

Augmenting Symbolic Execution with Artificial Intelligence: A Systematic Review and Research Agenda

Maksim Iavich ^{1,*}, Tamari Kuchukhidze ² and Audrius Lopata ²

¹ Department of Computer Science, Caucasus University, 0102 Tbilisi, Georgia

² Faculty of Informatics, Kaunas University of Technology, LT-44249 Kaunas, Lithuania; tamari.kuchukhidze@ktu.lt or tamari.kuchukhidze@gmail.com (T.K.); audrius.lopata@ktu.lt (A.L.)

* Correspondence: miavich@cu.edu.ge

Abstract

Symbolic execution is a key program analysis technique for ensuring the quality, security and safety of embedded, cyber-physical and electronic systems, but it is limited by path explosion and costly constraint solving. This study systematically reviews how artificial intelligence (AI) is used to guide symbolic execution. Following the PRISMA 2020 guidelines, five bibliographic sources were searched, yielding 679 records; after removing 170 duplicates and screening 509 titles and abstracts, 54 full texts were assessed and 37 studies met the inclusion and quality criteria. We propose a taxonomy that classifies these studies by the stage of symbolic execution they support, mainly path selection, constraint solving, state management and environment modeling, and by how AI is integrated, covering reinforcement learning, graph neural networks, neural constraint solving and large language models. The evidence is strongest for learned path selection and learned solver strategies, whereas learned state management, neural constraint approximation and learned function summarization remain largely unevaluated, and only four studies were evaluated on embedded, IoT or safety-critical software. We discuss applications in firmware and IoT security, automotive and aerospace verification and smart contract auditing, and derive a research agenda covering explainable AI guidance, cross-domain generalization and real-time adaptability for dependable embedded systems.

Keywords: static analysis; symbolic execution; embedded software; artificial intelligence; embedded systems; cyber-physical systems; program verification; neuro-symbolic computing

Academic Editor: Paolo Renna

Received: 17 September 2026

Revised: 3 October 2026

Accepted: 7 October 2026

Published: 8 October 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Today, software is a vital component of modern electronic and cyber-physical infrastructure, controlling critical operations in energy distribution, healthcare, transportation, and financial platforms [1,2]. The dependability, security, and safety of the software that manages electronic systems have transcended technical issues to become essential foundations of public welfare and economic stability as these systems grow more software-driven and interconnected [3]. Numerous high-profile incidents in a variety of domains, from data breaches exposing private information to system outages interfering with essential services and safety-critical failures in automotive and aerospace systems, illustrate the dire repercussions of software failures [4,5]. These incidents demonstrate the critical

need for trustworthy techniques that can proactively detect and mitigate software flaws prior to deployment [6].

One fundamental strategy for dealing with this issue is symbolic execution. Formally, it involves examining programs by considering inputs as symbolic variables instead of actual values, allowing for the methodical investigation of various program paths without running the program with particular test cases [7,8]. By gathering path constraints that describe the circumstances under which each path operates, symbolic execution allows for the investigation of a wider range of potential behaviors, in contrast to dynamic testing, which assesses software behavior given specific inputs and execution paths [9]. Symbolic execution tools create structured models of program semantics by converting programs into intermediate representations like control flow graphs and static single assignment forms. These models can be examined to identify a variety of problems, such as logic errors, memory mismanagement, security flaws, and safety property violations [10]. In embedded and electronic systems, where early defect discovery greatly lowers development costs and decreases safety hazards, this skill is especially helpful [11].

The incorporation of symbolic execution into contemporary software engineering workflows enhances its strategic significance. It facilitates a “shift-left” approach in which verification operations take place continually rather than being postponed to final validation phases by supporting the DevSecOps paradigm, which integrates security and quality assurance techniques directly into the development process [12]. Symbolic execution technologies can function as automated quality gates in continuous integration and deployment pipelines, assessing code modifications to stop flawed or unsafe software from making it to production environments [13]. Integrating program analysis into DevSecOps pipelines is intended to shorten the time needed to find and fix vulnerabilities compared with relying only on post-development evaluations [14].

Simultaneously, the scope and capabilities of symbolic execution have increased due to the increasing use of artificial intelligence in electronic and edge computing platforms. As machine learning components are increasingly included in safety- and security-critical systems, formal analysis has been extended to new types of risks, such as the verification of neural networks embedded in software [15,16]. Concurrently, AI methods have started to enhance symbolic execution itself, learning to direct exploration, speed up the resolution of constraints, and represent complicated settings in ways that overcome conventional scalability constraints [17,18].

Reliability, security and safety are the three basic quality imperatives that symbolic execution contributes to, which helps explain its significance. From a reliability standpoint, methods like path exploration and state merging methodically find flaws like buffer overflows, race conditions, and resource leaks that might result in runtime problems [19]. In terms of security, symbolic execution underpins advanced vulnerability discovery methodologies, tracing the propagation of untrusted inputs through software logic to detect critical vulnerabilities [20]. Lastly, symbolic execution is being used more and more in safety-critical fields like automotive electronics and avionics, which are governed by standards like ISO 26262 and DO-178C, to confirm that there are no dangerous runtime behaviors and to produce proof for certification [21,22].

There are still significant, enduring barriers to the general adoption and optimal efficacy of symbolic execution, especially for embedded and cyber-physical systems, despite these shown benefits and increasing maturity. The most basic of them is the path explosion problem, which restricts scalability to real-world software due to the exponential development of symbolic states with program complexity [9,23]. Another significant issue is constraint solving, which pushes SMT solvers beyond effective reasoning due to intricate nonlinear and floating-point operations [24]. Applicability to new platforms is limited by the significant manual labor required for environment modeling, especially for hardware

interfaces, system calls, and real-time operating system services [25,26]. The application of classical symbolic execution to embedded software is severely hampered by these interrelated issues: path explosion, intricate limitations, and difficult environment modeling, which encourages the investigation of more intelligent, guided methods.

By learning to make better choices regarding exploration, constraint solving, state management, and environment modeling, artificial intelligence and machine learning have emerged as viable augmentative technologies that overcome these shortcomings [27]. AI methods complement conventional methods rather than taking the place of symbolic execution; they learn from data and experience to concentrate analytic resources where they are most useful [28,29]. From large language models for environment modeling to reinforcement learning for path selection, this integration has resulted in a wide range of methods that aim to increase the scalability and applicability of symbolic execution to embedded systems [30,31].

Complementing our earlier survey of static analysis techniques for embedded, cyber-physical, and electronic software [32], this review offers a methodical analysis of AI-guided symbolic execution methodologies with an emphasis on the same class of systems.

This survey's main contributions are:

1. A comprehensive analysis of research on AI-guided symbolic execution that emphasizes embedded contexts and integrates results from large language models, graph neural networks, reinforcement learning and neuro-symbolic techniques.
2. A detailed taxonomy that sorts enhancement techniques into five main groups: path selection and exploration, constraint solving, large language model integration, state management and hybrid architectures. It looks at how they work, their pros and cons and how they can be used in embedded systems.
3. A comparison of industrial tools and academic prototypes, examining how they employ AI, what domains they operate in, their advantages and disadvantages, and how well they analyze embedded software.
4. An analysis of embedded system case studies and benchmarks, with a focus on blockchain-enabled systems, AI-enabled embedded systems, automotive and aerospace verification, firmware security, and problems unique to each sector.
5. A critical assessment of research gaps and open issues, including future directions in explainability and trust, cross-domain generalization, real-time adaptability, development lifecycle integration, and ethical implications for AI components.

This study focuses on embedded, cyber-physical, and electronic systems, from developing research objectives and choosing application domains to analyzing scalability, real-time constraints, and certification concerns [33]. This examination is framed by the following research questions:

- RQ1: In terms of path explosion, constraint solving, and environment modeling, what is a thorough taxonomy of AI-guided symbolic execution techniques and how do they overcome the basic shortcomings of conventional symbolic execution?
- RQ2: How are these methods used, specialized, and assessed in important embedded domains such as blockchain-enabled devices, AI-enabled embedded systems, automotive and aerospace systems, and firmware security?
- RQ3: In terms of reliability, resource adaption, generalization, and integration into development processes, what are the main unresolved issues and potential future research avenues for AI-guided symbolic execution?

This paper's remaining sections are structured as follows. Important background information on symbolic execution principles, including fundamental ideas, algorithmic variations, conventional optimizations, and significant constraints in embedded settings, is given in Section 2. The methodical literature review approach used to find and choose

pertinent research (following PRISMA 2020), together with its threats to validity, is explained in Section 3. Section 4 presents a taxonomy of AI-guided symbolic execution methods organized by the stage of symbolic execution they support. In Section 5, the tool ecosystem is surveyed and industry frameworks and academic prototypes are compared. Section 6 explores applications in electronic and embedded systems. Evaluation findings and performance insights from the literature are summarized in Section 7. Future research directions and open challenges are covered in Section 8. Practical implications for embedded systems deployment and certification are discussed in Section 9. A summary of the paper's main conclusions and a future direction for the area are presented in Section 10.

2. Symbolic Execution Fundamentals

By treating inputs as symbolic variables rather than concrete values, symbolic execution is a program analysis technique that investigates various program paths. In the 1970s, the fundamental concept was codified, providing the theoretical foundation for over half a century of subsequent research [7]. Since then, symbolic execution has developed into a practical technique for software verification, security analysis and automated test generation. To make symbolic execution appropriate for real-world software, a number of tools and techniques have been developed.

The strategy has been further validated by industrial use. To identify security flaws in file parsers and other Windows components, Microsoft built an internal whitebox fuzzing system that combined fuzzing with symbolic execution [20]. By identifying numerous flaws in Windows and Office applications that had escaped detection by traditional testing techniques, this strategy supported Microsoft's software security initiatives and demonstrated that symbolic execution could scale to large commercial software.

When it comes to verifying safety-critical components that run automobiles, medical devices and industrial infrastructure, symbolic execution may be advantageous for embedded and cyber-physical systems. However, using symbolic execution in embedded software presents many challenges. Studies examining firmware for commercial embedded devices have identified several challenges, including hardware interactions, real-time constraints and the difficulty of analyzing code running on resource-constrained microcontrollers [26]. These findings highlight the mismatch between the analysis requirements of embedded systems and the capabilities of symbolic execution.

This section lays the groundwork for comprehending how AI can address the limitations of symbolic execution, algorithmic variations, traditional optimizations and fundamental ideas, especially in embedded scenarios.

2.1. Core Concepts

2.1.1. Symbolic Versus Concrete Execution

Concrete execution follows a single path determined by specific input values. Symbolic execution instead treats inputs as symbolic variables: at each branch that depends on them, the analysis forks into one state per feasible outcome, and each state accumulates a path constraint, a logical formula over the inputs that encodes the branch decisions taken [7,9]. This allows many program paths to be examined in a single analysis run.

KLEE, which operates on the LLVM intermediate representation, scaled symbolic execution to real-world C programs: on the GNU Coreutils suite it achieved high line coverage and found previously unknown bugs in thoroughly tested utilities [19].

In situations where source code is not available, symbolic execution has been expanded to binary analysis through subsequent research. Frameworks that support symbolic execution of binaries for several architectures, including x86, ARM, MIPS and PowerPC, elevate binary code to intermediate representations, enabling architecture-agnostic

analysis [34]. This functionality is particularly crucial for embedded devices, as firmware is often delivered as binary images without accompanying source code.

Device drivers and other kernel components can be analyzed while the surrounding system environment is actually executed thanks to selective symbolic execution in full-system environments [35]. Evaluations show that many device drivers from popular operating systems can be successfully analyzed, enabling analysis of code that interacts closely with hardware and system services.

One of the main issues in program analysis research is the basic trade-off between symbolic and concrete execution. Although symbolic execution has advanced considerably, scalability issues still prevent it from being used in large and complex software systems, according to thorough analyses of symbolic execution techniques [9,10]. This finding encourages further investigation into optimization and guidance techniques.

2.1.2. Path Constraints and SMT Solving

Path constraints are expressed in the standardized language of SMT-LIB [36], typically using the theories of bit-vectors for fixed-width machine integers, arrays for memory and floating-point arithmetic for real-number computations.

An SMT solver decides whether a path constraint is satisfiable and, if so, returns a model that serves as a concrete test input. Z3 combines Boolean satisfiability solving with decision procedures for many theories [37]; cvc5 additionally produces independently checkable proofs [38]; and solvers specialized for bit-vectors and arrays often outperform general-purpose solvers on bit-level problems [39].

At each branch that depends on symbolic values, the executor asks the solver whether the condition and its negation are feasible under the current path constraint, prunes infeasible paths and, when a path terminates, obtains a satisfying assignment that is used as a test input.

Solver performance has a significant influence on symbolic execution's overall efficiency. Retrospective evaluations of symbolic execution have shown that constraint solving typically contributes significantly to execution time in tools like KLEE [40]. While linear constraints over tiny bit-widths solve rapidly, constraints using nonlinear arithmetic, symbolic memory indices or floating-point operations may need much longer solver times.

2.1.3. Intermediate Representations: IR, CFG, SSA

Symbolic execution tools rarely analyze source code directly; they operate on intermediate representations (IRs) that preserve program semantics while simplifying analysis. Source-level tools commonly use the LLVM IR, as KLEE does [19,41], whereas binary-level frameworks lift machine code from several architectures into a common IR, at some loss of precision for complex instructions [42].

From the IR, symbolic executors build control flow graphs, whose nodes are basic blocks and whose edges are possible control transfers; for binaries, this requires precise disassembly. Many executors also use static single assignment (SSA) form, in which each variable is assigned exactly once and phi functions select values at control-flow merge points; this makes data flow explicit and simplifies constraint generation [43].

2.2. Variants and Approaches

2.2.1. Static Symbolic Execution

Static symbolic execution attempts to explore all possible pathways through the program abstractly by analyzing programs only at the symbolic level, without physical execution. Early systems successfully created test cases that revealed errors in intricate pointer manipulation code, demonstrating this method on systems code that included

implementations of common Unix utilities [44]. Static symbolic execution has the potential benefit of completeness as it can, in theory, prove program characteristics and explore all possible pathways. This is in line with official safety-critical software verification objectives. Static analysis, however, has serious drawbacks. Because each conditional branch generates forks and loops with symbolic boundaries produce limitless path families, path explosion is most severe in static analysis. Loop invariants, which over-approximate attainable states but can induce imprecision and false positives, are abstraction methods needed to handle loops.

System calls, dynamic memory allocation and operating system services are among the environmental interactions that static analysis finds difficult to handle. Static symbolic execution must overestimate the consequences of these interactions in the absence of actual execution to ground them, which frequently results in state explosion or loss of accuracy. According to thorough surveys, solely static symbolic execution is seldom employed alone for large-scale software analysis; instead, it is either applied selectively to crucial code sections or supplemented with dynamic approaches [10].

2.2.2. Dynamic Symbolic Execution and Concolic Execution

Concrete execution and symbolic reasoning are interwoven in dynamic symbolic execution, also known as concolic execution from the mix of concrete and symbolic. This method generates fresh inputs that force execution into undiscovered pathways by methodically negating branch conditions after executing programs concretely with random inputs while performing symbolic execution along the observed path [45].

By individually negating each branch condition and creating a new path for each negated condition, generational search, a very successful search technique for concolic execution, generates several new inputs from each iteration [20]. By studying a large number of applications in a variety of fields and finding multiple security-critical flaws, this method scaled concolic execution to industrial practice.

Through advances like handling pointer limitations and dynamic memory allocation, extensions to concolic execution manage multi-threaded programs and sophisticated data structures, making it possible to analyze programs with object-oriented characteristics [46]. Researchers may apply and contrast various path selection heuristics using extensible frameworks for testing concolic execution search techniques, which advances our knowledge of what makes concolic exploration successful [47].

Concolic execution's strength is its ability to manage interactions with the environment. System calls, library functions and hardware interactions that would be challenging to express symbolically are automatically resolved by its concrete execution. Programs that interact substantially with their surroundings, including embedded systems with hardware peripherals, can be analyzed using this hybrid technique.

2.2.3. Hybrid Symbolic Execution

Strategically combining symbolic execution with complementary methods like fuzzing, abstract interpretation or model checking is known as hybrid symbolic execution. Systems that combine fuzzing with symbolic execution in automated exploit creation pipelines have proven successful in security applications [48]. To effectively investigate several shallow pathways, one hybrid strategy starts with coverage-guided fuzzing. Symbolic execution is used to provide inputs that cross sophisticated checks in code sections where fuzzing stops [49]. Driller reported that this combination found more vulnerabilities in the DARPA Cyber Grand Challenge binaries than fuzzing alone [49].

Symbolic analysis is only applied to program sections where it is most advantageous in selective symbolic execution. This method allows for the study of many drivers on

several platforms by symbolically analyzing device driver code while really running the remainder of the operating system [35].

Hybrid analysis integrates real hardware execution and symbolic execution for embedded systems [50]. The system offloads to physical hardware in order to retrieve tangible values that resolve symbolic state when symbolic execution hits hardware interaction points, such as reading memory-mapped registers or handling interrupts. This properly models hardware behavior while maintaining the viability of symbolic exploration.

The practical division of work is a recurring subject in hybrid techniques. Concrete execution resolves interactions with the environment; symbolic execution addresses complicated limitations and fuzzing effectively manages shallow exploration. Each method makes up for the shortcomings of the others, resulting in systems that are more resilient and scalable.

2.3. Traditional Optimizations (Non-AI)

2.3.1. Search Strategies

The algorithm's choice of which symbolic state to investigate next and the search strategy has a significant impact on the efficacy of symbolic execution. Evolution from basic heuristics to adaptable methods is documented by analyses of search strategies across various symbolic executors [9]. By keeping one active path at a time, depth-first search reduces memory use by thoroughly exploring paths before turning around. It can, however, become stuck exploring endless or incredibly deep pathways, never making it to shallow branching that could have serious weaknesses. Breadth-first search ensures shallow problems are found early by exploring all routes at a given depth before going deeper. But when the number of paths increases, memory needs increase significantly, sometimes using up all of the available memory.

For exploration, random search selects states equally at random. It is straightforward and impervious to pathological worst-case behaviors, but it does not concentrate on pathways that might result in additional coverage or defects. States that are likely to cover new instructions are given priority in coverage-guided search. Compared to depth-first or random techniques, two-level scheduling that alternates concentrated exploitation of interesting states with random path finding delivers greater final coverage across benchmark suites [19].

Another important tactic is generational search, which works on concolic execution by choosing a single concrete execution route, solving each to provide a new set of test inputs and negating each branch condition encountered along that path to create numerous new symbolic constraints [20]. This method effectively investigates several path alternatives and proved crucial in identifying security flaws in sizable commercial applications.

There is not a single search approach that works for all benchmark suites, according to comparative research [51]. Program structure, the kind of bug of interest and the resources at hand all influence the optimal methodology, which encourages adaptive and learning methods.

2.3.2. State Merging and Function Summarization

When many symbolic states converge at the same program point, they are combined by state merging. For a number of benchmark applications, effective state merging strategies show notable state count reductions [52].

Because the resultant path condition is a disjunction of the conditions of the merged pathways, merging decreases states but may raise constraint complexity. Compared to the individual conjunctive formulae, these disjunctive formulations might be significantly

more challenging for SMT solvers. Heuristics that use projected solver cost to determine whether to merge enhance net performance without causing pathological slowdowns.

In benchmark programs, limited state merging at function return points. Combining states leaving functions with distinct return values but the same post-return control flow, which reduces the overall number of states while only slightly lengthening the solving time [53].

Function summarization reuses the summary at each call location after computing a function's symbolic input-output connection once. Scalable analysis of programs with deep call chains is made possible by summaries, as shown by compositional dynamic test generation [54].

Depending on the objectives of the study, summaries might be precise, over-approximate or under-approximate. Pre-computing summaries for frequently used library functions is a natural application of this idea.

2.3.3. Lazy Initialization and Constraint Caching

Symbolic pointer problems are addressed with lazy initialization. A symbolic pointer's target is not assumed when it is declared. The executor only takes into account potential targets at dereference, creating new symbolic objects or non-deterministically selecting from already-allocated ones [55]. This method is crucial for examining system code that heavily utilizes pointers and allows examination of programs that manipulate intricately connected data structures [19].

Solver query results are stored in constraint caching for later use. Exact-match and subsumption caching, which detect when one constraint logically implies another, allow caching systems to achieve notable speedups [19,56].

Canonicalization is converting constraints to standard form such that semantically comparable constraints are regarded as identical. It is essential to cache efficacy. Cache hit rates are increased by normalizing restrictions by sorting conjuncts, removing superfluous words and using simplification prior to cache query.

Modern SMT solvers feature incremental solving, which opens up another dimension of reuse. When investigating pathways with common prefixes, solvers preserve context across related queries by enabling push and pop of constraints to reuse work from shared prefixes.

2.3.4. Parallel and Distributed Symbolic Execution

Multi-core processors are used in parallel symbolic execution to explore paths simultaneously. Parallel symbolic execution engines such as Cloud9 distribute the exploration across clusters of commodity machines to cope with path explosion [57]. Termination detection, load balancing and state partitioning to reduce communication overhead are important issues. These issues are addressed by hierarchical work-stealing scheduling in conjunction with distributed shared memory abstractions that migrate states only when required.

Machine clusters are included in distributed symbolic execution. Other approaches partition the symbolic execution tree using constraints on path prefixes so that the resulting sub-trees can be explored in parallel [58]. Communication cost is the main barrier, as transferring symbolic states necessitates communicating large quantities of heap and constraint data. Although they can assist, state differencing (sending just changes since the last checkpoint) and compression cannot completely reduce overhead. Distributed techniques show potential but are still being researched for embedded firmware analysis, where binary sizes are frequently small, but path counts are large.

2.4. Key Limitations in Embedded Contexts

2.4.1. Path Explosion in State-Rich Embedded Software

The state-space complexity of embedded systems is higher than that of desktop software with similar line counts. The combination of hardware state, interrupt-driven concurrency and continuous control loops that characterize embedded software presents difficulties for tools using dynamic analysis techniques, according to studies examining firmware from commercial embedded devices [26].

Hardware statefulness significantly increases this complexity. The hardware registers that embedded code continuously interacts with to retain state during executions include configuration registers, sensor data registers and timer values. The permanent state introduced by these registers necessitates symbolic modeling. As new symbolic variables are added by each read from these registers and register values are altered by each write, the state count grows multiplicatively.

The issue is made worse by interrupt-driven concurrency. The number of interleavings that result from multiple interrupt service routines preempting mainline execution at any time increases combinatorially with the size of the code and the number of interrupt sources. For all but the smallest examples, symbolic execution that accurately simulates every possible interleaving becomes unmanageable.

Control loops use iteration to produce explosions. With each iteration, loops with symbolic conditions produce new branches, which causes path counts to increase exponentially. Symbolic execution must either use abstraction techniques that compromise precision or bound iterations, which may miss bugs that appear only after many cycles, because control loops may run indefinitely in deployed systems.

Interrupt-driven code in particular can produce path counts that surpass practical analysis limits [59]. For tractable analysis, state merging and abstraction become crucial, but full coverage is still unattainable for all but the smallest firmware images.

2.4.2. Constraint Solving for Non-Linear Arithmetic and Hardware Models

SMT solving is difficult because of the numerical intensity of embedded software. Multiplication, division and floating-point arithmetic are frequently used in control algorithms, sensor fusion and signal processing, operations that push SMT solvers beyond effective reasoning. Problems become theory fragments with worst-case exponential complexity even when symbolic variables are simply multiplied [60].

Because SAT solvers encode linear arithmetic constraints over fixed-width bit-vectors as Boolean circuits using conflict-driven clause learning, these constraints solve effectively. Such encoding is resisted by nonlinear constraints, which are handled by bit-blasting or case splitting techniques and can grow exponentially in size with bit width when symbolic variables are multiplied.

Problems with complicated semantics, such as rounding modes, special values and exceptions, are exacerbated by floating-point arithmetic. Even relatively simple floating-point constraints can cause solver timeouts because of reasoning about rounding errors and special cases, making precise floating-point modeling extremely difficult [61].

More complexity is added by hardware modeling. Peripherals that combine continuous physics and discrete logic communicate with embedded programs. It is necessary to think about duty cycles, frequencies and timing limitations while modeling pulse-width modulation. It is necessary to reason about quantization error, sample rates and input properties while modeling analog-to-digital conversion. These hybrid discrete-continuous models require either integration using specialist continuous mathematics solvers or over-approximation at the expense of precision because they are not covered by native SMT solver theories.

Alternative methods substitute high-level semantic models for hardware abstraction layer functions for accurate hardware modeling [62]. By compromising some fidelity, this method makes it possible to analyze software symbolically, which was previously impossible because of complicated peripheral models. This makes it easier to find weaknesses in industrial control firmware.

2.4.3. Environment Modeling: Syscalls, Interrupts, Peripherals

Accurate environment models, such as operating system services, interrupt controllers and hardware peripherals, are necessary for symbolic execution of embedded applications. These models are incredibly challenging to create and flawed models undermine the validity of analysis by producing both false positives and false negatives. According to studies, faulty environment models may render significant chunks of investigated paths in embedded firmware analysis impractical [26].

System calls present the first modeling challenge. Real-time operating systems include several system requirements for task management, inter-task communication, timing services and device I/O. Reliable symbolic analysis requires precise capture due to the complex semantics of each system call. Modeling queue operations requires reasoning about priority interactions, blocking timeouts and queue state, all of which interact subtly. It typically takes a lot of manual labor to accurately mimic operating system calls and some models may remain incomplete even after significant progress [63].

Different modeling issues are brought forth by interruptions. Symbolic execution must take into account any potential interleaving with mainline code when asynchronous interrupts occur. Interrupt-driven embedded software typically contains a very large number of executable paths, which makes exhaustive analysis hard for all but trivial systems [59]. Although these bounds may overlook issues that only appear under particular interleaving, researchers investigate bounding approaches that limit interrupt occurrences taken into consideration.

