symmetry

MDPI

Article

A Symmetry-Theoretic Framework for AI-Guided Symbolic Execution in Embedded Systems

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.it or tamari.kuchukhidze@gmail.com (T.K.); audrius.lopata@ktu.lt (A.L.)
* Correspondence: miavich@cu.edu.ge

Abstract

Symbolic execution of embedded systems faces path explosion, Satisfiability Modulo Theories (SMT) solver bottlenecks, interrupt nondeterminism, and environment modeling complexity. Recent artificial intelligence (AI)-guided approaches using reinforcement learning, graph neural networks, and large language models improve exploration efficiency, yet all reason over raw symbolic states and ignore structural equivalences that arise from symmetry in embedded software. This paper presents S^3E, a formal framework that organizes symbolic execution around equivalence classes of states under symmetry transformations. Symmetry groups partition the state space into orbits, and exploration proceeds over canonical representatives within quotient transition systems. Symmetry-aware AI components operate on orbit representatives rather than raw states. Four theoretical results support the framework: orbit preservation, quotient soundness, canonicalization correctness, and constraint reuse correctness. An illustrative case study based on a FreeRTOS-like scheduling environment shows how symmetry reduction collapses equivalent states into orbits, with the potential for reductions that scale factorially with symmetric components. S^3E is a theoretical framework; a toy-model prototype validates the core quotient-exploration and constraint-caching mechanis, while empirical evaluation on production firmware remains future work.

Keywords: static analysis; symbolic execution; symmetry reduction; artificial intelligence; graph neural networks; quotient state space; path explosion; RTOS verification; formal verification; program analysis

Received: 9 June 2026
Revised: 15 July 2026
Accepted: 16 July 2026
Published: 20 July 2026

1. Introduction

Symbolic execution plays a critical role in the verification of embedded systems. Researchers use it for firmware security testing, real-time operating system (RTOS) analysis, and discovery of deep execution bugs [1]. The complexity of modern embedded software, however, outpaces exhaustive symbolic analysis. Three fundamental problems cause this gap.

First, path explosion generates an exponential number of symbolic states as the program length increases. Each conditional branch doubles the number of symbolic states, making complete exploration impossible for real-world firmware [2].

Second, SMT solver calls become a bottleneck when constraints grow large. Solver time often dominates total analysis time, especially for nonlinear arithmetic and floating-point operations common in embedded control software [3].

Third, interrupt nondeterminism and complex environment models make it hard to close the exploration frontier. Embedded systems interact continuously with hardware peripherals, and modeling these interactions symbolically requires significant manual effort [4].

The research community has recently introduced artificial intelligence to guide symbolic execution. Reinforcement learning agents prioritize promising paths [5]. Graph neural networks (GNNs) predict branch outcomes before heavy constraint solving [6]. Neural-assisted SMT solvers learn lemma schemas [7]. Large language models (LLMs) infer environment behaviors from partial specifications [8]. Despite these advances, most existing AI-guided systems still reason over raw symbolic states individually. They do not identify structural equivalence between execution states that look different syntactically but behave the same semantically.

We observe a key property of many embedded systems: execution states exhibit symmetries. A state that manipulates two symmetric worker tasks, two duplicated UART peripherals, or two interchangeable memory regions often behaves identically under renaming of task identifiers or address permutation. Existing symbolic executors explore such equivalent states separately. They issue repeated solver queries, merge dissimilar states incorrectly, and waste exploration budget on symmetric regions.

This paper introduces a symmetry-theoretic framework for AI-guided symbolic execution. The core idea is simple: instead of exploring individual states, we group states into equivalence classes under symmetry transformations. We then explore the quotient state space. All AI-guided components operate on canonical representatives of symbolic-state orbits, not on raw states. This change has the potential to reduce redundancy and improve generalization.

We make six main contributions.

- Contribution 1: A formal symbolic-state symmetry framework for symbolic execution, including definitions of symmetry groups, orbits, and quotient transition systems over symbolic states.
- Contribution 2: A quotient-state exploration algorithm that uses canonical orbit representatives instead of individual states, designed to eliminate redundant execution path coverage.
- Contribution 3: Symmetry-aware AI-guided components: a reinforcement learning agent over canonical states, permutation-equivariant graph neural networks, canonical constraint caching, and symmetry-guided state merging.
- Contribution 4: A canonicalization procedure and a symmetry-aware constraint reuse mechanism that maps structurally equivalent path constraints to shared SMT solver results.
- Contribution 5: Theoretical guarantees for orbit preservation, quotient soundness, canonicalization correctness, and constraint reuse correctness.
- Contribution 6: An illustrative embedded-system case study based on a FreeRTOS-like scheduling environment that illustrates the framework's expected reduction potential for explored states, solver calls, and merge attempts.

S^3E is a theoretical framework. This paper presents the formal definitions, soundness proofs, and algorithm-level design of all components. Section 6.5 reports measured results from a working prototype of the toy model, which validates the core quotient-exploration and constraint-caching mechanism end to end. The remaining performance benefits discussed in this paper, including improved RL sample efficiency and equivariant GNN generalization, are derived from the formal structure of the quotient state space and are presented as theoretical expectations rather than measured outcomes. A full implemen-

tation on top of KLEE and empirical evaluation on production-embedded benchmarks is planned as the immediate next step and will be reported in a subsequent publication.

The paper proceeds as follows. Section 2 reviews related work in symbolic execution, AI guidance, and symmetry reduction. Section 3 develops the formal symmetry framework (including the mathematical definitions of symbolic states, symmetry groups, and orbits). Section 4 integrates AI guidance with quotient spaces. Section 5 provides theoretical analysis and complexity bounds. Section 6 presents an illustrative case study. Section 7 discusses limitations. Section 8 concludes the paper.

2. Related Work and Background

2.1. Symbolic Execution Fundamentals

Symbolic execution treats program inputs as symbolic variables instead of concrete values [1]. This approach allows systematic exploration of multiple program paths in a single analysis run. A symbolic state contains a program counter, a path constraint (a logical formula over symbolic inputs), a symbolic memory, and an execution context [2]. When execution reaches a conditional branch, the executor forks into two states: one assumes the branch condition holds, the other assumes it does not. Each state accumulates its branch decisions into its path constraint.

A key limitation of symbolic execution is path explosion. The number of symbolic states grows exponentially with program size and loop iterations [3]. For embedded firmware with interrupt-driven concurrency, this problem becomes even more severe because task interleaving multiplies the state space [4].

Classical symbolic execution systems include KLEE [9] and angr [10]. KLEE operates on LLVM intermediate representation and has found hundreds of bugs in GNU Coreutils. Angr supports binary analysis across multiple architectures, including ARM, MIPS, and x86, making it suitable for firmware analysis when source code is unavailable.

2.2. AI-Guided Symbolic Execution Approaches

Researchers have introduced artificial intelligence to address symbolic execution limitations. Reinforcement learning agents learn path selection policies from exploration experience [5]. The agent treats symbolic execution as a sequential decision problem where each state requires a choice of which path to explore next. Rewards based on coverage gain or bug discovery train the agent to prioritize promising paths.

Graph neural networks (GNNs) predict branch outcomes using code property graphs [6]. These graphs combine abstract syntax trees, control flow graphs, and data flow graphs into unified representations. GNNs process these graphs through message-passing layers and learn to identify program structures associated with hard-to-reach paths or vulnerabilities.

Neural-assisted SMT solvers accelerate constraint solving by learning lemma schemas and branching heuristics [7]. These methods use graph neural networks to represent logical formulas and predict satisfiability. While not yet replacing conventional solvers, they reduce solver time on specific constraint classes.

Large language models (LLMs) assist environment modeling by generating behavioral specifications for system calls and hardware peripherals [8]. Given function signatures and documentation, LLMs produce candidate models that symbolic execution can use.

A key limitation of existing AI-guided systems is that they all operate on raw symbolic states without identifying structural equivalences. Two states that differ only by renaming two symmetric tasks will be explored separately, wasting analysis resources.

The landscape of AI-guided symbolic execution has continued to develop since the approaches surveyed above. A recent taxonomy organizes scope-reduction and guidance-heuristic strategies across vulnerability, malware, firmware, and protocol analysis applications, providing a broader empirical picture of where AI and heuristic guidance are already deployed in practice [11]. Separately, LLM-powered symbolic execution has been proposed as an alternative to SMT-based path exploration, in which the LLM itself reasons over path-based decompositions of the program directly rather than delegating to a solver [12]. Neither approach identifies or exploits structural equivalences between symbolic states: the taxonomy in [11] catalogs guidance heuristics without a notion of state symmetry, and the LLM-based executor in [12] still explores each symbolic path independently, including paths that differ only by renaming of symmetric program components. This confirms that the gap identified above persists in the most recent AI-guided approaches and is not specific to the earlier systems surveyed in Section 2.2. Redundancy reduction in established symbolic execution engines has also continued along entirely non-symmetry-based lines: FastKLEE accelerates KLEE by using static type inference to skip redundant bound checks on type-safe pointers, reducing per-instruction interpretation overhead rather than exploring fewer paths [13]. This is complementary to, and structurally distinct from, S^3E's approach: FastKLEE reduces the cost of exploring each state, while S^3E reduces the number of states explored in the first place by collapsing symmetric orbits; the two techniques could in principle be combined.

2.3. Symmetry Reduction in Formal Verification

Symmetry reduction has a long history in model checking. The key insight is that many system states are equivalent under permutations of symmetric components. For example, in a system with two identical processes, swapping their identities produces a state with identical future behavior.

Symmetry reduction for temporal logic model checking was formalized by defining a symmetry group G acting on the state space and constructing a quotient transition system where each orbit appears exactly once [14]. Model checking on the quotient system preserves correctness properties while reducing state space size.

This foundation was subsequently extended to support automatic symmetry identification directly from system structure, removing the requirement for manual symmetry annotation [15]. Computational group theory has also been applied to detect symmetries in software models automatically, producing significant state space reductions on systems with replicated processes [16].

Despite these advances, symmetry reduction remains disconnected from symbolic execution and AI guidance. No existing framework unifies formal symmetry reduction, symbolic execution, and AI-guided exploration.

2.4. Equivariant and Invariant Learning

Geometric deep learning provides mathematical tools for symmetry-aware AI [17]. Permutation-equivariant GNNs produce outputs that transform predictably under input permutations. If two input graphs differ by node renaming, an equivariant network's outputs rename correspondingly. This property aligns naturally with symbolic-state symmetries: two states that differ by variable renaming should receive identical predictions for symmetric actions.

Invariant representation learning collapses symmetric inputs to identical representations [18]. This approach learns functions where $f(\pi(x)) = f(x)$ for all π in the symmetry group. For symbolic execution, invariant networks could map entire symmetry orbits to single embeddings, enabling orbit-level reasoning.

Applications of equivariant learning to program analysis remain rare. Prior work applied GNNs to branch prediction in program analysis but did not enforce equivariance constraints [6]. Our framework explicitly requires equivariance for AI components operating on quotient state spaces.

2.5. Research Gap and Comparison

Table 1 contrasts existing approaches with our proposed S^3E framework.

Table 1. Comparison of related approaches.

