Sound and Precise Symbolic Automata Model for Stateful Software Systems

Xinlong Wu[1], Ruiyu Zhou[1], Peisen Yao[2], and Qingkai Shi[1(✉)]

[1] State Key Laboratory of Novel Software Technology, Nanjing University, China
[2] State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
{xinlongwu, ruiyuzhou}@smail.nju.edu.cn; pyaoaa@zju.edu.cn;
qingkaishi@nju.edu.cn

Abstract. Verifying stateful software systems remains challenging due to complex control structures and intricate state interactions, often necessitating pre-existing behavioral models. We introduce `Seal`, an abstract interpreter that automatically derives sound and precise symbolic finite automata models. In tests on real-world applications, `Seal` generates sound and precise automata within minutes and, when employed in dynamic model checking, achieves 1.6×–3.4× code coverage compared to the state of the art. These findings demonstrate that `Seal` can effectively connect code to model-based verification for stateful software systems.

Keywords: Abstract interpretation · Symbolic finite automata · Dynamic model checking.

1 Introduction

Modern software is rich in stateful components, such as parsers that consume streaming data, protocol endpoints that arbitrate request-reply sessions, and embedded controllers that react to continuous sensor feeds. For example, Fig. 1 shows a simplified stateful system and the corresponding automaton that controls a drone to flip. The flip maneuver is divided into four states as defined in line 1. The system transitions between states based on the drone's angle, as measured by the sensors in line 4. Directly model-checking specific properties of such a system is often challenging, because of not only the sophisticated code constructs (e.g., pointer operations and special data structures in the omitted code in lines 7, 13, and 19), but also the stateful behavior that requires particular input sequences to drive the system to a deep state. As such, we often need to acquire a behavioral model, e.g., the automaton in Fig. 1, before model checking.

A convenient, automata–theoretic view represents such behavioral models as symbolic finite automata (SFA) [63]. An SFA abstracts a stateful system as a finite set of high-level states (e.g., Start, Roll, Pitch, Done), with transitions guarded by first-order predicates over an (often infinite) input domain. This differs from classic control-flow or Floyd-Hoare automata in the context of software model checking e.g., [18,31,32,33,53,57]. For example, Fig. 2(a-b) shows a control-flow automaton where each state ℓ_i denotes a program point before line i

```
1. enum FlipState { Start=0, Roll=1, Pitch=2, Done=3 };
2. FlipState _state = FlipState::Start;
3. while (1) {
4.     int angle = read_gyro();
5.     switch(_state) {
6.     case FlipState::Start:
7.          ... /*code for business logic omitted*/
8.          if (angle >= 45)       _state = FlipState::Roll;
9.          else if (angle <= -45) _state = FlipState::Pitch;
10.         else                   ;
11.         break;
12.    case FlipState::Roll:
13.         ... /*code for business logic omitted*/
14.         if (angle >= 45)       ;
15.         else if (angle <= -45) _state = FlipState::Pitch;
16.         else                   _state = FlipState::Done;
17.         break;
18.    case FlipState::Pitch:
19.         ... /*code for business logic omitted*/
20.         if (angle >= 45)       _state = FlipState::Roll;
21.         else if (angle <= -45) ;
22.         else                   _state = FlipState::Done;
23.         break;
24.    case FlipState::Done:
25.         return;
26.    }}
```

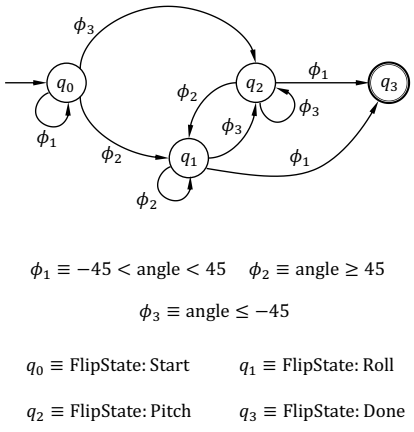
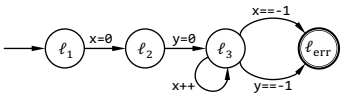
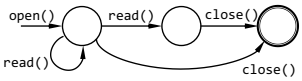


Fig. 1: A simplified code snippet from a stateful autopilot system for drones, and the automata for controlling drones.

```
1. x = 0;
2. y = 0;
3. while(...) { x++; }
4. assert(x != 1);
5. assert(y != 1);
```





(a) (b) (c)

Fig. 2: (a)-(b) Control-flow automaton. (c) Typestates for a file handler.

and the state-transition conditions are statements or predicates in the code. SFA also differs from typestates [58], which specify valid sequences of operations on a type instance. For instance, Fig. 2(c) shows the typestates of a file handler, specifying that `open()-read()-read()-close()` is a valid operation sequence.

In addition to the many promising, theoretical (decidability and closure properties) properties [25,20,38,64,63], a key advantage of SFA models is their ability to generate input sequences that conform to transition constraints, thereby enabling techniques such as dynamic model checking [27,37] to explore deep states for property verification. Without a sound and precise model to guide input generation, random inputs tend to keep the system in shallow states and fail to exercise deep-state properties. SFAs have therefore been used in a range of applications, including text or regular-expression processing [60,1], software analysis and verification [70,12,51,34,69], constraint solving [21,61] and stateful testing or fuzzing [19,41,5], among many others.

Existing Work. To enable such a broad range of applications, researchers have proposed automated techniques for acquiring SFA or its specialized forms. However, we observe that all existing work suffers from unsoundness or imprecision issues, as discussed below.

A typical line of these techniques [23,4,47,14,6,44,36,35,54,29,11] extends Angluin's L* algorithm [2] to actively learn symbolic automata from a hypothetical teacher who serves as an oracle of the target automaton and can answer membership queries and equivalence queries. Membership queries test whether the target automaton accepts a word. Equivalence queries test whether a learned automaton is equivalent to the target. In practice, the unavailability of a perfect teacher often results in us learning an unsound and imprecise automaton.

The second class leverages static program analysis to extract automata directly from source code [66,13,39,13,55,7]. However, they often rely on restrictive assumptions that limit their applicability or compromise their soundness. For instance, Xie et al. [66] can soundly handle only one of the four numerical program patterns they identify and have to enumerate all program paths, leading to both the unsoundness and the path explosion problems. Chen et al. [13] assume that the source code follows a simple syntactic pattern in which we can easily identify a single state variable of an enumerated type.

Our Approach. This paper presents `Seal`, a static analysis framework for inferring sound and precise SFA models from the implementation of stateful systems. Our key insight is that implementing a stateful system often results in *structural separation*: distinct states are realized by disjoint code regions. For example, in the drone controller in Fig. 1, the states `Start` and `Roll` are implemented by separate blocks responsible for initialization and roll control, respectively. This observation motivates a program partitioning strategy in which each sub-program corresponds to a candidate state, and control-flow dependencies between sub-programs induce the state transitions.

`Seal` iteratively explores the program using symbolic execution and fixed-point computation. Starting from the initial environment, the analysis explores a sub-program π_1 and computes its post-condition; if that post-condition enables execution of another partition π_2 in the next iteration, `Seal` records a transition from π_1 to π_2 guarded by the derived condition. To ensure convergence, we maintain disjoint sub-programs: if two sub-programs overlap, we extract their intersection as a separate component and refine the partition accordingly.

To balance precision and scalability, `Seal` employs a mixed abstract domain. The abstraction is adaptive at two levels — (1) *across iterations*: variables can switch from a precise symbolic representation (e.g., the exact path condition over input symbols) to a coarser abstraction (e.g., intervals) when needed for convergence; (2) *within an iteration*: abstraction is selected on a per-variable basis, so precision is degraded only for variables that cause divergence.

To sum up, this work makes the following contributions:

- We introduce `Seal`, the first static analysis framework for inferring sound and precise SFAs from the code of stateful software systems.
- We evaluate `Seal` on 12 real-world stateful software systems. `Seal` infers precise and sound automata for all systems within five minutes. When used for dynamic model checking, we achieve 1.6×–3.4× code coverage compared to the state of the art and uncover over ten previously unknown bugs.

2 Preliminaries

Unlike traditional finite automata, symbolic automata, as defined below, allow transitions to carry "predicates" over an algebra whose domain may be "infinite".

Definition 1 (Symbolic Finite Automata (SFA) [63]). *A symbolic finite automaton (SFA) is a tuple* $(Q, q_0, F, \Psi_D, \Delta)$ *where* Q *is a finite set of states,* $q_0 \in Q$ *is the start state,* $F \subseteq Q$ *is a set of final states,* Ψ_D *is a set of unary first-order predicate that constrains an input variable (the set of all possible inputs is denoted by the domain* D*), and* $\Delta \subseteq Q \times \Psi_D \times Q$ *is a finite set of state transitions, each of which is denoted as* (p, ϕ, q)*.*

Definition 2 (Symbolic Extended Finite Automata (SEFA) [24]). *A symbolic extended finite automaton (SEFA),* $(Q, q_0, F, \Psi_D, \Delta)$*, is the same as an SFA except that a state transition can read* n $(n \geq 1)$ *inputs, i.e.,* Ψ_D *is a set of* n*-ary first-order predicates. Each state transition is denoted by* $(p, \phi|_n, q)$*, which means we move the state from* p *to* q *after reading* n *inputs satisfying the constraint* ϕ*. The integer* n *is omitted when the context is clear.*

Example 1. Fig. 1 shows an SFA where each state transition reads and is constrained by a single input. SEFA allows a transition to read multiple inputs.

