lop*, `http://lcamtuf.coredump.cx/afl/`, Accessed: 2025-03-30.

[51] M. L. Laboratory, *Cyber grand challenge datasets*, `https://www.ll.mit.edu/r-d/datasets/cyber-grand-challenge-datasets`, Accessed: April 30, 2025.

[52] D. R. Engler and D. Dunbar, "Under-constrained execution: Making automatic code destruction easy and scalable," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, IEEE, 2007, pp. 360–371.

[53] D. A. Ramos and D. Engler, "{Under-constrained} symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[54] H. Rocha, R. Menezes, L. C. Cordeiro, and R. Barreto, "Map2check: Using symbolic execution and fuzzing: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26*, Springer, 2020, pp. 403–407.

[55] K. Ryan and C. Sturton, "Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, IEEE, 2023, pp. 110–121.

[56] Clang Documentation, *UndefinedBehaviorSanitizer*, `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, Accessed: 2025-03-21.

[57] Z. Huang, "Targeted symbolic execution for uaf vulnerabilities," in *2023 7th International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2023, pp. 282–289.

[58] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE transactions on software engineering*, no. 1, pp. 157–171, 1986.

[59] H. Tu, L. Jiang, and M. Böhme, "Vital: Vulnerability-oriented symbolic execution via type-unsafe pointer-guided monte carlo tree search," *arXiv preprint arXiv:2408.08772*, 2024.

[60] S. Baradaran, M. Heidari, A. Kamali, and M. Mouzarani, "A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes," *International Journal of Information Security*, vol. 22, no. 5, pp. 1277–1290, 2023.

[61] T. Menzies and Y. Hu, "Data mining for very busy people," *Computer*, vol. 36, no. 11, pp. 22–29, 2003.

[62] D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, and K. Wehrle, "Symbolic partial-order execution for testing multi-threaded programs," in *International Conference on Computer Aided Verification*, Springer, 2020, pp. 376–400.

[63] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 373–384, 2014.

[64] Memcached Development Team. "Memcached." Accessed. (), [Online]. Available: `https://www.memcached.org/` (visited on 07/19/2025).

[65] A. Vishnyakov, A. Fedotov, D. Kuts, *et al.*, "Sydr: Cutting edge dynamic symbolic execution," in *2020 Ivannikov ISPRAS Open Conference (ISPRAS)*, IEEE, 2020, pp. 46–54.

[66] F. Niskov, E. Kutovoy, and S. Kurmangaleev, "Enhancing s2e to analyze multi-thread programs,"

[67] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2003, pp. 179–190.

[68] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 531–548.

[69] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[70] M. Wang, W. Fei, M. Wang, and J. Cui, "Reinforcement learning guided symbolic execution for ethereum smart contracts," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2023, pp. 91–100.

[71] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[72] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse," in *NDSS 2021-Network and Distributed Systems Security*, 2021.

[73] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: Testing for divergences between software versions," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1181–1192.

[74] G. P. Farina, S. Chong, and M. Gaboardi, "Relational symbolic execution," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, 2019, pp. 1–14.

[75] C.-H. Bertrand Van Ouytsel, C. Crochet, T. Dam, and A. Legay, "Tool paper - sema: Symbolic execution toolchain for malware analysis," in May 2023, pp. 62–68, ISBN: 978-3-031-31107-9. DOI: 10.1007/978-3-031-31108-6_5.

[76] C.-H. B. V. Ouytsel and A. Legay, *Malware analysis with symbolic execution and graph kernel*, 2022. arXiv: 2204.05632 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2204.05632.

[77] V. Vouvoutsis, F. Casino, and C. Patsakis, "Beyond the sandbox: Leveraging symbolic execution for evasive malware classification," *Computers & Security*, vol. 149, p. 104 193, 2025.

[78] M. Botacin and A. Grégio, "Malware multiverse: From automatic logic bomb identification to automatic patching and tracing," *arXiv preprint arXiv:2109.06127*, 2021.