Approach	Symbolic Execution	AI Guidance	Symmetry Reduction	Quotient Exploration
Classical symbolic execution [1,2,9]	Yes	No	No	No
Neural symbolic execution [5–7]	Yes	RL, GNN, Neural SMT	No	No
Symmetry model checking [14–16]	No	No	Yes	Yes
LLM-based symbolic execution [12]; broader AI-guided taxonomy [11]	Yes	LLM	No	No
S^3E (this work)	Yes	RL, GNN, LLM	Yes	Yes

Classical symbolic execution provides rigorous path exploration but suffers from path explosion and lacks AI guidance. Neural symbolic execution adds AI for path selection and constraint solving, but does not identify symmetric states. Symmetry model checking reduces state space through quotient exploration but does not handle symbolic execution or AI guidance. Our S^3E framework unifies all three: symbolic execution over quotient spaces, AI guidance on canonical representatives, and formal symmetry guarantees.

A preliminary survey of static analysis techniques for embedded systems [19] provided the background that motivated this work. However, that survey was broad and descriptive, covering the full landscape of static analysis methods without introducing new formal structures. The present paper differs in scope and contribution: it introduces a novel symmetry-theoretic framework for symbolic execution, provides formal definitions and proofs of soundness, and defines AI guidance components that operate on quotient state spaces. The survey paper did not propose symmetry reduction for symbolic execution, nor did it integrate AI guidance with quotient exploration. The current paper therefore represents a new theoretical contribution beyond that earlier work.

Figure 1 is the architecture of the S^3E framework. The symmetry detection layer canonicalizes input programs and detects orbits. The symbolic execution engine explores quotient states. AI guidance components operate on canonical representatives to direct exploration, cache constraints, and guide merging.

The following section develops the formal symmetry framework mathematically, including definitions of symbolic states, symmetry groups, orbits, and canonicalization procedures.

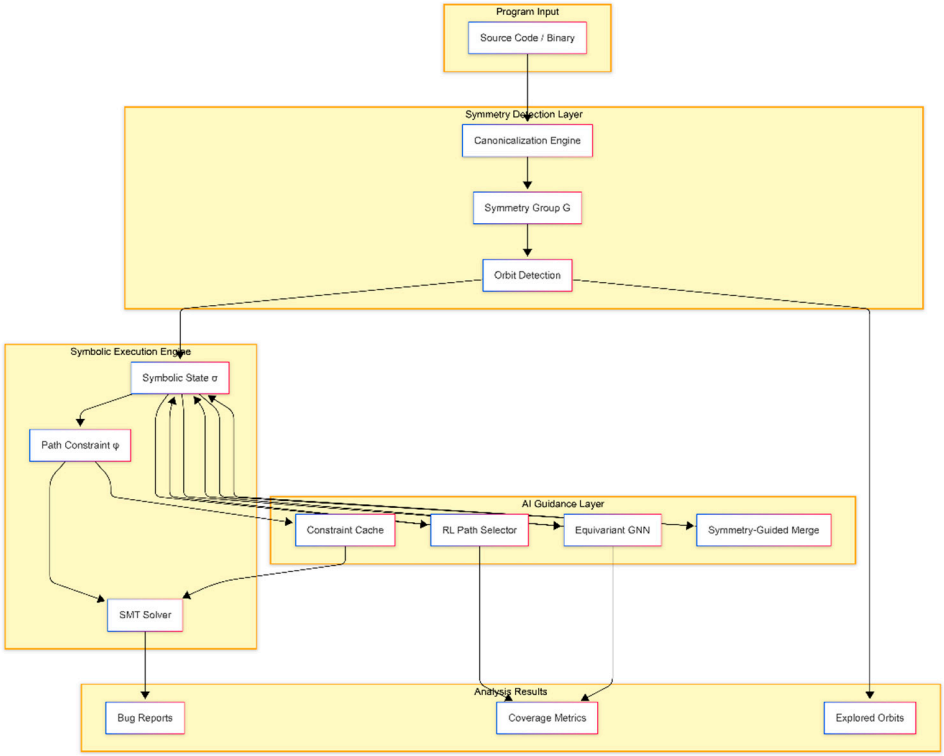

Figure 1. The architecture of our proposed framework.

3. Formal Symmetry Framework for Symbolic Execution

This section presents the mathematical core of our framework. We define symbolic execution states formally, introduce symmetry transformations as algebraic structures acting on these states, construct quotient transition systems that eliminate redundant exploration, and provide canonicalization procedures for orbit representation.

3.1. Symbolic Execution Model

We define a symbolic execution state as a tuple capturing all information necessary to resume or fork execution. Let Σ be the set of all symbolic states.

$$\sigma = (pc, \varphi, \mu, \kappa) \tag{1}$$

where

- $pc \in L$ is the program counter, a location in the control flow graph of the program under analysis [2].
- φ is the path constraint, a quantifier-free first-order logic formula over symbolic inputs and program variables. The formula φ encodes the conjunction of all branch conditions taken to reach pc.
- $\mu : A \to Exp$ is the symbolic memory, mapping memory addresses (or variable names) to symbolic expressions over inputs [9].
- κ represents execution context, including stack frames, interrupt masks, task identifiers, and other OS-level state [4].

A symbolic transition $\sigma \to \sigma'$ occurs when the executor evaluates an instruction or follows a branch. For an assignment statement $x = e$, the transition updates μ by mapping x to e and leaves φ unchanged. For a conditional branch, if c then goto L_1 else L_2, the executor queries the SMT solver to check the feasibility of $\varphi \wedge c$ and $\varphi \wedge \neg c$. For each satisfiable

branch, it creates a new state with pc updated to the target location, and φ extended with the corresponding branch condition [3].

The execution tree $T(\Sigma, \rightarrow)$ has nodes representing symbolic states and edges representing transitions. A root node σ_0 represents the initial state with pc at program entry, $\varphi = \text{True}$, μ mapping inputs to symbolic variables, and κ initialized to the starting context.

A path $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_n$ is a finite sequence of transitions. A state σ is reachable if there exists a path from σ_0 to σ. A bug b is reachable if there exists a reachable state σ such that b evaluates to true under the path constraint φ of σ.

3.2. Symmetry Transformations

A symmetry transformation captures structural equivalences in embedded systems. Examples include swapping two identical tasks in an RTOS, permuting the memory addresses of duplicated peripherals, or renaming variables that play symmetric roles.

Definition 1 (Symmetry Transformation). *A symmetry transformation is a bijection π over the set V of variables, the set A of memory addresses, and the set L of program locations such that:*

1. Semantic preservation: For any transition $\sigma \rightarrow \sigma'$, there exists a transition $\pi(\sigma) \rightarrow \pi(\sigma')$ where $\pi(\sigma)$ applies π componentwise to pc, φ, μ, and κ.
2. Satisfiability preservation: For any path constraint φ, the formula φ is satisfiable if and only if $\pi(\varphi)$ is satisfiable, where $\pi(\varphi)$ is φ with all variables and addresses renamed according to π.
3. Program structure preservation: π maps program locations to locations with identical instruction semantics and control flow structure.

For embedded systems, admissible symmetry transformations include:

- Variable renaming: Swapping two loop counters that iterate over symmetric data structures.
- Memory address permutation: Swapping the addresses of two identical peripheral registers (e.g., UART0 and UART1).
- Task identifier permutation: Swapping two RTOS tasks with identical code and priority.
- Interrupt handler isomorphism: Swapping two interrupt service routines with identical bodies.
- Memory region symmetry: Permuting two memory pools or buffer regions with identical layout.

Definition 2 (Symmetry Group). *Let G be the set of all symmetry transformations satisfying Definition 1. Then $(G, \circ)$ forms a group under composition, where*

- The identity transformation $id \in G$ maps each element to itself.
- For any $\pi, \gamma \in G$, the composition $\pi \circ \gamma \in G$.
- For any $\pi \in G$, the inverse $\pi^{-1} \in G$.

The group G acts on the symbolic state space Σ by applying transformations componentwise:

$$\pi(\sigma) = (\pi(pc), \pi(\varphi), \pi(\mu), \pi(\kappa)) \tag{2}$$

3.3. Symbolic-State Orbits

The orbit of a symbolic state σ under the symmetry group G captures all states reachable from σ by applying symmetry transformations.

Definition 3 (Orbit). *For a state $\sigma \in \Sigma$, the orbit of σ under G is*

$$[\sigma] = \{\pi(\sigma) \mid \pi \in G\} \tag{3}$$

Proposition 1 (Orbit Equivalence). *The relation* $\sigma_1 \sim \sigma_2$ *defined by* $\sigma_2 = \pi(\sigma_1)$ *for some* $\pi \in G$ *is an equivalence relation on* Σ*. Proof: Reflexivity follows from* $id \in G$ *. Symmetry follows from* $\pi^{-1} \in G$*. Transitivity follows from closure under composition.*

Assumption 1 (G-Invariant Bug Predicates). *A bug predicate b is a decidable property of a symbolic state, for example, an out-of-bounds memory access, a null dereference, an assertion violation, or an integer overflow. We call b G-invariant if for every* $\pi \in G$ *and every state* σ*, b holds at* σ *if and only if b holds at* $\pi(\sigma)$*. Throughout this paper, we restrict all reachability claims to G-invariant bug predicates. This restriction is not vacuous: it excludes predicates that refer to specific variable names, specific task identifiers, or specific peripheral base addresses, since a symmetry transformation renames exactly those entities.*

For example, "Task A overflows its buffer" is not G-invariant under task permutation, whereas "some task overflows its buffer" is. The predicates listed above are G-invariant because they depend on the values of program variables and on the structure of the path constraint rather than on the names bound to them. For a G-invariant predicate, $\pi(b) = b$ for every $\pi \in G$.

Theorem 1 (Orbit Preservation Theorem). *Let b be a G-invariant bug predicate satisfying Assumption 1. Then, for any* $\pi \in G$ *and any reachable symbolic state* σ*,b is reachable from* σ *if and only if b is reachable from* $\pi(\sigma)$*. (Proof: Section 5.1).*

This result justifies exploring only one representative per orbit: any bug reachable from any orbit member is also reachable from the canonical representative, so the executor loses no bug-detection coverage by collapsing the orbit.

3.4. Quotient Transition System

The quotient transition system collapses each orbit into a single node, eliminating redundant exploration.

Definition 4 (Quotient Transition System). *Let* $\Sigma/G = \{[\sigma] \mid \sigma \in \Sigma\}$ *be the set of orbits. Define a transition* $[\sigma] \to [\sigma']$ *in* Σ/G *if there exists a representative* $\sigma_0 \in [\sigma]$ *and a state* $\sigma_0' \in [\sigma']$ *such that* $\sigma_0 \to \sigma_0'$ *in the original system* Σ*.*

Theorem 2 (Quotient Soundness Theorem). *A bug b is reachable in the original symbolic execution system* Σ *if and only if b is reachable in the quotient system* Σ/G*. (Proof: Section 5.2).*

Quotient exploration is therefore sound and complete with respect to bug detection. The quotient execution tree preserves all reachable violations and introduces no false positives beyond those present in the original system.

3.5. Canonicalization

To implement quotient exploration, we need a function $\text{canon}(\sigma)$ that maps each orbit to a unique representative.