Remark 1. Since SEFA reads multiple inputs in a state transition while SFA reads a single input, SEFA is more expressive and can be seen as a natural extension of SFA. However, it is noteworthy that SEFA lacks many of the properties that SFA enjoys. For example, SEFA is not closed under Boolean operations; it is undecidable to check if two SEFAs are equivalent and if the intersection of two SEFAs is empty; and while nondeterminism does not add expressiveness to SFA, it does for SEFA.

Despite these theoretical divergence points, this work focuses on constructing these automata from stateful systems via static analysis rather than leveraging their specific closure or decidability properties. Thus, for the remainder of this paper, the term SFA will be used to encompass both models without further distinction. It should also be noted that in practical applications, techniques such as monadic decomposition [62] can be employed to reduce an SEFA to an SFA under specific conditions.

Definition 3 (Language). *A word* $w = a_1 a_2 \dots a_k$*, where each* $a_i \in D$ *stands for one input, can be accepted by an SFA* M *if and only if*

1. *the word* w *can be partitioned into disjoint sub-strings,* $w = b_1 b_2 \dots b_{k'}$*, where* b_i *is a sub-string, e.g.,* $a_2 a_3 a_4$*, and* $|b_i|$ *denotes the length;*
2. *and the automaton* M *contains a sequence of state transitions* $(q_{i-1}, \phi_i|_{|b_i|}, q_i)$ *for* $1 \leq i \leq k'$ *such that* b_i *satisfies* ϕ_i *and* $q_{k'}$ *is a final state.*

The language accepted by the automaton, denoted by $\mathcal{L}_M$*, is the set of all words that the automaton can accept.*

Table 1: Qualitative comparison with representative prior white-box methods.

Class	Approach	Assumptions	Soundness Guarantee	Scalability
Active Learning	SIGMA* [11]	Perfect Teacher	✗	Minutes†
Static Analysis	FSMEXTRACTOR [13]	Syntactic Patterns	✗	Seconds
Static Analysis	PROTEUS [65]	Almost None	✗	Hours‡
Static Analysis	SEAL (This Work)	Almost None	✓	Minutes

† The inference procedure of Sigma* does not guarantee termination.
‡ Path explosion often causes timeouts in real codebases.

Example 2. Assume the infinite input domain D is the set of all integers. The word 0 0 -50 0 belongs to the language of the SFA in Fig. 1. In the example, the word is actually a sequence of angles read from the sensors. The word can be accepted by the transitions (q_0, ϕ_1, q_0), (q_0, ϕ_1, q_0), (q_0, ϕ_3, q_2), and (q_2, ϕ_1, q_3), meaning that it allows a drone to complete a flip maneuver.

Definition 4 (Soundness). *Given a stateful software system that accepts and only accepts words from the set $\mathcal{L}$, we say its behavioral model, i.e., an SFA M, is sound if and only if $\mathcal{L} \subseteq \mathcal{L}_M$.*

Definition 5 (Precision). *Given two sound SFAs M_1 and M_2 of a stateful system, we say M_1 is more precise than M_2 if and only if $\mathcal{L}_{M_1} \subseteq \mathcal{L}_{M_2}$.*

Example 3. Fig. 3(a) shows a sound SFA for the stateful system in Fig. 1. It is sound because the transition constraint *true* can accept any word. However, it is significantly less precise than the ground-truth SFA in Fig. 1. For example, Fig. 3(a) accepts the word "0 0" while the ground-truth SFA does not. In contrast, the SFA in Fig. 3(b) is not sound because it cannot accept the word "0 0 -50 0", which, however, can be accepted by the stateful system as well as the ground-truth SFA.

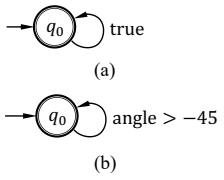


Fig. 3: Soundness and Precision.

Remark 2. Def. 5 gives an intuitive notion of precision. However, in practice, the precision of a static analysis and its resulting SFA depends heavily on exploration strategies rather than solely on the abstract domain it uses, making it hard to compare precision across different static analyses. In our evaluation, we run `Seal` on third-party applications and compare how effectively the inferred SFAs support them. As for soundness, we prove it in the following sections.

As briefly discussed in §1, existing work falls short in *efficiently* inferring a *sound* and *precise* SFA. Table 1 provides a qualitative comparison among representative approaches, showing that, to our best knowledge, our approach is the first one that can efficiently infer a sound and precise SFA from the code of a stateful system.

3 Overview

Our key insight is that stateful systems often exhibit structural separation: distinct states are realized by disjoint code regions. For example, in the drone controller in Fig. 1, the states `Start`, `Roll`, `Pitch`, and `Done` are implemented by separate blocks. This observation suggests a program partitioning technique in which each sub-program corresponds to a state, and state transitions are induced by inter-subprogram dependencies. To ensure the *scalability*, we merge as many paths as possible into a subprogram. Our work ensures *soundness* by fixed-point computation and, meanwhile, yields a *precise* solution through the application of a mixed abstract domain. Let us illustrate our idea via examples.

3.1 Basic Workflow

Fig. 4 lists the steps of SFA inference from the code in Fig. 1. In Fig. 4, the left part is an intermediate automaton, and the right part shows the program environment, which maps program variables, e.g., the variable $\_\mathsf{state}$, to symbolic formulae in an abstract domain. Our analysis preserves two invariants:

- Each state in the automaton represents a path set, denoted by π_i^j and meaning the program paths in the loop body going through lines i-j;
- Each transition is labeled with a program environment, which is the analysis result of the source state and the precondition of the destination state.

Step 1. We apply a precise symbolic abstract domain to begin the analysis, which updates the program environment, computes the path condition, and prunes infeasible paths. Before the analysis enters the while-loop, we have an initial program environment $\mathbb{E}_0$ where $\_\mathsf{state}_0 = \mathsf{Start}$. The subscript in $\_\mathsf{state}_0$ distinguishes the variable $\_\mathsf{state}$ in different program environments. Since $\_\mathsf{state}_0 = \mathsf{Start}$, we analyze the path set π_6^{11} in the first loop iteration. Thus, we create the first intermediate automaton, which includes a state π_6^{11} and the incoming edge labeled by the environment $\mathbb{E}_0$.

We update the program environment during the analysis, yielding $\mathbb{E}_1$ after the first iteration. As shown in the first row of Fig. 4, the value of $\_\mathsf{state}$ after the first iteration is recorded as $\_\mathsf{state}_1$. Due to the code at lines 8-10, $\_\mathsf{state}_1 = \mathsf{Roll}$ if the input $\mathsf{angle} \geq 45$, Pitch if $\mathsf{angle} \leq -45$, Start otherwise.

Step 2. Since $\_\mathsf{state}_1$ can be any value in $\{\ \mathsf{Start}, \mathsf{Roll}, \mathsf{Pitch}\ \}$, in the second loop iteration, we analyze the path set π_6^{23}, which yields $\mathbb{E}_2$. Thus, we create a state π_6^{23} as the destination of the $\mathbb{E}_1$ transition and the source of the $\mathbb{E}_2$ transition.

Step 3. Our analysis partitions a program into disjoint path sets. Steps 1 and 2 produce two overlapping path sets, π_6^{11} and π_6^{23}. Thus, we partition π_6^{23} into π_6^{11} and π_{12}^{23}, as shown by the third intermediate automaton. When partitioning the path set, the states and transitions are partitioned accordingly. Since the outgoing state transition is labeled with an environment that represents the analysis results of a path set, $\mathbb{E}_2$ is partitioned into $\mathbb{E}_{21}$ and $\mathbb{E}_{22}$: $\mathbb{E}_{21}$ denotes the

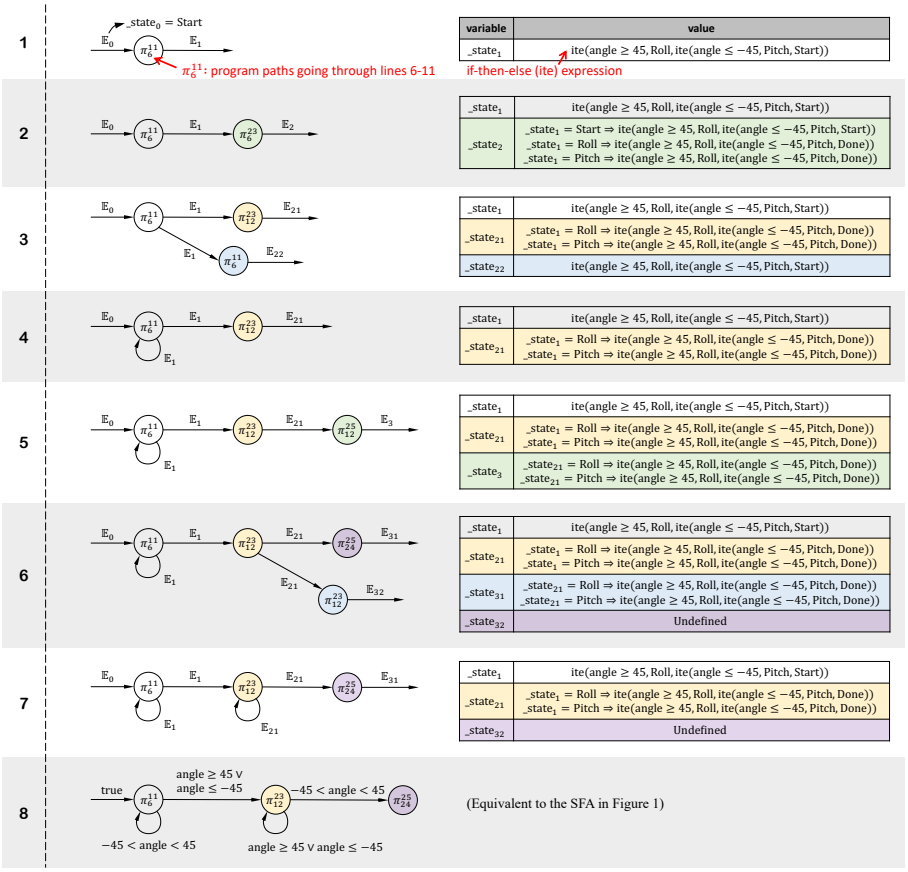


Fig. 4: Steps for inferring SFA from the code snippet in Fig. 1(a).

environment after analyzing the path set π_{12}^{23}, and $\mathbb{E}_{22}$ denotes the environment after π_6^{11}. The values of $\_\mathsf{state}_{21}$ and $\_\mathsf{state}_{22}$ are listed in the table.