Perhaps the hardest modeling problem is memory-mapped peripherals. Hardware events, not merely software writes, cause peripheral registers at specific memory addresses to modify their state. Counter registers automatically increase depending on clock cycles, while status registers show data availability dependent on external events. Both program-driven and hardware-driven state changes. The latter, occurring non-deterministically from the program perspective, must be captured in order to model these phenomena symbolically.

One solution is to use hardware offloading techniques that transfer peripheral interactions to real hardware [50]. The operation is transferred to real hardware when symbolic execution reaches memory-mapped I/O access, yielding tangible outcomes and carrying on with tangible values. These guarantees correct peripheral modeling without the need for symbolic models, but it adds latency from hardware communication and compromise the ability to symbolically explore many hardware states. These methods, which are sometimes called “firmware re-hosting,” give up some execution fidelity in order to analyze code that would otherwise be unmanageable because of complicated or unavailable hardware.

2.4.4. Resource Constraints in Embedded Analysis Contexts

Resource constraints imposed by embedded development environments may clash with symbolic execution specifications. A significant amount of memory may be needed for embedded firmware analysis, especially as state space grows. The enormous overhead of keeping many symbolic states is demonstrated by documentation of distributed state management, which emphasizes how state storage can use a lot of memory [64].

Path explosion and memory consumption are compounded. Program memory image, path constraints, and metadata copies are kept in each symbolic state. Before taking solver memory and tool overhead into consideration, memory requirements for firmware with numerous active states may surpass available capacity simply for state storage. Memory depletion becomes a real worry when path counts are high.

Time restrictions are just as important. While symbolic execution of moderately complex functions can take significantly longer, industrial continuous integration pipelines generally assume analysis completion within realistic timeframes per commit. It has been demonstrated that benchmark program analysis takes a significant amount of time for individual programs [19]. Analysis times may increase for embedded firmware of the same complexity with additional hardware interactions, which could make incorporation into development workflows difficult.

The underlying issue arises from embedded systems' real-time nature. By examining paths at rates orders of magnitude slower than actual execution, symbolic execution radically changes timing behavior. The relative ordering of events may be drastically altered by paths that execute in microseconds on hardware and take much longer to explore symbolically. Because symbolic exploration imposes exaggerated operation delays, race conditions that depend on precise timing may be completely ignored.

Timing constraints are central to embedded control software design [65], but symbolic execution tools rarely model them during analysis. To determine whether deadlines could be missed under any practical timetable, paths are labeled with projected execution times. Although encouraging, this work is still in its early stages and has not been incorporated into widely used symbolic execution tools. AI-guided methods concentrate resources on routes where timing is crucial since the basic conflict between thorough exploration and temporal integrity is still unsolved. The application of classical symbolic execution to embedded software is severely hampered by these interrelated limitations: path explosion, complicated constraints, difficult environment modeling and resource constraints, which spur the investigation of more intelligent, guided methods.

3. Methodology: Systematic Literature Review

To provide a complete, impartial, and reproducible synthesis of the literature on AI-guided symbolic execution for embedded and cyber-physical systems, this review followed the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) standards [66,67] in their 2020 version (PRISMA 2020). PRISMA's methodical approach to defining research questions, search strings, and inclusion criteria is directly applicable to systematically mapping a technical field in which a large volume of research, conference papers, journal articles and preprints may be subject to selection bias. Using PRISMA promotes transparency and reproducibility in the literature selection process. The PRISMA flow diagram describing the paper selection process is shown in Figure 1. The approach was carried out in three stages: developing precise research questions based on gaps found, methodically locating and obtaining pertinent material from chosen scholarly sources, and applying strict inclusion and exclusion criteria to the gathered works. Sections 3.1–3.6 describe, respectively, the research questions, the search strategy, the eligibility criteria and screening procedure, the PRISMA flow, the quality assessment and synthesis, and the threats to validity. The completed PRISMA 2020 checklist, the search log with the exact query for each source, and the list of included studies with their quality scores are provided in Appendix A.

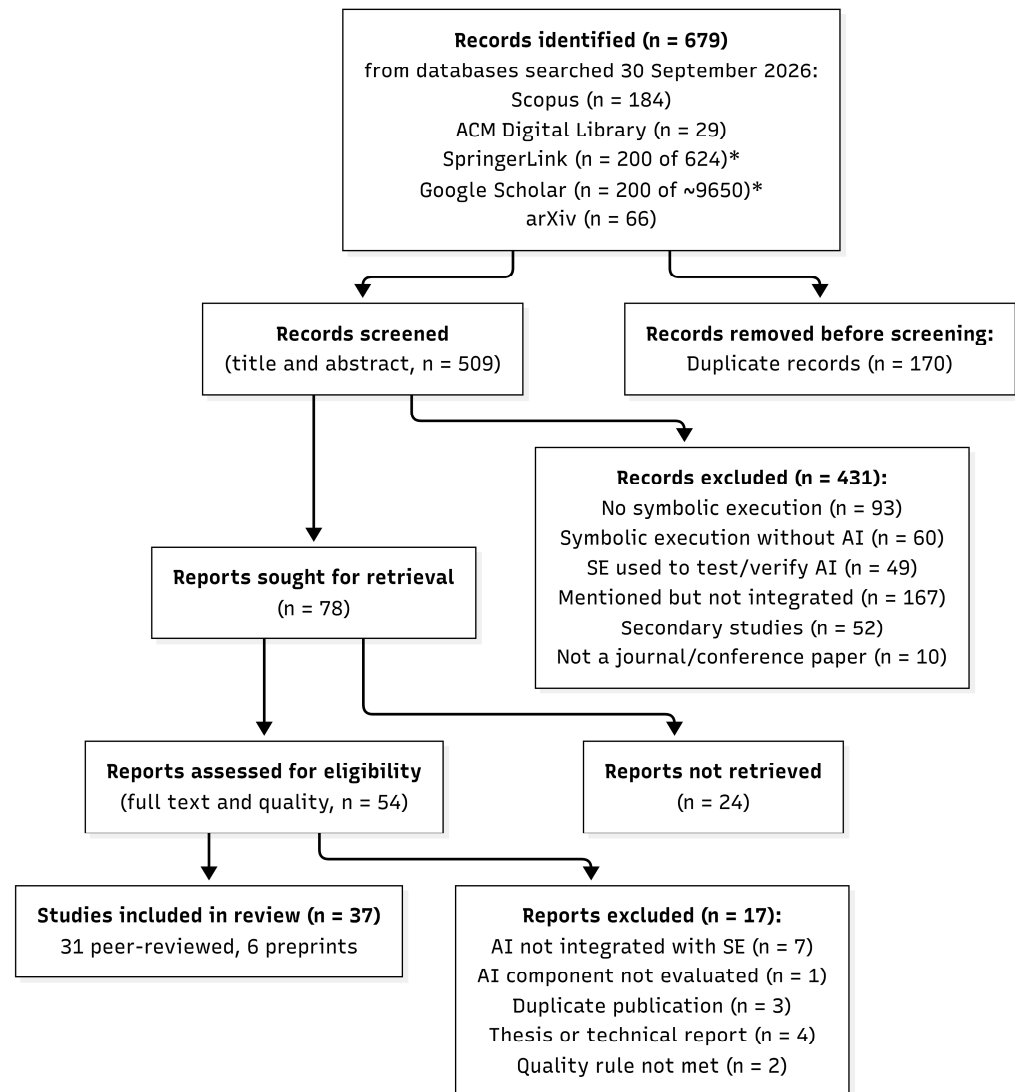


Figure 1. PRISMA Flow Diagram of the Systematic Literature Selection Process (PRISMA 2020 template; search date 30 September 2026). * First 200 results sorted by relevance (full-text search engines); IEEE Xplore was not accessible, and IEEE content was covered through Scopus.

3.1. Research Questions

Three research topics that address the conceptual, technical, and practical aspects of AI-guided symbolic execution served as the basis for the selection of relevant literature and the analysis that followed. The first question aims to develop a thorough taxonomy of AI-guided symbolic execution methods and comprehend how they overcome the basic drawbacks of conventional symbolic execution. Firmware security, automotive and aerospace systems, blockchain-enabled devices, and AI-enabled embedded systems are just a few of the important embedded sectors where the second question looks at how these methods are used, specialized, and assessed. The third inquiry explores the main unresolved issues and potential avenues for future study in AI-guided symbolic execution, with a focus on reliability, resource adaptability, generalization, and workflow integration.

These inquiries were developed in response to gaps in the current body of knowledge that were found in previous surveys of the subject, including the absence of systematic taxonomies and comparative studies of AI-guided symbolic execution methods for embedded systems.

3.2. Research Process

Major scientific databases and preprint repositories were used to compile research articles. According to Table 1, the ACM Digital Library, arXiv, IEEE Xplore, SpringerLink, Scopus, and Google Scholar were among the key sources. The search strategy combined two concept groups, symbolic execution techniques and artificial intelligence techniques; terms describing embedded and cyber–physical application areas were used to classify the included studies rather than to filter the search.

Table 1. Reference Scientific Literature Databases and Academic Search Engines, with the number of records retrieved and screened (search date: 30 September 2026; publication years 2020–2025). † First 200 results sorted by relevance (full-text search engines). ‡ HTTP error 420 at the time of the search; IEEE content was covered through Scopus (Section 3.2).

Database/Search Engine	Type	URL	Records Retrieved	Records Screened
Scopus	Search Engine	https://www.scopus.com	184	184
Google Scholar	Search Engine	https://scholar.google.com	about 9650	200 †
IEEE Xplore	Digital Library	https://ieeexplore.ieee.org	– ‡	–
ACM Digital Library	Digital Library	https://dl.acm.org	29	29
SpringerLink	Digital Library	https://link.springer.com	624	200 †
arXiv	Preprint Repository	https://arxiv.org	66	66

The search terms were arranged in two concept groups. The symbolic execution group comprised “symbolic execution”, “concolic execution” and “dynamic symbolic execution”; the artificial intelligence group comprised “machine learning”, “deep learning”, “reinforcement learning”, “graph neural network”, “large language model”, “LLM”, “neural network” and “neuro-symbolic”. Terms from the application domain, such as “embedded systems”, “cyber-physical systems”, “IoT”, “firmware”, “automotive”, “aerospace”, “ISO 26262”, “DO-178C” and “safety-critical”, were not used as a mandatory filter, because only a small number of AI-guided symbolic execution studies are evaluated on embedded targets; they were used instead to classify the included studies (Section 3.5).

Boolean operators were used to create comprehensive search strings from these keyword groupings. The two groups were combined with AND: (symbolic execution OR concolic execution) AND (at least one AI term). Scopus was searched using titles, abstracts and keywords; the ACM Digital Library and arXiv in abstracts; and SpringerLink and Google Scholar, which offer no abstract-only search, in the full text. The exact query submitted to each source, with its fields and filters, is listed in Table A1 (Appendix A); for example, the Scopus query was: TITLE-ABS-KEY (“symbolic execution” OR “concolic” OR “dynamic symbolic execution”) AND (“machine learning” OR “deep learning” OR “reinforcement learning” OR “graph neural network*” OR “large language model*” OR “LLM*” OR “neural network*” OR “neuro-symbolic”)) AND PUBYEAR > 2019 AND PUBYEAR < 2026 AND (LIMIT-TO (DOCTYPE, “ar”) OR LIMIT-TO (DOCTYPE, “cp”)) AND (LIMIT-TO (LANGUAGE, “English”)).

All searches were run on 30 September 2026 and limited to publications from 2020 to 2025 written in English. They returned 184 records in Scopus, 29 in the ACM Digital Library, 624 in SpringerLink, about 9650 in Google Scholar, and 66 in arXiv (Table 1). Because SpringerLink and Google Scholar search the full text, most of their results mention symbolic execution only in passing; for these two sources, the first 200 results sorted by relevance were screened. IEEE Xplore could not be queried directly because access was blocked from the authors’ location at the time of the search (HTTP error 420). IEEE journals and conference proceedings are indexed in Scopus, and 60 of the 184 Scopus records

were published in IEEE or joint IEEE/ACM venues; IEEE content was therefore covered through the Scopus search.

Preprint servers were included because many important results in the rapidly developing field of AI-guided program analysis first appear there. Preprints are marked as such in the classification of studies in Section 4.7 and conclusions that rest on preprints alone are identified (Section 3.6).

3.3. Inclusion and Exclusion Criteria

Predetermined inclusion and exclusion criteria were used to evaluate the applicability of candidate publications. Works that did not meet the requirements for inclusion were not taken into account. While works released between 2020 and 2025 are the main focus of this study, important pre-2020 works that created core techniques, like the original KLEE, SAGE, and early symbolic execution frameworks, are cited where necessary to offer appropriate historical context. Our own related survey of static analysis techniques [32], published in 2026, is cited only as background and is not part of the reviewed literature. Such pre-2020 works are cited as background and are not counted in the review corpus. No studies were added from outside the database search. Eight studies discussed in the previous version of this article were not retrieved by the search: four were published before 2020 and four do not involve symbolic execution; they are likewise cited as background only (see Section 4).

Before screening, the records from all sources were merged, and duplicates were removed automatically by DOI and by normalised title, followed by a manual check of all remaining title pairs with more than 85% similarity; 170 duplicates were removed, leaving 509 unique records.

A rational, multi-step process was followed for filtering according to the inclusion criteria. The first phase was eliminating publications that were thought to be unrelated to the research topics based on information acquired from titles and abstracts. Articles that specifically addressed at least one of the fundamental methods or research questions in their abstracts and titles were kept. To support this phase, an AI assistant first read every title and abstract and proposed a decision together with an exclusion reason; records without an abstract were judged from their title, and doubtful records were passed on to full-text assessment. The AI assistant was used to manage the volume of records and to apply the exclusion criteria consistently; it was not used to make final decisions, which remained with the human reviewer. Accordingly, one of the authors examined every proposed decision together with its recorded reason and either confirmed or revised it before it was adopted. No second, independent human screening was performed (see Section 3.6). In total, 431 records were excluded at this stage (93 did not involve symbolic execution, 60 used symbolic execution without AI, 49 used symbolic execution to test or verify AI models, 167 mentioned both without integrating them, 52 were secondary studies and 10 were not journal or conference papers), and 78 were retained for full-text assessment. Every decision and its reason is recorded in the screening spreadsheet; full-text exclusions are listed in Table A2.

The remaining articles' full texts were assessed in the second phase. Works that were deemed out of scope or that only briefly discussed AI-guided symbolic execution were eliminated. Excluded were papers that concentrated solely on traditional symbolic execution without any artificial intelligence elements or on AI methods without any application to program analysis. Studies that use symbolic execution or SMT solving to test or verify neural networks, without AI improving symbolic execution, were also excluded because they address the reverse research direction.

In the third step, the remaining articles were assessed against four quality criteria: the work includes empirical evaluation, formal analysis or a solid theoretical framework;

the publication is peer-reviewed or from a reputable preprint server such as arXiv; the study offers enough technical detail to make the methodology understandable; and the article presents a novel methodology or a notable improvement to an existing technique. Each criterion was scored 0 or 1, and a study was retained if it satisfied the first criterion and scored at least 3 of 4 points (Section 3.5).

Table 2 provides a summary of the formal inclusion and exclusion criteria.

Table 2. Inclusion and Exclusion Criteria.

Inclusion Criteria	Exclusion Criteria
Published in one of the selected databases (Table 1)	Not written in English
Published between 2020 and 2025 (with exceptions for seminal pre-2020 works)	Published before 2020 (with exceptions for seminal, foundational works)
At least one search term appears in the title, abstract or keywords	Full text not available
Addresses the stated research questions	Lacks technical detail or is a duplicate of another study
Provides sufficient technical depth, empirical results or novel theoretical framework	Provides only superficial treatment of the topic
Published in peer-reviewed journals, conference proceedings or reputable preprint servers	Published in predatory or non-credible venues

3.4. PRISMA Study Selection Flow Diagram

The PRISMA 2020 guidelines were followed throughout the systematic literature selection process, as seen in Figure 1. After searching five significant academic databases and preprint repositories, 679 records were initially found. 509 records were left for title and abstract screening after 170 duplicates were eliminated. 431 records were removed as a result of this screening because they were thought to be unrelated to the research objectives or to not deal with AI-guided symbolic execution. The remaining 78 publications were sought in full text. The full texts of 54 of them were accessible and were evaluated for eligibility using the inclusion criteria listed in Table 2; the other 24, most of them published in subscription-only venues, are reported as not retrieved in Figure 1.

Then, 17 full-text papers were excluded after evaluation (7 did not integrate AI with symbolic execution, 1 did not evaluate its AI component, 3 were duplicate publications of included studies, 4 were theses or technical reports, and 2 did not meet the quality rule). No studies were identified through other methods. As a result, the systematic review's final corpus consisted of 37 publications. Of these, 31 are peer-reviewed journal or conference papers and 6 are preprints, and 4 were evaluated on embedded, IoT or safety-critical software. The included studies are listed and classified in Table A3, and the classification in Section 4 marks which of the studies discussed in this article belong to the corpus; all other references in this article are background, standards or methodological sources and are not part of the review corpus.

3.5. Quality Assessment and Synthesis

The quality criteria in Section 3.3 guided the selection of publications. A four-point quality rating was used to evaluate methodological rigor and minimize bias in each full-text publication. The four criteria are those listed in Section 3.3: (Q1) empirical evaluation, formal analysis or a solid theoretical framework; (Q2) peer-reviewed publication, with preprints scoring 0; (Q3) sufficient technical detail; and (Q4) a novel method or a notable improvement of an existing technique. Each criterion scores 0 or 1, and a study was included in the final synthesis only if it satisfied Q1 and scored at least 3 of 4 points. Quality

scores were proposed by the AI assistant from the full text; every proposed score was then examined by one of the authors and either confirmed or revised before a study was admitted to the corpus. The final quality judgment therefore reflects the authors' assessment, not the assistant's output. With an average quality score of 3.8, the chosen papers demonstrated a high level of overall methodological rigor (31 studies scored 4 and 6 scored 3; all studies scoring 3 are preprints, which receive 0 on Q2). The score of every included study is given in Table A4. In addition, Section 4 records for each AI-guided symbolic execution study its publication status and whether it was evaluated on embedded targets, the two quality aspects most relevant to the conclusions of this review.

The selected literature was synthesized narratively and organized according to the taxonomy described in Section 4. The reference list combines the AI-guided symbolic execution studies reviewed in Sections 4–7 with background references, such as standards, seminal symbolic execution tools and methodological works. The studies of the review corpus are listed in Table A3 and marked in the classification in Section 4. The taxonomy described in Section 4 was used to classify the final corpus of 37 publications. The current state of research was reflected in the distribution across taxonomic categories (11 on path selection, 10 on constraint solving, 2 on state management, 4 on environment modeling and 10 on other stages, namely test and input generation, hybrid fuzzing, specification inference, engine configuration and intermediate-representation optimization; Table A3). Most AI-guided studies address path selection or constraint solving, and few address state management or environment modeling.

The main emphasis was on recent developments between 2020 and 2025, with particular attention paid to groundbreaking research that established fundamental ideas in early artificial intelligence integration and symbolic execution. Papers published prior to 2020 were cited as background if they offered fundamental methods (e.g., early symbolic execution frameworks, the original KLEE, DART, and SAGE systems) or were important sources that still have an impact on current research. This methodology guaranteed thorough coverage of both the latest advancements in artificial intelligence-guided methodologies, which are the main emphasis of this survey, and the historical underpinnings of symbolic execution.

The results are summarized in the following sections, which include a taxonomy of AI-guided symbolic execution techniques, a comparison of tools and frameworks, an analysis of embedded system applications, a synthesis of evaluation results and performance insights, and a discussion of open challenges and future research directions.

3.6. Threats to Validity

3.6.1. Search Coverage

IEEE Xplore could not be searched directly and was covered through Scopus (Section 3.2), and for the two full-text search engines, SpringerLink and Google Scholar, only the first 200 results by relevance were screened. Relevant studies that appear only in IEEE Xplore, or rank low in these two engines, may therefore be missing. The exact queries, dates and counts are reported (Table A1) so that the search can be repeated and extended. In addition, the full texts of 24 reports, most of them from subscription-only venues, were not accessible to the authors. Several of these studies are still discussed as background in Sections 4 and 5; nevertheless, work published in subscription venues, in particular IEEE and Springer, may be under-represented in the corpus.

3.6.2. Construct Validity

Terminology in this field is not standardized: comparable techniques are described as AI-guided, learning-based, neuro-symbolic or machine-learning-steered symbolic execution, and concolic testing is sometimes reported separately from symbolic execution.

Keyword-based searches may therefore miss relevant studies or retrieve irrelevant ones. Assigning studies to taxonomy categories also involves judgement; to limit overlap, the classification in Section 4 assigns each study once to its primary stage of symbolic execution and records its AI technique and integration pattern separately.

3.6.3. Internal Validity

Screening, data extraction and the mapping of claims to sources involve subjective decisions and are prone to error. Title/abstract screening, full-text assessment and quality scoring were carried out with the support of an AI assistant, which recorded a decision and reason for every record. The AI assistant was used to manage the volume of records and to apply the exclusion criteria consistently; it was not used to make final decisions, which remained with the human reviewer. Every proposed decision and every proposed quality score was examined by one of the authors and either confirmed or revised before it was adopted, so the final corpus reflects the authors' judgment rather than the assistant's output. Because no second, independent human screening was performed, inter-rater agreement cannot be reported. The AI suggestions may still carry errors, particularly for the 182 records without an abstract. During the revision of this review, every citation was checked against the cited publication, and claims that the cited work did not support were corrected, reformulated as open questions or removed.

3.6.4. Publication Bias and Preprints

Positive results are more likely to be published than negative ones, which may overstate the benefits of AI guidance. Preprints were included because the field, in particular work on large language models, moves quickly; they are marked as such in Section 4, and no conclusion of this review rests on preprint evidence alone.

3.6.5. External Validity

Most evaluated studies use general-purpose programs or SMT benchmarks rather than embedded firmware (Section 4). Statements about the suitability of these techniques for embedded systems are therefore inferences from that evidence and are presented as such; they should be tested on embedded targets before being relied upon.

3.6.6. Conclusion Validity

The reviewed studies differ in benchmarks, baselines, time budgets and metrics, which precludes a meta-analysis. Reported gains are quoted with their original setting and are not directly comparable across studies. Finally, studies published after the literature search are not covered (search date: 30 September 2026; publication years 2020–2025), which matters most for the rapidly developing work on large language models.

4. Taxonomy of AI-Guided Symbolic Execution Techniques

Artificial intelligence and symbolic execution have been used to create a wide range of methods that use learned guidance to solve basic scaling issues. By learning to make better choices regarding exploration, constraint solving, state management and environment modeling, these techniques complement conventional methods rather than taking the place of symbolic execution. Two complementary dimensions are used in the taxonomy. Path selection (Section 4.1), constraint resolution (Section 4.2), environment modeling (Section 4.3.1), and state management (Section 4.4) are the main stages of symbolic execution that AI enables. Hybrid AI-symbolic architectures (Section 4.5) explain integration patterns rather than an additional stage, and large language models (Section 4.3) are studied collectively because their applications span stages. Table 3 summarizes the

evidence obtained for each technique. Table 4 then classifies each study once by its primary stage and notes its AI approach, integration pattern, publication status, and whether it was assessed on embedded targets in order to prevent overlapping categories. Results that are solely backed by preprints are labeled as such and are not regarded as conclusive.

Table 3. AI-guided symbolic execution techniques and the evidence found for each in this review.

Technique	AI Method	Path Explosion Handling	Constraint Solving Efficiency	Embedded Suitability	Limitations
Learned path selection	Q-learning and DQN [31,68]; learned heuristics [28,29,69,70]	Prioritizes states; reported path reductions and coverage gains [28,31]	Indirect	Not yet evaluated on firmware	Training data or per-program learning cost
Graph neural networks	GNNs on code graphs [71,72]	Used for vulnerability prediction; little evaluation inside symbolic execution	N/A	Not evaluated	Needs labelled training data
Learned constraint solving	DNN satisfiability classification [73]; learned strategies [74,75]; solver selection [76]	N/A	Up to 100× vs. default Z3 [74]; up to 198.4% PAR-2 gain [76] (solver benchmarks)	Not evaluated on embedded constraints	Unsound unless confirmed by an exact solver
Approximation of intractable constraints	No evaluated neural approach found; incomplete non-neural techniques [77]	N/A	Research gap	Research gap	No evidence yet
Large language models	Environment models, harnesses, specifications [30,78,79]	N/A	N/A	Promising for environment modeling; limited evaluation	Nondeterministic; outputs need validation
State merging and pruning	Hand-designed cost models [80,81]; learned pruning [82]	Reduces states at merge points	Merging can complicate constraints	Not evaluated	Learned merging not yet evaluated
Function summarization	Compositional summaries [54]; no learned symbolic summaries found	Avoids re-analysis at call sites	N/A	Not evaluated	Research gap
Hybrid fuzzing and symbolic execution	Selective symbolic execution [49]; fuzzers learned from symbolic execution [83]	Complements fuzzing on hard branches	N/A	Silent memory corruption hampers feedback [26]	Learned switching not yet evaluated

AI-guided symbolic execution methods are categorized by their fundamental methodology, path explosion management, constraint solving efficiency, embedded appropriateness and restrictions in the taxonomy shown in Table 3. Although the table offers a high-level comparison, a more thorough architectural view is necessary to comprehend how different AI modules interact with the symbolic execution engine.

Table 4 applies this classification to the AI-guided symbolic execution studies discussed in Sections 4 and 5. Two patterns stand out: most studies target path selection or constraint solving and almost none were evaluated on embedded targets.

Table 4. Classification of AI-guided symbolic execution studies by stage, technique and evidence status.