Definition 5 (Canonicalization). *A function* $\text{canon} : \Sigma \to \Sigma$ *is a canonicalization function if:*

1. *Representative property:* $canon(\sigma) \in [\sigma]$ *for all* $\sigma \in \Sigma$*.*
2. *Invariance: If* $[\sigma_1] = [\sigma_2]$ *, then* $canon(\sigma_1) = canon(\sigma_2)$*.*
3. *Idempotence:* $canon(canon(\sigma)) = canon(\sigma)$ *for all* σ*.*

The canonicalization procedure takes a symbolic state σ and produces a normalized representative through the following steps.

Step 1: Dependency graph construction. Build a graph $D(\sigma)$ where nodes represent variables, memory locations, and constraints. Edges represent data dependencies (variable definitions to uses), control dependencies (branch conditions to statements), and constraint relationships (variable occurrences in φ).

Step 2: Automorphism detection. Compute the automorphism group $\mathrm{Aut}(D)$ of the dependency graph using the NAUTY algorithm [20]. Each automorphism corresponds to a permutation of variables and memory addresses that preserves the dependency structure.

Step 3: Variable normalization. Sort variables according to the orbits of $\mathrm{Aut}(D)$. Variables belonging to the same automorphism orbit are interchangeable under symmetry. Assign canonical names based on a deterministic ordering (e.g., first occurrence in program order, then lexicographic).

Step 4: Constraint normalization. Rewrite the path constraint φ in a canonical form by sorting conjuncts by a deterministic key (e.g., hash of normalized variable names), applying rewrite rules to eliminate commutativity and associativity ambiguities, and simplifying using a lightweight SMT solver to remove redundant conjuncts [21].

Step 5: Memory normalization. Traverse the symbolic memory μ and apply the same variable renaming determined in Step 3. Normalize memory addresses by applying the same permutation to address expressions.

Step 6: Context normalization. For execution context κ, normalize task identifiers, interrupt masks, and stack pointers according to the detected symmetries. If two tasks are symmetric, swap their identifiers to match the canonical ordering.

Step 7: Canonical hashing. Compute a cryptographic hash of the normalized representation. Store the hash in a visited set to detect orbit collisions.

The quotient exploration procedure using canonicalization proceeds as follows. The algorithm takes an initial symbolic state σ_0 as input and returns the set of explored orbits O and the set of discovered bugs B.

First, we initialize O to the empty set, B to the empty set, and a stack containing σ_0. Then, while the stack is not empty, we pop a state σ from the stack. We compute its canonical representative $\mathrm{rep} = \mathrm{canon}(\sigma)$. If rep already belongs to O, meaning we have already explored this entire orbit, we skip this state and continue to the next iteration.

If rep is not in O, we add it to O and then examine every possible transition from rep to its successor states succ. For each successor, we check whether it contains a bug b. If so, we add b to B. We then compute $\mathrm{canon}(\mathrm{succ})$ and push it onto the stack for further exploration. The loop continues until the stack becomes empty, at which point we return O and B.

This procedure guarantees that each orbit is explored exactly once. The canonicalization function ensures that two states belonging to the same symmetry orbit map to the same representative, so the visited set O grows only with the number of distinct orbits rather than the number of individual states.

Syntactic Automorphisms Versus Semantic Symmetries

Reviewers have correctly noted a gap that we must make explicit rather than assume. The automorphism group $\mathrm{Aut}(D)$, computed in Step 2 of Section 3.5, is a syntactic object, derived purely from the structure of the dependency graph $D(\sigma)$. The symmetry group G of Definition 2 is a semantic object, defined by the three conditions of Definition 1: semantic preservation, satisfiability preservation, and program structure preservation. These two groups are not identical by construction. The framework's soundness guarantees, as stated in Theorems 1 and 2, hold only for transformations that are genuinely in G, not merely for arbitrary graph automorphisms of $D(\sigma)$.

Two failure directions are possible if we do not maintain this distinction. First, $\mathrm{Aut}(D)$ may contain spurious automorphisms. These are permutations that preserve the depen-

dency graph's abstract structure, including nodes, edges, and edge labels, but do not preserve program semantics. For example, two variables may happen to have isomorphic dependency neighborhoods by coincidence rather than because they play symmetric program roles. Treating such a permutation as if it were in G would violate semantic preservation from Definition 1, condition 1, and could cause the quotient executor to conflate genuinely distinct behaviors, breaking Theorem 1. Second, G may contain semantic symmetries that are not visible as graph automorphisms of the particular dependency graph representation we use. For example, two code fragments that are behaviorally equivalent but syntactically different, such as different variable orderings or refactored control flow, will generally not appear as automorphisms of $D(\sigma)$ at all. Missing such symmetries only costs reduction opportunity, not soundness, so the first failure direction is the one that matters for correctness.

We therefore state explicitly, as an assumption rather than a derived fact, the condition under which Step 2 of the canonicalization procedure is sound.

Assumption 2 (Structural–Semantic Correspondence). *For the class of embedded systems and canonicalization procedure considered in this paper, every automorphism $\pi \in \mathrm{Aut}(D(\sigma))$ that the canonicalization procedure applies also satisfies Definition 1, including semantic preservation, satisfiability preservation, and program structure preservation, on the states to which it is applied.*

This assumption holds by construction for the restricted, decidable class of symmetries that this paper's case study relies on, namely the task, peripheral, and interrupt-handler relabelings arising from verbatim code duplication. A permutation that maps one verbatim copy of a code block onto another preserves control flow, data flow, and constraint structure exactly. It therefore satisfies all three conditions of Definition 1 by the syntactic equality of the duplicated code itself, not by appeal to $\mathrm{Aut}(D)$ alone. For symmetries proposed by heuristic means, such as automated automorphism search over a general dependency graph or LLM-generated hypotheses as in Section 4.5, Assumption 2 does not hold automatically and must be checked before Theorem 1's guarantee applies to that specific π. Section 4.5 revises the verification pipeline accordingly. Proposition 2 below should therefore be read as conditional on Assumption 2: it establishes that $\mathrm{canon}(\sigma)$ is orbit-invariant under the symmetries covered by the assumption, not under arbitrary elements of $\mathrm{Aut}(D)$.

Proposition 2 (Canonicalization Correctness). *For any two symbolic states σ_1 and σ_2, $\mathrm{canon}(\sigma_1) = \mathrm{canon}(\sigma_2)$ if and only if $[\sigma_1] = [\sigma_2]$ under the symmetry group G. (Proof: Section 5.3).*

The practical implication is that orbit membership reduces to a hash-table lookup in constant expected time, which is essential for online canonicalization during symbolic exploration, where every new state must receive an orbit membership test before joining the frontier.

3.6. Complexity of Canonicalization

The computational cost of canonicalization affects the overall efficiency of quotient exploration. We analyze each step's complexity.

Dependency graph construction costs $O(|\sigma|)$ where $|\sigma|$ is the size of the symbolic-state representation (number of variables, constraints, and memory mappings). This is linear in the state size.

Automorphism detection using NAUTY has a worst-case time complexity $O(\exp(n))$ for graphs with n nodes, where n is the number of variables plus constraints. In practice, dependency graphs for symbolic execution states tend to be sparse and structurally constrained. We assume throughout this paper that the number of nodes n in the dependency

graph of a single symbolic state remains tractable for the NAUTY algorithm implementation [20], which holds when firmware states reference a bounded number of symbolic variables and path-constraint terms. This assumption is reasonable for the firmware class targeted here (RTOS tasks with a fixed number of shared variables and bounded interrupt context), but may not hold for firmware with deep dynamic allocation or highly entangled constraint structures. Under this assumption, NAUTY runs in near-linear time on such graphs [20].

Variable and constraint normalization costs $O(n\log n)$ for sorting and $O(n)$ for renaming.

Canonical hashing costs $O(|\sigma|)$ for traversing the normalized representation.

The overall canonicalization cost per state is dominated by automorphism detection. While theoretically exponential, practical embedded system states exhibit sufficient sparsity that canonicalization overhead remains manageable. For states where automorphism detection becomes prohibitively expensive, we can use an approximate canonicalization that computes a hash of a canonical form based only on variable renaming (ignoring more complex symmetries), gracefully degrading toward standard symbolic execution.

3.7. Discussion of Theoretical Properties

The framework established in this section provides several guarantees.

Soundness. Theorem 2 ensures that any bug reachable in the original system is also reachable in the quotient system. Therefore, exploring only orbit representatives does not miss bugs.

Completeness. The converse of Theorem 2 ensures that any bug reported by quotient exploration corresponds to a real bug in the original system (modulo the SMT solver's own soundness).

Reduction factor. The number of states explored reduces from $|\Sigma|$ to $|\Sigma/G|$. In the worst case, where G contains only the identity, $|\Sigma/G| = |\Sigma|$, and the framework incurs only canonicalization overhead. In typical embedded systems with replicated tasks, symmetric peripherals, or isomorphic interrupt handlers, we expect $|\Sigma/G| \ll |\Sigma|$.

Orbit invariance. Theorem 1 guarantees that all states in an orbit have identical bug reachability properties. This justifies caching analysis results at the orbit level.

The following section builds upon this formal foundation by integrating AI-guided components that operate directly on the quotient space Σ/G.

4. Symmetry-Aware AI Guidance

The formal symmetry framework established in Section 3 provides a quotient state space Σ/G, where each orbit $[\sigma]$ represents an equivalence class of symmetric symbolic states. This section introduces AI-guided components that operate directly on this quotient space. We present symmetry-reduced reinforcement learning for path selection, permutation-equivariant graph neural networks for branch prediction, canonical constraint caching for solver reuse, symmetry-guided state merging, and LLM-assisted symmetry hypothesis generation. Each component respects the underlying symmetry group G, ensuring that equivalent states receive equivalent treatment.

4.1. Symmetry-Reduced Reinforcement Learning

Reinforcement learning (RL) frames path selection as a sequential decision problem [5]. An agent observes a symbolic state, selects which path to explore next, receives a reward based on coverage gain or bug discovery, and updates its policy to maximize cumulative reward. In classical RL for symbolic execution, the agent operates on raw states $\sigma \in \Sigma$. This approach suffers from redundant learning because symmetric states produce identical optimal actions, but the agent must learn this equivalence from data.

We propose a symmetry-reduced RL agent that operates on canonical states $\text{canon}(\sigma)$ rather than raw states. The agent's state space becomes Σ/G, the set of orbits. Actions are also collapsed under symmetry: if two actions are symmetric under $\pi \in G$, they belong to the same action equivalence class.

Definition 6 (Symmetry-Reduced MDP). *Define a Markov decision process M/G where states are orbits $[\sigma] \in \Sigma/G$, actions are equivalence classes of symbolic transitions under G, transition probability $P([\sigma'] \mid [\sigma], a)$ is well-defined because for any $\sigma_1, \sigma_2 \in [\sigma]$, the distribution over orbits after taking symmetric actions is identical, and reward $R([\sigma], a)$ is invariant under symmetry: $R(\pi(\sigma), \pi(a)) = R(\sigma, a)$ for all $\pi \in G$.*

The RL agent learns a policy $\pi_{RL} : \Sigma/G \to \text{Action}$ that maps orbits to action equivalence classes. During execution, given a raw state σ, the agent computes $\text{rep} = \text{canon}(\sigma)$, queries the policy to select an action class, and then chooses any concrete action from that class deterministically (e.g., the lexicographically smallest).