Step 4. Our analysis generates only one state for a path set. After Step 3, we have two states representing the same path set π_6^{11} and the same resulting program environments, i.e., $\mathbb{E}_1 = \mathbb{E}_{22}$ due to $\_\mathsf{state}_1 = \_\mathsf{state}_{22}$. In this case, we can safely merge the two states and transitions into a single one.

Step 5. Our analysis continues with the program environment $\mathbb{E}_{21}$, in which $\_\mathsf{state}_{21}$ can be any value in $\{$ Roll, Pitch, Done $\}$. Thus, in the next loop iteration, we analyze the path set π_{12}^{25}, yielding the environment $\mathbb{E}_3$. As such, we create a corresponding state to denote π_{12}^{25}, which has an outgoing transition labeled $\mathbb{E}_3$.

Step 6. Similar to Step 3, since $\pi_{12}^{23} \subseteq \pi_{12}^{25}$, we partition π_{12}^{25} into π_{12}^{23} and π_{24}^{25} such that all states represent disjoint path sets, as illustrated by the sixth automaton. Meanwhile, the environment $\mathbb{E}_3$ is partitioned accordingly into $\mathbb{E}_{31}$ and $\mathbb{E}_{32}$: the former is associated with the state π_{12}^{23} and the latter with π_{24}^{25}.

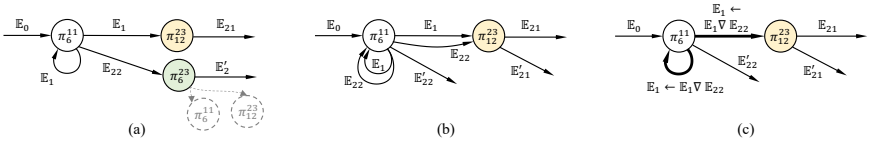


Fig. 5: Illustration of the fixed-point computation.

Step 7. Similar to Step 4, we merge the states that represent the same path set into a single state, as illustrated in the seventh automaton.

Step 8. This step translates the guarded environment automaton into SFA, i.e., translating the program environment over each state transition into formulae over inputs. Specifically, the environment over each transition is translated into the path condition of the destination state, yielding the final SFA. For instance, consider the transition from π_6^{11} to π_{12}^{23}, which is labeled with $\mathbb{E}_1$. Since the path condition of the path set π_{12}^{23} is $\mathsf{\_state}_1 = \mathsf{Roll} \vee \mathsf{\_state}_1 = \mathsf{Pitch}$ due to the branches at line 12 and line 18, we get the path condition $\mathsf{angle} \geq 45 \vee \mathsf{angle} \leq -45$ after replacing the variable $\mathsf{\_state}_1$ in the path condition with its value in $\mathbb{E}_1$. The final SFA produced by our approach is equivalent to the one shown in Fig. 1, which essentially merges the two equivalent states q_1 and q_2.

3.2 Achieving Soundness by Fixed-Point Computation

Our analysis performs a fixed-point computation to ensure soundness. To illustrate, let us revisit Step 4, where two states representing the path set π_6^{11}, as well as their outgoing transitions, can be safely merged because $\mathbb{E}_1 = \mathbb{E}_{22}$. The merging is "safe" because the environment after the path set π_6^{11} reaches a fixed point (i.e., the new environment $\mathbb{E}_{22}$ equals the old $\mathbb{E}_1$). Further analysis with this fixed-point environment will not yield new results.

Fig. 5(a) shows that if we do not merge the transitions and continue the analysis using $\mathbb{E}_{22}$ as the precondition, we will re-analyze the path set π_6^{23} again. As discussed, the path set π_6^{23} is split, and states representing the same path set are merged, yielding Fig. 5(b).

In the fixed-point computation, whenever multiple transitions occur between two states, we merge them into a single transition. Consider Fig. 5(b) and the transitions from π_6^{11} to itself as an example. The transitions are labeled with $\mathbb{E}_1$ and $\mathbb{E}_{22}$. If the two environments are equivalent, we merge them into a single transition. If they are not equivalent, the widening operator of the applied abstract domain is used to merge them into a common over-approximation, denoted as $\mathbb{E}_1 \nabla \mathbb{E}_{22}$ in Fig. 5(c). The widening operators guarantee the convergence of this procedure, after which there is at most one transition between any two states, and an over-approximated program environment labels each transition.

Remark 3. In practice, we do not need to apply the widening operator immediately when we have two transitions between two states. Instead, we can apply the standard "delayed widening" strategy [9]. That is, we can keep applying a

precise join operator and use the widening operator to enforce convergence only when we find a state transition hard to converge via joins.

3.3 Achieving Precision by Mixed Abstract Domain

Although the widening operators can facilitate convergence in fixed-point computation, they can lead to significant precision loss. To preserve precision, we adopt a mixed abstract domain, with a two-fold design.

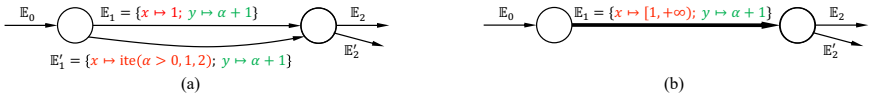


Fig. 6: Mixed abstract domain. (a) An example where we need to convert the precise symbolic domain to intervals. (b) After applying the mixed domain.

First, in different analysis phases, we may use different domains for the same variable. Our analysis always starts by computing the exact symbolic expression for each program variable. The analysis falls back to using a classic abstract domain, e.g., intervals, only when we find that computing a state transition is hard to converge. For example, in Fig. 6(a), we have calculated two transitions between the two states, labeled $\mathbb{E}_1$ and $\mathbb{E}'_1$, respectively. The fixed-point computation asks us to merge the two state transitions into a sound one, but the symbolic values of x are hard to merge. In this case, we convert the precise symbolic abstract domain of x into the intervals, $[1, 1]$ and $[1, 2]$. After merging the intervals with the classic widening operator [16], we get the result $[1, +\infty)$ in Fig. 6(b). Second, in the same analysis phase, we may use different domains for different variables. As mentioned above, to facilitate fixed-point computation, we convert the abstract domain of x to intervals, while y remains in the symbolic abstract domain.

Remark 4. The example above demonstrates a conversion from precise symbolic abstract domains to intervals, specifically to exploit the widening operator for fixed-point computation. Nevertheless, the interval domain represents only one of many viable alternatives; it can be replaced by other classical abstract domains, such as Boxes [30] and Octagons [49], each equipped with its own well-defined widening operator for the same purpose.

4 Detailed Design

This section formalizes our approach. §4.1 introduces the core language and related abstraction. §4.2 and §4.3 present the algorithm and its guarantee for constructing a guarded-environment automaton. §4.4 discusses the conversion from guarded-environment automaton to SFA.

$$
\begin{aligned}
\textit{Expression } E &:= k \mid x \mid x_1 \otimes x_2 \\
\textit{Inputs } I &:= x_1, x_2, \ldots, x_k \leftarrow \textit{inputs}(k) \\
\textit{Command } C &:= x \leftarrow E \mid \textit{return } k \mid \chi : \textit{if } (x) \textit{ then } C_1;\ \textit{else } C_2;\ \textit{endif} \mid C_1; C_2 \\
\textit{Program } P &:= \chi : I;\ C;\ \textit{goto } \chi;
\end{aligned}
$$

Fig. 7: Syntactical representation of stateful systems.

4.1 Program Abstraction

We consider stateful systems implemented in the restricted C-like syntax shown in Fig. 7, the typical pattern in existing implementations. A program P consists of a loop where each iteration reads inputs and updates the state as per the inputs and current state. We enforce this structure by requiring that input statements dominate all other statements in the loop body.

Expressions include constants k, variables x, and the binary expression, where $\otimes$ ranges over arithmetic, logical, and relational operators. Commands include assignments, branches (each labeled with a unique identifier χ), returns (where returning zero denotes acceptance, non-zero for rejection), and sequential composition. The input command reads $k > 0$ values into fresh variables at the start of each iteration. In practice, we handle function calls via clone-based interprocedural analysis [42], pointers using standard pointer analysis [67], and inner loops by transforming them into conditional branches via abstract interpretation.