Study	Symbolic Execution Stage	AI Technique	Integration Pattern	Publication Status	Evaluated on Embedded Targets	In Review Corpus
Wu et al. [31]	Path selection	Tabular Q-learning	Online search guidance	Peer-reviewed (short paper)	No	Not retrieved
Paduraru et al. [68]	Path selection	Deep Q-network with LSTM	Online search guidance	Peer-reviewed (workshop)	No (x86 binaries)	Yes
Cha et al. [28,69]	Path selection	Learned parametric heuristic	Offline-learned heuristic	Peer-reviewed	No	Yes [28,69] pre-2020
Learch [29]	Path selection	Supervised learning	Offline-learned heuristic	Peer-reviewed	No	Yes
Chameleon [70]	Path selection	Online heuristic learning	Within-campaign adaptation	Peer-reviewed	No	No (pre-2020)
PySymGym [27]	Path selection	Training infrastructure for navigation agents	Search guidance (infrastructure)	Peer-reviewed (workshop)	No	Not retrieved
SymTuner [84]	Engine configuration	Adaptive parameter tuning	Within-campaign adaptation	Peer-reviewed	No	Yes
Cha and Oh [82]	State management	Learned state-pruning strategy	Offline-learned heuristic	Peer-reviewed	No	Yes
DeepSolver [73]	Constraint solving	Neural satisfiability classification	Solver front-end	Preprint	No (trained on avionics and automotive models; evaluated on general-purpose Java programs)	Yes
FastSMT [74]	Constraint solving	Learned tactic strategies	Solver guidance	Peer-reviewed	No (SMT benchmarks)	No (pre-2020)
Chen et al. [75]	Constraint solving	Learned solving-strategy synthesis	Solver guidance	Peer-reviewed	No	Yes
MachSMT [76]	Constraint solving	Algorithm selection	Solver selection	Peer-reviewed	No (SMT benchmarks)	No (no SE - symbolic execution)
Hula et al. [85]	Constraint solving	Graph neural network scheduling	Solver selection	Peer-reviewed	No	No (no SE)
Deng et al. [30]	Environment modeling	Large language model	LLM-generated function models	Peer-reviewed (journal)	Partly (sensor-driven systems)	Yes
LLM-powered symbolic execution [86]	Several stages	Large language model	LLM inside the engine	Peer-reviewed	No	Yes
Wu et al. [17]	Test generation	Large language model	LLM combined with path-aware execution	Preprint	No	Yes (ICPC 2026 version)
Xu et al. [87]	Input seeding	Large language model	LLM-generated seed inputs	Peer-reviewed	No	Not retrieved
Wang et al. [88]	Code preparation (Python)	Large language model	LLM-generated code for execution	Preprint	No	Yes
Yang et al. [78]	Specification inference	Large language model	LLM output validated by execution	Preprint	Partly (aerospace code)	Yes
Wu et al. [89]	Invariant inference	Large language model	LLM output checked by model checking	Peer-reviewed	No	No (no SE)
Yu et al. [90]	Invariant inference	Reinforcement learning with SMT	Learning checked by SMT solving	Peer-reviewed	No	No (no SE)

ILF [83]	Hybrid fuzzing	Imitation learning from symbolic execution	Knowledge transfer to a fuzzer	Peer-reviewed	No (smart contracts)	No (pre-2020)
----------	----------------	--	--------------------------------	---------------	----------------------	---------------

Table 4 lists the AI-guided studies discussed in this article; the last column shows whether each belongs to the review corpus. Four studies were published before 2020 and four do not involve symbolic execution; they are discussed as background. The full texts of three were not accessible (Section 3.4). The complete corpus of 37 studies, with stage, AI technique, publication status, embedded evaluation and quality score, is given in Table A3. Corpus studies that do not appear in Table 4 are summarized in Section 4.7.

The symbolic execution engine (blue), as shown in Figure 2, is the central component of the analysis pipeline. It creates path constraints, converts program code into symbolic states, and calls SMT solvers to generate test cases and bug reports. AI guidance modules (orange) enhance this process in several ways: large language models create environment models for hardware interactions and system calls; neural SMT and approximation networks speed up constraint solving; learned merging and summarization optimize state management and reinforcement learning and graph neural networks guide path selection toward promising regions. Continuous learning is made possible by a feedback loop (green), in which the results of execution influence and enhance subsequent AI guidance decisions. This design emphasizes that AI is an addition to symbolic reasoning rather than its replacement. The modules are not equally mature: as Sections 4.1–4.4 show, learned path selection has the strongest evidence, whereas learned constraint approximation, learned state merging and learned function summarization remain largely unevaluated and should be read as research directions.

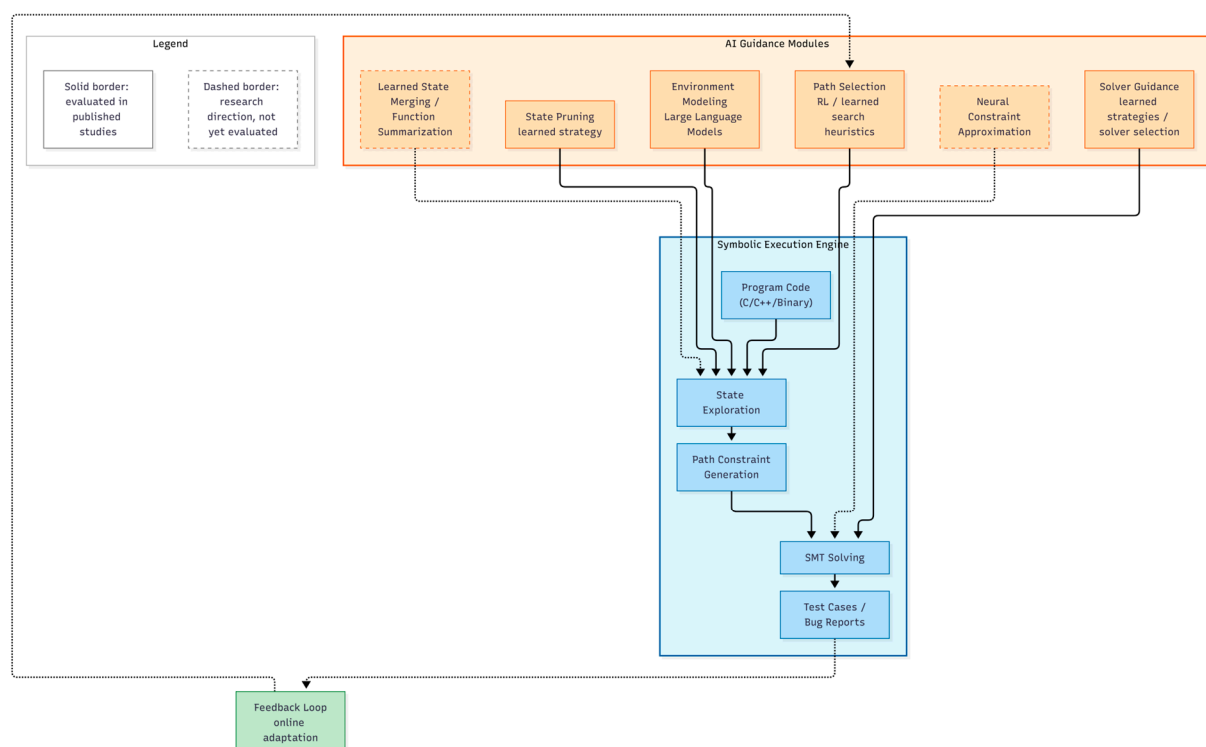


Figure 2. Architecture of AI-Guided Symbolic Execution. Dashed borders mark components that are research directions rather than evaluated techniques. Blue: symbolic execution engine; orange: AI guidance modules; green: feedback loop. Solid arrows show the flow within the engine and the interactions of evaluated components; dotted arrows show the interactions of research directions and the feedback loop.

4.1. AI for Path Selection and Exploration

Out of the millions of possible symbolic states, path selection chooses which one to investigate next. Conventional search techniques use fixed heuristics, which may work well for some applications but poorly for others. AI-guided path selection adapts to program structure and analytical objectives by learning from exploration experience to generate better judgments on the fly.

4.1.1. Reinforcement Learning for Dynamic Path Prioritization

Path selection is framed by reinforcement learning as a sequential decision problem in which an agent selects which state to investigate, gets rewarded according to results like increased coverage or the discovery of a bug and learns a strategy that maximizes cumulative reward over time. Instead of adhering to predetermined guidelines, this method allows symbolic execution to modify its search strategy based on experience. According to the concept of reinforcement learning, program states must be represented as feature vectors that encapsulate pertinent attributes [29,69]. Static program properties taken from the control flow graph, path length, branch history, coverage data and solver-related metrics like constraint complexity are examples of these characteristics. In the simplest form, the agent maintains value estimates for branch choices: Wu et al. use tabular Q-learning to steer KLEE toward a target statement, rewarding the agent when execution reaches one of the statement's dominators, and report in an initial evaluation that more than 90% of the explored paths and instructions were avoided compared with KLEE's default search [31]. Deep variants replace the value table with a neural network; Paduraru et al. embed a deep Q-network in an LSTM over the sequence of branch points of a concolic engine for x86 binaries [68].

Symbolic execution may need to balance coverage, bug detection and resource usage. In the RL-based studies we reviewed, this trade-off is expressed through the reward function rather than through learned families of multi-objective policies; for example, the reward functions in the tool of Paduraru et al. are configurable so that users can define their own test targets [68]. Transfer across programs is a second open issue. Cha et al. learn a parameterized search heuristic separately for each subject program [28], whereas Learch trains a state-selection model offline on data generated by symbolic execution of training programs and applies it to other programs [29]. We did not find evaluations of multi-objective or transfer-learned RL policies on embedded firmware.

The learning-based path-selection studies differ in how they define state, action and reward. States are tabular branch choices in Wu et al. [31], sequences of branch points encoded by an LSTM in Paduraru et al. [68], and vectors of hand-crafted features such as depth, coverage and constraint statistics in Cha et al. and Learch [29,69]. The action is the choice of the next state to explore in Wu et al. and Learch [29,31], and the choice of the branch to negate in the concolic tools of Paduraru et al. and Cha et al. [68,69]. The learning signal is reaching a dominator of the target statement [31], user-configurable test targets [68], the estimated contribution of a state to coverage within the time budget [29], or the final branch coverage obtained with a heuristic whose weights are optimized per program by search rather than learned by reinforcement [69]. Because these definitions differ, the reported gains are not directly comparable across studies (Section 3.6).

4.1.2. Graph Neural Networks Operating on Code Property Graphs

Graph neural networks develop representations that direct exploration by utilizing the inherent graph structure of programs. Program elements, such as basic blocks or statements, can be represented as nodes in a graph, while relationships, such as control flow, data flow and calling context, are captured by edges.

Code property graphs encapsulate syntactic and semantic program information by combining several graph representations into cohesive structures [91]. Data flow graphs monitor value propagation; control flow graphs record execution ordering and abstract syntax trees offer hierarchical structure. Learning algorithms can simultaneously access complementary knowledge sources by combining these into a single graph.

Graph neural networks learn node representations from this structure by message passing. Such representations have been used successfully to predict which functions are likely to be vulnerable [71]; by contrast, we found little direct evidence that graph neural networks have been evaluated for predicting constraint difficulty or branch coverage inside a symbolic executor.

For path selection, a learned model can score each state on the current frontier and prioritize the highest-scoring states. Learch, for example, estimates each state's expected contribution to coverage within a time budget [29], although it relies on engineered state features rather than a graph neural network. Infrastructure for training such navigation models for a symbolic execution engine has also been proposed [27].

Another use of graph networks for guiding is vulnerability prediction. Networks trained on labelled vulnerable code can recognize functions with similar characteristics, both in C code [71] and in smart contracts [72]. Using such predictions to prioritize symbolic exploration of high-risk regions is a plausible combination, but we found few studies that evaluate it end to end.

4.1.3. Heuristic Learning from Past Executions

Techniques that learn heuristics directly from traces of execution provide lightweight alternatives with low training overhead, going beyond reinforcement learning and graph networks [69]. These methods take patterns from fruitful investigations and apply them to direct symbolic execution in the future.

Cha et al. represent a search heuristic as a parametric function over features of symbolic states and learn its parameters automatically from the outcomes of previous runs [28,69]; the learned heuristics outperformed manually crafted ones on the reported benchmarks [28].

Similarity-based reuse of exploration results across comparable code fragments is conceivable and could build on learned measures of code similarity [92], but it has not been evaluated for symbolic execution.

Learning has also been applied to the configuration of the engine rather than to state priorities alone: SymTuner adaptively tunes the external parameters of a symbolic execution engine during a testing campaign [84]. We did not find evaluated ensemble approaches that combine several learned state-selection predictors.

Heuristics can also be adjusted during a single testing campaign: Chameleon learns new search heuristics from the knowledge accumulated during concolic testing and switches between them on the fly [70]. As exploration continues, lightweight learning algorithms update state priorities in real-time without the need for pretraining and can quickly adjust to program-specific features. When training data from comparable programs is not available, this method is especially useful for one-time analytical jobs.

4.2. Neural and Hybrid Constraint Solving

One of the main performance limitations in symbolic execution is constraint solving, which frequently takes up a significant amount of analysis time. By learning common constraint patterns, estimating challenging constraints or combining neural reasoning with conventional SMT solving, neural and hybrid techniques seek to speed up solving.

4.2.1. Neural SMT Solvers

Learning-based approaches to constraint solving either predict satisfiability directly or guide a conventional solver. DeepSolver trains a deep neural network on previously solved path conditions and uses it to classify the satisfiability of new path conditions during symbolic execution [73]; because such predictions carry no correctness guarantee, they are best used as a filter in front of an exact solver. FastSMT instead learns a policy over the tactics of the Z3 solver and synthesizes it into an interpretable strategy that guides the unmodified solver; on its benchmarks it solved 17% more formulas and was up to 100 times faster than the default Z3 strategy [74]. A related approach synthesizes solving strategies tailored to the constraints of a particular program under symbolic execution [75]. MachSMT learns to predict which of several SMT solvers will be fastest on a given formula and is explicitly positioned as a complement to, not a replacement for, conventional solvers [76].

Logical formulas can be represented as graphs whose nodes are variables and clauses; NeuroSAT learns to predict the satisfiability of Boolean formulas in this form from single-bit supervision [93]. The message traversing these graphs gains the ability to identify structural patterns associated with satisfiability. Such predictions are approximate and have been demonstrated mainly on synthetic Boolean benchmarks rather than on constraints produced by symbolic execution.

Graph neural networks have been used to schedule SMT solvers for given formulas [85], an approach that could be applied to the bit-vector constraints that model fixed-width integer arithmetic in programs. Networks can be trained to predict satisfiability and recommend variable assignments using corpora of constraints from symbolic execution. These predictions reduce the number of cases the solver must investigate by directing branching heuristics toward promising pathways when used with conventional solvers.

It is possible to incorporate neural guidance into current SMT solvers without completely replacing them [94]. The neural network effectively learns solver heuristics from data by advising the solver on which variables to split on and which theoretical lemmas to apply. This method may enhance performance for frequently occurring constraint patterns while maintaining the solver's soundness and completeness.

A natural safeguard is to let an exact solver confirm any result a network proposes, so that incorrect predictions cost time but not soundness. Beyond preliminary work such as DeepSolver [73], we did not find peer-reviewed evaluations of this pattern on symbolic-execution constraints.

4.2.2. Approximation Networks for Intractable Constraints

Certain constraint classes, such as nonlinear arithmetic, floating-point computations and requirements using sophisticated theories, might result in exponential blowup or infinite timeouts. Neural models can learn to predict satisfiability from data [93], but we did not find evaluated approaches that train networks to compute sound over-approximations of such constraints for symbolic execution.

For SMT solvers, nonlinear constraints requiring the multiplication of symbolic variables present unique challenges. Incomplete techniques that trade completeness for speed have been proposed for nonlinear integer arithmetic [77]; such techniques, rather than neural approximations, are currently the practical option when exact solving times out.

Through exception semantics, special values and rounding, floating-point limitations add further complexity. Current practice relies on exact encodings such as improved bit-blasting [61], and a recent empirical study shows that floating-point support in symbolic executors remains limited [24].

Constraint reuse is better established. Green normalizes constraints so that previously computed solver results can be reused within and across analyses [56]. Because such

reuse requires syntactic or semantic equivalence between constraints, DeepSolver trains a classifier on stored solutions in order to extend reuse to constraints that are similar but not equivalent [73].

A hybrid scheme in which a learned model first screens paths and an exact solver is invoked only for paths used to generate tests would preserve the soundness of generated tests; we did not, however, find published evaluations of such an approximate-then-refine pipeline.

4.2.3. Neuro-Symbolic Integration: Combining Logical and Neural Reasoning

The complementing capabilities of neural networks include pattern recognition, approximation and data learning. Also, symbolic reasoning: soundness, interpretability and handling of complex logic are combined in neuro-symbolic techniques [95]. Such integration could, in principle, retain the guarantees of symbolic reasoning while speeding up analysis.

Neuro-symbolic AI surveys describe architectures in which neural components propose candidates and symbolic components check them [96,97]. In symbolic execution, the closest realized instances are learned classifiers or strategies whose outputs are confirmed by an exact solver [73,74]; we did not find evaluated systems in which solver feedback is used to retrain the neural component during analysis.

Learned selection among solving strategies is documented for conventional solvers [74,76], and neuro-symbolic approaches have been surveyed for the Internet of Things [98]; selecting between neural and symbolic reasoning for each constraint has not yet been evaluated for symbolic execution.

Combinations of LLMs with SMT solvers have been applied in other domains, for example, automated financial and legal compliance analysis [99], but differentiable SMT solving for training the components of a symbolic executor has not been demonstrated.

4.3. Large Language Models in Symbolic Execution

For jobs requiring semantic knowledge beyond typical program analysis, large language models trained on enormous datasets of text and code have become effective tools [17]. They are useful for enhancing symbolic execution because of their capacity to comprehend and produce code, deduce purpose and model intricate patterns.

4.3.1. LLM-Assisted Environment Modeling

A crucial bottleneck for symbolic execution is still environment modeling, which accurately depicts system calls, library operations and hardware interactions [26]. By learning from existing code, documentation and execution traces, large language models (LLMs) have the potential to automatically create or improve environment models [86].

LLMs have been used to generate models of unknown functions in sensor-driven systems for security verification [30]. The LLM creates candidate models in a format appropriate for symbolic execution given function signatures and accessible documentation. These models capture resource requirements, side effects and input-output relationships that would otherwise need to be manually specified.

Interactive refinement, in which an analyst iteratively corrects LLM-generated models, is a natural extension, but we did not find studies that evaluate its accuracy or the manual effort it saves.

Peripheral behavior can also be inferred automatically without LLMs: P2IM builds models of peripheral interfaces from the firmware's own accesses to memory-mapped I/O, enabling hardware-independent firmware testing [100]. Whether LLMs can infer comparable models from firmware code and documentation has not yet been evaluated in the studies we reviewed.

Automated model creation with feedback loops combines LLM generation with symbolic execution validation [78]. Models are proposed by the LLM, tested on real-world examples through symbolic execution and failures are reported back to the LLM for improvement. Without human intervention, this automated cycle gradually increases the accuracy of the model.

4.3.2. Prompt-Based Harness and Test Generation

In symbolic execution, test harnesses need a lot of manual labor. A test harness is code that calls the function under test, marks its inputs as symbolic and checks properties; in KLEE, for example, inputs are made symbolic by explicit calls in the harness [19]. Large language models can generate such harnesses from function signatures and usage examples.

The closest large-scale evidence concerns fuzz drivers, which have the same structure: a study of LLM-generated fuzz drivers for 86 API targets from 30 C projects found that the generated drivers can match industry-written drivers in fuzzing outcomes, but that LLMs struggle with APIs that have intricate usage requirements [79]. LLM-based code generation has also been used to support symbolic execution of Python programs [88]. The saving in manual effort for symbolic-execution harnesses specifically has not yet been quantified.

PALM combines symbolic path enumeration with LLM-based test generation: it enumerates program paths at the AST level, turns each path into an executable program variant whose assertions encode the branch decisions along the path, and asks an LLM to generate an input for that path; each generated test is executed against the variant, and the first failing assertion is returned to the LLM for up to five refinement rounds [17]. On 124 HumanEval-Java programs, this raised path coverage by 35.0% over direct prompting of the same model (GPT-4o-mini), whereas with a stronger model (o4-mini) direct prompting achieved slightly higher coverage than PALM [17].

Invariant inference from code and documentation is used to identify program properties that are likely to hold at various stages, such as loop invariants, class invariants or method postconditions. These invariants are candidates for formal specification or guidance for symbolic exploration toward potentially violating routes. LLM-inferred invariants can capture programmer intent that may not be explicitly disclosed.

Test oracle generation allows for automated verification of functional correctness by predicting expected program outputs for generated inputs [101]. These predictions can be improved through several iterations of symbolic execution and feedback, even though they are not entirely accurate. They also serve as oracles for finding bugs.

4.3.3. Invariant and Precondition Inference

Program invariants and function preconditions capture essential properties that apply at all stages of execution [102]. These properties guide symbolic execution by defining what must be true for correctness of analysis, what can be assumed for efficiency and what conditions lead to interesting behavior.

Invariant inference can use LLMs to generate candidate invariants from code; in the neuro-symbolic approach of Wu et al., LLM-proposed loop invariants are checked with bounded model checking [89]. Using such invariants to speed up symbolic execution, for example through state merging, has not been evaluated in the studies we reviewed.

Precondition inference determines the conditions under which functions can be safely called. Weakest preconditions have been inferred for restricted program classes such as non-deterministic linear array programs [103], and symbolic execution has been combined with LLMs to generate program specifications automatically [78].

Loop invariant inference addresses one of the most difficult problems in program verification; reinforcement learning combined with SMT solving has been used to infer loop invariants [90]. Whether such invariants accelerate symbolic execution of loops has not yet been measured.

Chain-of-thought prompting may improve the quality of LLM-proposed invariants, but we found no evaluation of this effect on embedded software.

4.4. Learning-Based State Management

State management decisions—when to merge states, how to aggregate features and how to model memory—have a direct impact on the scalability of symbolic execution. Learning-based approaches could make these decisions by predicting which states are similar, which features are worth aggregating and which memory behaviors are important for analysis; as this section shows, however, most published techniques in this area still rely on hand-designed cost models.

4.4.1. State Merging via Similarity Learning

State merging reduces roadblocks by merging similar states at control flow junctions [52]. However, merging indiscriminately creates complex constraints that slow down the solution. Similarity learning decides when states are similar enough that the benefits of merging outweigh the costs.

Deciding when to merge requires estimating costs and benefits. Kuznetsov et al. estimate the effect of a merge on future solver queries with a static analysis and merge only when this is predicted to pay off [80], while veritesting merges paths across multiple branches by alternating between dynamic and static symbolic execution [81]. Both rely on hand-designed cost models rather than learned similarity functions.

Learning merge decisions from data is a natural extension of these cost models, but we did not find peer-reviewed studies that train neural networks to make merging decisions, nor evaluations of learned merging on embedded benchmarks. A related, learning-based form of state management is state pruning: Cha and Oh learn a strategy for aggressively pruning symbolic states [82].

Merging could also be framed as a sequential decision problem for reinforcement learning, with rewards based on solver time and exploration progress. Any such policy would have to respect the conditions under which merging is sound, for which formal foundations exist [104], and would benefit from encodings such as quantifier-based merging that limit constraint growth [52].

4.4.2. Function Summarization Using Sequence Models

Function summaries avoid re-analyzing a function at every call site by reusing a symbolic description of its effect; compositional dynamic test generation computes such summaries on the fly and reuses them across calling contexts [54].

Transformer models have been trained to summarize source code in natural language [105], but such summaries are descriptive text, not symbolic input-output relations, and cannot be consumed by a symbolic executor.

We did not find evaluated approaches in which sequence or graph models learn symbolic function summaries, whether flat, hierarchical or incremental, and we therefore treat learned summarization as an open research direction rather than an established technique.

4.4.3. Memory Modeling with Predictive Networks

State explosion is largely caused by memory modeling, which represents heap-allocated data structures, pointers and symbolic addresses [106]. Whether learned models can simplify memory representations is largely unexplored.

Existing memory models reduce this cost with hand-designed representations, for example by segmenting memory into independent objects [107] or by making the memory model a parameter of a compositional symbolic execution engine [108].

Using sequence models to predict memory accesses along symbolic paths, for example to support speculative memory modeling, has not been evaluated in the studies we reviewed and remains a research direction.

4.5. Hybrid AI–Symbolic Architectures

Complete designs strongly combine AI components with symbolic execution engines, going beyond individual techniques. When exact reasoning is required, these hybrid systems rely on symbolic execution, but they also use AI for guiding, prediction and approximation.

4.5.1. AI-Guided Fuzzing + Symbolic Execution

The combination of fuzzing and symbolic execution exploits the strengths of each technique: fuzzing efficiently explores many shallow paths, while symbolic execution copes with complex constraints that block random mutation. Driller invokes symbolic execution selectively when fuzzing stops making progress [49], and whitebox fuzzing systems such as SAGE apply symbolic reasoning at scale [20]. Learning could decide when to switch between techniques and how to focus each one, but the evidence for learned switching is still limited.

A related, better-documented use of learning transfers knowledge from symbolic execution to fuzzing: ILF learns a fuzzing policy from inputs generated by symbolic execution and applies it to smart contracts [83]. We did not find evaluations of classifiers that learn, from many fuzzing campaigns, when to switch to symbolic execution.

Symbolic execution has also been used to guide fuzzing of hardware designs [109]. Reinforcement learning policies that allocate time between fuzzing and symbolic execution are a natural next step, but we found no published evaluation of them.

For embedded devices, hybrid approaches face an additional obstacle: memory corruptions often do not cause observable crashes on such devices, which weakens the feedback that fuzzing relies on [26]. Learned models of hardware-interaction costs could help decide when symbolic execution is worth its overhead, but we found no evaluations on real embedded devices.

4.5.2. ML-Primed Symbolic Exploration

Machine learning can also be applied before symbolic execution starts. Learned vulnerability predictors flag functions that are likely to be vulnerable [71], and learned search heuristics capture which exploration choices were productive on previously analyzed programs [28,29]. Using such predictions to choose where symbolic execution should start, or which functions it should target first, is a plausible combination, but we found little direct evaluation of it and none on embedded benchmarks.