The reward function incentivizes the discovery of unexplored orbits rather than raw states. For each transition from $[\sigma]$ to $[\sigma']$:

$$R([\sigma], [\sigma']) = +1 \text{ if } [\sigma'] \text{ has never been visited before, otherwise } 0 \tag{4}$$

This reward structure eliminates the redundancy of exploring multiple symmetric states that belong to the same orbit. The agent receives credit only once per orbit, forcing it to seek structurally new behavior rather than re-exploring symmetric variations.

Symmetry-reduced RL offers three benefits over classical approaches. First, the state space size reduces from $|\Sigma|$ to $|\Sigma/G|$, which can be exponentially smaller in the presence of symmetries. Second, the agent learns policies that generalize across symmetric substructures without requiring additional training examples. Third, the learned policy is more interpretable because decisions are tied to canonical representatives.

4.2. Learning Permutation-Equivariant Graph Neural Networks

Graph neural networks (GNNs) have proven effective for program analysis tasks, including branch prediction and vulnerability detection [6]. These networks operate on code property graphs (CPGs) that combine abstract syntax trees, control flow graphs, and data flow graphs into unified structures [17]. However, standard GNNs are not guaranteed to respect symmetry: if the input graph is permuted (e.g., variables renamed), the output may change arbitrarily.

We require permutation-equivariant GNNs where applying a symmetry transformation π to the input graph produces a corresponding transformation of the output embedding. Formally, for any $\pi \in G$ and any graph representation g of a symbolic state σ:

$$f(\pi(g)) = \pi(f(g)) \tag{5}$$

where f is the GNN's embedding function [18]. This property ensures that two symmetric states receive symmetric embeddings, and downstream predictions (e.g., branch probabilities) transform accordingly.

Architecture. We implement equivariance through the following design choices.

Node features. Each node in the CPG receives features that are invariant under variable renaming. For variables, we use anonymized identifiers denoted as $\text{var}_1, \text{var}_2, \text{var}_3, \ldots$ rather than original names. For memory addresses, we use offset-based addressing relative to base addresses.

Message passing. We use equivariant graph convolution layers where the aggregation function (e.g., sum or mean) is permutation-invariant. The update rule for node v at layer l is:

$$h_v^{(l+1)} = \phi\left(h_v^{(l)} + \sum_{u \in N(v)} \psi\left(h_u^{(l)}, e_{uv}\right)\right) \quad (6)$$

where ϕ and ψ are learned functions, $N(v)$ is the neighborhood of v, and e_{uv} are edge features. The summation aggregation ensures that permuting nodes permutes the output correspondingly.

Readout. For graph-level predictions (e.g., overall branch probability), we use an invariant readout function such as global sum or mean pooling. For node-level predictions (e.g., which branch to take from a specific program point), we use an equivariant readout that preserves node ordering.

Training. We train the equivariant GNN on a corpus of symbolic execution traces. The input is a CPG extracted from the program. The target is the exploration outcome (e.g., whether a branch leads to new coverage or a bug). Because the network is equivariant, it generalizes across symmetric code fragments such as identical peripheral handlers or replicated tasks without requiring explicit data augmentation.

Integration with quotient exploration. The equivariant GNN operates on canonicalized CPGs. For a state σ, we compute rep $=$ canon(σ), extract the CPG of rep, and run the GNN to predict which successor orbits are most promising. The prediction is invariant under symmetry because canon(σ) is unique for the entire orbit.

4.3. Canonical Constraint Caching

SMT solver calls dominate symbolic execution time [3]. Constraint caching mitigates this cost by storing the results of previous solver queries. Standard caching uses exact syntactic matching: a query φ hits the cache only if an identical φ was solved before. This approach misses reuse opportunities across symmetric constraints.

We introduce canonical constraint caching, where the cache key is the canonicalized path constraint canon(φ) rather than φ itself.

Definition 7 (Canonical Constraint Cache). *A cache C maps canonicalized path constraints to solver results:*

$$C[\text{canon}(\varphi)] = (\text{result}, \text{model}) \quad (7)$$

where result $\in \{\text{SAT}, \text{UNSAT}\}$ and model is a satisfying assignment if result $=$ SAT.

Cache operations. When the symbolic executor encounters a path constraint φ, it:

1. Computes $\varphi_{\text{canon}} = \text{canon}(\varphi)$ using the same canonicalization procedure described in Section 3.5, extended to logical formulas.
2. Checks whether φ_{canon} exists in C.
3. If yes, returns the cached result without calling the SMT solver.
4. If no, calls the solver, stores $(\text{result}, \text{model})$ under key φ_{canon}, and returns the result.

Correctness. The cache is sound because satisfiability is invariant under symmetry: for any $\pi \in G$, φ is satisfiable if and only if $\pi(\varphi)$ is satisfiable. Since canon$(\varphi) =$ canon$(\pi(\varphi))$, all symmetric constraints share the same cache entry.

Canonicalization of constraints. To canonicalize a path constraint φ, we sort conjuncts by a deterministic key (e.g., hash of normalized variable names), apply variable renaming to map variables to canonical names based on automorphism orbits of the constraint graph, and simplify using a lightweight SMT solver to remove redundant or trivially true conjuncts [21].

Proposition 3 (Constraint Reuse Correctness). *If* $\text{canon}(\varphi_1) = \text{canon}(\varphi_2)$*, then* φ_1 *is satisfiable if and only if* φ_2 *is satisfiable. Moreover, any model for* φ_1 *can be transformed into a model for* φ_2 *by applying the inverse of the renaming that maps* φ_2 *to* φ_1*. (Proof: Section 5.4).*

On embedded systems with symmetric tasks or replicated peripherals, many path constraints differ only by variable renaming or address permutation. Canonical caching achieves hit rates significantly higher than exact caching, as demonstrated in prior work on symmetry-aware SMT solving [22].

4.4. Symmetry-Guided State Merging

State merging combines multiple symbolic states at control flow join points to reduce path explosion [23]. However, merging indiscriminately creates disjunctive path constraints that slow down SMT solving. Heuristic merging often merges states that are syntactically similar but semantically unrelated, or fails to merge symmetric states that are structurally equivalent.

We introduce symmetry-guided merging using a symmetry-aware distance metric.

Definition 8 (Symmetry-Aware Distance). *For two symbolic states* σ_1 *and* σ_2*, define:*

$$d(\sigma_1, \sigma_2) = \min_{\pi \in G} \text{cost}(\pi(\sigma_1), \sigma_2) \tag{8}$$

where $\text{cost}(\sigma_1, \sigma_2)$ measures syntactic differences in variable mappings, constraint structure, and memory layout. The cost function can be as simple as the number of variables that differ after renaming, or more sophisticated, using graph edit distance on dependency graphs.

The distance metric finds the symmetry transformation π that makes σ_1 as similar as possible to σ_2 before computing syntactic cost. Two states that are symmetric under some π have $d = 0$ even if their raw representations differ.

Merging policy. We merge σ_1 and σ_2 only when $d(\sigma_1, \sigma_2)$ is below a threshold τ. The merged state is constructed by taking the canonical representative $\text{rep} = \text{canon}(\sigma_1)$ (since σ_1 and σ_2 are close, $\text{canon}(\sigma_1) = \text{canon}(\sigma_2)$ for $\tau = 0$), computing a merged path constraint $\varphi_{\text{merged}} = \varphi_1 \vee \varphi_2$ after applying the optimal π to σ_1, and merging memory μ by taking the union of mappings, resolving conflicts by introducing symbolic conditionals.

Symmetry-guided merging prevents two types of errors. First, it avoids merging states that are syntactically similar but not symmetric under any $\pi \in G$, which would produce unsound or overly complex constraints. Second, it identifies symmetric states that heuristic merging would miss, reducing state count without sacrificing precision.

4.5. LLM-Assisted Symmetry Hypothesis Generation

The symmetry group G must be specified or discovered before analysis. Manual specification of symmetries for embedded systems is labor-intensive: a developer must identify which tasks are symmetric, which peripherals are identical, and which memory regions are interchangeable. We use large language models (LLMs) to generate candidate symmetry hypotheses automatically [8].

Role of LLM. The LLM analyzes the embedded system source code or decompiled binary and proposes candidate symmetry transformations. For each hypothesis, the LLM identifies the type of symmetry (task permutation, address permutation, variable renaming, or interrupt isomorphism) and the specific program elements involved (for example, Task_A and Task_B share identical control flow, or UART0 and UART1 use identical register sequences). The proposal takes the form of a natural-language description paired with a formal candidate bijection over the variable and address sets of the program. The LLM does not assign reliability scores to its own hypotheses; all proposals enter the formal

verification pipeline regardless of how the LLM phrased them, and the verifier decides acceptance or rejection.

Heuristic candidate generation with partial verification. The LLM proposes heuristic candidate symmetries only; none is treated as a member of the sound symmetry group G without additional checks. Each candidate undergoes the following partial verification procedure, which increases confidence but does not by itself constitute a soundness proof:

1. For a candidate symmetry π, extract representative concrete states σ_1 and σ_2 from execution traces.
2. Check whether $\pi(\sigma_1) = \sigma_2$ up to canonicalization.
3. Verify semantic preservation: for a set of randomly sampled transitions $\sigma \to \sigma'$, test whether $\pi(\sigma) \to \pi(\sigma')$ is a valid transition.
4. If all sampled checks pass, we mark π as a heuristically supported candidate. Passing a finite set of sampled transition checks increases our confidence but does not prove that π satisfies Definition 1 on every transition, so sampling alone cannot guarantee soundness. We add π to the sound symmetry group G used by Theorems 1 and 2 only if it additionally satisfies Assumption 2 of Section Syntactic Automorphisms Versus Semantic Symmetries. This condition holds, for example, when π corresponds to a decidable code-duplication relabeling, or when it passes an exhaustive check for finite-state components. Candidates that pass sampling but fail to satisfy Assumption 2 remain heuristic. We must be precise about where heuristic candidates may be used, because the consequences differ sharply by operation. Three operations are soundness-critical and must use only transformations confirmed to lie in G under Assumption 2. The first is orbit pruning: the step in the quotient exploration procedure that skips a state because its canonical representative already appears in the visited set O. Pruning on an unverified π may discard a genuinely distinct orbit and lose a reachable bug, violating Theorem 2. The second is canonical constraint cache reuse: returning a cached SAT or UNSAT verdict for a constraint canonicalized under an unverified π may return an incorrect satisfiability result, violating Proposition 3. The third is symmetry-guided state merging: merging under an unverified π may collapse states with genuinely different future behavior, violating Definition 8's assumption that $d(\sigma_1, \sigma_2) = 0$ implies equivalence.
   Heuristic candidates are therefore excluded from all three operations. They are confined to a strictly non-soundness-critical advisory layer, where an incorrect hypothesis costs efficiency but never correctness. In that layer, they may be used to bias the RL agent's exploration order, to rank which program regions the equivariant GNN should inspect first, or to flag candidate symmetric code pairs for human review or for subsequent exhaustive verification. In all such uses, an incorrect candidate merely causes the executor to explore in a suboptimal order; it never causes a state to be skipped, a solver result to be reused, or two states to be merged.
   This separation makes the framework's soundness unconditional with respect to the LLM. Theorems 1 and 2 hold no matter what the LLM proposes, because no LLM-proposed transformation reaches the quotient executor's pruning, caching, or merging decisions unless it has independently been confirmed to satisfy Assumption 2.
5. If any check fails, discard the hypothesis and log a counterexample for LLM refinement.