Mixed Abstract Domain. For SFA inference, we maintain a symbolic program environment $\mathbb{E} \subseteq \mathbb{V} \times \tilde{\mathbb{V}}$ that maps variables $x \in \mathbb{V}$ to abstract values $\tilde{x} \in \tilde{\mathbb{V}}$ in a mixed domain comprising precise symbolic formulas over input symbols α_i and traditional intervals: $\tilde{x} := k \mid \alpha_i \mid \tilde{x}_1 \otimes \tilde{x}_2 \mid \mathsf{ite}(\chi, \tilde{x}_1, \tilde{x}_2) \mid (k_1, k_2)$, where:

- k is a concrete integer constant;
- $\alpha_{i \geq 0}$ denotes the $(i+1)$-th input in the current iteration; $\alpha_{i<0}$ denotes the $(k' + i + 1)$-th input from $\lceil -i/k' \rceil$ iterations ago, where k' is the number of inputs read in each iteration;
- $\tilde{x}_1 \otimes \tilde{x}_2$ represents symbolic arithmetic, logical, or relational expressions;
- $\mathsf{ite}(\chi, \tilde{x}_1, \tilde{x}_2)$ is an if-then-else expression, denoting path-dependent values;
- (k_1, k_2) represents an interval between two integers.

Example 4. Assume that each loop iteration reads one input, and after a loop iteration, $\mathbb{E}(x_1) = (0, 5)$ and $\mathbb{E}(x_2) = \alpha_0$. In the next iteration, all input subscripts are decremented: α_0 becomes α_{-1}. If the second iteration executes $x_3 \leftarrow x_1 + x_2$, we derive $\mathbb{E}(x_3) = (0, 5) + \alpha_{-1}$, capturing the dependency on a previous input.

In what follows, we use $\mathbb{E}[a/b]$ to mean replacing all occurrences of b with a, and $\mathbb{E}[x \mapsto \tilde{x}]$ to mean an update that maps the variable x to a new abstract value $\tilde{x}$. Besides, the domain supports two key operations. The *join* operation $\sqcup_\chi$ precisely merges environments from different branches:

$$\forall x.(\mathbb{E}_1 \sqcup_\chi \mathbb{E}_2)(x) = \mathsf{ite}(\chi, \mathbb{E}_1(x), \mathbb{E}_2(x)) \tag{1}$$

The *widening* operation ∇ ensures termination by over-approximating symbolic values with intervals when its two operands differ:

$$\tilde{x}_1 \nabla \tilde{x}_2 = \begin{cases} \tilde{x}_1 & \text{if } \tilde{x}_1 \equiv \tilde{x}_2 \\ \mathsf{ToInt}(\tilde{x}_1) \nabla_{\mathbf{int}} \mathsf{ToInt}(\tilde{x}_2) & \text{otherwise} \end{cases}, \tag{2}$$

where $\nabla_{\mathbf{int}}$ is standard interval widening [16] and ToInt converts symbolic values to intervals via optimization modulo theories (OMT) [68].

Path-Set Algebra. Recall that our approach partitions the loop body (see Fig. 7) into multiple path sets, each of which is treated as an SFA state. Since the algorithm involves multiple operations over the path sets, we formally define the path-set algebra as a tuple $(\mathbb{P}, \Pi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg, \wp)$ where:

- $\mathbb{P}$ denotes the set of all control-flow paths through the loop body;
- Π is the set of *path-set conditions* — Boolean formulas over branch labels χ; each path-set condition represents a subset of $\mathbb{P}$;
- $\bot$ and $\top$ are the empty and universal path-set conditions;
- $[\![\_]\!] : \Pi \mapsto 2^{\mathbb{P}}$ is the denotation function mapping path-set conditions to path sets; formally, we have $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathbb{P}$;
- $\vee$, $\wedge$, $\neg$ are the standard Boolean operations and satisfy: $[\![\pi \wedge \pi']\!] = [\![\pi]\!] \cap [\![\pi']\!], [\![\pi \vee \pi']\!] = [\![\pi]\!] \cup [\![\pi']\!], [\![\neg\pi]\!] = \mathbb{P} \setminus [\![\pi]\!]$.
- $\wp(\pi) = \pi[\mathbb{E}(\chi)/\chi]$ maps a path-set condition π to the path condition (first-order logic formula over inputs) by substituting branch labels χ in π with their symbolic values in the program environment $\mathbb{E}$.

Example 5. If $\mathbb{E}(\chi_1) = \alpha_0 > 5$, $\mathbb{E}(\chi_2) = \alpha_1 < 0$, and $\pi = \chi_1 \wedge \neg\chi_2$, $\wp(\pi) = (\alpha_0 > 5) \wedge \neg(\alpha_1 < 0)$, which can then be used to check the path feasibility.

4.2 Guarded-Environment Automata

We now describe the construction of a guarded-environment automaton that over-approximates the program's behavior. The states denote the disjoint sets of paths in the loop body. The transitions are guarded by the program environment, representing the post- and pre-conditions of the current and next loop iterations.

Definition 6 (Guarded-Environment Automata). *A guarded–environment automaton is a tuple* $(Q, q_0, F, 2^{\mathbb{V}\times\tilde{\mathbb{V}}}, \Delta)$ *where*

- $Q \subseteq \Pi$ *is a finite set of states: each state is a path-set condition, denoting a set of control-flow paths through the loop body;*
- $q_0 \in Q$ *is the initial state;*
- $F \subseteq Q$ *is a set of final states: each final state denotes a set of paths reaching a return statement and exiting the loop;*
- $2^{\mathbb{V}\times\tilde{\mathbb{V}}}$ *denotes a set of symbolic program environments;*
- $\Delta \subseteq Q \times 2^{\mathbb{V}\times\tilde{\mathbb{V}}} \times Q$ *denotes transitions, each guarded by a symbolic environment* $\mathbb{E} \subseteq \mathbb{V} \times \tilde{\mathbb{V}}$ *and denoted as* $(\pi, \mathbb{E}, \pi')$.

Algorithm 1: Building the Guarded-Environment Automaton

```
1 procedure buildGuardedEnvironmentAutomaton()
2     𝕎 = {(⊥, ∅)};  Δ = ∅ ;                    /* worklist of states and transition set */
3     while 𝕎 ≠ ∅ do
4         (π0, 𝔼0) ← pop element from 𝕎;
5         (π1, 𝔼1), (π2, 𝔼2), ··· ← analyze one loop iteration from 𝔼0;
6         Δ ← Δ ∪ {(π0, 𝔼0, πi)} for all i > 0;
7         𝕎 ← 𝕎 ∪ {(πi, 𝔼i)} for non-final state πi;
8         forall i > 0 do refine(𝕎, Δ, πi) ;              /* refine partition, §4.3 */
9     return Δ;
```

Alg. 1 infers a guarded-environment automaton by symbolically evaluating the program. It explores feasible paths, computes the symbolic program environments, and refines the path-set partitions (Lines 3–8 of Alg. 1):

1. Line 4: Pop $(\pi_0, \mathbb{E}_0)$ from the worklist, where $[\![\pi_0]\!]$ is a path set and $\mathbb{E}_0$ is the program environment resulting from symbolically executing paths in $[\![\pi_0]\!]$;
2. Line 5: Symbolically execute one loop iteration using $\mathbb{E}_0$ as the precondition, yielding a list of feasible path sets and program environments, $(\pi_i, \mathbb{E}_i)$;
3. Line 6: Add state transitions $(\pi_0, \mathbb{E}_0, \pi_i)$ into the automaton;
4. Line 7: Add $(\pi_i, \mathbb{E}_i)$ into the worklist for further exploration if π_i does not represent a final state (i.e., not end with a return statement);
5. Line 8: Refine the path set partition to ensure soundness and termination.

Fig. 8 defines the symbolic evaluation rules used in Line 5 of Alg. 1 via judgments of the form $\mathbb{E}, \pi \vdash C : \mathbb{E}', \pi'$, which reads: executing command C from environment $\mathbb{E}$ along paths satisfying π yields environment $\mathbb{E}'$ and a new path-set condition π'. The Init rule processes input statements by renaming previous iteration's inputs (decrementing subscripts by k, the number of inputs) and binding fresh symbolic inputs $\alpha_0, \ldots, \alpha_{k-1}$ to the input variables. This enables tracking data flow across iterations. The Assign rule evaluates the right-hand side expression symbolically. Constants are mapped directly, variable references look up the current symbolic value, and operations compose symbolic values using the same operator. The Ret rule marks the current path as final without modifying the environment. The Seq rule threads the environment and path-set through sequential composition.

The Branch rule handles conditionals by first recording the branch condition in the environment ($\mathbb{E}[\chi \mapsto \tilde{x}]$). This enables later conversion from a path-set condition to a path condition, thereby checking path feasibility via an SMT solver. It then analyzes both branches, each with the path-set refined by adding the appropriate branch constraint (χ or $\neg\chi$). The three cases in the conclusion handle: (1) when the else branch is infeasible, return only the then branch result; (2) when the then branch is infeasible, return only the else branch result; (3) otherwise, join both branches using Equation (1).