Learned configuration of the engine is better supported: SymTuner adapts the external parameters of a symbolic execution engine during testing [84]. We did not find evaluations of models pretrained on embedded software corpora for this purpose.

Feedback loops between learning and exploration are best documented within a single campaign, where Chameleon learns new heuristics from accumulated exploration results [70]; improvement across multiple analysis runs on related code bases has not yet been evaluated.

4.5.3. Continuous Learning Symbolic Executors

Continuous learning systems would accumulate knowledge across analysis runs and retrain their models for path selection, state management or constraint solving as they analyze more programs. Learned representations of symbolic expressions [110] and learned search heuristics [28,29] are possible building blocks, but we did not find evaluated cross-run learning repositories for symbolic execution.

Within-run adaptation is better supported: Chameleon adapts its search heuristics during a single campaign [70]. Combining such online adaptation with knowledge accumulated across runs (lifelong learning) has not yet been evaluated.

Active learning has been applied to path exploration strategies for symbolic execution [111]; using it to choose which programs to analyze next in order to improve a guidance model remains unexplored.

Foundation models pretrained on programs and analysis traces are sometimes proposed as a way to absorb many specialized learning techniques. Current evidence is narrower: LLMs have been used to generate test inputs that seed symbolic execution [87] and to power parts of the symbolic execution process [86], but we found no foundation model trained specifically for symbolic execution or evaluated on embedded benchmarks.

4.6. Critical Assessment of AI-Guided Symbolic Execution

Even though AI integration has been shown to have many benefits, there are still some important problems that need to be looked at closely. First, AI guidance doesn't fix the path explosion problem. It just makes it less of a problem by smartly prioritizing. The underlying state space is still exponential, and learned policies may still miss important paths that are not in their training distribution.

Second, the soundness guarantees that are often associated with symbolic execution become less trustworthy when neural approaches for solving constraints introduce uncertainty. Without additional layers of validation, approximations that speed up solving are not safe for safety-critical verification because they may miss potential paths or produce false positives.

Third, because certifying authorities want tools that are deterministic, explainable, and have predictable failure mechanisms, the deployment of AI-guided approaches in safety-critical domains governed by standards like ISO 26262 and DO-178C remains problematic. These requirements are not entirely met by current AI components.

Finally, there is still little generalization across different embedded architectures and application domains; rules that perform well on one platform frequently don't function well on another, necessitating extensive retraining or adaption.

These drawbacks show that AI should be carefully integrated in safety-critical situations because it is an augmentative technology rather than a replacement for rigorous symbolic reasoning.

4.7. Further Studies in the Review Corpus

Twenty-five studies of the review corpus (Table A3) are not covered in Sections 4.1 to 4.6. They are summarized here by the stage of symbolic execution they support.

Path selection. Several studies learn state-selection strategies for KLEE. MS-ALS trains a feed-forward network that predicts the reward of symbolic states and uses active learning to select informative states for labelling; on eight real-world programs it achieved 16.4% more average coverage than the second-best strategy [111]. TLS transfers such a network across program families by freezing part of its layers and retraining on family-specific data, increasing branch coverage on `grep` by about 15 percentage points over random search strategies [112]. DLF selects dynamically among an ensemble of models by sample similarity and extracts features from the control flow graph with a gated graph

neural network [113], and FeatMaker generates state features automatically from path conditions instead of relying on manually designed features [114]. SyML learns patterns of vulnerable paths from execution traces of binaries and uses its predictions to prioritize exploration [115]. For smart contracts, SmarTest uses a statistical language model of vulnerable transaction sequences to prioritize symbolic paths and found zero-day vulnerabilities [116], and NumScout uses an LLM to prune functions that are unrelated to numerical defects before symbolic execution [117]. Yang et al. studied in simulation how imperfect satisfiability predictions are best used to prioritize paths, and found that the design of the analysis tool and the path-selection algorithm can affect performance as much as the accuracy of the predictor [118].

Constraint solving. SMTimer predicts the solving time of a query so that hopeless queries can be abandoned early; on constraint models collected from angr and KLEE its adaptive predictor reached an F1-score of 0.74 and improved solving efficiency by 1.25 to 3 times in a simulation [119]. MLBSE replaces exact solving of nonlinear and black-box path conditions with a sampling-and-learning optimization inside Symbolic PathFinder and reports an estimated confidence in satisfiability when no solution is found [120]. SIRISMT synthesizes Z3 solving strategies with machine learning and improved code coverage by up to 11.8% when used in KLEE on Coreutils programs [121]. ICON, from the group that developed DeepSolver [73], classifies path conditions with neural networks; trained on path conditions collected from the TCAS, Wheel Brake System and MerArbiter models, it was faster than Z3, Green, GreenTrie and FastSMT on nine further Java subjects [122]. For Android apps, random-forest classifiers predict unsatisfiable paths from path-level statistics with 95% balanced accuracy, allowing the TIRO analysis tool to skip 51% of the paths [123]. Sibyl uses graph neural networks to select the most suitable SMT solver for each query and was evaluated, among other settings, inside KLEE [124]. Related to state management, Molina et al. learn to decide whether partially symbolic heap structures in generalized symbolic execution can be extended to valid structures, replacing manually written feasibility checks [125].

Environment modeling. ADFEmu is the study in the corpus most directly aimed at embedded systems: it combines concolic execution with LLMs to emulate direct memory access (DMA) peripherals during firmware re-hosting and, on real firmware samples, reached a 100% re-hosting success rate and explored 5.31% more paths than the state of the art [126]. A preprint by the group of Deng et al. [30] applies LLM-based modeling of unknown functions to WebAssembly software for medical devices [127]. AutoStub uses genetic programming to derive symbolic stubs for external functions from sampled input-output pairs and approximated 55% of the evaluated functions with over 90% accuracy [128].

Test generation, hybrid fuzzing and other stages. Cottontail uses an LLM inside concolic execution to generate highly structured test inputs [129], and HyLLfuzz replaces the concolic solver of hybrid fuzzing with an LLM that reasons over code slices, covering 31% to 59% more branches than three state-of-the-art hybrid fuzzers on 14 C programs [130]. An earlier deep-learning approach predicts which paths are better suited to fuzzing and which to symbolic execution [131]. For unit testing of Rust, a different tool, also named PALM, combines path constraints obtained by program analysis with LLM-based test generation [132]. SymGPT lets an LLM translate the rules of ERC standards into constraints that are checked by symbolic execution of smart contracts, and reported 5783 rule violations in 4000 contracts with a false discovery rate of 2.0% [133]. LIFT uses an LLM to rewrite time-intensive blocks of the intermediate representation executed by the symbolic engine, reducing execution time by up to 53.5% [134]. Baton learns classification models for the conditionals of a conditional execution graph to guide random testing and uses

targeted concolic testing for the remaining conditionals; on three Java benchmarks it improved branch coverage over concolic testing by 2.3% to 29.9% [135].

Apart from ADFEmu and, in part, the medical-device study [127], none of these studies was evaluated on embedded targets; ICON used avionics and automotive models only to train its classifiers [122], which confirms the gap discussed in Section 8.

5. Tool Landscape and Framework Comparison

The methods discussed in Section 4 have been used to a range of tools, from established frameworks supporting real-world analysis workflows to academic prototypes showcasing particular AI integrations. The field of AI-enhanced symbolic execution tools is surveyed in this section, which ends with a comparative analysis that highlights each tool's advantages, disadvantages and suitable application areas.

5.1. Academic Prototypes

Several prototypes investigating the integration of AI with symbolic execution have been generated by academic research. These tools establish feasibility and characterize the performance of novel approaches. However, they usually place more emphasis on showcasing new techniques than on production readiness.

5.1.1. KLEE with Machine Learning Extensions

KLEE's modular architecture and broad use make it a foundation for many machine learning enhancements [23]. The best-documented ones replace KLEE's hand-crafted search heuristics with learned ones: Leach learns a state-selection model offline [29], Cha et al. learn parametric search heuristics [28,69], and Wu et al. steer KLEE with Q-learning [31]; SymTuner additionally learns to tune the engine's external parameters [84]. These studies report coverage or bug-finding gains on general-purpose C programs; none of them was evaluated on embedded firmware.

KLEE extensions that preprocess target programs into code property graphs and use graph neural network predictions to prioritize states were not described in the studies we reviewed; code property graphs and GNN-based vulnerability prediction exist independently [71,91].

Similarly, we found no published KLEE extensions that replace the default solver with neural approximate-then-refine solving or that use learned similarity metrics for state merging. KLEE's constraint-solving chain relies on caching and exact solvers [19], which learned solver strategies could complement [74,76].

5.1.2. Angr with AI-Guided Exploration

The Python-based design and plugin system of angr make it a convenient platform for experimenting with learning-enhanced exploration at the binary level [34]. Reinforcement learning formulations, in which binary code regions are states and exploration choices are actions, are a natural fit, but we did not find evaluated reinforcement learning extensions of angr in the literature we reviewed.

Learned seed selection for concolic execution, and LLM-generated models for unresolved system calls and library functions [30], could likewise be integrated into angr; published evaluations of such integrations are not yet available.

5.1.3. SymLM and Execution-Aware Learned Representations

SymLM is sometimes described as a neuro-symbolic executor, but it addresses a different task: it predicts function names in stripped binaries from context-sensitive, execution-aware code embeddings [136]. It shows how representations learned from execution

behavior can support binary analysis, but it does not maintain symbolic path constraints or provide soundness guarantees, and it is therefore not a symbolic executor.

Learned loop invariants, which could let a symbolic executor avoid explicit unrolling, have been studied in verification settings, where candidate invariants proposed by LLMs or by reinforcement learning are checked with bounded model checking or SMT solving [89,90]; we did not find symbolic executors that integrate such invariants.

Fully differentiable symbolic executors, in which gradients flow from analysis outcomes back into guidance components, have not, to our knowledge, been implemented or evaluated; they remain a conceptual direction.

5.1.4. Custom Reinforcement Learning-Enhanced Tools

Several research prototypes apply reinforcement learning directly to the decisions of a symbolic executor: Q-learning controlling the branch choices of KLEE when targeting a statement [31], a deep Q-network over sequences of branch points in a concolic engine for x86 binaries [68], and infrastructure for training navigation agents for a symbolic execution engine [27]. To our knowledge, none of these prototypes has been evaluated on embedded firmware, and Wu et al. report only an initial evaluation [31].

Multi-objective policies and libraries of transferable policies are established ideas in the general reinforcement learning literature, but we did not find symbolic execution tools that implement them.

5.2. Industrial and Open-Source Tools

Industrial and established open-source tools prioritize reliability, scalability and usability. As the following subsections show, their published versions contain few or no learned components; they matter for this review mainly as the platforms into which AI guidance would have to be integrated.

5.2.1. Mayhem and ForAllSecure

Mayhem was introduced as a system for finding exploitable bugs in binary code that combines online and offline symbolic execution [137]. Its later cyber reasoning system, developed at Carnegie Mellon University and ForAllSecure (Pittsburgh, PA, USA), combined symbolic execution with fuzzing and won the DARPA Cyber Grand Challenge in 2016 [138].

The peer-reviewed descriptions of Mayhem present it as a hybrid of symbolic execution and fuzzing [137,138]; we did not find published evidence that its path prioritization relies on learned models. Mayhem is therefore included here as a reference point for hybrid symbolic execution rather than as an example of AI-guided symbolic execution.

Descriptions of later commercial extensions, for example to embedded targets, are not documented in the peer-reviewed literature and are not assessed in this review.

5.2.2. S2E with Plugin-Based AI Guidance

S2E's plugin architecture enables the integration of AI guiding components into a symbolic execution platform that is already operational. Its plugins can observe and influence path selection, so learned components could in principle be added without modifying the core engine [139].

We did not find published S2E plugins that use neural approximations for constraint caching or online learning for path priorities; the documented plugins implement conventional search strategies and analyses [139].

Likewise, LLM-based environment modeling has been demonstrated in other analysis settings [30,86], but not, to our knowledge, as S2E plugins.

5.2.3. Firmware Analysis Frameworks with Symbolic and AI Components

Specialized frameworks for firmware analysis adapt symbolic execution to the special features of embedded systems, such as resource limits, real-time constraints and hardware interfaces [140]; as discussed below, AI components are not part of the published versions of these frameworks.

FIE extends KLEE to firmware for the MSP430 family of microcontrollers; applied to 99 open-source firmware programs, it verified memory safety for the majority of them and found 21 bugs [141]. We found no published neural extensions of FIE.

Instead of attempting accurate hardware modeling, HALucinator replaces hardware abstraction layer functions with high-level models [62]. These models are written by hand; generating them with LLMs is a promising direction but was not part of the published system.

Avatar combines symbolic execution in an emulator with execution on the real device by forwarding peripheral accesses to the hardware [50]. What to forward is configured by the analyst; we found no published work that learns these offloading decisions.

Benchmarks for reproducible embedded-systems experiments exist [142], but we found no evaluation on them showing that learned guidance improves firmware analysis.

5.3. Comparative Analysis

The variety of methods, target domains and design tradeoffs in this developing subject are reflected in the variety of AI-enhanced symbolic execution tools. Table 5 lists representative tools together with the role that learning actually plays in each, as documented in the cited publications.

Table 5. Representative symbolic execution tools and the documented role of learning in each.

Tool	Type	AI Technique	Supported Platforms	Key Strength	Known Limitation
Learch (on KLEE) [29]	Academic	Learned state-selection model trained offline on symbolic-execution data	C programs (LLVM)	Estimates each state's contribution to coverage within a time budget	Needs training data from prior symbolic-execution runs; not evaluated on firmware
ParaDySE [28,69]	Academic	Automatically learned parametric search heuristics	C programs	Generated heuristics outperformed manually crafted ones in coverage and bug finding on the reported benchmarks [28]	Heuristic is learned per subject program, at additional cost
Chameleon [70]	Academic	Online learning of search heuristics during concolic testing	C programs	Adapts heuristics within a single campaign without pre-training	Evaluated on general-purpose C programs only
SymTuner [84]	Academic	Adaptive tuning of external engine parameters	C programs (KLEE)	Automates engine configuration during testing	Tunes parameters rather than state selection
Q-learning-guided KLEE [31]	Academic prototype	Tabular Q-learning for branch choices	C programs	Large reduction in explored paths toward a target statement	Initial evaluation only; target-directed
DQN concolic tool [68]	Academic, open source	Deep Q-network within an LSTM over branch sequences	x86 binaries	Configurable reward functions for custom test targets	No evaluation on embedded targets reported
DeepSolver [73]	Academic (preprint)	Deep neural network classifying path-condition satisfiability	Path conditions from symbolic execution	Extends solution reuse beyond equivalent constraints	Predictions not guaranteed correct; not yet peer reviewed
FastSMT [74]	Academic	Learned policy synthesized into Z3 strategies	SMT formulas (QF_NRA, QF_BV, QF_NIA)	Solved 17% more formulas; up to 100 times faster than default Z3	Strategies are specific to the training distribution

MachSMT [76]	Academic, open source	Learned algorithm selection among SMT solvers	SMT-LIB formulas	Improved on competition winners in many divisions	Selects solvers; does not replace them
Mayhem [137,138]	Academic origin, later commercial	None documented in peer-reviewed literature (hybrid symbolic execution and fuzzing)	Binaries	Won the DARPA Cyber Grand Challenge (2016)	Proprietary; no published learned components
S2E [35,139]	Academic, open source	None built in; plugin architecture	Full-system binaries, drivers	Mature platform for in-vivo multi-path analysis	Learned guidance would require custom plugins
FIE [141]	Academic	None	MSP430 firmware	Verified memory safety for most of 99 firmware programs; found 21 bugs	Architecture-specific
HALucinator [62]	Academic, open source	None (hand-written HAL models)	Firmware using hardware abstraction layers	Avoids detailed peripheral modeling	Models must be written manually
SymLM [136]	Academic	Execution-aware embeddings for function-name prediction	Stripped binaries	Learned representations useful for binary analysis	Not a symbolic executor; no soundness guarantees

The comparison reveals several trends. Most of the AI integration identified in this review concerns path selection and exploration direction, reflecting the importance of search strategies for symbolic execution performance. Learned search heuristics have outperformed manually crafted heuristics in the evaluations reported by their authors [28,29,69], but these evaluations used general-purpose C programs rather than embedded firmware.

Learning-based support for constraint solving is at an earlier stage. Learned strategy selection and solver selection have produced measurable gains [74,76], and learned satisfiability classification has been explored for symbolic execution [73], but learning currently complements rather than replaces conventional SMT solvers.

Firmware analysis tools are distinguished mainly by embedded-specific modeling of peripherals, interrupts and memory layouts rather than by learned components; in the published literature, AI-guided symbolic execution has not yet been integrated into these tools.

6. Applications in Embedded and Electronic Systems

Some of the most difficult verification and security concerns in current computing are solved by the convergence of AI-guided symbolic execution with embedded systems applications. There is an urgent need for rigorous analysis approaches that can scale to complex, resource-constrained situations because embedded software is rapidly regulating safety-critical activities in consumer electronics, medical devices, autos and industrial infrastructure. One of the most prominent application areas for AI-enhanced symbolic execution is IoT and embedded device firmware security, which is discussed in this section.

6.1. Firmware and IoT Device Security

A vast attack surface has been generated by the growth of Internet of Things devices and firmware weaknesses in embedded systems allow for the remote compromise of everything from industrial controllers to domestic appliances. Promising approaches for finding these vulnerabilities prior to deployment and for reviewing deployed firmware in

cases where source code is not available are provided by AI-guided symbolic execution [33].

6.1.1. Vulnerability Detection in RTOS-Based Firmware

Many embedded devices are built on real-time operating systems that involve hardware abstraction, task scheduling and inter-task communication. The firmware of these systems exhibits complex behaviors caused by resource sharing, task preemption and timing constraints, which lead to vulnerabilities [63].

Memory security vulnerabilities in RTOS-based software are particularly difficult to analyze. Device compromise or failure can be caused by buffer overflows in network packet processing, use-after-free errors in dynamic memory management and integer overflows in control calculations [143]. Because these vulnerabilities typically depend on specific task interleaving and external events that are difficult to replicate deterministically, standard testing often fails to trigger them.

When applied to RTOS firmware, symbolic execution involves solving parallel models of these systems. The analysis must consider all potential task interleaving and interrupt service routines; this state space expands combinatorially with the number of tasks and possible preemption points. Learned path selection could focus exploration on the task interleaving most likely to expose race conditions and deadlocks, but we found no evaluation of this on RTOS firmware; such failures are currently studied with techniques such as fault-injection simulation [144].

Software analysis is further complicated by hardware abstraction layers. While RTOSs offer consistent APIs for accessing devices, the underlying hardware implementations of microcontrollers vary significantly. Symbolic implementations require accurate models of these hardware interactions, which has historically required a significant amount of manual labor. LLM-assisted modeling of hardware abstraction layer functions from documentation and existing code could reduce this manual labor [30], but its effect has not yet been quantified.

The memory protection mechanisms in modern microcontrollers make software analysis difficult. The memory regions that code can access are affected by memory protection units, privileged and non-privileged execution modes and access control configurations. To identify vulnerabilities that bypass these protection mechanisms, symbolic execution must model them; the automatic inference of peripheral models from firmware [100] suggests that parts of this configuration could also be inferred automatically.

Automated firmware testing that exploits firmware-hardware interaction patterns has been applied to real firmware [25], and symbolic-execution tools such as FIE have found bugs in firmware programs [141]. We did not, however, find published reports of AI-guided symbolic execution uncovering vulnerabilities in commercial industrial control firmware.

6.1.2. Analysis of Wireless Protocol Stacks

Billions of devices have software that implements wireless communication protocols such as Bluetooth, Zigbee, Wi-Fi and cellular technology. Protocol stacks are notoriously stateful, complex and susceptible to remote attacks [145]. The ability to methodically explore protocol state spaces and find implementation errors that deviate from specifications is possible with symbolic execution.

An examination of the Bluetooth stack reveals its complexities. The Bluetooth protocol consists of several layers (radio frequency, baseband, link management, logical link control and application profiles), each with complex state machines and timing constraints. Attacks ranging from complete device control to unauthorized pairing of

devices have been made possible by flaws in the Bluetooth implementation [146]. The concurrency of multiple connection states, the asynchronous nature of radio events and the constraints of low-power operation must be handled by symbolic execution of these layers.

Similar difficulties arise with the Zigbee protocol stack, which is widely used in industrial sensor networks and home automation. Devices that transmit messages and maintain routing tables that change as the network topology changes, making their mesh networking capabilities even more complex. Attacks that threaten entire building automation systems have been made possible by flaws in the Zigbee implementation. The study of mesh routing paths and security key exchange, which are likely to involve implementation issues, could be prioritized by AI-driven symbolic execution [147].

The state space of protocol implementations grows exponentially with the number of protocol states, message types and concurrent connections. For practical protocol stacks, traditional symbolic execution rapidly becomes unmanageable [146]. By identifying the protocol states and message sequences that are most likely to reveal vulnerabilities, reinforcement learning for path selection concentrates analysis resources on high-risk protocol implementation areas.

Symbolic guidance has also been combined with mutation-based fuzzing, for example, for hardware designs [109]; applying it to protocol implementations follows the same principle. While symbolic execution resolves limitations for messages that cause intricate protocol behaviors, fuzzing rapidly generates a large number of message sequences. Based on the difficulty of the remaining unexplored states and the observed coverage progress, learned switching policies can decide when to employ each strategy.

Formal models of protocol specifications provide additional guidance for analysis. When protocol specifications are available in formal languages, they can be used to generate expected behaviors against which implementations are checked [148]. LLMs can assist in deriving behavioral models from natural language specifications, bridging the gap between human-readable standards and machine-checkable formalisms.

6.2. Automotive and Aerospace Verification

Safety-critical transportation systems, such those in cars and airplanes, need to be much more reliable than ordinary computers. Strict regulations, such as ISO 26262 for automobiles [21] and DO-178C for airplanes [22], demand extensive testing. This enables the use of sophisticated methods like symbolic execution, which may thoroughly assess software and demonstrate its usefulness.

6.2.1. ECU Software Verification and Safety Standards

Modern cars use many small computers, called electronic control units (ECUs), to control almost everything, from engine management to braking, steering, airbag deployment and even self-driving features. If the software in these systems fails, the consequences can be fatal. That is why there is a strict standard called ISO 26262. It defines safety levels, called automotive safety integrity levels (ASILs), based on how risky a failure would be [21].

ISO 26262 requires proving that software will operate safely in all possible situations. This is usually done through a mix of testing, code review and static analysis. Symbolic execution is a technique that complements these methods by examining many possible paths the software can take and by creating tests for hard-to-reach edge cases [22]. Since modern cars have a huge amount of software on multiple ECUs, it is impossible to test everything. That is where AI-driven symbolic execution comes in. It helps focus analysis on the most critical parts of the software, those with the highest safety ratings or where errors are most likely to occur.

Timing is also critical. Automotive systems must respond within tight deadlines to maintain safety. If a task takes too long, it could indicate a problem. Time-based symbolic execution evaluates the worst-case times for each path and checks whether any of them could cause a missed deadline. This helps developers fix risky parts early [21].

Vehicles also communicate over specialized networks, such as CAN, LIN and Automotive Ethernet. To verify that the software handles these issues correctly, symbolic execution can simulate network behavior, such as message timing and error handling, so testing can be done without physical hardware.

Finally, ISO 26262 requires that software with different safety ratings not interfere with each other. For example, a low-risk function should not cause problems for a high-risk function due to memory, CPU time or network access overhead. Symbolic execution, guided by artificial intelligence, can check for these types of conflicts by learning how resources are shared.

6.2.2. Flight Control Code Analysis Using AI-Symbolic Hybrids

Of all the engineering disciplines, aviation software must undergo the most rigorous safety testing. The rules based on system criticality are set out in a standard known as DO-178C [149]. The most critical systems, such as engine management and flight control, are classified as Level A, while the least critical systems are classified as Level E. These high-risk systems require extensive testing and formal demonstration of correctness.

Testing flight control software is quite expensive; in fact, it often costs more than any other aspect of avionics development. By automatically generating tests and verifying program attributes, a method known as symbolic execution can be useful. However, flight control laws are complex, involving feedback loops, nonlinear functions and a large number of floating-point calculations. Traditional symbolic execution techniques push this complexity to its limits.

By handling complex mathematics more efficiently, AI-driven techniques can help analyze flight control code. Neural networks can learn to predict the results of complex calculations with small, controllable errors. This makes it possible to explore code paths that might otherwise be exhausted using symbolic tools. AI predictions can be verified using precise calculations when accurate results are required for certification.

Flight control systems have to deal with a variety of potential problems, including sensor failures, actuator limitations and changing flight conditions. It is difficult to test every possible combination. The most dangerous situations, such as combinations of sensor defects and control surface limitations that can lead to loss of control, can be the focus of AI-driven symbolic execution [22].

Flight software is built in layers, with multiple backups, different implementations and monitoring systems. As components interact, verification becomes more difficult. This is overcome by composite verification, which tests each component independently before integrating the conclusions. AI-generated summaries of component functions could speed up this process, although the summaries would themselves need verification.

Formal approaches are now accepted by certification bodies as proof of software correctness. Hybrid techniques using AI learning and symbolic reasoning can yield conclusions that are rigorous enough for certification. The use of AI guidance in the development of certified software is possible when symbolic tools can validate AI predictions.

6.3. Smart Contract and Blockchain Embedded Code

Blockchain technology has come a long way since its inception as a coin, enabling decentralized applications running on virtual machines. As the technology expands into the Internet of Things, blockchain nodes will increasingly need to function on small,

resource-constrained devices. Blockchain technology and embedded devices combine to create new and complex verification challenges [150]. The software on these devices must be both secure and efficient, but their limited processing and memory capabilities make traditional verification methods difficult. AI-driven symbolic execution offers a potential method that makes blockchain applications reliable on even the smallest devices by deftly examining software paths to identify errors and vulnerabilities.