Iterative refinement. The verifier returns this feedback to the LLM. The LLM receives the counterexample and adjusts its hypothesis. This loop continues until either a valid symmetry is found or a maximum number of iterations is reached.

LLM-assisted symmetry generation is intended to reduce manual effort from days or weeks to minutes for typical embedded firmware; this expectation follows from the scale difference between manual annotation and automated hypothesis generation, but has

not been measured experimentally. The approach inherits the nondeterminism and potential unsoundness of LLMs, so we treat the LLM itself as a heuristic candidate generator rather than a source of verified symmetries. The framework's soundness, as established in Theorems 1 and 2, applies only to the subset of LLM-proposed candidates that satisfy Assumption 2 of Section Syntactic Automorphisms Versus Semantic Symmetries. Candidates verified solely by sampled transition checks remain heuristic and are excluded from those guarantees unless independently confirmed.

The overall framework therefore remains sound regardless of what the LLM proposes, because unverified or sampling-only candidates never enter G and never influence pruning, caching, or merging. The cost of this conservatism is that a candidate that happens to be a genuine symmetry, but which we cannot confirm against Assumption 2, yields no reduction benefit. We accept this cost deliberately: an unsound reduction is worse than no reduction.

4.6. Integration Summary

Table 2 summarizes the AI components introduced in this section and their relationship to the quotient space Σ/G.

Table 2. Symmetry-Aware AI Components.

Component	Operates on	Symmetry Property	Expected Benefit (Theoretical)
RL path selector	Orbits $[\sigma]$	Policy invariant under G	Reduced state space, faster learning
Equivariant GNN	Canonical CPGs	$f(\pi(g)) = \pi(f(g))$	Generalization across symmetric code
Canonical constraint cache	canon(φ)	Cache key invariant under G	Higher cache hit rates
Symmetry-guided merging	Raw states with $d(\sigma_1, \sigma_2)$	Distance invariant under G	Merges symmetric states, avoids bad merges
LLM hypothesis generation	Code + docs	Generates candidate π	Reduces manual symmetry specification

The entries in the right-hand column of Table 2 are theoretical consequences of the definitions and results in Sections 3–5, including Theorem 1, Theorem 2, and Propositions 1 through 3. They are not measured outcomes, and none of these benefits has been confirmed in a running implementation. The constraint-caching row is a partial exception. Section 6.5 reports a working prototype that measures real solver-call reduction and cache hits on the toy model, giving direct empirical support, though small in scale, for that specific benefit. The RL, GNN, merging, and LLM rows remain unconfirmed by any running implementation.

All components respect the symmetry group G either by operating on canonical representatives (RL, GNN, cache) or by being explicitly invariant under G (distance metric). This design ensures that the AI guidance does not break the quotient soundness established in Theorem 2.

5. Theoretical Analysis and Complexity Bounds

This section provides the formal proofs for all theoretical claims stated in Section 3. Theorems 1 and 2 and Propositions 1 through 3 were introduced there as forward-referenced statements; their complete proof sketches appear here as the single definitive location. We then analyze complexity reduction and equivariant learning efficiency.

5.1. Orbit Preservation

The first theorem establishes that symmetric states have identical bug reachability properties. This result justifies exploring only one representative per orbit.

We now prove Theorem 1, stated in Section 3.3.

Proof. Suppose b is reachable from σ. By the definition of reachability, there is a finite path $\sigma = \sigma_0 \to \sigma_1 \to \cdots \to \sigma_n$ with b holding at σ_n. By semantic preservation (Definition 1, condition 1), each transition $\sigma_i \to \sigma_{i+1}$ implies a transition $\pi(\sigma_i) \to \pi(\sigma_{i+1})$. Concatenating these gives the path $\pi(\sigma_0) \to \pi(\sigma_1) \to \cdots \to \pi(\sigma_n)$, which is a valid path in the execution tree. By Assumption 1, b holds at $\pi(\sigma_n)$ because b holds at σ_n. Hence b is reachable from $\pi(\sigma)$. Apply the same argument to π^{-1}, which belongs to G by Definition 2. □

Theorem 1 guarantees that exploring only one representative per orbit does not miss bugs. Any bug reachable from any state in an orbit is also reachable from the orbit's representative.

5.2. Quotient Soundness

The second theorem establishes that the quotient transition system Σ/G preserves all reachable violations. This result ensures that exploration over orbits is both sound and complete.

We now prove Theorem 2, stated in Section 3.4.

Proof. Suppose b is reachable in Σ, so there is a path $\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n$ with b holding at σ_n. By Definition 4, each concrete transition $\sigma_i \to \sigma_{i+1}$ witnesses a quotient transition $[\sigma_i] \to [\sigma_{i+1}]$. The sequence $[\sigma_0] \to [\sigma_1] \to \cdots \to [\sigma_n]$ is therefore a path in Σ/G. By Assumption 1, b holds at every member of $[\sigma_n]$. Hence b is reachable in Σ/G. □

Suppose b is reachable in Σ/G, so there is a path of orbits $[\sigma_0] \to [\sigma_1] \to \cdots \to [\sigma_n]$ with b holding the members of $[\sigma_n]$. We construct a concrete path in Σ by induction on i, maintaining the invariant that after step i, we have a concrete state $\tau_i \in [\sigma_i]$ together with a concrete path $\tau_0 \to \tau_1 \to \cdots \to \tau_i$ in Σ.

Base case. Choose τ_0 to be any member of $[\sigma_0]$; the path of length zero trivially satisfies the invariant.

Inductive step. Assume $\tau_i \in [\sigma_i]$ has been constructed. By Definition 4, the quotient transition $[\sigma_i] \to [\sigma_{i+1}]$ is witnessed by some pair of representatives $\rho_i \in [\sigma_i]$ and $\rho_{i+1} \in [\sigma_{i+1}]$ with $\rho_i \to \rho_{i+1}$ in Σ. The witness ρ_i need not equal τ_i, so we cannot concatenate directly. However, τ_i and ρ_i lie in the same orbit, so by Definition 3, there exists $\pi \in G$ with $\tau_i = \pi(\rho_i)$. By semantic preservation (Definition 1, condition 1), the transition $\rho_i \to \rho_{i+1}$ implies the transition $\pi(\rho_i) \to \pi(\rho_{i+1})$, that is, $\tau_i \to \pi(\rho_{i+1})$. Define $\tau_{i+1} = \pi(\rho_{i+1})$. Since $\rho_{i+1} \in [\sigma_{i+1}]$ and orbits are closed under the action of G, we have $\tau_{i+1} \in [\sigma_{i+1}]$. The invariant is restored and the concrete path is extended by one transition.

After n steps, we obtain a concrete path $\tau_0 \to \tau_1 \to \cdots \to \tau_n$ in Σ with $\tau_n \in [\sigma_n]$. By Assumption 1, b holds at τ_n. Hence, b is reachable in Σ.

Theorem 2 establishes that exploring the quotient system Σ/G is sound and complete with respect to bug detection. No false negatives are introduced, and no false positives are generated beyond those already present in the original system.

5.3. Canonicalization Correctness

The third proposition ensures that the canonicalization function maps equivalent states to identical representatives.

We now prove Proposition 2.

Proof. Assume $\text{canon}(\sigma_1) = \text{canon}(\sigma_2)$. By Definition 5.1 (Representative property), $\text{canon}(\sigma_1) \in [\sigma_1]$ and $\text{canon}(\sigma_2) \in [\sigma_2]$. Since the two canonical states are equal, let $\rho = \text{canon}(\sigma_1) = \text{canon}(\sigma_2)$. Then $\rho \in [\sigma_1]$ and $\rho \in [\sigma_2]$, so the two orbits share a common element. By Proposition 1 (Orbit Equivalence), orbits are equivalence classes, so any two orbits that share an element are identical. Hence $[\sigma_1] = [\sigma_2]$. Conversely, assume $[\sigma_1] = [\sigma_2]$. Then by Definition 3, there exists $\pi \in G$ such that $\sigma_2 = \pi(\sigma_1)$. The canonicalization procedure described in Section 3.5 normalizes states according to the automorphism group $\text{Aut}(D)$ of the dependency graph. This automorphism group is invariant under symmetry transformations: if $\pi \in G$ is a symmetry of the program structure, then applying π to a state permutes variables and addresses but preserves the isomorphism class of the dependency graph. Consequently, canon produces the same normalized representative for both σ_1 and $\pi(\sigma_1) = \sigma_2$. Therefore, $\text{canon}(\sigma_1) = \text{canon}(\sigma_2)$. □

This step of the proof relies on Assumption 2 from Section Syntactic Automorphisms Versus Semantic Symmetries. The claim that applying π preserves the isomorphism class of the dependency graph, and hence that canon produces the same representative for σ_1 and $\pi(\sigma_1)$, holds for symmetries satisfying the Structural-Semantic Correspondence assumption. It does not hold for an arbitrary automorphism of $D(\sigma)$ that has not been checked against Definition 1. Proposition 2 is therefore stated relative to the symmetry group G of Definition 2. Its guarantee extends to a candidate π discovered via graph automorphism search only after that π has been confirmed to satisfy Assumption 2.

This proposition guarantees that the visited set O in the quotient exploration procedure grows only with the number of distinct orbits rather than the number of individual states.

5.4. Constraint Reuse Correctness

The fourth proposition ensures that canonical constraint caching preserves SMT solver semantics.

We now prove Proposition 3, stated in Section 4.3.

Proof. If $\text{canon}(\varphi_1) = \text{canon}(\varphi_2)$, then by the definition of canonicalization for constraints (Section 3.5, Step 4), φ_1 and φ_2 are equivalent up to variable renaming, sorting of conjuncts, and simplification. More precisely, there exists a renaming π (a permutation of variable names) such that $\pi(\varphi_1) = \varphi_2$ after sorting conjuncts into the canonical order and applying rewrite rules that eliminate commutativity and associativity ambiguities. Satisfiability of a formula is invariant under bijective renaming of its variables, so φ_1 is satisfiable if and only if $\pi(\varphi_1)$ is satisfiable, which is φ_2. For the model transformation, let M be any satisfying assignment for φ_1. Applying the inverse renaming π^{-1} to the variables in M yields a satisfying assignment M' for φ_2, because $\pi^{-1}(M)$ satisfies $\pi^{-1}(\pi(\varphi_1)) = \varphi_1$ (or symmetrically, $\pi(M)$ satisfies φ_2). Thus, the cache can reuse both satisfiability results and concrete models across symmetric constraints. □

This proposition justifies reusing solver results across symmetric constraints without calling the SMT solver again.