Lemma 1. *Assume Line 8 of Alg. 1 preserves soundness. The environment* $\mathbb{E}$ *in each transition* $(\pi, \mathbb{E}, \_) \in \Delta$ *or worklist item* $(\pi, \mathbb{E}) \in \mathbb{W}$ *soundly abstracts the program's behavior after executing any path in* $[\![\pi]\!]$.

$$\frac{\pi = \top \quad \mathbb{E} = \mathbb{E}_0[\alpha_{i-k}/\alpha_i][x_i \mapsto \alpha_{i-1}]}{\mathbb{E}_0, \pi \vdash x_1, \ldots, x_k \leftarrow \mathit{inputs}(k) : \mathbb{E}, \pi}\ \textsc{Init} \qquad \frac{\mathbb{E}_1, \pi_1 \vdash C_1 : \mathbb{E}_2, \pi_2 \quad \mathbb{E}_2, \pi_2 \vdash C_2 : \mathbb{E}_3, \pi_3}{\mathbb{E}_1, \pi_1 \vdash C_1; C_2 : \mathbb{E}_3, \pi_3}\ \textsc{Seq}$$

$$\frac{}{\mathbb{E}, \pi \vdash \mathit{return}\ k : \mathbb{E}, \pi}\ \textsc{Ret} \qquad \frac{\mathbb{E}(x_1) = \tilde{x}_1 \quad \mathbb{E}(x_2) = \tilde{x}_2}{\mathbb{E}, \pi \vdash x \leftarrow E : \begin{cases} \mathbb{E}[x \mapsto k], \pi & E := k \\ \mathbb{E}[x \mapsto \tilde{x}_1], \pi & E := x_1 \\ \mathbb{E}[x \mapsto \tilde{x}_1 \otimes \tilde{x}_2], \pi & E := x_1 \otimes x_2 \end{cases}}\ \textsc{Assign}$$

$$\frac{\mathbb{E}(x) = \tilde{x} \quad \mathbb{E}[\chi \mapsto \tilde{x}], \pi \wedge \chi \vdash C_1 : \mathbb{E}_1, \pi_1 \quad \mathbb{E}[\chi \mapsto \tilde{x}], \pi \wedge \neg\chi \vdash C_2 : \mathbb{E}_2, \pi_2}{\mathbb{E}, \pi \vdash \chi : \mathit{if}\ (x)\ \mathit{then}\ C_1;\ \mathit{else}\ C_2 : \begin{cases} \mathbb{E}_1, \pi_1 & \text{if unsat}(\wp(\pi) \wedge \neg\tilde{x}) \\ \mathbb{E}_2, \pi_2 & \text{if unsat}(\wp(\pi) \wedge \tilde{x}) \\ \mathbb{E}_1 \sqcup_\chi \mathbb{E}_2, \pi_1 \vee \pi_2 & \text{otherwise} \end{cases}}\ \textsc{Branch}$$

Fig. 8: Symbolic evaluation rules.

Proof. By induction on the number of while-loop iterations, the base case holds trivially for the initial worklist item $(\bot, \emptyset)$. For the inductive case, assume the invariant holds for $(\pi_0, \mathbb{E}_0)$. Each evaluation rule preserves soundness. For example, Assign performs exact symbolic evaluation; Branch joins feasible branches, and the path-set condition π_i precisely characterizes the paths analyzed.

4.3 Partition Refinement and Sound Termination

To construct a sound guarded-environment automaton, two issues remain: (1) Lemma 1 assumes Line 8 of Alg. 1 preserves soundness, and (2) Alg. 1 may not terminate due to unbounded worklist growth. We address both issues with the refinement procedure called at Line 8 of Alg. 1, which (1) enforces disjoint path sets and (2) ensures convergence to a finite fixpoint. Alg. 2 implements the refinement, where Line 2 ensures the disjointness of path sets, and Lines 2-3 together ensure the sound termination.

Path-Set Partition Refinement. Lines 4-10 maintain the invariant that all path-sets are pairwise disjoint. When a newly discovered path-set π overlaps with an existing π', Lines 5-8 partition them into three disjoint pieces: $\pi \wedge \pi'$ (intersection), $\pi \wedge \neg\pi'$ (difference), and $\pi' \wedge \neg\pi$ (difference).

When partitioning a path set π into subsets π_i, lines 9-10 also partition the environment $\mathbb{E}$ into corresponding sub-environments $\mathbb{E}_i$, via formula simplification [22], which removes infeasible ite branches based on the refined condition π_i:

$$\forall x \in \mathrm{dom}(\mathbb{E}).\mathbb{E}_i(x) = \mathrm{simplify}(\mathbb{E}(x), \pi_i) \tag{3}$$

Example 6. Assume $\mathbb{E}(x) = \mathsf{ite}(\chi, \tilde{x}_1, \tilde{x}_2)$ and $\pi = \chi \vee \neg\chi$ is partitioned into $\pi_1 = \chi$ and $\pi_2 = \neg\chi$. Thus, $\mathbb{E}_1(x) = \tilde{x}_1$ is the result of simplifying $\mathbb{E}(x)$ under the condition π_1. Similarly, $\mathbb{E}_2(x) = \tilde{x}_2$.

Lemma 2. *Path-set partitioning preserves Lemma 1: each partitioned environment soundly abstracts its corresponding path-set.*

Proof. Equation (3) performs formula simplification that removes infeasible values based on the refined path-set condition. Since simplification only removes values inconsistent with π_i, it preserves soundness for paths in $[\![\pi_i]\!]$.

Algorithm 2: Partition refinement for termination

```
1  procedure refine(𝕎, Δ, π)
2      abstractStates(𝕎, Δ, π) → π is partitioned into a list of disjoint π_i with (π_i, 𝔼_i);
3      abstractTransitions(𝕎, Δ, (π_i, 𝔼_i)) for each (π_i, 𝔼_i);
4  procedure abstractStates(𝕎, Δ, π)
5      assume ∃π' ≠ π: [[π' ∧ π]] ≠ ∅, [[π' ∧ ¬π]] ≠ ∅, and [[π ∧ ¬π']] ≠ ∅;
6      π_1 ← π ∧ π';  π_2 ← π ∧ ¬π';  π_3 ← ¬π ∧ π';
7      replace (_, _, π) ∈ Δ with (_, _, π_1) and (_, _, π_2);        /* π → π_1, π_2 */
8      replace (_, _, π') ∈ Δ with (_, _, π_1) and (_, _, π_3);       /* π' → π_1, π_3 */
9      replace (π, 𝔼) in 𝕎 and Δ with (π_1, 𝔼_1) and (π_2, 𝔼_2) ;     /* 𝔼 → 𝔼_1, 𝔼_2 */
10     replace (π', 𝔼') in 𝕎 and Δ with (π_1, 𝔼'_1) and (π_3, 𝔼'_3) ;  /* 𝔼' → 𝔼'_1, 𝔼'_3 */
11 procedure abstractTransitions(𝕎, Δ, (π, 𝔼) ∈ 𝕎)
12     foreach transition pair (π', 𝔼_1, π), (π', 𝔼_2, π) ∈ Δ do
13         assume (π', 𝔼_2, π) ∈ Δ is a new transition added due to lines 7-10;
14         replace both with (π', 𝔼_1 ∇ 𝔼_2, π);                    /* widen environments */
15         if 𝔼_1 ∇ 𝔼_2 = 𝔼_1 then 𝕎 ← 𝕎 \ {(π, 𝔼)};                 /* fixed point reached */
```

Lemma 3. *The automaton has at most n states, where n is the number of distinct control-flow paths in the loop body.*

Proof. Each state corresponds to a unique path-set condition. The partition refinement maintains disjointness, so the maximum number of states is bounded by the number of paths in the control flow graph, which is finite.

Transition Abstraction via Widening. Lines 11-15 handle the case where multiple transitions occur between two states. Line 14 merges the transitions into a single transition by widening the symbolic program environments using Equation (2). As shown in line 15, when the widened environment equals one of the environments ($\mathbb{E}_1 \nabla \mathbb{E}_2 = \mathbb{E}_1$), we have reached a fixed point: $\mathbb{E}_1$ over-approximates $\mathbb{E}_2$, so further analysis from $\mathbb{E}_2$ subsumes analysis from $\mathbb{E}_1$. We therefore remove $(\pi, \mathbb{E})$ from the worklist to ensure the algorithm terminates.

Lemma 4. *Transition abstraction via widening preserves Lemma 1.*

Proof. The widening operator ∇ is sound by definition: $\mathbb{E}_1 \nabla \mathbb{E}_2$ over-approximates both $\mathbb{E}_1$ and $\mathbb{E}_2$. Therefore, the merged environment soundly abstracts states reachable via either transition.

Sound Termination. As defined, the widening operator is degenerated into the standard interval widening if a precise join operation does not converge (see Remark 4). Since the intervals' widening operator ensures convergence [16], there exists at most one fixed-point transition between two states. Combined with Lemma 3, the algorithm must terminate with a finite number of states and state transitions.

Theorem 1. *Alg. 1 terminates and produces a guarded-environment automaton where each transition $(\pi, \mathbb{E}, \pi')$ satisfies: $\mathbb{E}$ soundly abstracts the program state after executing any trace in $[\![\pi]\!]$.*

Proof. Termination follows from Lemma 3 (finite states) and the convergence of interval widening. Soundness follows from Lemmas 1, 2, and 4.