6.3.1. Resource-Constrained Blockchain Nodes

Despite memory, processing power and power constraints, lightweight blockchain nodes designed for IoT devices can still participate in a distributed ledger [151]. These nodes, built on hardware similar to microcontrollers, perform important tasks. These tasks include validating transactions, maintaining a partial record of the blockchain state and executing smart contracts.

The amount of processing and storage that smart contracts can use is limited by strict gas limits that these IoT devices must adhere to [152]. If certain limits are violated, the contract is aborted and resources are wasted.

To verify that all possible paths to execution remain within the available gas, smart contract code can be checked using symbolic execution. By checking all paths taken by the code, it identifies contracts that may run out of gas under specific input conditions. By using artificial intelligence to guide this analysis, which can initially focus on the routes that are most likely to approach gas constraints, developers can identify worst-case scenarios for resource consumption.

To maintain security and viability, the consensus methods of built-in blockchain nodes must manage issues such as lost messages, network outages and even attempts to interfere with the system. As the number of nodes and network conditions increase, the number of different states that these protocols can enter increases exponentially [153]. A form of artificial intelligence called reinforcement learning can apply path selection to examine the most important protocol interactions for fault tolerance, helping to identify situations where consensus might fail.

Since light nodes often rely on full nodes to validate transactions, trust assumptions are made that must be properly tested. Merkle proof is an important verification technique that allows a light node to verify that a given transaction is part of a block without having to store all of the block's data [154]. The embedded code must demonstrate that this verification procedure is accurate. These procedures can be verified by symbolic execution, which guarantees that malicious full nodes cannot mislead light clients.

Embedded devices based on blockchains use custom data structures designed for memory efficiency [155]. These structures can be checked for flaws that could corrupt data and ensure that state transitions preserve data integrity using symbolic execution in conjunction with memory modeling. Because predictive memory networks can predict access patterns, modeling memory usage for blockchain applications is less difficult.

6.3.2. Formal Verification of Decentralized Embedded Logic

Smart contracts are blockchain programs that operate exactly as written and cannot be changed once they are implemented. Because of this immutability, any errors or vulnerabilities during deployment could cause irreversible financial losses; therefore, thorough verification is essential before deployment. Formal verification has become an essential tool for ensuring the security and intended functionality of smart contracts, particularly through symbolic execution [156].

Most smart contracts run on the Ethereum virtual machine and their bytecode presents unique analytical challenges due to their stack-based architecture, dynamic jump instructions and susceptibility to re-entry threats. Symbolic execution can investigate all possible behaviors and identify potential security flaws by looking at a contract's bytecode. Since complex contracts with multiple interactions may result in an explosion of possible courses, artificial intelligence can help focus research on the most promising or risky areas [157].

One of the most devastating vulnerabilities in smart contracts is reentrancy, where a contract calls an untrusted external contract, which then recursively calls the original contract until state updates are complete. Symbolic execution can detect these

vulnerabilities by examining the various interfaces of the contract's calls, which could lead to an inconsistent state. Graph neural networks trained on labelled contracts can detect reentrancy and other vulnerability types [72]; using such predictions to focus symbolic exploration on the highest-risk call sequences is plausible but has not been evaluated in the studies we reviewed.

Access control vulnerabilities arise when unauthorized users invoke privileged functions. Symbolic execution can verify that all possible paths to a sensitive function require proper authorization by detecting missing or invalid permission checks [158]. Machine learning can help by inferring expected access control properties from common code samples and symbolic execution can then verify whether the actual implementation meets these expectations.

Financial invariants must hold true for all potential contract executions, such as guaranteeing that the total supply of tokens is constant or that exchange rates stay constant. These invariants can be verified throughout the entire contract state space by symbolic execution that is guided by learned behavior. This allows for the identification of potential violations caused by rounding or arithmetic errors. Specialized methods aid in the analysis of floating-point approximations for contracts involving intricate financial computations.

Finally, modern decentralized applications consist of many interacting contracts, so verification must extend beyond individual contracts to the entire system. The symbolic execution of many interacting contracts must account for all possible invocation sequences and the state interactions between them [159]. Using learned function summaries, composite verification allows for scalable analysis of these composite applications by verifying components in isolation and then combining their verified properties.

6.4. AI-Enabled Embedded Systems

Verifying the accuracy, security and safety of AI components operating on embedded hardware becomes crucial as AI shifts from cloud servers to edge devices. Techniques for examining AI models and the systems that incorporate them are provided by AI-guided symbolic execution.

6.4.1. Verification of ML Inference Pipelines on Edge Devices

When machine learning models are used on small embedded devices, such as sensors, wearables or industrial controllers. They must make predictions under tight resource constraints. These models are frequently optimized using methods like pruning, which removes superfluous connections and quantization, which lowers the precision of numbers, in order to accommodate limited memory and processing power [160]. Although these optimizations make deployment possible, they may also introduce numerical errors and affect prediction accuracy, which may have an impact on safety-critical decisions in autonomous systems or medical devices, for example.

To operate effectively on microcontrollers, quantized neural networks employ low-precision arithmetic, such as binary weights or 8-bit integers. However, the deployed model may behave differently from the original floating-point version due to quantization errors introduced by the decreased precision. SMT-based verification can check whether these errors can cause incorrect classifications for any input within the expected range [16].

Numerical stability is another important issue for security-critical applications. Small numerical errors can propagate through deep networks, potentially distorting classification results.

Memory safety in inference implementations is important because models operate in a highly constrained environment. Issues such as buffer overflows in tensor operations,

use-after-free errors in memory pooling and integer overflows in index calculations can all pose a threat to system safety [19]. Symbolic execution can verify that assembly code or low-level implementations operate safely in allocated memory regions for all possible executions, providing reliable guarantees of memory safety.

The timing behavior of inference affects system responsiveness, especially in real-time applications that require guaranteed inference latency. In some model implementations, data-dependent execution paths can introduce timing variations.

Hardware-specific optimizations, such as the use of SIMD instructions, custom accelerators or specialized memory layouts. These make verification difficult because they introduce architecture-dependent behaviors [160]. Different processor families exhibit unique instruction sets and vector widths may vary statically or dynamically between platforms. Symbolic execution must accurately model these hardware characteristics to be valid.

6.4.2. Neural-Symbolic Co-Analysis for Autonomous Systems

Autonomous systems, such as self-driving cars and drones, combine well-studied components such as neural networks with traditional control software, creating verification challenges that span both paradigms [161]. These hybrid systems require new analysis techniques that can address the interplay between neural perception and symbolic control.

In autonomous vehicles, the perception components perform tasks such as object detection, lane tracking and sign recognition using deep neural networks that process sensor inputs and produce outputs for the control system. These networks are difficult to verify with traditional methods due to their complexity and nonlinear behavior. Symbolic abstraction techniques can over-approximate the behavior of the network, creating simplified representations that allow formal reasoning about the range of possible outputs for given ranges of inputs. This approach allows for verification even when precise analysis is impractical.

Control software takes the perceptual output and generates commands for actuators such as brakes and steering. While traditional control code is subject to symbolic execution, its behavior depends on the perceptual output, which may be imprecise. Neuro-symbolic analysis combines the symbolic execution of control code with abstract representations of perceptual uncertainty, which identifies scenarios where perceptual errors can lead to dangerous control decisions.

Comprehensive verification of autonomous systems requires verification of properties that span both perception and control. For example, a safety feature might require that if an obstacle appears in the vehicle's path, the braking system be activated at a safe distance. Verification of such features requires consideration of perception accuracy, control latency and vehicle dynamics.

Simulation-based testing, while widely used for autonomous systems, cannot exhaustively explore the entire input space. Symbolic execution complements simulation by examining corner cases: unusual lighting conditions, rare object configurations or sensor failures that can lead to unsafe behavior. Simulators such as CARLA [162] and toolkits such as VerifAI, which supports the formal design and analysis of AI-based systems through scenario-based simulation [161], can be combined with symbolic analysis of control code to explore such corner cases.

Autonomous systems' operational monitoring uses proven features to verify system behavior during operation. Monitors derived from symbolic analysis detect deviations from expected behavior, activating safe fallback modes when necessary. Research on neuro-symbolic frameworks aims to improve interpretation and testing by moving from

black-box machine learning to whitebox, human-readable reasoning, which increases the efficiency of operational monitoring.

Certification of autonomous systems remains an open challenge, but neuro-symbolic analysis provides the basis for warranty cases. By setting limits on perceptual errors, checking the correctness of control software and analyzing their interactions, this technique helps create the evidence needed to certify safety-critical autonomous functions [163]. The neuro-symbolic paradigm replaces stochastic layers with interpretable symbolic AI, which promotes determinism and supports the rigorous verification required for certification.

7. Evaluation and Performance Insights

Rigorous assessment techniques that quantify both advancements over conventional methods and the expenses associated with learning components are necessary for evaluating the efficacy of AI-guided symbolic execution. This section examines the evaluation benchmark suites, summarizes quantitative findings from the literature on the benefits of integrating AI and examines the trade-offs and overheads that influence real-world deployment choices.

7.1. Benchmark Suites

Standardized benchmarks offer a standard framework for assessing AI-guided methods and allow for the fair comparison of various symbolic execution strategies [164]. The research community has created a number of benchmark suites that focus on various facets of symbolic execution efficiency, ranging from embedded-specific issues to general program analysis.

7.1.1. General-Purpose Benchmarks: SV-COMP and DARPA Cyber Grand Challenge

The International Competition on Software Verification or SV-COMP, serves as a benchmark for the yearly evaluation of software verification tools, including symbolic execution techniques. Hundreds of C programs with specific correctness properties, from straightforward reachability tests to intricate memory safety and termination checks, are included in the competition. SV-COMP benchmarks are widely used by researchers and developers to track advancements over time and compare symbolic execution tools to other verification methods.

The competition divides verification assignments into several categories. The ReachSafety category includes programs where the verification task is to determine whether particular program places are reachable. These benchmarks assess the capacity to resolve restrictions and explore paths on typical control flow patterns [164]. The MemSafety category focuses on memory safety features including buffer overflows, memory leaks and incorrect pointer dereferences. These qualities are especially important for embedded software written in dangerous languages like C. The NoOverflows category looks for integer overflows, which can occur when arithmetic operations on fixed-width integers unexpectedly wrap [165]. This is a common vulnerability in embedded systems.

Although SV-COMP is widely used, we did not find studies in our corpus that evaluate AI-guided symbolic executors on its benchmarks; the learned search strategies discussed in Section 4 were evaluated on other sets of open-source programs.

The DARPA Cyber Grand Challenge generated a set of binaries primarily for testing automated vulnerability discovery methods. These challenge binaries contain known vulnerabilities that range from straightforward buffer overflows to intricate logic errors demanding sophisticated symbolic reasoning [138]. For assessing symbolic execution on security-critical tasks, the corpus continues to be a common benchmark. Every binary has

at least one memory corruption security issue that is activated by network input and they all function as network services that carry out practical tasks via innovative protocols.

The Cyber Grand Challenge binaries assess a tool's ability to manage different architectures, obfuscated code and practical vulnerability patterns. For example, the S2E symbolic execution platform, which was used to automatically detect vulnerabilities and generate proofs of vulnerability, was an essential component of CodeJitsu's Cyber Reasoning System. On these binaries, the hybrid approach of Driller, which combines fuzzing with selective symbolic execution, found more vulnerabilities than fuzzing alone [49].

7.1.2. Embedded-Specific Benchmarks

The hardware interface, real-time constraints and resource limitations typical of embedded software are frequently absent from general-purpose benchmarks. This gap is filled by specialized embedded benchmarks, which offer practical firmware targets for assessment.

Frequently used embedded benchmarks are FreeRTOS apps. A popular real-time microcontroller operating system, FreeRTOS offers a stable platform with clearly defined tasking and communication primitives. FreeRTOS applications, ranging from task schedulers to network protocol implementations, can serve as evaluation targets [166], although we found no standardized symbolic-execution benchmark suite built on FreeRTOS. These benchmarks test embedded software's concurrency and timing features, which put conventional symbolic execution to the test.

Zephyr RTOS benchmarks provide an additional platform for embedded examination. Zephyr, a Linux Foundation project aimed at devices with limited resources, contains sensor drivers, Bluetooth stack implementations and power management code that mimics actual IoT firmware [167].

TizenRT benchmarks target embedded systems running a lightweight RTOS derived from the Tizen platform [168]. These benchmarks include multimedia processing, networking stacks and application frameworks more complex than typical microcontroller firmware, providing a bridge between deeply embedded and general-purpose software.

Capture-the-flag style competitions such as the Embedded Security Challenge provide vulnerable firmware targets, but they are not designed as standardized benchmarks for symbolic execution tools.

7.1.3. Custom Industrial Case Studies

The true test of any verification technique is its performance on genuine industrial software. We did not, however, find peer-reviewed industrial case studies of AI-guided symbolic execution; the evidence discussed in this review comes from academic prototypes evaluated on open-source programs and benchmarks. Industrial case studies, including automotive, aerospace, medical and industrial-control software, are therefore an important gap for future work (Section 8).

7.2. Measured Gains from AI Integration

The quantitative evaluation of AI-guided symbolic execution is focused on three key metrics: path explosion reduction, constraint solving speedup and bug detection rate enhancement. The reported gains depend strongly on the technique, benchmark, baseline and time budget, so the figures below are not directly comparable across studies, and very few of them were obtained on embedded software.

7.2.1. Reduction in Path Explosion

AI-guided path selection addresses the fundamental scalability problem of path explosion, which is the exponential rise in symbolic states with program complexity. Evaluations quantify the decrease in states investigated while preserving or enhancing coverage.

Reported reductions are specific to each study's setting. In a target-directed setting, Q-learning guidance avoided more than 90% of the paths and instructions explored by the default strategy of KLEE in an initial evaluation [31], and automatically generated search heuristics outperformed manually crafted ones in branch coverage and bug finding on open-source C programs [28]. None of these evaluations used embedded benchmarks.

Function summaries reduce redundant analysis of repeated calls [54]; as noted in Section 4.4.2, we found no evaluations of learned summaries.

7.2.2. Speedup in Constraint Solving

Constraint solving dominates symbolic execution time and typically consumes a large portion of the total execution time [40]. AI-guided approaches accelerate solution through neural approximation, learning caching and more intelligent solver interaction.

Learned solver strategies offer the best-documented speedups. FastSMT reported solving 17% more formulas and runtimes up to 100 times faster than the default Z3 strategy on benchmarks in the QF_NRA, QF_BV and QF_NIA theories [74], and MachSMT reported up to a 198.4% improvement in PAR-2 score over SMT-COMP winners in some divisions [76]. These results were obtained on solver benchmarks, not on constraints collected from embedded software.

Constraint reuse reduces solver calls when constraints recur [56], and DeepSolver reported that its learned classifier classified path conditions more efficiently than conventional solving and existing reuse frameworks [73]; the latter result has not yet been peer reviewed.

Learned branching heuristics inside SMT solvers have been proposed, for example reinforcement-learning-guided solving [94], but we found no evaluation of their effect on constraints produced by symbolic execution.

7.2.3. Improvement in Bug Detection Rates

The ultimate measure of symbolic execution's effectiveness is its ability to find defects. Learned search heuristics have improved bug finding on open-source C programs: automatically generated heuristics outperformed manually crafted ones in both branch coverage and bug finding [28], and adaptively changing heuristics improved both measures further [70]. We found no evaluation of RL-guided exploration on embedded benchmarks with known vulnerabilities.

Graph neural networks detect vulnerable functions more accurately than earlier learning-based detectors (Devign reported an average of 10.51% higher accuracy [71]), but they have not been compared directly with symbolic execution for bug finding in the studies we reviewed.

LLM-assisted environment modeling could make previously unanalyzable firmware code reachable, but we found no evaluation that quantifies the resulting increase in reachable paths or discovered vulnerabilities.

7.3. Overheads and Trade-Offs

AI guidance does not come free. It is necessary to balance the improvements made against the training needs, runtime costs and precision considerations. Determining whether and how to use AI-guided symbolic execution requires an understanding of these overheads.

7.3.1. Training Time and Data Requirements

Gains on minor or one-time analytical tasks may be outweighed by the training data and calculation time required for learning-based advice. The price varies greatly depending on the method. To provide good policies, reinforcement learning for path selection usually has to be trained on a large corpus of programs. Depending on the intricacy of the program and the computational resources available, training time might vary from hours to days. Once trained, a model can in principle be reused on other programs, spreading the cost of training over several analyses; Learch, for example, is trained offline and then applied to new programs [29]. This amortization can make RL feasible for companies examining several connected initiatives.

To learn predictive models, graph neural networks need labeled training data or algorithms with known exploration results. It is necessary to perform symbolic execution on a representative corpus in order to generate this training data, which is a computationally costly procedure [29]. However, current work on foundation models lessens the requirement for task-specific training and pretrained models may be used across businesses.

Online learning methods adjust during a single analysis run, completely avoiding pre-training. Although these techniques require less data, they could not perform as well as offline-trained models. Despite perhaps smaller benefits, online learning offers a useful substitute for one-time analytical jobs where pre-training data is not accessible.

LLM-based methods have inference costs per query but do not require training. If models are queried frequently, LLM inference latency may dominate analysis time. Although this expense may be reduced by caching LLM answers and batching requests, LLM overhead is still substantial for applications that need a large number of models, such as environment modeling for many library functions.

7.3.2. Runtime Performance Versus Precision

Runtime overhead from feature extraction, model inference, and learning updates is introduced by AI guidance. The benefits of more targeted investigation must be weighed against this expense. In order to calculate state values, reinforcement learning strategies need to extract features from each symbolic state and run a forward pass through a neural network. When examining many states, the per-state cost of feature extraction and inference can become substantial; whether the reduction in explored states outweighs this overhead depends on the program and the time budget, and the studies we reviewed rarely report the overhead separately.

At exploration decision points, graph neural network inference on code property graphs introduces overhead. Preprocessing is necessary for building graphs and calculating node embeddings, which can be time-consuming for bigger programs. This preprocessing expense may dominate runtime for shorter analyses, but it is spread out over numerous exploration choices for longer analyses.

Updates from online learning increase the computing cost of analysis. Every update takes up time that could be used for exploration because it necessitates digesting recent experience and modifying model parameters. It is necessary to adjust the trade-off between learning frequency and exploration progress according to program features and analysis time.

When accurate reasoning is replaced by approximations, precision trade-offs occur. False positives in bug reports might result from neural constraint approximation inaccurately classifying some constraints as satisfiable when they are not. This trade-off could be beneficial for security analysis, where false positives are acceptable provided they are examined. Approximation must be used in conjunction with precise testing for verification, when soundness is necessary to uphold promises.

7.3.3. Generalization Across Different Embedded Platforms

AI-guided symbolic execution has significant challenges when switching between many embedded platforms since these platforms have quite different software ecosystems, peripherals and architectures [11]. Applying a guiding guideline that was learned on one platform to another may be challenging for tool developers and users.

Learned path selection strategies are fundamentally impacted by changes in architecture. Due to variations in instruction semantics, calling conventions and compiler optimizations, policies learned on one set of instructions may perform poorly on another. Generalization is not certain, even if transfer learning approaches can assist in adapting rules across architectures with little further training. In order to improve the portability of AI-powered path selectors across various program representations, tools such as PySymGym are actively investigating language-independent, graph-based data representations [27].

Even if two device families share a microcontroller core, hardware peripheral models learnt from one may not apply to the other. Peripheral register layouts, interrupt behavior and timing characteristics vary significantly across manufacturers and even across product lines from the same manufacturer. Recent research has turned to large language models to address this challenge. LLM-generated models based on documentation can adapt to new devices, but they require device-specific prompts and validation. LLMs have been used to generate models of unknown functions automatically [30], although the reduction in manual effort has not been independently quantified.

Platform-specific variations in real-time operating system APIs have an influence on environment modeling. Models for the queue operations of one RTOS are not instantly applicable to the message queues of another RTOS, even when their functionality is similar. While there may be some transfer because POSIX-like APIs are comparable, learnt API models are likely to require separate training or modeling for each RTOS; we found no evaluated framework that automates this.

Compiler and optimization modifications have a significant influence on binary-level analysis. Code produced with different optimization settings or several compilers, which show different control flow and data layouts, might confound learned guidance. According to studies dating back to 2015, standard compiler optimizations must be implemented judiciously during symbolic execution. Implementing optimizations in a planned order may potentially cause slowdowns as compared to no optimization, depending on the search algorithms and constraint caching techniques used, according to research using KLEE on LLVM optimization flags [51]. While training on several compiler settings increases robustness, application to unexpected circumstances is not assured.

Practical applicability is impacted by the wide variations in tool support for embedded platforms. Only LLVM-based analysis is supported by some AI-guided symbolic executors, necessitating the availability of source code. Others, such as angr, offer binary analysis for several architectures but lack models of platform-specific characteristics [34]. Certain architectures are still more supported than others and platform coverage is still inconsistent. Approaches that perform concrete execution on the physical device and symbolic execution on a host, as in Avatar [50], help to overcome resource constraints. However, tool developers continue to pay less attention to more complex designs.

These overheads and trade-offs affect practical decisions on the use of AI-guided symbolic execution. Large-scale, repetitive analytical tasks can benefit from AI guidance when training expenses can be recouped and approximation is made possible by accuracy criteria. When soundness is important and training data is unavailable, traditional approaches may still be more appropriate for one-time assessments of different platforms. By understanding these trade-offs, practitioners may select the optimal solution for their

particular verification challenges by balancing the benefits of automation against platform coverage limitations.

8. Open Challenges and Future Directions

The integration of AI with symbolic execution for embedded systems is still a developing topic with many unresolved issues, despite the notable advancements mentioned in earlier sections. These issues cover a wide range of technological, methodological and social aspects, from practical worries about implementation in industrial workflows to basic queries about how to foster confidence in AI-guided analysis. This section highlights important unresolved issues and suggests possible avenues for further investigation.

Rather than listing research topics in the abstract, Table 6 derives the research agenda from the evidence gaps identified in Sections 4–7: each row links a finding of this review to the open problem it leaves and to the subsection in which the corresponding direction is discussed.

Table 6. Research agenda derived from the findings of this review.

Findings of this Review	Key Evidence	Evidence Type	Open Gap	Research Direction	Section
Learned path selection is the best-supported technique	[28,29,31,68–70]	Peer-reviewed; general-purpose programs	No evaluation on firmware or RTOS code	Embedded benchmarks; transfer across architectures	Sections 7.1.2 and 8.3
Learned solver strategies and solver selection give large gains	[74–76]	Peer-reviewed; SMT benchmarks	Not measured on constraints from embedded code	Evaluation on constraint corpora produced by symbolic execution	Section 8.2
Neural satisfiability prediction is unsound and largely unreviewed	[73,93]	Preprints	No soundness guarantee; unclear certification path	Exact-solver confirmation and tool-qualification evidence	Section 8.1
Learned state merging, function summarization and constraint approximation are unevaluated	None found	Research gap	Techniques proposed but not evaluated	Learned merge and summary models with soundness checks	Sections 4.4 and 8.2.2
LLM-based environment modeling is promising but little quantified	[30,78,86]	Mixed; several preprints	Validation, nondeterminism and effort savings unmeasured	Validated LLM-generated models with measured manual effort	Sections 8.1 and 8.3.2
No peer-reviewed industrial case studies	None found	Research gap	No industrial evidence for AI-guided symbolic execution	Industrial evaluations producing certification evidence	Sections 8.1.2 and 8.4.2
Online adaptation exists within a campaign, not across runs	[49,70,83]	Peer-reviewed	No cross-run learning; learned fuzzing switch unevaluated	Continuous learning integrated into CI/CD pipelines	Section 8.4.1

8.1. Explainability and Trust

Clarification and trust issues become crucial as AI components play more and more important roles in directing symbolic execution. To be confident in the outcomes of an AI-

guided tool, developers, certifiers and regulators need to know why it makes certain exploration decisions.

8.1.1. Interpreting AI-Guided Symbolic Decisions

Modern symbolic executors are often guided by neural networks and reinforcement learning policies that function as black boxes, making choices based on intricate patterns discovered through training data. Human analysts may not be able to understand the reasoning behind such a system's decisions to merge specific states or prioritize some paths over others. For practical adoption, this opacity poses a number of challenges.

When the basis for decisions is unknown, debugging unexpected tool behavior becomes challenging. Understanding why an AI-guided symbolic executor misses a known vulnerability or spends too much time on ineffective paths necessitates analyzing the internal logic of the learned policy. Saliency maps, feature attribution and counterfactual explanations are examples of explainability techniques for neural networks that have been developed mainly for computer vision and natural language processing, not for program analysis decisions. It is still up for debate how to apply these methods to the field of symbolic execution, where choices are made using intricate program state representations.

For high-assurance applications, the guidance system itself must be verified. The accuracy of the AI guidance determines whether a symbolic executor can claim to have reached a specific coverage or to have discovered every bug of a specific kind. Neural network policy formal verification is a developing field with methods for demonstrating small network properties but limited scalability to program analysis sizes. A way forward is provided by neuro-symbolic approaches that integrate learned components with verifiable symbolic reasoning, but more work is needed to integrate them into useful tools [33].

Models of human-AI collaboration could offer intermediate answers. Instead of treating AI guidance as independent, interactive systems may explain recommended judgments to human analysts, who would then approve or modify them. For this form of collaboration, mechanisms for incorporating human input into learning policies and interfaces that effectively communicate complex symbolic execution notions would be required. Research on explainable AI for program analysis is still in its early stages and much more work needs to be done to identify the most useful explanations and how to generate them.

8.1.2. Certification in Safety-Critical Domains

Examples of safety-critical businesses that are subject to regulatory frameworks requiring documented proof of verification efficacy are the automotive, aircraft and medical device sectors. AI-guided symbolic execution must adhere to specific protocols and evidence in order to be accepted as part of certification, such as DO-178C for airborne software and ISO 26262 for automotive systems [21,22].

Qualifying AI components as verification tools presents new challenges. AI-guided tools provide novel failure scenarios that are not addressed by present guidelines, despite the fact that certification authorities have decades of experience with formal methodologies, testing and static analysis. Conventional tool qualification measures do not address robustness issues since learnt policies may exhibit unexpected behavior when inputs outside of their training distribution are applied. Research is desperately needed to develop qualification frameworks that solve these problems and foster creativity.