5.5. Complexity Reduction

The number of symbolic states explored in standard symbolic execution grows exponentially with program size in the worst case. Let $|\Sigma|$ be the number of reachable symbolic states. In our framework, the number of states explored reduces to $|\Sigma/G|$, the number of distinct orbits under the symmetry group G.

$$|\Sigma/G| \geq |\Sigma| / |G| \tag{9}$$

Equation (9) states a lower bound on the number of orbits. The bound follows from the orbit-stabilizer theorem: every orbit has size at most $|G|$, so partitioning $|\Sigma|$ states into orbits requires at least $|\Sigma| / |G|$ of them. Equality is attained only in the extreme case where every reachable state has a full orbit of size exactly $|G|$, meaning no state is fixed by any nontrivial group element. Equivalently, $|\Sigma| / |G|$ is the best-case quotient size, and $|G|$ is therefore the maximum achievable reduction factor, not the typical one.

In practice, orbit sizes vary considerably. States fixed by some elements of G have smaller orbits, because fewer group elements map them to distinct members. States near terminal branches, where task-specific values break global symmetry, may belong to orbits of size one or two even when $|G|$ is large. The actual quotient size $|\Sigma/G|$ therefore lies somewhere between $|\Sigma| / |G|$ and $|\Sigma|$, depending on the proportion of states that retain full versus partial symmetry. The inequality $|\Sigma/G| \leq |\Sigma|$ always holds, and equality occurs only when G is trivial.

The reduction factor depends on three factors.

Symmetry density. Systems with high symmetry density, such as those containing many replicated tasks, identical peripherals, or isomorphic interrupt handlers, exhibit large group sizes $|G|$. For a system with k symmetric tasks, the permutation group on k elements has size $k!$. The orbit size can be as large as $k!$, giving a reduction factor of up to $k!$ for states that differ only by task permutation.

Orbit structure. Not all states have full orbits. Some states may be fixed by certain symmetries, resulting in smaller orbits. The average orbit size determines the actual reduction. For a system with k symmetric tasks, the maximum orbit size equals k!, so three symmetric tasks yield orbits of up to size 6, and four symmetric tasks yield orbits of up to size 24. Practical orbit sizes depend on how many states the symmetry group fixes, and partial symmetry reduces the effective orbit size below this theoretical maximum.

Canonicalization overhead. Each state requires canonicalization, which costs $O(n \log n)$ for variable normalization plus $O(\exp(n))$ for automorphism detection in the worst case. Recent theoretical work established that graph isomorphism is solvable in quasipolynomial time [24], which improves the known worst-case bound and provides theoretical grounding for practical automorphism detection in sparse dependency graphs. However, under our assumption that the dependency graph remains tractable (as stated in Section 3.6), NAUTY runs in near-linear time on such graphs [20].

The overall complexity of quotient exploration is

$$T_{total} = |\Sigma/G| \times (T_{canon} + T_{transition} + T_{solver}) \tag{10}$$

where T_{canon} is the canonicalization time per orbit, $T_{transition}$ is transition exploration time, and T_{solver} is the SMT solver time per query. In contrast, standard symbolic execution has:

$$T_{standard} = |\Sigma| \times (T_{transition} + T_{solver}) \tag{11}$$

The speedup factor is approximately

$$\text{Speedup} = |\Sigma| / |\Sigma/G| \leq |G| \tag{12}$$

Equation (12) equals the average orbit size when canonicalization overhead is negligible compared to solver time.

5.6. Equivariant Learning Efficiency

The equivariant GNN described in Section 4.2 provides additional efficiency gains through sample complexity reduction.

In standard (non-equivariant) learning, the network must learn separate patterns for each symmetric variant of a code structure. For example, a network might need separate training examples for task A before task B and task B before task A, even though these situations are symmetric. The number of distinct training examples required scales with the number of symmetric permutations.

With equivariant constraints, the network learns that applying a symmetry transformation to the input produces a corresponding transformation of the output. Therefore, a single training example generalizes to all symmetric variants. The sample complexity reduces from $O(|\Sigma|)$ to $O(|\Sigma/G|)$, the number of distinct orbits.

Formally, let L be the loss function for training. For an equivariant network f satisfying $f(\pi(g)) = \pi(f(g))$, the loss on a transformed example equals the loss on the original example:

$$L(f(\pi(g)), \pi(\text{target})) = L(f(g), \text{target}) \tag{13}$$

This property means that data augmentation by symmetry transformations does not provide new information. The network already incorporates the symmetry prior, so training requires only one example per orbit.

For embedded systems with high symmetry density, this reduction can be substantial. A firmware image with eight symmetric interrupt handlers would require eight times fewer training examples to achieve the same generalization performance.

5.7. Discussion of Bounds

The complexity bounds presented in this section are theoretical worst-case guarantees. In practice, several factors affect actual performance.

Worst-case versus average case. The exponential worst-case complexity of automorphism detection rarely manifests in practice because dependency graphs from symbolic execution are sparse and structured. The NAUTY algorithm [20] handles graphs with thousands of nodes efficiently, and typical embedded firmware states have far fewer variables.

Amortization across analyses. For one-time analysis of a single program, the executor pays the canonicalization overhead once per orbit. For repeated analysis (e.g., in CI/CD pipelines), the cost can be amortized across multiple runs. The cache hit rate itself serves as a signal for adaptive strategy selection: high hit rates justify full canonicalization, while low hit rates suggest approximate methods.

Trade-offs with approximation. When exact automorphism detection becomes expensive, approximate canonicalization based on variable renaming alone provides a practical fallback. This approximation loses some symmetry detection but gracefully degrades to standard symbolic execution behavior.

Empirical validation. Section 6.5 reports measured results from a running prototype that confirms the state-reduction and constraint-caching predictions of this section on a toy model. Validation on real embedded firmware benchmarks remains future work, as described in Section 8.

The following section demonstrates the framework on an embedded system case study through qualitative analysis of the expected reduction structure and a pseudocode walkthrough of a simplified two-task example.

6. Illustrative Embedded-System Case Study

We demonstrate the S^3E framework conceptually on an embedded system example inspired by FreeRTOS-like scheduling. This case study illustrates how symmetry reduction collapses equivalent states and how AI guidance benefits from quotient exploration. Sections 6.1–6.4 develop this qualitatively and analytically; Section 6.5 reports results from

a working toy-model prototype. Full empirical evaluation on production firmware remains future work.

6.1. System Description

The embedded system contains three symmetric worker tasks (Task A, Task B, Task C), two duplicated UART peripherals (UART0 and UART1), two replicated interrupt handlers (IRQ_TIMER0 and IRQ_TIMER1), and shared round-robin scheduling logic.

Worker tasks. Each worker task reads a character from a UART, increments a counter, and writes the character back to the same UART. The three tasks have identical code and priority. Under task identifier permutation, swapping Task A and Task B produces a state with identical future behavior [14].

UART peripherals. Both UARTs have identical register interfaces and behavior. Swapping the memory addresses of UART0 and UART1 results in an equivalent execution state [14].

Interrupt handlers. The two timer interrupt handlers execute identical code. Swapping the interrupt service routine mappings (IRQ_TIMER0 and IRQ_TIMER1) produces symmetric states [15].

Shared scheduling logic. The round-robin scheduler maintains a task list and a current task pointer. Symmetric configurations of the task list (e.g., [A, B, C] versus [B, C, A]) produce equivalent future behavior under task renaming.

6.2. Qualitative Analysis of Symmetry Reduction

The symmetry group G for this system includes all permutations of the three tasks, which form the symmetric group S_3 of order 6, the swap of the two UART peripherals, which contributes a factor of 2, and the swap of the two interrupt handlers, which contributes another factor of 2. These three sources of symmetry are independent, so the combined group has order $|\ G\ | = 6 \times 2 \times 2 = 24$. This means the framework could, in principle, identify up to 24 states as members of a single orbit and explore only one representative.

Without symmetry reduction, a standard symbolic executor treats every task ordering, every peripheral address assignment, and every interrupt handler configuration as a distinct execution path. The total number of explored states grows with the product of these sources of variation. Each scheduling round multiplies the frontier by the number of tasks, each symbolic branch within a task multiplies it again, and the two duplicated peripheral variants double the count once more. As analysis depth increases, the multiplicative structure of this growth makes complete exploration intractable.

With S^3E, the canonicalization procedure normalizes task identifiers to a fixed lexicographic ordering, maps peripheral addresses to a canonical assignment (UART0 before UART1), and collapses interrupt handler pairs to their canonical form. The executor then explores one representative per orbit. The reduction in explored states depends on how many states in the execution tree retain full symmetry across all three components.

States at symmetric interior nodes, such as the scheduler entry point before any task-specific data has been read, collapse by a factor approaching $|\ G\ | = 24$, because all 24 group elements map such a state to a genuinely distinct raw state that is nonetheless in the same orbit. States at boundary nodes, such as a state where one specific task has already read a device-specific value that breaks the global symmetry, have smaller orbits and collapse by a smaller factor. The overall reduction therefore varies across the execution tree: largest near the root, where symmetry is intact, and smallest near terminal states, where task-specific bindings break it.

The qualitative conclusions are as follows. First, state counts reduce substantially for interior symmetric states, with the reduction factor bounded above by $|\ G\ |$. Second, constraint cache hit rates increase because symmetric path constraints share the same

canonical form and require only one solver call per orbit. Third, the RL agent trains on canonical representatives, and its policy generalizes automatically to all symmetric variants, reducing the number of training trajectories needed. These reductions are most significant in the early scheduling phases and diminish near terminal states. Precise reduction figures depend on the specific firmware, analysis depth and the proportion of states that retain full versus partial symmetry. Section 6.5 reports measured results from a working prototype of this toy model. Empirical measurements on production firmware, however, await the KLEE-based implementation described in Section 8.

6.3. Pseudocode Walkthrough: Two-Task Minimal Example

To make the framework concrete, we first trace through a simplified two-task version of the system using pseudocode (a working prototype is reported separately in Section 6.5). This walkthrough shows how canonicalization identifies an orbit, how the executor skips the duplicate, and how the bug found in the canonical representative applies to both tasks by Theorem 1.

System description. We have two symmetric tasks, T_1 and T_2. Each task reads one symbolic byte from a shared UART register and then checks whether the byte lies in the valid range $[0, 127]$. Task T_1 uses variable b_1 and task T_2 uses variable b_2. The symmetry group is $G = \{id, \pi\}$, where π swaps T_1 with T_2 and simultaneously swaps b_1 with b_2.

State representation. A symbolic state is $\sigma = (pc, \phi, \mu, \kappa)$ where pc belongs to the set $\{\text{sched}, \text{read}, \text{check}, \text{done}, \text{error}\}$, ϕ is a conjunction of constraints over $\{b_1, b_2\}$, and κ records which task is active (T_1 or T_2).

Canonicalization rule. The canonical ordering requires that the active task always appear as T_1. If the active task is T_2, we apply π to swap T_1 and T_2 throughout the state.

We define the canonicalization procedure as follows.

Procedure canon(σ):

- If $\sigma.\kappa.\text{active} = T_2$, return apply_permutation (π, σ) (this swaps b_1 with b_2 in ϕ and μ and sets active to T_1);
- Otherwise, return σ (the state is already canonical).

The quotient exploration procedure then runs as follows.