Algorithm 3: Converting to SFA

```
1 procedure toSFA(Δ)
2     replace each (π, 𝔼, π′) ∈ Δ with (π, ℘(π′), π′);
3     foreach (π, ϕ, π′) ∈ Δ do
4         a) rmLookback((π, ϕ, π′), Δ); b) eliminate α_{i<0} in ϕ via quantifier elimination;
5 procedure rmLookback((π, ϕ, π′), Δ)
      /* The underscore symbol _ means a don't-care parameter.                    */
6     if ϕ has form f(α_{i<0}) ∧ h(_) then
7         (_, ϕ′, _) ∈ Δ ← the nth transition before (π, ϕ, π′) where n = ⌈−i/k⌉;
8         if (π, ϕ, π′) post-dominates (_, ϕ′, _) then
9             replace ϕ with h(_), ϕ′ with ϕ′ ∧ f(α_{i+n×k});  /* move condition backward */
```

4.4 From Guarded-Environment Automaton to SFA

Finally, Alg. 3 converts the guarded-environment automaton to SFA, where each state transition is guarded by predicates over input symbols only from the current loop iteration. To this end, Line 2 replaces the symbolic environment over state transitions with path conditions (see the final step in Fig. 4), and Line 4 applies a two–step normalization to eliminate look-back symbols $\alpha_{i<0}$ that reference previous inputs. First, it moves transition constraints backward when post-dominance guarantees all paths reaching the current transition must pass through an earlier transition (see Example 7). For the remaining look-back symbols where post-dominance does not hold, the second step applies quantifier elimination (QE) [26] to eliminate look-back symbols, obtaining an over-approximation that depends only on current inputs (see Example 8).

Example 7. The tail transition in Fig. 9(a) contains a look-back symbol α_{-1} that refers to the previous input. Since the tail transition post-dominates previous ones, we can safely move it back, yielding $0 < \alpha_0 < 5$ and $\alpha_0 > 0$, respectively.

Example 8. The look-back symbol in Fig. 9(b) can be eliminated by quainter elimination. Applying $QE(\exists\alpha_{-1}.\alpha_{-1} < 1 \wedge \alpha_{-1} + \alpha_0 > 0)$ yields the post-QE condition $\alpha_0 > -1$. While the QE result is sound, it enlarges the transition guard. For example, the word "-10, 1" acceptable by the post-QE transitions is not in the scope of the original transition guards.

Theorem 2. *If the implementation of a stateful system accepts input word $w = a_1 \dots a_n$, then the inferred automaton after Line 2, Algorithm 3 accepts w.*

Proof. If w is accepted, execution follows a sequence of paths $p_1, \dots, p_n$ where $|[\![\pi_i]\!]| = 1$ for the path-set π_i corresponding to p_i. By Theorem 1, for each consecutive pair (p_i, p_{i+1}) there exists a transition $(\pi'_i, \mathbb{E}, \pi'_{i+1})$ with $[\![p_i]\!] \subseteq [\![\pi'_i]\!]$, where $\mathbb{E}$ soundly abstracts the state after p_i. The path condition $\phi'_{i+1} = \pi'_{i+1}[\mathbb{E}(\chi)/\chi]$ over-approximates the concrete condition required to follow p_{i+1}. Therefore, the sequence of transitions labeled by $\phi'_1, \dots, \phi'_n$ accepts w.

Theorem 3. *Lines 3-4 of Alg. 3 preserve Theorem 2's soundness guarantee.*

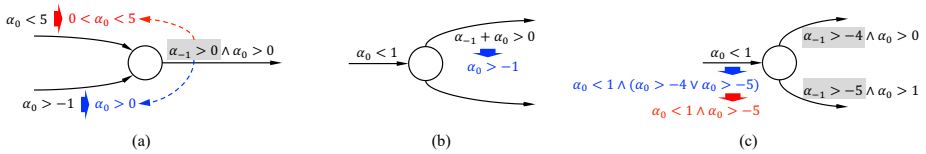


Fig. 9: Lookback elimination.

Proof. The post-dominance check ensures that moving constraints backward is sound: every accepting run must pass through the earlier transition. Quantifier elimination produces a sound over-approximation. Therefore, the resulting automaton still accepts all words accepted by the implementation.

Possible Enhancements to Precision. While Example 8 shows that eagerly applying QE compromises precision, this issue is particularly prevalent in stateful systems characterized by deep temporal dependencies, which leads to frequent look-back symbols. Since these systems deviate from a strict SFA structure, the inferred SFA is inherently an imprecise approximation. However, this loss of precision could be avoided in certain scenarios. We discuss two of them below.

First, as shown in Fig. 9(c), the two tail transitions do not post-dominate the initial transition. The constraints over look-back symbols could also be moved backward in the form of a disjunction, $\alpha_0 > -4 \vee \alpha_0 > -5$. This approach ensures the result remains sound for both tail transitions while maintaining exact precision for the bottom transition.

Second, given two consecutive transitions $(\pi_0, \alpha_0 < 1, \pi_1), (\pi_1, \alpha_{-1} + \alpha_0 > 0, \pi_2)$ and assuming that there is no other incoming transitions to and outgoing transitions from π_1, we can merge them into a single transition without look-backs: $(\pi_0, \alpha_0 < 1 \wedge \alpha_0 + \alpha_1 > 0, \pi_2)$. This merge does not lose precision and still conforms to the SEFA definition.

5 Evaluation

We implemented our approach, `Seal`, atop LLVM [40]. Programs are first compiled to LLVM IR, which serves as input to `Seal` for SFA inference. During analysis, we employ Z3 [21] to compute symbolic expressions and evaluate path conditions. Before analyzing each program, a necessary and only manual, preprocessing step is identifying APIs that read external inputs (e.g., `read_gyro()` in Fig. 1(a)). This involves annotating several (typically < 5) functions per system, which serves as a one-time configuration. Our evaluation aims to answer the following three research questions:

- **RQ1** (Efficiency): How long does it take to infer an SFA from the source code? How much more efficient is `Seal` than the state of the art?
- **RQ2** (Effectiveness): How similar is an inferred SFA compared to the ground-truth automaton? An ablation study is also included.

Table 2: Subjects used in the evaluation.

ID	Name	Description	ID	Name	Description
1	ORP	Octave resource protocol	7	Rotate	Copter rotation control
2	MAVLINK	Protocol for drones	8	Flip	Copter flipping control
3	TINY	Serial interface protocol	9	RTL	Return-to-launch
4	SML	Smart meter protocol	10	SmartRTL	Enhanced RTL
5	MIDI	Musical device protocol	11	Throw	Launch via throw
6	KISS	Amateur radio protocol	12	Takeoff	Plane taking-off control

– **RQ3** (Usefulness): When used in applications such as dynamic model checking, does the inferred SFA improve the efficacy of the applications?

Baselines. Table 1 provides a qualitative comparison between the state of the art and our approach. In this section, we provide a quantitative comparison. Specifically, the baselines include the following.

– Sigma* [11] is a white-box active learning technique originally designed for inferring a symbolic finite transducer (SFT), an extension of SFA.
– FSMExtractor [13] is a pattern-sensitive program analyzer that assumes simple and specific patterns of SFA implementations.
– Proteus [65] infers a path-dependency automaton, a variant of SFA. It does not provide soundness guarantees and is not scalable due to path explosion.

For RQ1, we measure the time required to infer automata using each tool. We set a 3-hour timeout for all baselines. This threshold is generous, as `Seal` completes inference in under a few minutes for all subjects.

For RQ2, we assess the similarity between inferred automata and manually constructed ground-truth automata. Direct structural comparison is infeasible because it is undecidable to check the equivalence of symbolic automata [24]. Instead, we adopt a behavioral similarity metric: we generate 10,000 input words accepted by the ground-truth automaton. For each word w, let $|w|$ and $\|w\|$ denote the number of states traversed by the ground-truth and inferred automata, respectively, when processing w. If the inferred automaton rejects w, we set $\|w\| = 0$. We define the similarity score for w as $\min(|w|, \|w\|)/\max(|w|, \|w\|)$, and report the average similarity across all words.

There may be many ways to define such a similarity metric. However, we argue that our metric provides insights into the quality of an automaton, particularly its internal structure, by counting the distinct states required to accept a word. Suppose we infer a trivially sound one-state automaton that accepts everything. The similarity score is $1/\max(|w|, 1)$, which is low. So the metric does penalize trivial over-approximation.

To further evaluate RQ2, we conduct an ablation study to isolate the impact of our mixed abstract domain. We show that using a single abstract domain either prevents convergence within the time budget or results in poor precision.

For RQ3, we apply the inferred automata to dynamic model checking [27]. We measure the improvements in code coverage and bug-finding capability.

Table 3: Efficiency and effectiveness of automaton inference (RQ1 and RQ2).