To produce evidence for certification, more than only tool outputs are required. Certification dossiers must contain arguments showing the tool was used correctly, the results are authentic and the verification method is appropriate for the application. For

AI-guided symbolic execution, this means documenting training data attributes, policy performance across relevant program types and actions taken to avoid overfitting or distribution shift. As standards bodies begin to consider these issues, working groups are looking into how existing frameworks might accommodate AI-enhanced development and verification tools.

Certification is more challenging for systems with mixed criticality, where components of different safety standards coexist. Because of interference or shared resources, AI advice given to lower-criticality components may affect analysis of higher-criticality components. Certification must take these relationships into account, which may require alternative analysis settings or guarantees that AI guiding decisions for lower-criticality code do not affect higher-criticality verification.

Developing certification-aware training techniques that optimize for attributes that certification authorities care about, like coverage of particular code structures or the absence of specific problem types, instead of generic coverage metrics is one of the future research priorities. It would be possible to evaluate tools for these areas meaningfully if benchmark suites certified to represent safety-critical software characteristics were created. In order to achieve regulatory acceptability, it will be crucial to interact with certification authorities early in the tool development process rather than after the fact.

8.2. Real-Time and Resource-Aware Adaptation

Both during analysis and in the deployed systems they examine, symbolic execution tools must adhere to the stringent resource limits that embedded systems work under. Lightweight models and an understanding of analysis costs are necessary to adapt AI guidance to these limitations.

8.2.1. Lightweight ML Models for On-Device Guidance

The majority of current AI-guided symbolic executors operate on development workstations with plenty of processing power. However, other applications, such as runtime monitoring, just-in-time verification or field security checks, need analysis on the embedded devices themselves. These situations require machine learning models that are compact enough to run on technology comparable to microcontrollers.

Quantization, pruning and knowledge distillation are model compression approaches designed for embedded machine learning that can significantly reduce neural network sizes while maintaining accuracy. It is necessary to comprehend which model components are crucial for decision quality and how compression influences exploration results in order to apply these methods to guidance models for symbolic execution.

New application scenarios are made possible by the deployment of TinyML guiding models on embedded devices. In principle, a symbolic executor running on the device could analyze code as it runs and flag deviations from intended behavior; we found no implementation of this idea. Models operating within stringent latency budgets and memory footprints would be necessary for such runtime verification; this would be difficult but possibly possible with specialized architectures.

Co-designing hardware and software for on-device guiding may result in specific accelerators for symbolic execution choices. Lightweight hardware for path prediction and constraint approximation could speed up on-device analysis in the same way as neural processing units speed up inference for vision tasks. We found no published accelerators of this kind.

8.2.2. Balancing Analysis Depth with System Constraints

By its very nature, symbolic execution compromises resource consumption for depth of analysis. In order to allocate limited time and memory to the most worthwhile

investigation while adhering to system limits, AI guidance must make these decisions intelligently.

Budget-aware exploration chooses routes that maximize advantage within the constraints of available resources after learning to forecast the expense of investigating various routes and the anticipated reward. Such considerations are naturally included in reinforcement learning formulations through reward functions that encourage coverage or bug discovery while penalizing resource use. More complex cost models that take into consideration memory utilization, solver time and other embedded analysis-specific limitations must be developed in future study.

For time-constrained settings, anytime algorithms that yield valuable results even when interrupted before completion are crucial. Guidance policies that give priority to early discoveries are necessary for symbolic executors that can return partial results after analysis time expires, such as vulnerabilities discovered thus far or coverage attained. Learning rules that optimize for anytime performance require different training objectives and evaluation metrics than those that aim for final coverage.

When memory or time constraints are reached, graceful degradation under resource demand guarantees that the analysis deteriorates in predictable ways. Resource-aware symbolic executors may switch to lighter-weight analysis modes, cache results for future resumption or provide conservative estimates of remaining exploration potential instead of crashing or providing useless results. For AI-guided systems, designing such graceful degradation methods necessitates careful consideration of what information to save and how to recover when resources become available again.

8.3. Cross-Domain Generalization

Applicability across the broad ecosystem of embedded systems is limited because AI guidance trained on one set of programs or platforms frequently works badly on others. For practical implementation, generalization must be improved.

8.3.1. Transfer Learning Across Firmware Architectures

A bewildering range of processor architectures are used in embedded systems, including many proprietary cores, RISC-V for open hardware, ARM Cortex-M for low-power devices and different DSPs for signal processing. Due to variations in instruction sets, memory models and compiler protocols, guidance policies learned on one architecture might not apply to another [169].

By aligning feature representations across architectures, domain adaptation techniques improve the performance of policies learnt on source domains on targets. For binary analysis tasks, where the same vulnerability patterns appear differently across instruction sets, representation learning that aligns code across architectures has shown promise [170]. It is necessary to create architecture-agnostic program representations that abstract away syntactic differences while maintaining semantic information in order to extend these methods to symbolic execution guidance.

By sharing representations while preserving architecture-specific output layers, multi-task learning trains policies that function effectively across several architectures concurrently. This method necessitates training data from a variety of architectures, which may be hard to get by on some systems. Limited real data could be supplemented by synthetic data generation via compiler variants and instruction set emulation, although maintaining fidelity is still difficult.

Another route to cross-architecture learning is provided by progressive neural networks, which increase capacity for new architectures while retaining learnt information from earlier ones. These architectures enable transfer without catastrophic forgetting by adding new columns for each target while freezing previously learnt

columns [171]. Although comparatively unexplored, applications to symbolic execution guiding show promise.

8.3.2. Few-Shot Learning for Rare Hardware Platforms

Large training corpora for AI guiding are impractical since many embedded systems are created in very small quantities or for specific applications. For these situations, few-shot learning methods that generalize from a limited number of examples are crucial.

By optimizing for speedy adaptation, meta-learning techniques teach models to learn from sparse input. An initialization that can be refined with a few instances on new platforms is learned by a meta-learner who is exposed to numerous learning tasks, such as the examination of various programs. Few-shot and meta-learning techniques are well established in machine learning [172], but we found no evaluations of them for guiding symbolic execution on new hardware architectures.

Limited genuine data can be supplemented via synthetic program development. Although they are not entirely representative of real applications, compiler fuzzing approaches that build a variety of programs for target architectures generate training data that exercises compiler behaviors and language features [173]. Compared to guidance rules learned on restricted real data alone, those trained on a combination of real and synthetic data may be more general.

To optimize model improvement with the least amount of human labor, active learning chooses which samples to label. Active learning could concentrate labeling on the most instructive programs for uncommon platforms where acquiring labeled analysis data is costly, requiring manual verification of exploration outcomes. This method necessitates the creation of acquisition functions that calculate the degree to which a new example will enhance guidance quality. It is a difficult but manageable line of inquiry.

8.4. Integration into Development Lifecycles

The practical limitations of industrial software engineering must be met and AI-guided symbolic execution must seamlessly fit into current development workflows in order to be widely adopted.

8.4.1. CI/CD Pipelines for Embedded Software

With each code change, software development, testing and analysis are automated by continuous integration and continuous deployment pipelines. These pipelines' latency, reliability and usability requirements must be met in order to incorporate AI-guided symbolic execution.

To save repetitive effort, incremental analysis techniques make advantage of the outcomes from earlier pipeline runs. Only the impacted functions and their dependencies require reanalysis when code is modified. In order to prioritize the exploration of modified code while maintaining information about unaltered parts, learned guidance must adjust to incremental settings. Since learning components usually anticipate starting from zero, creating incremental iterations of AI-guided algorithms is an open challenge.

Regression test selection reduces pipeline delay by determining which tests to run based on code changes. Analogous methods would choose which analysis configurations, such as guidance policies, depth bounds and solver settings, to apply to modified code for symbolic execution. Pipeline efficiency could be significantly increased by learning which analysis setups are more likely to detect defects in modified code.

For developers to be accepted, false positive management is essential. AI guidance may bring more false positives due to approximation mistakes and symbolic execution can identify more possible problems that are not truly flaws. To prevent overburdening developers, tools must include efficient triage processes, such as automatically classifying

reported issues by likelihood and severity. A possible approach is to use previous data to learn how to differentiate between real issues and false positives.

8.4.2. Regulatory Acceptance and Standards Evolution

For safety-critical areas, regulatory acceptance is crucial, as covered in Section 8.1.2. The development of standards to support AI-enhanced verification is required in addition to the qualification of specific tools.

Frameworks for tool qualification must change to accommodate AI components. Deterministic tools with well-understood failure modes are assumed by current frameworks such as ISO 26262's tool confidence level evaluation. These frameworks do not handle the probabilistic behavior and distributional changes brought about by AI guidance [21,22]. It is a multi-stakeholder challenge that calls for cooperation between tool developers, users and certifiers in order to develop new qualification methodologies that take these characteristics into account while retaining safety assurance.

AI in safety-critical systems is starting to be addressed by standards groups. Efforts to address AI in aviation are starting to appear, as are documents addressing functional safety and AI systems. The acceptance of AI-guided verification tools in regulated industries will be influenced by these developing standards. Participating in these standardization initiatives guarantees that future regulatory requirements will be met by research directions.

In addition to tool outputs, evidence packages for certification must record the creation and verification of AI components. This entails recording training data attributes, performance across representative benchmarks, overfitting prevention strategies and validation on safety-relevant aspects for learnt guidance policies. Creating uniform forms for this kind of documentation would make certification across various technologies and projects easier.

8.5. Ethical and Security of AI Components

The security and ethical implications of AI components are becoming increasingly important as they are incorporated into verification systems. The completeness and dependability of analysis may be impacted by these elements' potential introduction of additional biases and vulnerabilities.

8.5.1. Adversarial Attacks on Guiding AI Models

Adversarial attacks, carefully constructed inputs that result in incorrect classification or bad decisions, may pose a threat to the neural networks that direct symbolic execution. Such attacks could have major repercussions in the context of symbolic execution guidance.

Programs deliberately created to trick guidance models into avoiding vulnerable paths could be created by an adversary controlling the code under analysis [174]. An attacker could include program elements that a reinforcement learning strategy has learned to emphasize into malicious code in order to avoid detection. For security-critical systems, it is crucial to comprehend these weaknesses and create reliable training techniques [175].

8.5.2. Training-Data Bias and Uneven Exploration Coverage

Machine learning models pick up biases from their training data. Biased training data may result in systematic under-verification of specific code types for symbolic execution guidance, raising questions about completeness and fairness.

When certain program types are overrepresented and others are underrepresented in training corpora, representation bias takes place. Guidance models may function poorly

on embedded firmware if training data primarily consists of desktop programs, which would result in less complete embedded system verification. Programs from all pertinent domains must be systematically gathered in order to ensure diverse and representative training data.

Label bias occurs when human annotators' biases are reflected in the ground truth utilized for training, such as which functions contain defects and which paths are worth investigating. Models may learn to prioritize some vulnerability categories while ignoring others if historical bug data overrepresents those types. It needs interdisciplinary cooperation with subject matter experts to develop objective labeling procedures and comprehend how label bias impacts guidance quality.

If guidance models learn to concentrate on code patterns that are similar to those that are previously known to have flaws, they may ignore new vulnerability classes and develop confirmation bias. In reinforcement learning, exploration-exploitation tradeoffs must strike a balance between exploiting well-known patterns and investigating potentially new ones [176]. It is a constant effort to design incentive systems that promote diversity in vulnerabilities found rather than merely coverage of known patterns.

Such bias has a concrete mechanism in symbolic execution: a learned search heuristic trained mostly on desktop C utilities encodes their branch and loop patterns, and on firmware it may systematically deprioritize paths through interrupt handlers or code that accesses memory-mapped peripherals. The effect can be measured by comparing coverage per code category (for example, interrupt handlers, peripheral drivers and application logic) between learned and conventional search on the same target, a comparison that none of the reviewed studies reports.

9. Practical Implications for Embedded Systems

AI combined with symbolic execution has significant implications for embedded software development, testing, and certification. It is critical to comprehend these consequences for both researchers developing practical applications of these technologies and practitioners considering their use.

Firmware testing and security vulnerability detection. AI-guided symbolic execution has been particularly helpful in firmware security analysis. Symbolic execution has identified defects in firmware [141], and learnt path selection and LLM-based environment modeling are the AI approaches most likely to expand this to bigger firmware, but their usefulness on embedded targets has yet to be proved. The amount of human effort generally required for firmware analysis can be decreased by concentrating the study on code parts that interface with hardware and automatically constructing models for peripheral behavior. This makes it possible to do more thorough security evaluations faster.

IoT Security in Environments with Limited Resources. AI guidance provides a means for practical verification in IoT devices constrained by limited memory and power, where conventional symbolic execution would deplete available resources. Lightweight learned models that focus on exploration based on the unique features of a device can work within the resource limits that are common in embedded development workflows. Nonetheless, the implementation of AI components directly on devices continues to pose a significant challenge, necessitating further investigation into model compression and efficient inference.

Incorporation into DevSecOps workflows. There is a demand for verification tools that are compatible with continuous integration and continuous deployment pipelines as a result of the shift-left movement in software development. AI-guided symbolic execution tools must function well in settings where dependability, speed and usability are critical. Learned switching strategies that adapt fuzzing and symbolic execution to

observed progress might offer the responsiveness needed for short feedback cycles, but they have not yet been examined. But the fact that AI components are not always predictable and that model training takes time makes it hard to get the reproducibility that automated pipelines need.

Limits on Certification in Safety-Critical Areas. One of the most significant challenges to adoption is that safety-critical applications are governed by standards such as DO-178C for aviation and ISO 26262 for automotive systems. According to these requirements, tools must be evaluated for known failure mechanisms, determinism, and predictability. These requirements are not fully satisfied by the AI-assisted symbolic execution techniques available today. Safety-critical domains will only be able to apply AI in lower-integrity applications or in supporting roles where AI guidance supplements rather than replaces conventional verification methods until certification frameworks adapt to incorporate AI-enhanced technologies.

10. Conclusions

Program analysis for embedded and cyber-physical systems has advanced significantly with the combination of artificial intelligence and symbolic execution. The panorama of AI-guided symbolic execution approaches, their applications, assessments, and the unresolved issues that shape the future research agenda have all been methodically addressed in this survey. This synthesis yields several important conclusions.

AI does not replace symbolic execution; it complements it, and the evidence of enhancement is now strongest for taught path selection and learned solver techniques. The essential strength of symbolic execution is its capacity to rigorously reason about program behavior through SMT solutions and route restrictions, which AI techniques cannot match with the same assurances of soundness. AI functions as a potent augmentative technology that learns to make better choices about exploration, approximation, and resource allocation rather than taking the place of these fundamental systems. The clearest evidence involves learned path selection and learned solver techniques; LLM-based environment modeling is intriguing but poorly appraised, while neural constraint approximation, learned state merging and learned function summarization remain mostly research topics. In each instance, AI maintains its fundamental capacity for reasoning while enhancing the efficacy of symbolic execution. The outcome is a synergy rather than a replacement: neuro-symbolic systems that integrate the logical accuracy of symbolic analysis with the pattern recognition of learning.

For embedded systems, hybrid AI-symbolic techniques show the greatest promise. Hardware interface, real-time limits, resource limitations, and safety-critical requirements are among the distinctive features of embedded software that necessitate analysis techniques that can manage complexity without sacrificing reliability. Pure symbolic execution has scalability issues, and pure AI methods lack the formal assurances required for certification. The best of both worlds is provided by hybrid techniques that closely combine symbolic reasoning and learning. Neural guidance concentrates analysis on paths that are most likely to have faults or violate characteristics; symbolic execution fully checks those paths. For complex arithmetic, learned approximations speed up constraint solving; precise solvers verify outcomes where soundness is important. Hardware-dependent code can be analyzed using models created by LLM; symbolic execution verifies these models against actual behavior. Whether such hybrid approaches surpass either paradigm alone on embedded software has not yet been demonstrated; evaluating them on realistic firmware is the most pressing gap identified by this review.

Therefore, the utility of these approaches is yet promising. Firmware security, smart contract analysis, and safety-critical verification have all benefited from symbolic

execution; however, AI-guided variants have primarily been assessed on general-purpose benchmarks, and we were unable to locate any peer-reviewed industrial case studies of AI-guided symbolic execution in automotive, aerospace, or medical settings.

But there are still many obstacles to overcome. Before AI-guided technologies are widely used in regulated industries, explainability and trust must be addressed. Frameworks for qualifying AI components under safety criteria are desperately needed, and developers and certifiers must comprehend why tools make specific exploratory decisions. Lightweight models that can operate on embedded devices and algorithms that strike a compromise between analytic depth and system limitations are necessary for real-time and resource-aware adaptation. In order to make AI guidance useful across the wide range of embedded platforms, cross-domain generalization necessitates transfer learning and few-shot approaches. Tools that are compatible with CI/CD pipelines, efficiently handle false positives, and satisfy regulatory requirements are necessary for integration into development lifecycles. Researchers and practitioners must continue to focus on ethical and security issues related to AI components themselves, such as adversarial assaults, bias in training data, and fairness in verification conclusions. Section 8, which provides a thorough research plan for the discipline, delves further into these issues.

Adaptive, self-improving verification technologies are a long-term goal. Online adaptation within a single campaign has been demonstrated [70], but symbolic executors that learn across analysis runs and foundation models trained specifically for symbolic execution have not yet been reported.

In response, the scientific community has to work together. For equitable comparison and progress monitoring, open benchmarks that reflect realistic embedded software are crucial. Although useful, the hardware interaction, parallelism, and real-time features of commercial embedded systems covered in Section 2.4 are not adequately captured by current benchmarks. It takes ongoing financing and community effort to create and maintain such benchmarks. In addition to the benchmark survey in Section 7.1, standardized evaluation methodologies that take into consideration the special characteristics of embedded analysis—coverage of hardware-dependent code, detection of embedded-specific vulnerabilities, and performance under resource constraints—would enable meaningful comparison across approaches. To overcome the complex issues listed in this study, interdisciplinary cooperation between program analysis researchers, machine learning experts, embedded systems engineers, and certification authorities is required.

The ultimate goal is to develop adaptive, self-improving verification tools for embedded software of the future. Imagine symbolic executors that pick up knowledge from each analysis run and apply it to new platforms and applications. Tools that modify their tactics in real time in response to resources and observed progress. Systems that use hardware traces and documentation to create their own environment models. Verifiers that provide human analysts with an explanation of their choices and take criticism into account to make improvements. In addition to finding more flaws more quickly, these technologies would offer the proof and assurance required for certification in areas where safety is crucial.

By continuing to advance the research directions described in Section 8, this goal can be realized. The groundwork has been established: prototypes illustrate the promise and we know how AI can direct symbolic execution, but evaluations on embedded software are still needed. Whether this foundation can support the lofty objective of fully automated, reliable verification for the increasingly complex embedded systems that underpin modern society will be determined in the upcoming years. The opportunity is as great as the stakes. We can create verification tools that match the complexity of the systems they examine by fusing the rigor of symbolic reasoning with the flexibility of

machine learning. This ensures that the software governing our automobiles, medical equipment, industrial infrastructure, and communication networks operates dependably, securely and safely.

Author Contributions: Conceptualization, M.I.; methodology, M.I. and T.K.; software, M.I. and T.K.; validation M.I., T.K. and A.L.; formal analysis, M.I. and T.K.; investigation, M.I. and T.K.; resources, M.I. and T.K.; data curation, T.K.; writing—original draft preparation, T.K.; writing—review and editing, M.I., T.K. and A.L.; visualization, T.K.; supervision, M.I., T.K. and A.L.; project administration, A.L.; funding acquisition, A.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the European Union under the Horizon Europe programme, grant agreement No. 101244751 (project AISSAM—“Automated Vulnerability Detection in Software Development Using AI Techniques”; call HORIZON-WIDERA-2024-TALENTS-02; type of action: HORIZON TMA MSCA Postdoctoral Fellowships).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

This appendix reports the search log (Table A1), the reports excluded at the full-text stage (Table A2), the studies of the review corpus (Table A3), the quality scores of all reports assessed in full text (Table A4) and the completed PRISMA 2020 checklist (Table A5). Reference numbers refer to the reference list of this article.

Table A1. Search log. In the queries, * is the truncation wildcard of the respective database.

Source	Date	Exact Query	Fields/Filters	Results	Screened	Export
Scopus	30 September 2026	TITLE-ABS-KEY ((“symbolic execution” OR “concolic” OR “dynamic symbolic execution”) AND (“machine learning” OR “deep learning” OR “reinforcement learning” OR “graph neural network*” OR “large language model*” OR “LLM*” OR “neural network*” OR “neuro-symbolic”)) AND PUBYEAR > 2019 AND PUBYEAR < 2026 AND (LIMIT-TO (DOCTYPE, “ar”) OR LIMIT-TO (DOCTYPE, “cp”)) AND (LIMIT-TO (LANGUAGE, “English”))	Title, abstract, keywords; 2020–2025; English; articles and conference papers	184	184	CSV with citation information, abstracts and keywords
		[[Abstract: “symbolic execution”] OR [Abstract: “concolic”]] AND [[Abstract: “machine learning”] OR [Abstract: “deep learning”] OR [Abstract: “reinforcement learning”] OR [Abstract: “graph neural network”] OR [Abstract:				
ACM Digital Library	30 September 2026		Abstract; ACM Full-Text Collection (849,684 records); 1 January 2020–31 December 2025	29	29	Results page copied to Word (export is a premium feature in the Basic Edition)

			“large language model”] OR [Abstract: “neural network”] OR [Abstract: “neuro-symbolic”] AND [E-Publication Date: (1 January 2020 TO 31 December 2025)]			
SpringerLink	30 September 2026	“symbolic execution” OR “concolic”) AND (“machine learning” OR “deep learning” OR “reinforcement learning” OR “graph neural network” OR “large language model” OR “neural network” OR “neuro-symbolic”)	Full text (no abstract-only search available); 2020–2025; English; Conference paper, Article, Research article; sorted by relevance	624	200	CSV (Download results); first 200 by relevance screened
Google Scholar	30 Sep 2026	“symbolic execution” OR “concolic execution” “machine learning” OR “deep learning” OR “reinforcement learning” OR “graph neural network” OR “large language model” OR “neural network”	Full text; 2020–2025; citations and patents excluded; sorted by relevance	About 9650	200	First 10 pages (20 results each) copied to Word
arXiv	30 September 2026	abs: (“symbolic execution” OR concolic) AND abs: (“machine learning” OR “deep learning” OR “reinforcement learning” OR “graph neural network” OR “large language model” OR “neural network” OR “neuro-symbolic”) AND submittedDate: [202001010000 TO 202512312359]	Abstract; submitted 1 January 2020–31 December 2025 (arXiv API query)	66	66	Atom XML from https://export.arxiv.org/api/query (accessed on 30 September 2026),, max_results = 200
IEEE Xplore	30 September 2026	(“All Metadata”: “symbolic execution” OR “All Metadata”: “concolic”) AND (“All Metadata”: “machine learning” OR ... OR “All Metadata”: “neuro- symbolic”) (planned)	Not run: HTTP error 420 on two browsers and two networks (Wi-Fi and mobile data)	Not available		IEEE content covered through Scopus: 60 of the 184 Scopus records are from IEEE or joint IEEE/ACM venues

Table A2. Reports excluded at the full-text stage ($n = 17$).

ID	Study	Year	Reason	Explanation
S069	ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System	2025	F4: Thesis or technical report	ATLANTIS AIXCC final report (151 pages): technical report, not a journal/conference paper; concolic executor is one component, not separately evaluated
S107	Boosting symbolic execution for vulnerability detection	2025	F4: Thesis or technical report	PhD dissertation (SMU ETD collection), not a journal/conference paper; related Cottontail paper S140 is included

S120	Combining Dynamic Symbolic Execution, Machine Learning and Search-Based Testing to Automatically Generate Test Cases for Classes	2020	F5: Quality rule not met (Q1 = 1 and total $\geq$ 3)	Tardis: classifier prioritises path conditions in DSE; evaluated only on the paper's own example program (Q1 = 0) $\rightarrow$ fails quality rule (F5); arXiv preprint
S151	DeFiAligner: Leveraging Symbolic Analysis and Large Language Models for Inconsistency Detection in Decentralized Finance	2024	F1: AI not integrated with symbolic execution	DeFiAligner: symbolic expressions from SEVM are passed one-way to LLMs for comparison with documentation; LLM does not guide or extend SE (same rule as S154)
S154	Detecting Multi-Parameter Constraint Inconsistencies in Python Data Science Libraries	2024	F1: AI not integrated with symbolic execution	MPDetector: SE extracts code constraints, LLM extracts doc constraints, then compared; LLM does not improve SE. Same decision applies to its journal version S245
S179	Efficient Multiplex Symbolic Execution with Adaptive Search Strategy	2020	F2: AI component not evaluated	2-page ASE 2020 NIER paper: only the redundancy filter is evaluated; the online-learned search strategy is described as future work
S218	Generating and understanding tests via path-aware symbolic execution with llms	2025	F3: Duplicate publication of an included study	arXiv version of PALM (S340): same authors, same system; link resolves to the PALM PDF
S241	Hybrid Fuzzing with Deep Reinforcement Learning for Efficient Search Space Reduction	2022	F4: Thesis or technical report	Korean thesis record (KISTI ScienceON, DIKO = domestic dissertation), not a journal/conference paper
S243	Hyperion: Unveiling DApp Inconsistencies Using LLM and Dataflow-Guided Symbolic Execution	2025	F1: AI not integrated with symbolic execution	Hyperion: fine-tuned LLaMA2 analyses front-end descriptions, symbolic execution analyses bytecode separately, results compared; LLM does not guide or extend SE (same rule as S151/S154)
S245	Identifying Multi-parameter Constraint Errors in Python Data Science Library API Documentation	2025	F1: AI not integrated with symbolic execution	Journal version (PACMSE) of S154/MPDetector: same method, LLM does not improve SE
S251	Improving search-based test case generation with local search using adaptive simulated annealing and dynamic symbolic execution	2022	F1: AI not integrated with symbolic execution	Adaptive simulated annealing (metaheuristic) plus DSE in search-based testing; no AI/ML component
S257	Integrating symbolic execution and machine learning for automated test generation in system-level programming languages vision	2025	F5: Quality rule not met (Q1 = 1 and total $\geq$ 3)	CEUR vision paper: evaluated only on minimal C/Rust examples (Q1 = 0) $\rightarrow$ fails quality rule (F5)
S342	Parallelizing Path Exploration and Optimizing Constraint Solving for Efficient Symbolic Execution	2020	F4: Thesis or technical report	Doctoral dissertation (ProQuest), not a journal/conference paper
S373	RIVER 2.0: An open-source testing framework using AI techniques	2020	F3: Duplicate publication of an included study	RIVER 2.0 tool paper: evaluation tables reproduced from the team's earlier RL-concolic work (S336)
S383	Scalable Concolic Execution of COTS Binaries with the River Framework	2023	F1: AI not integrated with symbolic execution	River concolic fuzzer for COTS binaries (Triton + GDB); no AI component, RL only in related work

S455	Synergizing symbolic execution and fuzzing by function-level selective symbolization	2022	F1: AI not integrated with symbolic execution F3: Duplicate publication of an included study	FUSE: function-level selective symbolization combined with fuzzing; no AI component ASE 2020 extended abstract: preliminary version of S456 (ISSTA 2021), which is included
S457	Synthesizing Smart Solving Strategy for Symbolic Execution	2020		

Table A3. Studies included in the review corpus ($n = 37$).