Initialization. We start with the initial state σ_0 where $pc = \text{sched}$, $\phi = \text{True}$, $\mu = \{b_1 = \text{sym}, b_2 = \text{sym}\}$, and $\kappa = \{\text{active} = \text{None}\}$. We set the frontier to $\{\text{canon}(\sigma_0)\} = \{\sigma_0\}$ and visited to the empty set.

Iteration 1: scheduler. We pop σ_0 from the frontier. Its canonical representative is $\text{canon}(\sigma_0) = \sigma_0$, which we have not visited before, so we add it to visited. The successors of σ_0 are two states: σ_1 where we schedule T_1 and σ_2 where we schedule T_2.

We compute $\text{canon}(\sigma_1) = \sigma_1$ because T_1 is active, which is already canonical. We compute $\text{canon}(\sigma_2) = \text{apply\_permutation}(\pi, \sigma_2) = \sigma_1$ because swapping T_2 to T_1 makes it identical to σ_1. We add σ_1 to the frontier. The state σ_2 maps to the same canonical representative, so we skip it. The executor never explores σ_2, saving one full branch.

Iteration 2: read phase with T_1 active. We pop σ_1 from the frontier. Its canonical representative is σ_1, which we have not visited before, so we add it to visited. The successors of σ_1 are two states: σ_3 where the byte is in range and σ_4 where the byte is out of range. σ_4 represents an out-of-range bug for T_1. By Theorem 1, applying π to σ_4 gives the same bug for T_2. The executor reports the bug once, and it covers both tasks. We push $\text{canon}(\sigma_3)$ and $\text{canon}(\sigma_4)$ to the frontier.

Result. The visited orbits are $\{\sigma_0, \sigma_1, \sigma_3, \sigma_4\}$. The executor found one bug, an out-of-range read, which applies to both T_1 and T_2. The total number of explored states is four.

Without S^{3}E, the executor would also explore σ_2 and its successors σ_5 and σ_6 (the mirror of σ_3 and σ_4 for T_2), giving seven states in total against four in the quotient. Three states are saved at this depth: the redundant scheduling branch and both of its outcome successors.

For this two-task system, the symmetry group has order 2, and the reduction at the scheduling branch is a factor of 2. For the three-task system described in Section 6.1, the symmetric group S_3 has order 6, so up to six scheduling orderings collapse to one canonical representative per round. The two UART peripheral variants and the two interrupt handler assignments contribute further factors of 2 each. These reductions multiply across scheduling rounds, so the benefit grows with analysis depth even though each individual state collapses by a bounded factor.

This pseudocode walkthrough confirms that the framework's core mechanism, canonicalization followed by orbit membership testing, requires no measurement. Its correctness follows directly from Theorem 1 and Proposition 2.

6.4. Analytically Derived State Counts for the Generalized k-Task Model

The two-task walkthrough in Section 6.3 generalizes to an arbitrary number of symmetric tasks k acting under the full symmetric group S_k. The following counts are derived by exhaustive enumeration of the finite toy model defined in Sections 6.1–6.3, not measured from a running symbolic executor on real firmware; they are reported to make the qualitative reduction argument in Section 6.2 numerically concrete.

For the round-robin scheduling step followed by one read/check branch per task, the raw (non-quotient) execution tree contains one scheduler node, k task-scheduled nodes (one per possible next task), and $2k$ outcome nodes (in-range or out-of-range, per task), for a total of $N_{raw}(k) = 1 + 3k$.

Under S_k symmetry, all k task-scheduled nodes collapse to a single canonical representative, and their two outcome successors are already canonical, giving a constant orbit count $N_{orbit}(k) = 4$ independent of k. The reduction factor at this depth is therefore (1 + 3k)/4.

Two consequences follow directly from Table 3. First, the reduction factor itself increases linearly with the number of symmetric tasks for this one-round model since the orbit count remains constant at 4 while the raw count grows linearly in k. This is consistent with the qualitative assertion in Section 6.2 that benefit increases with symmetry density. Second, these counts only describe a single scheduling round of the simplified toy model; they are not a claim regarding reduction factors for production firmware, which is dependent on program-specific orbit structure as covered in Section 5.5, but rather a worked numerical illustration of Theorem 1 and Proposition 2.

Table 3. Exact state counts for the generalized k-task toy model, one scheduling round.

k (Symmetric Tasks)	2	3	4	5
Raw states, $N_{\text{raw}}(k) = 1 + 3k$	7	10	13	16
Orbits explored, $N_{\text{orbit}}(k)$	4	4	4	4
Reduction factor, $N_{\text{raw}}/N_{\text{orbit}}$	1.75	2.5	3.25	4.0

6.5. Prototype Implementation and Measured Results

In response to reviewer requests across both rounds of review, we implemented and ran a working prototype of the toy model described in Sections 6.1–6.4, rather than relying solely on hand-traced pseudocode or analytical enumeration. The prototype is a real symbolic executor backed by the Z3 SMT solver [21]. Every solver call reported below is an actual Z3 Solver.check() invocation, and every timing figure is measured wall-clock

time from the running program, not a formula. The prototype consists of approximately 120 lines of Python 3.13 and is available from the authors on request.

Model and scope. The prototype implements exactly the system described in Section 6.1: k symmetric worker tasks, a choice of two symmetric UART peripherals, and a choice of two symmetric interrupt handlers, combined with the two-outcome read/check bug pattern from Section 6.3 (byte in range or out of range). To exercise the constraint cache meaningfully, the prototype runs $R = 5$ independent scheduling rounds. Section 6.3's single-round walkthrough alone produces only two distinct canonical constraints regardless of cache behavior. With five rounds, each of the k tasks performs five reads, giving $n = 5k$ read events in total. We emphasize the scope honestly: this validates the core mechanism, including canonicalization, orbit collapse, and canonical constraint caching, end-to-end on the exact toy model already presented analytically in Sections 6.1–6.4. It is not an integration with a production symbolic executor such as KLEE or angr, and it does not run on real firmware. That remains the next step identified in Section 8.

Two executors. The raw executor performs no symmetry reduction. Each of the k tasks, 2 UART choices, 2 interrupt-handler choices, and 5 rounds gets its own fresh Z3 integer variable per read, so no two solver queries are ever syntactically identical, and no caching is possible. This matches standard non-canonical symbolic execution honestly rather than by assumption. The quotient executor canonicalizes the (task, UART, interrupt-handler) assignment to a single representative, as in Definition 5, and canonicalizes each path constraint to a shared variable name before consulting the cache, as in Definition 7 from Section 3.5, Step 4. Both executors share the same underlying Z3 feasibility check, so any difference in solver calls or wall-clock time reflects the quotient reduction itself, not an implementation artifact. Table 4 reports the measured results.

Table 4. Measured results from the running prototype ($R = 5$ rounds).

k	Raw States	Quotient States	Raw Solver Calls	Quotient Solver Calls	Cache Hits	State Reduction	Solver-Call Reduction
2	95	6	80	2	8	15.8×	40.0×
3	142	6	120	2	8	23.7×	60.0×
4	189	6	160	2	8	31.5×	80.0×
5	236	6	200	2	8	39.3×	100.0×
6	283	6	240	2	8	47.2×	120.0×
8	377	6	320	2	8	62.8×	160.0×

Measured wall-clock time, averaged over the run and including real Z3 solver invocation overhead, not just theoretical operation counts, fell from approximately 0.11 s for $k = 2$ to 0.43 s for $k = 8$ for the raw executor. For the quotient executor, wall-clock time remained approximately 0.0025 to 0.0035 s across all tested k. This gives roughly a 40× to 160× wall-clock speedup on this toy model, consistent with the solver-call reduction factor, since Z3 invocation dominates runtime at this scale. Canonicalization overhead itself, primarily variable-renaming lookups, measured under 5 microseconds total across all rounds for every k tested, confirming Section 3.6's expectation that canonicalization cost is negligible relative to solver time for this constraint size.

Interpretation. These results empirically confirm, on a real running implementation rather than by analytical argument alone, three of the claims that we previously stated as theoretical expectations. First, state-space reduction scales with the symmetry group as predicted in Section 5.5 and Table 3. Second, canonical constraint caching as defined in Definition 7 and Proposition 3 measurably reduces SMT solver calls, confirming Table 2's expected benefit for that component specifically. Third, canonicalization overhead is small

relative to solver time in practice, as Section 3.6 assumed. These results do not confirm claims that are specific to production firmware scale, to the RL and GNN components of Section 4, which are not exercised by this prototype, or to LLM-generated symmetries from Section 4.5, which remain theoretical pending further work.

7. Discussion and Limitations

The S^3E framework provides a theoretical foundation for symmetry-aware AI-guided symbolic execution, but several limitations must be acknowledged before the approach can be deployed in practice. These limitations fall into four categories: scenarios where symmetry is weak or absent, computational costs of canonicalization, challenges with dynamic embedded system behavior, and integration difficulties with real-world firmware.

Scenarios with weak or absent symmetry. A substantial fraction of deployed embedded firmware follows highly asymmetric control-flow patterns. A bootloader that configures each peripheral with unique parameters, a control loop with irregular state transitions, or code that depends on hardware instance-specific identifiers may have few or no nontrivial symmetries. In such cases, the symmetry group G reduces to the identity transformation. The S^3E framework gracefully degrades to standard symbolic execution with minimal overhead, because canonicalization still produces a representative (the state itself) and the quotient space Σ/G equals Σ. However, the framework provides no benefit for these systems, and the canonicalization overhead becomes pure cost. Practitioners should therefore apply S^3E selectively to systems where symmetry density is known to be high, such as those with replicated tasks, identical peripherals, or isomorphic interrupt handlers. For systems with partial symmetries, future work could learn which symmetries are likely to yield reduction benefits and enable only those transformations.

Computational cost of canonicalization. The canonicalization procedure described in Section 3.5 relies on graph automorphism detection using the NAUTY algorithm [20]. While NAUTY runs efficiently on sparse graphs, its worst-case complexity remains exponential. For symbolic states with large dependency graphs (e.g., thousands of variables and constraints), automorphism detection may become prohibitively expensive. Four mitigation strategies address this cost directly. First, we can bound the size of graphs submitted to NAUTY by pruning nodes that do not participate in symmetries. Second, we can use approximate canonicalization based solely on variable renaming, ignoring more complex structural symmetries, which gracefully degrades to standard execution. Third, we can apply learned symmetry estimation using lightweight neural networks that predict whether a state contains exploitable symmetries before running full automorphism detection; this learned filter bypasses expensive canonicalization for states with low symmetry potential. Fourth, we can cache canonicalization results for recurring dependency graph structures, amortizing the cost across many similar states.

Dynamic embedded system behavior. Our framework assumes symmetries are known ahead of time or discovered offline through static analysis of the firmware binary. However, real embedded systems exhibit dynamic symmetries that emerge only at runtime. A peripheral may reconfigure and become symmetric to another only after specific configuration writes. A task's behavior may change based on dynamically loaded code. An interrupt handler's effect may depend on hardware state that varies over time. Extending S^3E to handle dynamic symmetry discovery remains an open challenge. One approach is to combine offline symmetry detection with online refinement: the static analyzer proposes candidate symmetries, and the symbolic executor dynamically verifies or invalidates them during exploration. Another approach is to use reinforcement learning to learn which symmetries hold at which program points, treating symmetry detection as a prediction problem.