ID	# State/# Transition				Efficiency (in seconds)				Effectiveness (in similarity)			
	Seal	Sigma*	F10r	Proteus	Seal	Sigma*	F10r	Proteus	Seal	Sigma*	F10r	Proteus
1	6/12	1/2	4/8	42/102	127	4	13	1020	0.92	0.32	0.72	0.97
2	38/166	2/4	16/27	-	301	5	142	-	**0.90**	0.23	0.58	-
3	15/60	1/2	7/16	151/879	138	3	39	3907	**0.95**	0.21	0.65	0.95
4	32/91	3/8	19/34	-	278	5	69	-	**0.95**	0.20	0.77	-
5	15/120	1/2	3/6	765/4011	156	2	33	9001	0.91	0.11	0.83	0.96
6	7/15	1/2	4/9	24/143	96	2	10	459	0.98	0.33	0.67	0.99
7	8/23	1/2	5/15	-	66	1	12	-	**0.99**	0.29	0.69	-
8	12/32	2/3	6/13	-	71	3	12	-	**0.98**	0.26	0.70	-
9	15/50	2/5	6/17	-	109	6	13	-	**0.99**	0.19	0.81	-
10	12/81	1/2	5/10	-	231	7	10	-	**0.93**	0.19	0.76	-
11	10/30	1/2	6/9	-	110	5	15	-	**0.97**	0.11	0.65	-
12	9/38	1/2	3/7	-	111	5	11	-	**0.92**	0.27	0.48	-

Subjects & Environment. In the experiments, we include two typical categories of SFA applications, totaling 12, as shown in Table 2. The first six programs use SFA to parse network messages in certain protocols [50,46,59,56,48,43], and the remaining six are from the autopilot system, Ardupilot [3], and employ SFA to control autopilot systems, such as a copter. All experiments are run on a MacBook Pro (16-inch, 2019) equipped with an 8-core, 16-thread Intel Core i9 CPU with 2.30GHz speed and 32GB of memory.

5.1 RQ1: Efficiency

Table 3 shows the time cost, number of states, and transitions in the inferred automata. As reported, `Seal`, Sigma*, and FSMExtractor (abbr. F10r) can finish their inference in a few minutes and even seconds, whereas Proteus takes longer and often cannot finish because it must enumerate all program pathways, leading to the path-explosion problem. Sigma* and FSMExtractor are faster because of their designs for their original purposes. Sigma* is initially designed to infer symbolic transducers, which extend symbolic automata by enabling outputs during state transitions. Thus, its design relies on the code's output operations. When applied to SFA implementations, due to the absence of output operations, it often produces an overly conservative result like the one in Fig. 3(a). FSMExtractor is fast because it captures only simple code patterns and does not involve fixed-point computation to guarantee soundness.

5.2 RQ2: Effectiveness

Table 3 reports the estimated similarity between inferred automata and the ground truth, where values closer to one indicate greater similarity. `Seal` achieves high similarity (0.90–0.99, median 0.95), whereas Sigma* and FSMExtractor range from 0.11 to 0.83 (median 0.48). Although Proteus achieves comparable or better similarity when it succeeds, its reliance on precise per-path analysis rather than sound abstraction leads to path explosion, preventing it from completing more than half of the programs.

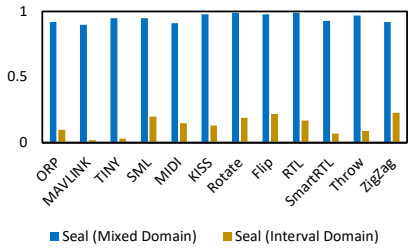


Fig. 10: Similarity in ablation study.

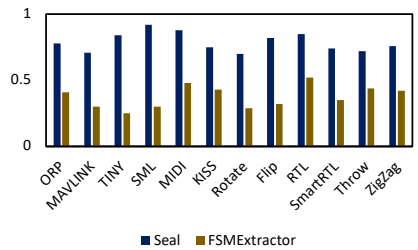


Fig. 11: Coverage in dyn. model checking.

As explained, `Seal` employs a mixed abstract domain to ensure soundness and precision. To assess its impact, we conducted an ablation study in which we removed the precise symbolic domain and retained only the interval domain to ensure termination (using only the symbolic domain would not converge). As shown in Fig. 10, this removal significantly degrades SFA quality under the similarity metrics, highlighting the importance of the mixed domain.

5.3 RQ3: Usefulness

We applied the inferred SFA to guide dynamic model checking, which *"consists of adapting model checking into a form of systematic testing that applies to industrial-scale software [27]"*. Compared to traditional testing, dynamic model checking provides better coverage because SFA allows us to generate specific input sequences that thoroughly test the deep states of a stateful system.

In this experiment, we use the SFAs inferred by FSMExtractor and our approach, as they consistently produce SFAs. To perform dynamic model checking on the stateful message parsers, BooFuzz [10] allows us to specify the message formats represented by the inferred SFAs. For autopilot systems, PGFuzz [52] enables us to generate input sequences from the inferred SFAs. Each testing procedure is performed on a three-hour budget.

Fig. 11 shows the coverage achieved by the dynamic model checking. The X-axis lists the 12 programs, and the Y-axis shows statement coverage. As shown in Fig. 11, since we can infer a better automaton, dynamic model checkers using the SFA inferred by our approach achieve 1.6×–3.4× coverage compared to those using FSMExtractor.

6 Related Work

In addition to the related work in §1 and §2, this section extends the discussion to other related works on abstract interpretation.

Reduced products [17] combine multiple abstract domains (e.g., intervals × congruences) to improve precision. However, the combination is typically fixed

and applied uniformly across all variables. In contrast, `Seal` introduces an explicit per-variable domain and only downgrades variables responsible for non-convergence, while the remaining variables stay symbolic. This selective demotion preserves symbolic guards whenever possible, but still enforces termination.

Trace partitioning [45,53] refines abstract states by splitting traces based on selected predicates, thereby improving precision by separating behaviors. While effective, trace partitioning adapts the *state-space partition* at the level of abstract states/traces; it lacks the ability to migrate variables between symbolic and concrete domains on a per-iteration basis, which is essential for stabilizing symbolic transitions in the presence of input-driven control.

Guided static analysis [28] applies standard static analysis "as is" to a sequence of restricted parts of the original program, termed as "program restrictions". It requires that the sequence of program restrictions generated by a non-decreasing chain of abstract states must converge to the original program in finitely many steps. In comparison, the program restrictions in our approach are disjoint and map directly to SFA states. Also, we do not apply static analysis "as is" but adjust the abstract domain of individual variables in a demand-driven manner during the analysis.

Finally, most symbolic analyses either represent a single execution prefix (as in symbolic execution) or infer loop invariants without an explicit model of *stream look-back*. `Seal`'s symbolic domain natively supports time-indexed input symbols, including negative indices, enabling precise modeling of cross-iteration dependencies. Predicate abstraction and CEGAR [8,15] typically begin with coarse abstractions and refine them globally in response to spurious counterexamples. In comparison, `Seal` performs local, demand-driven precision adjustment at the level of individual variables. This granularity is critical in SFA inference, where imprecision in a small set of numeric variables (e.g., counters) should not compromise the symbolic structure of unrelated input guards.

7 Conclusion

We have formally presented `Seal`, a static analysis framework that can efficiently infer sound and precise behavior models, in the form of SFA/SEFA, from the code of stateful systems. We apply the model to guide dynamic model checking, which demonstrates the usefulness of our approach in practice.

Data-Availability Statement. A public version of our tool is available at `https://doi.org/10.5281/zenodo.20133756`.

Acknowledgments. We sincerely thank the anonymous reviewers for their suggestions. This work was partially supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025-XDXM118), the 111 Center (No. B26023), and the National Natural Science Foundation of China (62302434 and U2341212). Qingkai Shi is the corresponding author.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alur, R., D'Antoni, L., Raghothaman, M.: Drex: A declarative language for efficiently evaluating regular string transformations. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 125–137. POPL '15, ACM (2015). https://doi.org/10.1145/2676726.2676981
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
3. ArduPilot: Ardupilot - versatile, trusted, open. https://www.ardupilot.org/ (2024)
4. Argyros, G., D'Antoni, L.: The learnability of symbolic automata. In: Proceedings of the 30th International Conference on Computer Aided Verification. pp. 427–445. CAV '18, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_23
5. Argyros, G., Stais, I., Jana, S., Keromytis, A.D., Kiayias, A.: Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1690–1701. CCS '16, ACM (2016). https://doi.org/10.1145/2976749.2978383
6. Argyros, G., Stais, I., Kiayias, A., Keromytis, A.D.: Back in black: Towards formal, black box analysis of sanitizers and filters. In: Proceedings of the 37th IEEE Symposium on Security and Privacy. pp. 91–109. S&P '16, IEEE (2016). https://doi.org/10.1109/SP.2016.14
7. Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful greybox fuzzing. In: Proceedings of the 31st USENIX Security Symposium. pp. 3255–3272. USENIX Security '22, USENIX (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/ba
8. Ball, T., Rajamani, S.K.: The slam toolkit. In: Proceedings of the 13th International Conference on Computer Aided Verification. pp. 260–264. CAV '01, Springer (2001). https://doi.org/3-540-44585-4_25
9. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 196–207. PLDI '03, ACM (2003). https://doi.org/10.1145/781131.781153
10. BooFuzz: A fork and successor of the sulley fuzzing framework. https://github.com/jtpereyda/boofuzz (2024)
11. Botinčan, M., Babić, D.: Sigma*: Symbolic learning of input-output specifications. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 443–456. POPL '13, ACM (2013). https://doi.org/10.1145/2429069.2429123
12. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Proceedings of the 16th International Conference on Computer Aided Verification. pp. 372–386. CAV '04, Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_29
13. Chen, Y., Song, L., Xing, X., Xu, F., Wu, W.: Automated finite state machine extraction. In: ACM Workshop on Forming an Ecosystem Around Software Transformation. pp. 9–15. FEAST '19, ACM (2019). https://doi.org/10.1145/3338502.3359760