ID	Ref.	Study	SE Stage	AI Technique	Status	Embedded Evaluation	Quality
S340	[17]	PALM: Path-aware LLM-based Test Generation with Comprehension	Test/input generation	Large language model	Peer-reviewed	No	4
S273	[29]	Learning to Explore Paths for Symbolic Execution	Path selection	Supervised learning (Learch)	Peer-reviewed	No	4
S284	[30]	LLM-Based Unknown Function Automated Modeling in Sensor-Driven Systems for Multi-Language Software Security Verification	Environment modeling	Fine-tuned large language model (20B)	Peer-reviewed	Partly	4
S336	[68]	Optimizing decision making in concolic execution using reinforcement learning	Path selection	Deep Q-network with LSTM	Peer-reviewed	No	4
S186	[69]	Enhancing dynamic symbolic execution by automatically learning search heuristics	Path selection	Learned parametric search heuristic	Peer-reviewed	No	4
S134	[73]	Constraint Solving with Deep Learning for Symbolic Execution	Constraint solving	Deep neural network (satisfiability classifier)	Preprint	No (trained on avionics and automotive models; evaluated on general-purpose Java programs)	3
S456	[75]	Synthesize solving strategy for symbolic execution	Constraint solving	Deep reinforcement learning + decision trees	Peer-reviewed	No	4
S259	[78]	Integrating Symbolic Execution with LLMs for Automated Generation of Program Specifications	Specification/invariant inference	Large language model	Preprint	Partly	3
S299	[82]	Making symbolic execution promising by learning aggressive state-pruning strategy	State management	Learned state-pruning strategy	Peer-reviewed	No	4
S453	[84]	SYMTUNER: Maximizing the power of symbolic execution by adaptively tuning external parameters	Engine configuration	Online learning (adaptive parameter sampling)	Peer-reviewed	No	4
S272	[86]	Large Language Model Powered Symbolic Execution	Constraint solving	Large language model	Peer-reviewed	No	4
S358	[88]	Python symbolic execution with llm-powered code generation	Constraint solving	Large language model (agent)	Preprint	No	3
S344	[111]	Path Exploration Strategy for Symbolic Execution based on Multi-strategy Active Learning	Path selection	Active learning + feedforward neural network	Peer-reviewed	No	4

S337	[112]	Optimizing Symbolic Execution Path Exploration with a Transfer Learning-Based Strategy	Path selection	Transfer learning (neural network, program families)	Peer-reviewed	No	4
S169	[113]	DLF: A Deep Active Ensemble Learning Framework for Test Case Generation	Path selection	Active ensemble learning + gated graph neural network	Peer-reviewed	No	4
S207	[114]	Featmaker: Automated feature engineering for search strategy of symbolic execution	Path selection	Automated feature generation + learned weights	Peer-reviewed	No	4
S451	[115]	SyML: Guiding symbolic execution toward vulnerable states through pattern learning {SmarTest}: Effectively hunting vulnerable transaction	Path selection	Supervised machine learning	Peer-reviewed	No	4
S422	[116]	sequences in smart contracts through language {Model-Guided} symbolic execution	Path selection	Statistical language model	Peer-reviewed	No	4
S322	[117]	NumScout: Unveiling Numerical Defects in Smart Contracts Using LLM-Pruning Symbolic Execution	Path selection	Large language model (function pruning)	Peer-reviewed	No	4
S204	[118]	Exploring strategies for guiding symbolic analysis with machine learning prediction	Path selection	ML satisfiability prediction (strategy study)	Peer-reviewed	No	4
S108	[119]	Boosting symbolic execution via constraint solving time prediction (experience paper)	Constraint solving	Machine learning (adaptive solving-time predictor)	Peer-reviewed	No	4
S294	[120]	Machine learning steered symbolic execution framework for complex software code	Constraint solving	Machine learning (sampling-learning optimisation solver)	Peer-reviewed	No	4
S404	[121]	Simulate, Refine and Integrate: Strategy Synthesis for Efficient SMT Solving	Constraint solving	Machine learning (strategy synthesis)	Peer-reviewed	No	4
S260	[122]	Intelligent Constraint Classification for Symbolic Execution	Constraint solving	Neural networks (path-condition classification)	Peer-reviewed	No (trained on avionics and automotive models; evaluated on general-purpose Java programs)	4
S003	[123]	Accelerating Symbolic Analysis for Android Apps	Constraint solving	Random forest (unsatisfiable-path prediction)	Peer-reviewed	No	4

S402	[124]	Sibyl: Improving Software Engineering Tools with SMT Selection	Constraint solving	Graph neural network (solver selection)	Peer-reviewed	No	4
S274	[125]	Learning to Prune Infeasible Paths in Generalized Symbolic Execution	State management	Machine learning classifier (heap feasibility)	Peer-reviewed	No	4
S012	[126]	ADFEmu: Enhancing Firmware Fuzzing with Direct Memory Access (DMA) Input Emulation Using Concolic Execution and Large Language Models (LLMs)	Environment modeling	Large language model + concolic execution	Peer-reviewed	Yes	4
S282	[127]	LLM-Based automated modeling in symbolic execution for securing medical software	Environment modeling	Fine-tuned large language model (20B)	Preprint	Partly	3
S091	[128]	Autostub: Genetic programming-based stub creation for symbolic execution	Environment modeling	Genetic programming	Peer-reviewed	No	4
S140	[129]	Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation	Test/input generation	Large language model	Preprint	No	3
S270	[130]	Large Language Model assisted Hybrid Fuzzing	Hybrid fuzzing	Large language model	Preprint	No	3
S147	[131]	Deep learning-based hybrid fuzz testing	Hybrid fuzzing	Deep learning (path suitability prediction)	Peer-reviewed	No	4
S341	[132]	PALM: Synergizing Program Analysis and LLMs to Enhance Rust Unit Test Coverage	Test/input generation	Large language model	Peer-reviewed	No	4
S449	[133]	SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models	Specification/invariant inference	Large language model	Peer-reviewed	No	4
S276	[134]	LIFT: Automating Symbolic Execution Optimization with Large Language Models for AI Networks	Other	Large language model	Peer-reviewed	No	4
S093	[135]	Baton: symphony of random testing and concolic testing through machine learning and taint analysis	Test/input generation	Learned classification models	Peer-reviewed	No	4

Table A4. Quality scores of all reports assessed in full text ($n = 54$).

ID	Study	Q1	Q2	Q3	Q4	Total	Decision
S003	Accelerating Symbolic Analysis for Android Apps	1	1	1	1	4	Included
S012	ADFEmu: Enhancing Firmware Fuzzing with Direct Memory Access (DMA) Input Emulation Using Concolic Execution and Large Language Models (LLMs)	1	1	1	1	4	Included
S069	ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System	-	-	-	-	-	F4

S091	Autostub: Genetic programming-based stub creation for symbolic execution	1	1	1	1	4	Included
S093	Baton: symphony of random testing and concolic testing through machine learning and taint analysis	1	1	1	1	4	Included
S107	Boosting symbolic execution for vulnerability detection	-	-	-	-	-	F4
S108	Boosting symbolic execution via constraint solving time prediction (experience paper)	1	1	1	1	4	Included
S120	Combining Dynamic Symbolic Execution, Machine Learning and Search-Based Testing to Automatically Generate Test Cases for Classes	0	0	1	1	2	F5
S134	Constraint Solving with Deep Learning for Symbolic Execution	1	0	1	1	3	Included
S140	Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation	1	0	1	1	3	Included
S147	Deep learning-based hybrid fuzz testing	1	1	1	1	4	Included
S151	DeFiAligner: Leveraging Symbolic Analysis and Large Language Models for Inconsistency Detection in Decentralized Finance	-	-	-	-	-	F1
S154	Detecting Multi-Parameter Constraint Inconsistencies in Python Data Science Libraries	-	-	-	-	-	F1
S169	DLF: A Deep Active Ensemble Learning Framework for Test Case Generation	1	1	1	1	4	Included
S179	Efficient Multiplex Symbolic Execution with Adaptive Search Strategy	-	-	-	-	-	F2
S186	Enhancing dynamic symbolic execution by automatically learning search heuristics	1	1	1	1	4	Included
S204	Exploring strategies for guiding symbolic analysis with machine learning prediction	1	1	1	1	4	Included
S207	Featmaker: Automated feature engineering for search strategy of symbolic execution	1	1	1	1	4	Included
S218	Generating and understanding tests via path-aware symbolic execution with llms	-	-	-	-	-	F3
S241	Hybrid Fuzzing with Deep Reinforcement Learning for Efficient Search Space Reduction	-	-	-	-	-	F4
S243	Hyperion: Unveiling DApp Inconsistencies Using LLM and Dataflow-Guided Symbolic Execution	-	-	-	-	-	F1
S245	Identifying Multi-parameter Constraint Errors in Python Data Science Library API Documentation	-	-	-	-	-	F1
S251	Improving search-based test case generation with local search using adaptive simulated annealing and dynamic symbolic execution	-	-	-	-	-	F1
S257	Integrating symbolic execution and machine learning for automated test generation in system-level programming languages vision	0	0	1	1	2	F5
S259	Integrating Symbolic Execution with LLMs for Automated Generation of Program Specifications	1	0	1	1	3	Included
S260	Intelligent Constraint Classification for Symbolic Execution	1	1	1	1	4	Included
S270	Large Language Model assisted Hybrid Fuzzing	1	0	1	1	3	Included
S272	Large Language Model Powered Symbolic Execution	1	1	1	1	4	Included
S273	Learning to Explore Paths for Symbolic Execution	1	1	1	1	4	Included
S274	Learning to Prune Infeasible Paths in Generalized Symbolic Execution	1	1	1	1	4	Included

S276	LIFT: Automating Symbolic Execution Optimization with Large Language Models for AI Networks	1	1	1	1	4	Included
S282	LLM-Based automated modeling in symbolic execution for securing medical software	1	0	1	1	3	Included
S284	LLM-Based Unknown Function Automated Modeling in Sensor-Driven Systems for Multi-Language Software Security Verification	1	1	1	1	4	Included
S294	Machine learning steered symbolic execution framework for complex software code	1	1	1	1	4	Included
S299	Making symbolic execution promising by learning aggressive state-pruning strategy	1	1	1	1	4	Included
S322	NumScout: Unveiling Numerical Defects in Smart Contracts Using LLM-Pruning Symbolic Execution	1	1	1	1	4	Included
S336	Optimizing decision making in concolic execution using reinforcement learning	1	1	1	1	4	Included
S337	Optimizing Symbolic Execution Path Exploration with a Transfer Learning-Based Strategy	1	1	1	1	4	Included
S340	PALM: Path-aware LLM-based Test Generation with Comprehension	1	1	1	1	4	Included
S341	PALM: Synergizing Program Analysis and LLMs to Enhance Rust Unit Test Coverage	1	1	1	1	4	Included
S342	Parallelizing Path Exploration and Optimizing Constraint Solving for Efficient Symbolic Execution	-	-	-	-	-	F4
S344	Path Exploration Strategy for Symbolic Execution based on Multi-strategy Active Learning	1	1	1	1	4	Included
S358	Python symbolic execution with llm-powered code generation	1	0	1	1	3	Included
S373	RIVER 2.0: An open-source testing framework using AI techniques	-	-	-	-	-	F3
S383	Scalable Concolic Execution of COTS Binaries with the River Framework	-	-	-	-	-	F1
S402	Sibyl: Improving Software Engineering Tools with SMT Selection	1	1	1	1	4	Included
S404	Simulate, Refine and Integrate: Strategy Synthesis for Efficient SMT Solving	1	1	1	1	4	Included
S422	{SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution	1	1	1	1	4	Included
S449	SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models	1	1	1	1	4	Included
S451	SyML: Guiding symbolic execution toward vulnerable states through pattern learning	1	1	1	1	4	Included
S453	SYMTUNER: Maximizing the power of symbolic execution by adaptively tuning external parameters	1	1	1	1	4	Included
S455	Synergizing symbolic execution and fuzzing by function-level selective symbolization	-	-	-	-	-	F1
S456	Synthesize solving strategy for symbolic execution	1	1	1	1	4	Included
S457	Synthesizing Smart Solving Strategy for Symbolic Execution	-	-	-	-	-	F3

Q1: empirical evaluation, formal analysis or solid theoretical framework; Q2: peer-reviewed (preprints score 0); Q3: sufficient technical detail; Q4: novel method or notable improvement. Inclusion rule: Q1 = 1 and total ≥ 3 . Reports excluded for reasons F1 to F4 were not scored.

Table A5. PRISMA 2020 checklist.

Section	Item	Checklist Item	Location in Manuscript
Title	1	Identify the report as a systematic review	Title
Abstract	2	Structured summary	Abstract
Introduction	3	Rationale	Section 1
	4	Objectives/research questions	Sections 1 and 3.1
Methods	5	Eligibility criteria	Section 3.3, Table 2
	6	Information sources and search date	Section 3.2, Table 1, Table A1
	7	Full search strategies	Section 3.2, Table A1
	8	Selection process (reviewers, automation tools)	Section 3.3 (AI-assisted screening, one author)
	9	Data collection process	Section 3.3; screening spreadsheet (available on request)
	10	Data items	Tables 4 and A3 (stage, AI technique, status, embedded evaluation)
	11	Study risk-of-bias/quality assessment	Section 3.5, Table A4
	12	Effect measures	Not applicable (no meta-analysis)
	13	Synthesis methods	Section 3.5 (narrative synthesis by taxonomy)
	14	Reporting bias assessment	Section 3.6
	15	Certainty assessment	Not performed; limitations in Section 3.6
Results	16	Study selection, flow diagram	Section 3.4, Figure 1, Table A2
	17	Study characteristics	Tables 4 and A3
	18	Risk of bias/quality in studies	Section 3.5, Table A4
	19	Results of individual studies	Sections 4 and 7
	20	Results of syntheses	Sections 4–7
	21	Reporting biases	Section 3.6
	22	Certainty of evidence	Sections 4.6 and 7 (qualitative)
Discussion	23	Discussion and limitations	Sections 3.6, 8 and 9
Other	24	Registration and protocol	Not registered; no protocol published
	25	Support/funding	Funding statement
	26	Competing interests	Conflicts of Interest statement
	27	Availability of data and materials	Appendix A; screening spreadsheet available from the corresponding author on request

References

- Rios, E.; Iturbe, E.; Rego, A.; Ferry, N.; Tigli, J.-Y.; Lavirotte, S.; Rocher, G.; Nguyen, P.; Song, H.; Dautov, R.; et al. The DYNABIC approach to resilience of critical infrastructures. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 1–8.
- Nguyen, P.H.; Rauniyar, A.; Niemi, T.V. Digital Twin-Based Security Orchestration, Automation and Response for IoT and CPS. In *IFIP International Internet of Things Conference*; Springer Nature: Cham, Switzerland, 2024; pp. 243–260.
- Kelemen, R.; Squillace, J.; Medvácz, Á.; Cappella, J.; Bucko, B.; Mazuch, M. Cybersecurity regulations and software resilience: Strengthening awareness and societal stability. *Soc. Sci.* **2025**, *14*, 578.
- Kaspersky ICS CERT. Threat Landscape for Industrial Automation Systems. Q3 2025. Kaspersky. 11 December 2025. Available online: <https://ics-cert.kaspersky.com/publications/reports/2025/12/11/threat-landscape-for-industrial-automation-systems-q3-2025/> (accessed on 8 June 2026).
- Kayan, H.; Nunes, M.; Rana, O.; Burnap, P.; Perera, C. Cybersecurity of industrial cyber-physical systems: A review. *ACM Comput. Surv. (CSUR)* **2022**, *54*, 1–35.
- Matloob, F.; Ghazal, T.M.; Taleb, N.; Aftab, S.; Ahmad, M.; Khan, M.A.; Abbas, S.; Soomro, T.R. Software defect prediction using ensemble learning: A systematic literature review. *IEEE Access* **2021**, *9*, 98754–98771.
- King, J.C. Symbolic execution and program testing. *Commun. ACM* **1976**, *19*, 385–394.
- Clarke, L.A. A Personal Retrospective on Symbolic Execution. *IEEE Trans. Softw. Eng.* **2025**, *51*, 706–709.
- Cadar, C.; Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM* **2013**, *56*, 82–90.

10. Baldoni, R.; Coppa, E.; D’elia, D.C.; Demetrescu, C.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–39.
11. Guo, W.S.; Wang, Y.; Yang, X.; Gao, M. Codecomb: Automated Test Case Generation and Defect Detecting for Embedded Software Based on Symbolic Execution. *J. Chin. Comput. Syst.* **2017**, *38*, 1250–1255.
12. Nicho, M.; Effiong, I.; McDermott, C.D. Shift-Left Security: Integrating Security in the Initial Phase of the DevOps Methodology. In *2025 9th International Conference on Cryptography, Security and Privacy (CSP)*; IEEE: New York, NY, USA, 2025; pp. 182–191.
13. Malik, G. Integrating threat intelligence with DevSecOps: Automating risk mitigation before code hits production. *Util. Math.* **2025**, *122*, 309–340.
14. Sinan, M. Support Security Control Management and Implementation in DevSecOps. Doctoral Dissertation, RMIT University, Melbourne, Australia, 2025.
15. Daggett, M.L.; Kokke, W.; Atkey, R.; Komendantskaya, E.; Slusarz, N.; Arnaboldi, L. Vehicle: Bridging the embedding gap in the verification of neuro-symbolic programs. *arXiv* **2024**, arXiv:2401.06379.
16. Sena, L.; Song, X.; Alves, E.; Bessa, I.; Manino, E.; Cordeiro, L. Verifying quantized neural networks using SMT-based model checking. *arXiv* **2021**, arXiv:2106.05997.
17. Wu, Y.; Zhou, X.; Humayun, A.; Gulzar, M.A.; Kim, M. Generating and understanding tests via path-aware symbolic execution with llms. *arXiv* **2025**, arXiv:2506.19287.
18. Abdullin, A.; Derakhshanfar, P.; Panichella, A. Test wars: A comparative study of SBST, symbolic execution, and LLM-based approaches to unit test generation. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*; IEEE: New York, NY, USA, 2025; pp. 221–232.
19. Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI* **2008**, *8*, 209–224.
20. Godefroid, P.; Levin, M.Y.; Molnar, D. SAGE: Whitebox fuzzing for security testing. *Commun. ACM* **2012**, *55*, 40–44.
21. Li, Y.; Liu, W.; Liu, Q.; Zheng, X.; Sun, K.; Huang, C. Complying with iso 26262 and iso/sae 21434: A safety and security co-analysis method for intelligent connected vehicle. *Sensors* **2024**, *24*, 1848.
22. Rierison, L. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*; CRC Press: Boca Raton, FL, USA, 2017.
23. Cadar, C.; Nowack, M. KLEE symbolic execution engine in 2019. *Int. J. Softw. Tools Technol. Transf.* **2021**, *23*, 867–870.
24. Yang, X.; Zhang, G.; Shuai, Z.; Chen, Z.; Wang, J. Symbolic execution of floating-point programs: How far are we? *J. Syst. Softw.* **2025**, *220*, 112242.
25. Ahn, S.; Malik, S. Automated firmware testing using firmware-hardware interaction patterns. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*; Association for Computing Machinery: New York, NY, USA, 2014; pp. 1–10.
26. Muench, M.; Stijohann, J.; Kargl, F.; Francillon, A.; Balzarotti, D. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *NDSS*; Internet Society: Fredericksburg, VA, USA, 2018.
27. Chistyakova, A.; Nigmatulin, M.; Shemetova, E.; Parfenov, D.; Akhmedov, D.; Grigorev, S. PySymGym: An Infrastructure to Train AI-Powered Navigation Assistant for Symbolic Execution Engine. In *2025 IEEE/ACM 1st International Workshop on Advancing Static Analysis for Researchers and Industry Practitioners in Software Engineering (STATIC)*; IEEE: New York, NY, USA, 2025; pp. 13–16.
28. Cha, S.; Hong, S.; Lee, J.; Oh, H. Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*; ACM: New York, NY, USA, 2018; pp. 1244–1254.
29. He, J.; Sivanrupan, G.; Tsankov, P.; Vechev, M. Learning to explore paths for symbolic execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 2526–2540.
30. Deng, L.; Zhong, Q.; Song, J.; Lei, H.; Li, W. Llm-based unknown function automated modeling in sensor-driven systems for multi-language software security verification. *Sensors* **2025**, *25*, 2683.
31. Wu, J.; Zhang, C.; Pu, G. Reinforcement learning guided symbolic execution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*; IEEE: New York, NY, USA, 2020.
32. Iavich, M.; Kuchukhidze, T.; Lopata, A. Static Analysis Techniques for Embedded, Cyber-Physical, and Electronic Software Systems: A Comprehensive Survey. *Electronics* **2026**, *15*, 918.
33. Bailey, J.; Nicholas, C. Symbolic Execution in Practice: A Survey of Applications in Vulnerability, Malware, Firmware, and Protocol Analysis. *arXiv* **2025**, arXiv:2508.06643.

34. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2016; pp. 138–157.
35. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Not.* **2011**, *46*, 265–278.
36. Barrett, C.; Fontaine, P.; Tinelli, C. *The SMT-LIB Standard: Version 2.6*; Technical Report; Department of Computer Science, The University of Iowa: Iowa City, IA, USA, 2017.
37. de Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*; LNCS 4963; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
38. Barbosa, H.; Reynolds, A.; Kremer, G.; Lachnitt, H.; Niemetz, A.; Nötzli, A.; Ozdemir, A.; Preiner, M.; Viswanathan, A.; Viteri, S.; et al. Flexible proof production in an industrial-strength SMT solver. In *International Joint Conference on Automated Reasoning*; Springer International Publishing: Cham, Switzerland, 2022; pp. 15–35.
39. Reisenberger, C. PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting with Look-Ahead. Master's Thesis, Johannes Kepler Universität Linz, Linz, Austria, 2014.
40. Liu, T.; Araújo, M.; d'Amorim, M.; Taghdiri, M. A comparative study of incremental constraint solving approaches in symbolic execution. In *Haifa Verification Conference*; Springer International Publishing: Cham, Switzerland, 2014; pp. 284–299.
41. Zhang, T.; Wang, P.; Guo, X. A survey of symbolic execution and its tool KLEE. *Procedia Comput. Sci.* **2020**, *166*, 330–334.
42. Kim, S.; Faerevaag, M.; Jung, M.; Jung, S.; Oh, D.; Lee, J.; Cha, S.K. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: New York, NY, USA, 2017; pp. 353–364.
43. Cytron, R.; Ferrante, J.; Rosen, B.K.; Wegman, M.N.; Zadeck, F.K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* **1991**, *13*, 451–490.
44. Cadar, C.; Ganesh, V.; Pawlowski, P.M.; Dill, D.L.; Engler, D.R. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2008**, *12*, 1–38.
45. Christakis, M.; Müller, P.; Wüstholtz, V. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 144–155.
46. Schemmel, D.; Büning, J.; Rodríguez, C.; Laprell, D.; Wehrle, K. Symbolic partial-order execution for testing multi-threaded programs. In *International Conference on Computer Aided Verification*; Springer International Publishing: Cham, Switzerland, 2020; pp. 376–400.
47. Chen, B.; Havlicek, C.; Yang, Z.; Cong, K.; Kannavara, R.; Xie, F. CRETE: A versatile binary-level concolic testing framework. In *International Conference on Fundamental Approaches to Software Engineering*; Springer International Publishing: Cham, Switzerland, 2018; pp. 281–298.
48. Brooks, T.N. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In *Science and Information Conference*; Springer International Publishing: Cham, Switzerland, 2018; pp. 1083–1102.
49. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*; Internet Society: Fredericksburg, VA, USA, 2016; Volume 16, pp. 1–16.
50. Zaddach, J.; Bruno, L.; Francillon, A.; Balzarotti, D. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS)*; Internet Society: Fredericksburg, VA, USA, 2014.
51. Dong, S.; Olivo, O.; Zhang, L.; Khurshid, S. Studying the influence of standard compiler optimizations on symbolic execution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*; IEEE: New York, NY, USA, 2015; pp. 205–215.
52. Trabish, D.; Rinetzky, N.; Shoham, S.; Sharma, V. State merging with quantifiers in symbolic execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 1140–1152.
53. Cadar, C. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 906–909.
54. Godefroid, P. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*; Association for Computing Machinery: New York, NY, USA, 2007; pp. 47–54.