[79] A. Fasano, T. Ballo, M. Muench, *et al.*, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 687–701, ISBN: 9781450382878. DOI: 10.1145/3433210.3453093. [Online]. Available: https://doi.org/10.1145/3433210.3453093.

[80] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.

[81] C. Liu, A. Mera, E. Kirda, M. Xu, and L. Lu, "{Co3}: Concolic co-execution for firmware," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5591–5608.

[82] T. Scharnowski, N. Bars, M. Schloegel, *et al.*, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.

[83] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2007–2024, ISBN: 978-1-939133-24-3. [Online].

Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou.

[84] E. Johnson, M. Bland, Y. Zhu, *et al.*, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 321–338.

[85] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th. Pearson Education, 2011.

[86] M. D. Liskov, J. D. Ramsdell, J. D. Guttman, and P. D. Rowe, *The cryptographic protocol shapes analyzer: A manual for CPSA 4*, CPSA Version 4.3, January 13, 2023, The MITRE Corporation, 2023. [Online]. Available: https://hackage.haskell.org/package/cpsa-4.4.1/src/doc/cpsa4manual.pdf (visited on 06/21/2025).

[87] *Cve-2014-0160 (heartbleed)*, Common Vulnerabilities and Exposures (CVE), Accessed: 03/26/2025. [Online]. Available: %5Curl%7Bhttps://www.cve.org/CVERecord?id=CVE-2014-0160%7D.

[88] OpenSSL, *Openssl security advisory [07 april 2014]*, Accessed: 03/26/2025, 2014. [Online]. Available: url%7Bhttps://openssl-library.org/news/secadv/20140407.txt%7D.

[89] *Cve-2014-3566 (poodle)*, Common Vulnerabilities and Exposures (CVE), Accessed: 03/26/2025. [Online]. Available: %5Curl%7Bhttps://www.cve.org/CVERecord?id=CVE-2014-3566%7D.

[90] O. Project, *OpenSSL Security Advisory: CVE-2024-12797*, https://openssl-library.org/news/secadv/20250211.txt, Accessed: 2025-03-26, Feb. 2025.

[91] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[92] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "{Mace}:{model-inference-assisted} concolic exploration for protocol and vulnerability discovery," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[93] R. Marcovich, O. Grumberg, and G. Nakibly, "Pise: Protocol inference using symbolic execution and automata learning," in *BlackHat USA*, 2022.

[94] S. Wen, Q. Meng, C. Feng, and C. Tang, "A model-guided symbolic execution approach for network protocol implementations and vulnerability detection," *PloS one*, vol. 12, no. 11, e0188229, 2017.

[95] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036.

[96] H. Asadian, P. Fiterău-Broştean, B. Jonsson, and K. Sagonas, "Applying symbolic execution to test implementations of a network protocol against its specification," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2022, pp. 70–81.

[97] Y. Sun, Z. Li, S. Lv, and L. Sun, "Spenny: Extensive ics protocol reverse analysis via field guided symbolic execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 6, pp. 4502–4518, 2022.

[98] M. Vanhoef and F. Piessens, "Symbolic execution of security protocol implementations: Handling cryptographic primitives," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[99] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[100] W. Wang, K. Liu, A. R. Chen, *et al.*, "Python symbolic execution with llm-powered code generation," *arXiv preprint arXiv:2409.09271*, 2024.

[101] Y. Li, R. Meng, and G. J. Duck, "Large language model powered symbolic execution," *arXiv preprint arXiv:2505.13452*, 2025.

[102] J. Chen, Z. Shao, S. Yang, *et al.*, "Numscout: Unveiling numerical defects in smart contracts using llm-pruning symbolic execution," *IEEE Transactions on Software Engineering*, 2025.

[103] J. Chen, L. Deng, Y. Qiu, P. Zhao, J. SONG, X. WANG, *et al.*, "Llm-based automated modeling in symbolic execution for securing medical software," *Jingcheng and WANG, Xiaopei, Llm-Based Automated Modeling in Symbolic Execution for Securing Medical Software*, 2024.

[104] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13, Washington, D.C.: USENIX Association, 2013, pp. 463–478, ISBN: 9781931971034.