14. Chubachi, K., Hendrian, D., Yoshinaka, R., Shinohara, A.: Query learning algorithm for residual symbolic finite automata. In: Proceedings of the 10th International Symposium on Games, Automata, Logics, and Formal Verification. pp. 140–153. G and ALF '19, Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.305.10
15. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of the 12th International Conference on Computer Aided Verification. pp. 154–169. CAV '00, Springer (2000). https://doi.org/10.1007/10722167_15
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM (1977). https://doi.org/10.1145/512950.512973
17. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 269–282. POPL '79, ACM (1979). https://doi.org/10.1145/567752.567778
18. Dalla Preda, M., Giacobazzi, R., Lakhotia, A., Mastroeni, I.: Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In: Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 329–341. POPL '15, ACM (2015). https://doi.org/10.1145/2676726.2676986
19. Daniele, C., Andarzian, S.B., Poll, E.: Fuzzers for stateful systems: Survey and research directions. ACM Computing Surveys (CSUR) **56**(9), 222:1–222:23 (2024). https://doi.org/10.1145/3648468
20. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 541–553. POPL '14, ACM (2014). https://doi.org/10.1145/2535838.2535849
21. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS '08, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
22. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. In: Proceedings of the 17th International Static Analysis Symposium. pp. 236–252. SAS '10, Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_15
23. Drews, S., D'Antoni, L.: Learning symbolic automata. In: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 173–189. TACAS '17, Springer (2017). https://doi.org/10.1007/978-3-662-54577-5_10
24. D'Antoni, L., Veanes, M.: Extended symbolic finite automata and transducers. Formal Methods in System Design **47**, 93–119 (2015). https://doi.org/10.1007/s10703-015-0233-4
25. D'Antoni, L., Veanes, M.: Forward bisimulations for nondeterministic symbolic finite automata. In: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 518–534. TACAS '17, Springer (2017). https://doi.org/10.1007/978-3-662-54577-5_30

26. Garcia-Contreras, I., Govind, V.K.H., Shoham, S., Gurfinkel, A.: Fast approximations of quantifier elimination. In: Proceedings of the 35th International Conference on Computer Aided Verification. pp. 64–86. CAV '23, Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_4
27. Godefroid, P.: Between testing and verification: Dynamic software model checking. In: Dependable Software Systems Engineering, pp. 99–116. IOS Press (2016). https://doi.org/10.3233/978-1-61499-627-9-99
28. Gopan, D., Reps, T.: Guided static analysis. In: Proceedings of the 14th International Static Analysis Symposium. pp. 349–365. SAS '07, Springer (2007). https://doi.org/10.1007/978-3-540-74061-2_22
29. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theoretical Computer Science **411**(47), 4029–4054 (2010). https://doi.org/10.1016/j.tcs.2010.07.008
30. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: Proceedings of the 17th International Static Analysis Symposium. pp. 287–303. SAS '10, Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_18
31. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proceedings of the 16th International Static Analysis Symposium. pp. 69–85. SAS '09, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
32. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 36–52. CAV '13, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
33. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 58–70. POPL '02, ACM (2002). https://doi.org/10.1145/503272.503279
34. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with {BEK}. In: Proceedings of the 20th USENIX Security Symposium. pp. 1–16. USENIX Security '11, USENIX (2011), https://www.usenix.org/conference/usenix-security-11/fast-and-precise-sanitizer-analysis-bek
35. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 263–277. VMCAI '11, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_19
36. Isberner, M., Howar, F., Steffen, B.: Inferring automata with state-local alphabet abstractions. In: Proceedings of the 5th NASA Formal Methods Symposium. pp. 124–138. NFM '13, Springer (2013). https://doi.org/10.1007/978-3-642-38088-4_9
37. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys (CSUR) **41**(4), 21:1–21:54 (2009). https://doi.org/10.1145/1592434.1592438
38. Keil, M., Thiemann, P.: Symbolic solving of extended regular expression inequalities. In: Proceedings of the 34th International Conference on Foundation of Software Technology and Theoretical Computer Science. pp. 175–186. FSTTCS '14, LIPIcs (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175
39. Kim, K., Kim, T., Warraich, E., Lee, B., Butler, K.R.B., Bianchi, A., Tian, D.J.: Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In: IEEE Symposium on Security and Privacy. pp. 2212–2229. S&P '22, IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833593

40. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization. pp. 75:1–75:12. CGO '04, IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665
41. Li, J., Li, S., Sun, G., Chen, T., Yu, H.: Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. IEEE Transactions on Information Forensics and Security **17**, 2673–2687 (2022). https://doi.org/10.1109/TIFS.2022.3192991
42. Li, L., Cifuentes, C., Keynes, N.: Boosting the performance of flow-sensitive points-to analysis using value flow. In: Proceedings of the 13th European Software Engineering Conference Held Jointly with the 19th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. pp. 343–353. ESEC/FSE '11, ACM (2011). https://doi.org/10.1145/2025113.2025160
43. LibKiss: Kiss protocol (keep it simple stupid) parsing library for amateur radio. https://github.com/memoryhole/libkiss (2023)
44. Maler, O., Mens, I.E.: A generic algorithm for learning symbolic automata from membership queries. In: Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, pp. 146–169. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_8
45. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Proceedings of the 14th European Symposium on Programming. pp. 5–20. ESOP '05, Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_2
46. MAVLink: Mavlink developer guide. https://mavlink.io/en/ (2024)
47. Mens, I.E., Maler, O.: Learning regular languages over large ordered alphabets. Logical Methods in Computer Science **11**(3), 1–22 (2015). https://doi.org/10.2168/LMCS-11(3:13)2015
48. MIDI: A c library for parsing midi messages. https://github.com/binarynate/midi-message-parser (2020)
49. Miné, A.: The octagon abstract domain. Higher-order and symbolic computation **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
50. Octave: Octave resource protocol guides. https://docs.octave.dev/docs/octave-resource-protocol-guides (2022)
51. Ohmann, P., Brooks, A., D'Antoni, L., Liblit, B.: Control-flow recovery from partial failure reports. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 390—405. PLDI '17, ACM (2017). https://doi.org/10.1145/3062341.3062368
52. PGFuzz: Policy-guided fuzzing for robotic vehicles. https://github.com/purseclab/pgfuzz (2023)
53. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems (TOPLAS) **29**(5), 26–77 (2007). https://doi.org/10.1145/1275497.1275501
54. Sheinvald, S.: Learning deterministic variable automata over infinite alphabets. In: Proceedings of the International Symposium on Formal Methods. pp. 633–650. FM '19, Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_37
55. Shimizu, T., Yoshida, N., Yamamoto, R., Takada, H.: Symbolic execution-based approach to extracting a micro state transition table. In: ACM SIGSOFT International Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things. pp. 1–6. TAV-CPS/IoT '19, ACM (2019). https://doi.org/10.1145/3341108.3342244

56. SML: Smart message language parser. https://github.com/olliiiver/sml_parser (2023)
57. Sotoudeh, M.: Automated verification of monotonic data structure traversals in c. In: Proceedings of the 37th International Conference on Computer Aided Verification. pp. 29–55. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_2
58. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering **SE-12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
59. TinyFrame: A simple library for building and parsing data frames for serial interfaces (like uart / rs232). https://github.com/mightypork/tinyframe (2021)
60. Veanes, M.: Applications of symbolic finite automata. In: Proceedings of the 18th International Conference on Implementation and Application of Automata. pp. 16–23. CIAA '13, Springer (2013). https://doi.org/10.1007/978-3-642-39274-0_3
61. Veanes, M., Bjørner, N., De Moura, L.: Symbolic automata constraint solving. In: Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 640–654. LPAR '17, Springer (2010). https://doi.org/10.1007/978-3-642-16242-8_45
62. Veanes, M., Bjørner, N., Nachmanson, L., Bereg, S.: Monadic decomposition. Journal of the ACM **64**(2), 1–28 (2017). https://doi.org/10.1145/3040488
63. Veanes, M., De Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation. pp. 498–507. ICST '10, IEEE (2010). https://doi.org/10.1109/ICST.2010.15
64. Watson, B.W.: Implementing and using finite automata toolkits. Natural Language Engineering **2**(4), 295–302 (1996). https://doi.org/10.1017/S135132499700154X
65. Xie, X., Chen, B., Liu, Y., Le, W., Li, X.: Proteus: Computing disjunctive loop summary via path dependency analysis. In: Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. pp. 61–72. FSE '16, ACM (2016). https://doi.org/10.1145/2950290.2950340
66. Xie, X., Chen, B., Zou, L., Liu, Y., Le, W., Li, X.: Automatic loop summarization via path dependency analysis. IEEE Transactions on Software Engineering **45**(6), 537–557 (2019). https://doi.org/10.1109/TSE.2017.2788018
67. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 351–363. POPL '05, ACM (2005). https://doi.org/10.1145/1040305.1040334
68. Yao, P., Shi, Q., Huang, H., Zhang, C.: Program analysis via efficient symbolic abstraction. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 118:1–118:32 (2021). https://doi.org/10.1145/3485495
69. Yaseen, N., Arzani, B., Beckett, R., Ciraci, S., Liu, V.: Aragog: Scalable runtime verification of shardable networked systems. In: Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation. pp. 701–718. OSDI '20, USENIX (2020), https://www.usenix.org/conference/osdi20/presentation/yaseen
70. Zhou, Z., Ye, Q., Delaware, B., Jagannathan, S.: A hat trick: Automatically verifying representation invariants using symbolic finite automata. Proceedings of the ACM on Programming Languages **8**(PLDI), 203:1–203:25 (2024). https://doi.org/10.1145/3656433