55. Rosner, N.; Geldenhuys, J.; Aguirre, N.M.; Visser, W.; Frias, M.F. BLISS: Improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Trans. Softw. Eng.* **2015**, *41*, 639–660.
56. Visser, W.; Geldenhuys, J.; Dwyer, M.B. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*; Association for Computing Machinery: New York, NY, USA, 2012; Article 58.
57. Bucur, S.; Ureche, V.; Zamfir, C.; Candea, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys)*; Association for Computing Machinery: New York, NY, USA, 2011; pp. 183–198.
58. Staats, M.; Păsăreanu, C.S. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*; Association for Computing Machinery: New York, NY, USA, 2010; pp. 183–194.
59. Regehr, J. Random testing of interrupt-driven software. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT)*; Association for Computing Machinery: New York, NY, USA, 2005; pp. 290–298.
60. Robere, R.; Kolokolova, A.; Ganesh, V. The proof complexity of SMT solvers. In *International Conference on Computer Aided Verification*; Springer International Publishing: Cham, Switzerland, 2018; pp. 275–293.
61. Brain, M.; Schanda, F.; Sun, Y. Building better bit-blasting for floating-point problems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer International Publishing: Cham, Switzerland, 2019; pp. 79–98.
62. Clements, A.A.; Gustafson, E.; Scharnowski, T.; Grosen, P.; Fritz, D.; Kruegel, C.; Vigna, G.; Bagchi, S.; Payer, M. {HALucinator}: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*; USENIX Association: Berkeley, CA, USA, 2020; pp. 1201–1218.
63. Subramanyan, P.; Malik, S.; Khattri, H.; Maiti, A.; Fung, J. Verifying information flow properties of firmware using symbolic execution. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*; IEEE: New York, NY, USA, 2016; pp. 337–342.
64. Karna, A.K.; Du, J.; Shen, H.; Zhong, H.; Gong, J.; Yu, H.; Ma, X.; Zhao, J. Tuning parallel symbolic execution engine for better performance. *Front. Comput. Sci.* **2018**, *12*, 86–100.
65. Ziegenbein, D.; Hamann, A. Timing-aware control software design for automotive systems. In *Proceedings of the 52nd Annual Design Automation Conference*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 1–6.
66. Moher, D.; Liberati, A.; Tetzlaff, J.; Altman, D.G.; Prisma Group. Preferred reporting items for systematic reviews and meta-analyses: The PRISMA statement. *Int. J. Surg.* **2010**, *8*, 336–341.
67. Page, M.J.; McKenzie, J.E.; Bossuyt, P.M.; Boutron, I.; Hoffmann, T.C.; Mulrow, C.D.; Shamseer, L.; Tetzlaff, J.M.; Akl, E.A.; Brennan, S.E.; et al. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *BMJ* **2021**, *372*, n71.
68. Paduraru, C.; Stefanescu, A.; Paduraru, M.G. Optimizing decision making in concolic execution using reinforcement learning. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), A-MOST Workshop*; IEEE: New York, NY, USA, 2020.
69. Cha, S.; Hong, S.; Bak, J.; Kim, J.; Lee, J.; Oh, H. Enhancing dynamic symbolic execution by automatically learning search heuristics. *IEEE Trans. Softw. Eng.* **2021**, *48*, 3640–3663.
70. Cha, S.; Oh, H. Concolic testing with adaptively changing search heuristics. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 235–245.
71. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 10197–10207.
72. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*; International Joint Conferences on Artificial Intelligence Organization: Vienna, Austria, 2021; pp. 3283–3290.
73. Wen, J.; Khan, M.; Che, M.; Yan, Y.; Yang, G. Constraint solving with deep learning for symbolic execution. *arXiv* **2020**, arXiv:2003.08350.
74. Balunović, M.; Bielik, P.; Vechev, M. Learning to solve SMT formulas. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 10317–10328.
75. Chen, Z.; Chen, Z.; Shuai, Z.; Zhang, G.; Pan, W.; Zhang, Y.; Wang, J. Synthesize solving strategy for symbolic execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 348–360.

76. Scott, J.; Niemetz, A.; Preiner, M.; Nejati, S.; Ganesh, V. MachSMT: A machine learning-based algorithm selector for SMT solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 12652; Springer: Berlin/Heidelberg, Germany, 2021; pp. 303–325.
77. Borralleras, C.; Larraz, D.; Rodríguez-Carbonell, E.; Oliveras, A.; Rubio, A. Incomplete SMT techniques for solving non-linear formulas over the integers. *ACM Trans. Comput. Log. (TOCL)* **2019**, *20*, 1–36.
78. Yang, F.; Ma, X.; Wang, S.; Xu, X.; Cao, Q.; Zhan, N.; Li, X.; Gu, B. Integrating Symbolic Execution with LLMs for Automated Generation of Program Specifications. *arXiv* **2025**, arXiv:2506.09550.
79. Zhang, C.; Zheng, Y.; Bai, M.; Li, Y.; Ma, W.; Xie, X.; Li, Y.; Sun, L.; Liu, Y. How effective are they? Exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 1223–1235.
80. Kuznetsov, V.; Kinder, J.; Bucur, S.; Candea, G. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*; Association for Computing Machinery: New York, NY, USA, 2012; pp. 193–204.
81. Avgerinos, T.; Rebert, A.; Cha, S.K.; Brumley, D. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*; Association for Computing Machinery: New York, NY, USA, 2014; pp. 1083–1094.
82. Cha, S.; Oh, H. Making symbolic execution promising by learning aggressive state-pruning strategy. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 147–158.
83. He, J.; Balunović, M.; Ambroladze, N.; Tsankov, P.; Vechev, M. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 531–548.
84. Cha, S.; Lee, M.; Lee, S.; Oh, H. SymTuner: Maximizing the power of symbolic execution by adaptively tuning external parameters. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 2068–2079.
85. Hůla, J.; Mojžíšek, D.; Janota, M. Graph neural networks for scheduling of SMT solvers. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*; IEEE: New York, NY, USA, 2021; pp. 447–451.
86. Li, Y.; Meng, R.; Duck, G.J. Large language model powered symbolic execution. *Proc. ACM Program. Lang.* **2025**, *9*, 3148–3176.
87. Xu, J.; Xu, J.; Chen, T.; Ma, X. Symbolic Execution with Test Cases Generated by Large Language Models. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*; IEEE: New York, NY, USA, 2024; pp. 228–237.
88. Wang, W.; Liu, K.; Chen, A.R.; Li, G.; Jin, Z.; Huang, G.; Ma, L. Python symbolic execution with llm-powered code generation. *arXiv* **2024**, arXiv:2409.09271.
89. Wu, G.; Cao, W.; Yao, Y.; Wei, H.; Chen, T.; Ma, X. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*; IEEE: New York, NY, USA, 2024; pp. 406–417.
90. Yu, S.; Wang, T.; Wang, J. Loop invariant inference through SMT solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 175–187.
91. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*; IEEE: New York, NY, USA, 2014; pp. 590–604.
92. Zhang, Z.; Saber, T. Machine learning approaches to code similarity measurement: A systematic review. *IEEE Access* **2025**, *13*, 51729–51764.
93. Selsam, D.; Lamm, M.; Bünz, B.; Liang, P.; de Moura, L.; Dill, D.L. Learning a SAT solver from single-bit supervision. *arXiv* **2018**, arXiv:1802.03685.
94. Lu, Z. AlphaSMT: A Reinforcement Learning Guided SMT Solver. Master's Thesis, University of Waterloo, Waterloo, ON, Canada, 2023.
95. Hitzler, P.; Eberhart, A.; Ebrahimi, M.; Sarker, M.K.; Zhou, L. Neuro-symbolic approaches in artificial intelligence. *Natl. Sci. Rev.* **2022**, *9*, nwac035.
96. Bhuyan, B.P.; Ramdane-Cherif, A.; Tomar, R.; Singh, T.P. Neuro-symbolic artificial intelligence: A survey. *Neural Comput. Appl.* **2024**, *36*, 12809–12844.
97. Colelough, B.C.; Regli, W. Neuro-symbolic AI in 2024: A systematic review. *arXiv* **2025**, arXiv:2501.05435.

98. Lu, Z.; Afridi, I.; Kang, H.J.; Ruchkin, I.; Zheng, X. Surveying neuro-symbolic approaches for reliable artificial intelligence of things. *J. Reliab. Intell. Environ.* **2024**, *10*, 257–279.
99. Hsia, Y.S.; Yu, F.; Jiang, J.H.R. Neuro-Symbolic Compliance: Integrating LLMs and SMT Solvers for Automated Financial Legal Analysis. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*; IEEE: New York, NY, USA, 2025; pp. 1–10.
100. Feng, B.; Mera, A.; Lu, L. {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*; USENIX Association: Berkeley, CA, USA, 2020; pp. 1237–1254.
101. Hossain, S.B.; Filieri, A.; Dwyer, M.B.; Elbaum, S.; Visser, W. Neural-based test oracle generation: A large-scale evaluation and lessons learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 120–132.
102. Bao, J.; Trivedi, N.; Pathak, D.; Hsu, J.; Roy, S. Data-driven invariant learning for probabilistic programs. *Form. Methods Syst. Des.* **2025**, *66*, 278–306.
103. Sumanth Prabhu, S.; D’Souza, D.; Chakraborty, S.; Venkatesh, R.; Fedyukovich, G. Weakest precondition inference for non-deterministic linear array programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer Nature: Cham, Switzerland, 2024; pp. 175–195.
104. Porncharoenwase, S.; Nelson, L.; Wang, X.; Torlak, E. A formal foundation for symbolic evaluation with merging. *Proc. ACM Program. Lang.* **2022**, *6*, 1–28.
105. Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 4998–5007.
106. Borzacchiello, L.; Coppa, E.; Cono D’Elia, D.; Demetrescu, C. Memory models in symbolic execution: Key ideas and new thoughts. *Softw. Test. Verif. Reliab.* **2019**, *29*, e1722.
107. Kapus, T.; Cadar, C. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 774–784.
108. Lööw, A.; Park, S.H.; Nantes-Sobrinho, D.; Ayoun, S.É.; Sjöstedt, O.; Gardner, P. Compositional Symbolic Execution for the Next 700 Memory Models. *Proc. ACM Program. Lang.* **2025**, *9*, 2815–2842.
109. Miftah, S.S.; Srivastava, A.; Kim, H.; Wei, S.; Basu, K. SymbFuzz: Symbolic Execution Guided Hardware Fuzzing. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*; IEEE: New York, NY, USA, 2025; pp. 1477–1490.
110. Allamanis, M.; Chanthirasegaran, P.; Kohli, P.; Sutton, C. Learning continuous semantic representations of symbolic expressions. In *Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia, 6–11 August 2017; Precup, D., Teh, Y.W., Eds.; PMLR: 2017; Volume 70, pp. 80–88.
111. He, L.; Zhang, D.; Zhu, D.; Zhang, J.; Wang, R.; Liu, J. Path exploration strategy for symbolic execution based on multi-strategy active learning. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2024; pp. 165–168.
112. Sun, T.; Zhu, D.; He, L.; Zhang, D. Optimizing symbolic execution path exploration with a transfer learning-based strategy. *Int. J. Comput. Commun. Control* **2025**, *20*, 5. <https://doi.org/10.15837/ijccc.2025.5.6885>.
113. Lu, Y.; Peng, Y.; Zhu, D. DLF: A deep active ensemble learning framework for test case generation. *Information* **2025**, *16*, 1109. <https://doi.org/10.3390/info16121109>.
114. Yoon, J.; Cha, S. FeatMaker: Automated feature engineering for search strategy of symbolic execution. *Proc. ACM Softw. Eng.* **2024**, *1*, 2447–2468. <https://doi.org/10.1145/3660815>.
115. Ruaro, N.; Zeng, K.; Dresel, L.; Polino, M.; Bao, T.; Continella, A.; Zanero, S.; Kruegel, C.; Vigna, G. SyML: Guiding symbolic execution toward vulnerable states through pattern learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2021. <https://doi.org/10.1145/3471621.3471865>.
116. So, S.; Hong, S.; Oh, H. SmartTest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In *Proceedings of the 30th USENIX Security Symposium*; USENIX Association: Berkeley, CA, USA, 2021.

117. Chen, J.; Shao, Z.; Yang, S.; Shen, Y.; Wang, Y.; Chen, T.; Shan, Z.; Zheng, Z. NumScout: Unveiling numerical defects in smart contracts using LLM-pruning symbolic execution. *IEEE Trans. Softw. Eng.* **2025**, *51*, 1538–1553. <https://doi.org/10.1109/TSE.2025.3555622>.
118. Yang, M.; Lie, D.; Papernot, N. Exploring strategies for guiding symbolic analysis with machine learning prediction. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*; IEEE: New York, NY, USA, 2024; pp. 659–669. <https://doi.org/10.1109/SANER60148.2024.00073>.
119. Luo, S.; Xu, H.; Bi, Y.; Wang, X.; Zhou, Y. Boosting symbolic execution via constraint solving time prediction (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*; Association for Computing Machinery: New York, NY, USA, 2021. <https://doi.org/10.1145/3460319.3464813>.
120. BuBu, L.; Liang, Y.; Xie, Z.; Qian, H.; Hu, Y.-Q.; Yu, Y.; Chen, X.; Li, X. Machine learning steered symbolic execution framework for complex software code. *Form. Asp. Comput.* **2021**, *33*, 301–323.
121. Zhou, B.; Wang, H.; Yao, Y.; Chen, T.; Xu, F.; Ma, X. Simulate, refine and integrate: Strategy synthesis for efficient SMT solving. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI)*; International Joint Conferences on Artificial Intelligence Organization: Vienna, Austria, 2025. <https://doi.org/10.24963/ijcai.2025/887>.
122. Wen, J.; Mahmud, T.; Che, M.; Yan, Y.; Yang, G. Intelligent constraint classification for symbolic execution. In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*; IEEE: New York, NY, USA, 2023. <https://doi.org/10.1109/SANER56733.2023.00023>.
123. Yang, M.; Lie, D.; Papernot, N. Accelerating symbolic analysis for Android apps. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*; IEEE: New York, NY, USA, 2021. <https://doi.org/10.1109/ASEW52652.2021.00021>.
124. Leeson, W.; Dwyer, M.B.; Filieri, A. Sibyl: Improving software engineering tools with SMT selection. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*; IEEE: New York, NY, USA, 2023. <https://doi.org/10.1109/ICSE48619.2023.00184>.
125. Molina, F.; Ponzio, P.; Aguirre, N.; Frias, M. Learning to prune infeasible paths in generalized symbolic execution. In *Proceedings of the 33rd IEEE International Symposium on Software Reliability Engineering (ISSRE)*; IEEE: New York, NY, USA, 2022. <https://doi.org/10.1109/ISSRE55969.2022.00054>.
126. Ding, Y.; Zhao, X.; Wu, Z.; Zhu, Y.; Bai, L.; Han, H. ADFEmu: Enhancing firmware fuzzing with direct memory access (DMA) input emulation using concolic execution and large language models (LLMs). *Comput. Mater. Contin.* **2025**, *84*, 5977–5993. <https://doi.org/10.32604/cmc.2025.065672>.
127. Chen, J.; Deng, L.; Qiu, Y.; Zhao, P.; Lei, H.; Song, J.; Wang, X. LLM-based automated modeling in symbolic execution for securing medical software. *SSRN* **2024**, preprint 4938953.
128. Mächtle, F.; Loose, N.; Serr, J.-N.; Sander, J.; Eisenbarth, T. AutoStub: Genetic programming-based stub creation for symbolic execution. In *2025 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*; IEEE: New York, NY, USA, 2025; pp. 19–22. <https://doi.org/10.1109/SBFT66712.2025.00011>.
129. Tu, H.; Lee, S.; Li, Y.; Chen, P.; Jiang, L.; Böhme, M. Cottontail: Large language model-driven concolic execution for highly structured test input generation. *arXiv* **2025**, arXiv:2504.17542.
130. Meng, R.; Duck, G.J.; Roychoudhury, A. Large language model assisted hybrid fuzzing. *arXiv* **2024**, arXiv:2412.15931.
131. Gao, F.; Wang, Y.; Situ, L.; Wang, L. Deep learning-based hybrid fuzz testing. *J. Softw.* **2021**, *32*, 988–1005. <https://doi.org/10.13328/j.cnki.jos.006225>.
132. Chu, B.; Feng, Y.; Liu, K.; Shi, H.; Nan, Z.; Guo, Z.; Xu, B. PALM: Synergizing program analysis and LLMs to enhance Rust unit test coverage. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: New York, NY, USA, 2025. <https://doi.org/10.1109/ASE63991.2025.00223>.
133. Xia, S.; He, M.; Shao, S.; Yu, T.; Zhang, Y.; Yoshida, N.; Song, L. SymGPT: Auditing smart contracts via combining symbolic execution with large language models. *Proc. ACM Program. Lang.* **2026**, *10*, 109. <https://doi.org/10.1145/3798217>.
134. Wang, R.; Li, K.; Xu, M.; Zhang, Y.; Xu, K.; Liu, C.; Xiao, Y.; Cheng, X. LIFT: Automating symbolic execution optimization with large language models for AI networks. In *Proceedings of the 2nd Workshop on Networks for AI Computing (NAIC)*; USENIX Association: Berkeley, CA, USA, 2025; pp. 37–42. <https://doi.org/10.1145/3748273.3749202>.
135. Chen, B.; Liu, Y.; Peng, X.; Wu, Y.; Qin, S. BATON: Symphony of random testing and concolic testing through machine learning and taint analysis. *Sci. China Inf. Sci.* **2023**, *66*, 132101. <https://doi.org/10.1007/s11432-020-3403-2>.

136. Jin, X.; Pei, K.; Won, J.Y.; Lin, Z. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1631–1645.
137. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing Mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*; IEEE: New York, NY, USA, 2012; pp. 380–394.
138. Avgerinos, T.; Brumley, D.; Davis, J.; Goulden, R.; Nighswander, T.; Rebert, A.; Williamson, N. The Mayhem cyber reasoning system. *IEEE Secur. Priv.* **2018**, *16*, 52–60.
139. Chipounov, V.; Kuznetsov, V.; Candea, G. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. (TOCS)* **2012**, *30*, 1–49.
140. Ul Haq, S.; Singh, Y.; Sharma, A.; Gupta, R.; Gupta, D. A survey on IoT & embedded device firmware security: Architecture, extraction techniques, and vulnerability analysis frameworks. *Discov. Internet Things* **2023**, *3*, 17.
141. Davidson, D.; Moench, B.; Ristenpart, T.; Jha, S. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium*; USENIX Association: Berkeley, CA, USA, 2013; pp. 463–478.
142. Olivier, P.; Ngo, X.H.; Francillon, A. BEERR: Bench of embedded system experiments for reproducible research. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*; IEEE: New York, NY, USA, 2022; pp. 332–339.
143. Xie, X.; Ye, J.; Wu, L.; Li, R. RTOSExtractor: Extracting user-defined functions in stripped RTOS-based firmware. In *2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*; IEEE: New York, NY, USA, 2022; pp. 87–96.
144. Alimi, N.; LAHBIB, Y.; MACHHOUT, M.; Tourki, R. An rtos-based fault injection simulator for embedded processors. *Int. J. Adv. Comput. Sci. Appl.* **2017**, *8*, 37.
145. Naidu, G.A.; Kumar, J. Wireless protocols: Wi-fi son, bluetooth, zigbee, z-wave, and wi-fi. In *Innovations in Electronics and Communication Engineering: Proceedings of the 7th ICIECE 2018*; Springer: Singapore, 2019; pp. 229–239.
146. Sharma, C.; Gondhi, N.K. Communication protocol stack for constrained IoT systems. In *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*; IEEE: New York, NY, USA, 2018; pp. 1–6.
147. Kambourakis, G.; Kolias, C.; Geneiatakis, D.; Karopoulos, G.; Makrakis, G.M.; Kounelis, I. A state-of-the-art review on the security of mainstream IoT wireless PAN protocol stacks. *Symmetry* **2020**, *12*, 579.
148. Bojjagani, S.; Reddy, Y.P.; Anuradha, T.; Rao, P.V.; Reddy, B.R.; Khan, M.K. Secure authentication and key management protocol for deployment of Internet of Vehicles (IoV) concerning intelligent transport systems. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 24698–24713.
149. Panchal, P.; Hein, L.; Myschik, S.; Holzapfel, F. *A Systematic and Agile Approach to Developing DO-178C Com-Pliant Model-Based Safety-Critical Software*; Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV: Bonn, Germany, 2024.
150. Kim, S.; Ryu, S. Analysis of blockchain smart contracts: Techniques and insights. In *2020 IEEE Secure Development (SecDev)*; IEEE: New York, NY, USA, 2020; pp. 65–73.
151. Misra, S.; Mukherjee, A.; Roy, A.; Saurabh, N.; Rahulamathavan, Y.; Rajarajan, M. Blockchain at the edge: Performance of resource-constrained IoT networks. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *32*, 174–183.
152. Khan, S.; Lee, W.K.; Hwang, S.O. AEchain: A lightweight blockchain for IoT applications. *IEEE Consum. Electron. Mag.* **2021**, *11*, 64–76.
153. Khalid, U.; Asim, M.; Baker, T.; Hung, P.C.; Tariq, M.A.; Rafferty, L. A decentralized lightweight blockchain-based authentication mechanism for IoT systems. *Clust. Comput.* **2020**, *23*, 2067–2087.
154. Tan, J.; Shi, J.; Wan, J.; Dai, H.N.; Jin, J.; Zhang, R. Blockchain-based data security and sharing for resource-constrained devices in manufacturing IoT. *IEEE Internet Things J.* **2024**, *11*, 25558–25567.
155. Lee, B.; Lee, J.H. Blockchain-based secure firmware update for embedded devices in an Internet of Things environment. *J. Supercomput.* **2017**, *73*, 1152–1167.
156. Duan, Z.; Mao, H.; Chen, Z.; Bai, X.; Hu, K.; Talpin, J.P. Formal modeling and verification of blockchain system. In *Proceedings of the 10th International Conference on Computer Modeling and Simulation*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 231–235.
157. He, J.; Li, S.; Wang, X.; Cheung, S.C.; Zhao, G.; Yang, J. Neural-febi: Accurate function identification in ethereum virtual machine bytecode. *J. Syst. Softw.* **2023**, *199*, 111627.
158. Ghaleb, A.; Rubin, J.; Pattabiraman, K. Achecker: Statically detecting smart contract access control vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: New York, NY, USA, 2023; pp. 945–956.

159. Xue, Y.; Ma, M.; Lin, Y.; Sui, Y.; Ye, J.; Peng, T. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*; IEEE: New York, NY, USA, 2020; pp. 1029–1040.
160. Wei, L.; Ma, Z.; Yang, C.; Yao, Q. Advances in the neural network quantization: A comprehensive review. *Appl. Sci.* **2024**, *14*, 7445.
161. Dreossi, T.; Fremont, D.J.; Ghosh, S.; Kim, E.; Ravanbakhsh, H.; Vazquez-Chanlatte, M.; Seshia, S.A. Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *International Conference on Computer Aided Verification*; Springer International Publishing: Cham, Switzerland, 2019; pp. 432–442.
162. Dosovitskiy, A.; Ros, G.; Codevilla, F.; Lopez, A.; Koltun, V. CARLA: An open urban driving simulator. In *Conference on Robot Learning*; PMLR: 2017; pp. 1–16.
163. Koopman, P.; Wagner, M. Challenges in autonomous vehicle testing and validation. *SAE Int. J. Transp. Saf.* **2016**, *4*, 15–24.
164. Beyer, D.; Strejček, J. Improvements in software verification and witness validation: SV-COMP 2025. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer Nature: Cham, Switzerland, 2025; pp. 151–186. https://doi.org/10.1007/978-3-031-90660-2_9.
165. McGowan, C.; Richards, M.; Sui, Y. SVF-SVC: Software verification using SVF (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer Nature: Cham, Switzerland, 2025; pp. 254–259.
166. Ramegowda, D.; Lin, M. Energy efficient mixed task handling on real-time embedded systems using FreeRTOS. *J. Syst. Archit.* **2022**, *131*, 102708.
167. Silva, M.; Cerdeira, D.; Pinto, S.; Gomes, T. Operating systems for internet of things low-end devices: Analysis and benchmarking. *IEEE Internet Things J.* **2019**, *6*, 10375–10383.
168. Hahm, S.-I.; Kim, J.; Jeong, A.; Yi, H.; Chang, S.; Kishore, S.N.; Chauhan, A.; Cherian, S.P. Reliable real-time operating system for IoT devices. *IEEE Internet Things J.* **2020**, *8*, 3705–3716.
169. Zeng, X.; Yang, Y.; Wen, Q.; Qin, S. Cross-Architecture Binary Code Similarity-Detection Method Based on Contextual Information. *Appl. Sci.* **2025**, *15*, 9458.
170. Kim, G.; Hong, S.; Franz, M.; Song, D. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 151–163.
171. Rusu, A.A.; Rabinowitz, N.C.; Desjardins, G.; Soyer, H.; Kirkpatrick, J.; Kavukcuoglu, K.; Pascanu, R.; Hadsell, R. Progressive neural networks. *arXiv* **2016**, arXiv:1606.04671.
172. Wang, Y.; Yao, Q.; Kwok, J.T.; Ni, L.M. Generalizing from a few examples: A survey on few-shot learning. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–34.
173. Yang, X.; Chen, Y.; Eide, E.; Regehr, J. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*; Association for Computing Machinery: New York, NY, USA, 2011; pp. 283–294.
174. Koch, L.; Begoli, E. Adversarial binaries: AI-guided instrumentation methods for malware detection evasion. *ACM Comput. Surv.* **2025**, *57*, 1–36.
175. Chen, X.; Hu, X.; Huang, Y.; Jiang, H.; Ji, W.; Jiang, Y.; Jiang, Y.; Liu, B.; Liu, H.; Li, X.; et al. Deep learning-based software engineering: Progress, challenges, and opportunities. *Sci. China Inf. Sci.* **2025**, *68*, 111102.
176. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.