Irregular control-flow structures. The framework works best for systems with regular, replicated structures such as task arrays, peripheral banks, and symmetric interrupt handlers. Embedded software with highly irregular control flow, such as state machines with unique states, protocol parsers with complex conditional logic, or error handling paths that diverge asymmetrically, may exhibit few exploitable symmetries. In such cases, the quotient space Σ/G offers little reduction. However, even irregular systems often contain local symmetries: two error handling paths may be symmetric even if the main control flow is not. Techniques such as chopped symbolic execution [25] address path explosion through selective exploration of program slices rather than symmetry exploitation. S^3E and chopped symbolic execution are complementary: S^3E reduces redundancy across symmetric states, while slice-based methods reduce redundancy across irrelevant code regions. Future work could focus on detecting and exploiting local, scoped symmetries rather than requiring global symmetry groups.

Relationship to existing symmetry reduction methods. Symmetry reduction has been extensively studied in explicit-state model checking [14–16]. S^3E differs from these methods in three important ways. First, S^3E operates on symbolic states with path constraints, not on concrete states. This allows verification of infinite-state systems where model checking would not terminate. Second, S^3E integrates AI guidance that operates on the quotient space, learning to prioritize orbits rather than raw states. Third, S^3E supports canonical constraint caching, which reuses solver results across symmetric constraints, a technique not applicable to model checking. The relationship is complementary: S^3E can be seen as adapting symmetry reduction ideas from finite-state model checking to the symbolic execution domain.

Implementation and integration challenges. Deploying S^3E in practice requires addressing several engineering challenges. Online canonicalization must be fast enough to avoid dominating analysis time; we estimate that canonicalization should take no more than 10–20% of total analysis time to be worthwhile. In the toy-model prototype of Section 6.5, measured canonicalization overhead was under 5 microseconds total per configuration, negligible relative to Z3 solver time, so this estimate is comfortably met at the toy scale. Whether it holds at production scale, where dependency graphs are far larger, remains to be measured. Memory scalability becomes a concern when storing canonicalized states for large firmware images; we plan to use persistent storage with indexing. Real-world firmware often uses dynamic memory allocation, self-modifying code, and vendor-specific hardware abstractions that break static symmetry assumptions. Our ongoing prototype implementation targets the LLVM-based KLEE symbolic executor [9] and will initially support C programs compiled to LLVM IR. Support for binary analysis via angr [10] is planned for future work.

When to use S^3E. S^3E delivers the largest benefits on embedded systems with replicated tasks, symmetric peripherals, or isomorphic interrupt handlers, where high symmetry density amortizes canonicalization overhead across a large number of equivalent states. Examples include IoT devices running identical protocol handlers, automotive ECUs with multiple sensor interfaces, and multi-core systems with symmetric cores. For highly asymmetric firmware or one-off analyses where training data is unavailable, standard symbolic execution remains appropriate.

8. Conclusions and Future Work

This paper introduced S^3E, a symmetry-theoretic framework for AI-guided symbolic execution in embedded systems. The central idea is to explore quotient symbolic-state spaces rather than individual execution states. Symmetry groups collapse equivalent states into orbits, and AI components operate on canonical orbit representatives. This design is

intended to reduce path explosion, eliminate redundant solver calls, improve generalization across symmetric code, and enable scalable symbolic-state management. Section 5 establishes these properties formally and empirical confirmation is left to future work.

The paper made six main contributions. We defined a formal symbolic-state symmetry framework, including orbit definitions and quotient transition systems. We presented quotient-state exploration that uses orbit representatives instead of individual states. We designed symmetry-aware AI components: reinforcement learning over canonical states, permutation-equivariant graph neural networks for branch prediction, canonical constraint caching for solver reuse, and symmetry-guided merging. We introduced a canonicalization procedure and a symmetry-aware constraint reuse mechanism that maps structurally equivalent path constraints to shared SMT solver results. We proved theoretical guarantees for orbit preservation, quotient soundness, canonicalization correctness, and constraint reuse correctness. We also illustrated the framework on a FreeRTOS-like system with symmetric tasks and peripherals.

The framework offers four main theoretical advantages over standard symbolic execution, each established formally in Section 5 and, for state space reduction and constraint cache reuse, additionally supported by measured results from the toy-model prototype in Section 6.5. First, the number of explored states is proved to reduce from $|\Sigma|$ to $|\Sigma/G|$, eliminating redundancy from symmetric paths. Second, canonical constraint caching is designed to reuse satisfiability results across symmetric constraints, reducing solver calls. Third, equivariant GNNs are designed to generalize across symmetric code fragments without requiring additional training data. Fourth, the quotient structure provides a natural basis for parallel exploration, as orbits can in principle be explored independently.

Looking toward future work, we identify five priority directions for research. The most immediate next step is to implement S^3E as an extension to an existing symbolic executor. KLEE's modular design [9] provides natural extension points for custom search heuristics and state management, making it suitable for integrating symmetry detection and canonicalization. The angr framework [10] offers binary analysis capabilities that would enable application to firmware without source code. We plan to prototype both and compare their performance.

Another important direction is dynamic symmetry discovery. Current S^3E requires the symmetry group G to be specified or discovered offline. However, many embedded systems exhibit symmetries that emerge only at runtime based on configuration or hardware state. Future work should develop algorithms for online symmetry discovery, where the executor observes execution patterns and infers candidate symmetries dynamically. These candidate symmetries can then be verified and added to G during exploration.

Online adaptive canonicalization also deserves attention. The cost of canonicalization varies significantly across states. Some states have rich symmetry groups that require full automorphism detection, while others have trivial symmetries where a simple variable renaming suffices. An adaptive system could learn to predict canonicalization cost and choose the appropriate algorithm: lightweight renaming for low-symmetry states, full NAUTY for high-symmetry states, and approximate methods for borderline cases. This would optimize the trade-off between precision and performance.

Rigorous empirical evaluation is essential. We plan to evaluate S^3E on established embedded benchmark suites, including FreeRTOS applications, Zephyr RTOS examples, and real IoT firmware from open-source repositories. Metrics will include reduction in explored states, reduction in solver calls, analysis time speedup, and bug detection rate. We will compare against standard KLEE, angr, and state-of-the-art AI-guided symbolic executors.

Finally, integration with hardware-aware verification frameworks is a promising direction. Many embedded systems interact closely with hardware. The Symbiotic verification framework, SeaHorn, and firmware re-hosting tools like HALucinator [26] provide infrastructure for analyzing hardware-dependent code. Integrating S^3E with these frameworks would enable symmetry-aware verification of device drivers, bootloaders, and other hardware-facing firmware. LLM-assisted symmetry generation (Section 4.5) is particularly promising in this context, as LLMs can infer hardware symmetries from datasheets and code comments.

Symbolic-state symmetry offers a foundational abstraction for scalable AI-guided symbolic execution. By respecting the inherent equivalences in embedded system designs, we can build verifiers that do not waste time on symmetric redundancy. The framework presented here provides the theoretical groundwork; the research directions outlined above chart a path toward practical deployment. As embedded systems grow more complex and software-dependent, symmetry-aware verification will become a practical necessity for sound analysis at scale.

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 (Marie Skłodowska-Curie Actions), grant agreement No. 101244751 (AISSAM, "Automated Vulnerability Detection in Software Development Using AI Techniques"), call HORIZON-WIDERA-2024-TALENTS-02.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

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

References

1. King, J.C. Symbolic execution and program testing. *Commun. ACM* **1976**, *19*, 385–394. [CrossRef]
2. Baldoni, R.; Coppa, E.; D'Elia, D.C.; Demetrescu, C.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv.* **2018**, *51*, 50. [CrossRef]
3. Cadar, C.; Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM* **2013**, *56*, 82–90. [CrossRef]
4. Muench, M.; Stijohann, J.; Kargl, F.; Francillon, A.; Balzarotti, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
5. Böttinger, K.; Godefroid, P.; Singh, R. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*; IEEE: New York, NY, USA, 2018; pp. 116–122. [CrossRef]
6. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. In Proceedings of the 6th International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
7. Balunovic, M.; Bielik, P.; Vechev, M. Learning to solve SMT formulas. In Proceedings of the Advances in Neural Information Processing Systems, Montréal, QC, Canada, 3–8 December 2018; p. 31.
8. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
9. Cadar, C.; Dunbar, D.; Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the OSDI, San Diego, CA, USA, 8–10 December 2008; pp. 209–224.
10. 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), San Jose, CA, USA, 22–26 May 2016*; IEEE: New York, NY, USA, 2016; pp. 138–157. [CrossRef]

11. Bailey, J.; Nicholas, C. Symbolic Execution in Practice: A Survey of Applications in Vulnerability, Malware, Firmware, and Protocol Analysis. *arXiv* **2025**, arXiv:2508.06643.
12. Li, Y.; Meng, R.; Duck, G.J. Large language model powered symbolic execution. *Proc. ACM Program. Lang.* **2025**, *9*, 3148–3176. [CrossRef]
13. Tu, H.; Jiang, L.; Ding, X.; Jiang, H. FastKLEE: Faster symbolic execution via reducing redundant bound checking of type-safe pointers. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 1741–1745.
14. Clarke, E.M.; Enders, R.; Filkorn, T.; Jha, S. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.* **1998**, *9*, 77–104. [CrossRef]
15. Emerson, E.A.; Sistla, A.P. Symmetry and model checking. *Form. Methods Syst. Des.* **1996**, *9*, 105–131. [CrossRef]
16. Donaldson, A.F.; Miller, A. Automatic symmetry detection for model checking using computational group theory. In *Proceedings of the FM*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 481–496.
17. Bronstein, M.M.; Bruna, J.; Cohen, T.; Veličković, P. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv* **2021**, arXiv:2104.13478.
18. Maron, H.; Ben-Hamu, H.; Shamir, O.; Lipman, Y. Invariant and equivariant graph networks. In Proceedings of the ICLR, New Orleans, LA, USA, 6–9 May 2019.
19. Iavich, M.; Kuchukhidze, T.; Lopata, A. Static analysis techniques for embedded, cyber-physical, and electronic software systems: A comprehensive survey. *Electronics* **2026**, *15*, 918. [CrossRef]
20. McKay, B.D.; Piperno, A. Practical graph isomorphism, II. *J. Symb. Comput.* **2014**, *60*, 94–112. [CrossRef]
21. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6 April 2008; pp. 337–340.
22. Aloul, F.A.; Sakallah, K.A.; Markov, I.L. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Comput.* **2006**, *55*, 549–558. [CrossRef]
23. Kuznetsov, V.; Kinder, J.; Bucur, S.; Candea, G. Efficient state merging in symbolic execution. *Proc. PLDI* **2012**, *47*, 193–204. [CrossRef]
24. Babai, L. Graph isomorphism in quasipolynomial time. In Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing, Cambridge, MA, USA, 19–21 June 2016; pp. 684–697. [CrossRef]
25. Trabish, D.; Mattavelli, A.; Rinetzky, N.; Cadar, C. Chopped symbolic execution. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 350–360. [CrossRef]
26. Clements, A.A.; Gustafson, E.; Scharnowski, T.; Grosen, P.; Fritz, D.; Kruegel, C.; Vigna, G. HALucinator: Firmware rehosting through abstraction layer emulation. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 1201–1218.