Discovering Blind-Trust Vulnerabilities in PLC Binaries via State Machine Recovery

Fangzhou Dong[†], Arvind S Raj[†], Efrén López-Morales[‡], Siyu Liu[†], Yan Shoshitaishvili[†],
Tiffany Bao[†], Adam Doupé[†], Muslum Ozgur Ozmen[†], Ruoyu Wang[†]
[†]Arizona State University, [‡]New Mexico State University
[†]{*bonniedong, arvindsraj, sliu274, yans, tbao, doupe, moozmen, fishw*}*@asu.edu*, [‡]*elopezm@nmsu.edu*

***Abstract*—Programmable Logic Controllers (PLCs) are industrial computers that control devices with real-world physical effects, and safety vulnerabilities in these systems can lead to catastrophic consequences. While prior research has proposed techniques to detect safety issues in PLC state machines, most approaches require access to design specifications or source code—resources often unavailable to analysts or end users.**

This paper targets a prevalent class of vulnerabilities, which we name Blind-Trust Vulnerabilities, caused by missing or incomplete safety checks on peripheral inputs. We introduce Ta'veren, a novel static analysis-based framework that identifies such vulnerabilities directly from PLC binaries without relying on firmware rehosting, which remains an open research problem in firmware analysis. Ta'veren recovers the finite state machines of the PLC binaries, enabling repeated safety analyses under various policy specifications. To abstract the state from program states to logic-related states, we leverage our insight that PLCs consistently use specific variables to represent internal states, thus allowing for aggressive state deduplication. This insight enables us to effectively deduplicate states without compromising soundness. We develop a prototype of Ta'veren and evaluate it on real-world PLC binaries. Our experiments show that Ta'veren efficiently recovers meaningful FSMs and uncovers critical safety violations with high effectiveness.

I. Introduction

Programmable Logic Controllers (PLCs) are industrial computers used in Industrial Control Systems (ICS) that control physical components, such as conveyor belts, compressors, packaging machines, and water treatment systems. With the explosive growth in building cyber-enabled industrial systems, PLCs have become increasingly prevalent: the global PLC market was valued at $16.3 billion in 2024 and is projected to grow to 1.5 times by 2033 [1].

Despite their wide adoption, PLCs have been involved in numerous safety-critical failures that have caused catastrophic incidents to society, spanning from financial loss [2], physical damage [3], destruction of critical infrastructure [4], to the loss of human life [5]–[7]. As these systems play a central role in controlling physical processes, it is imperative that PLC programs maintain safe behaviors under all input conditions.

Among all different types of inputs to PLC programs, inputs from peripherals such as sensors have been shown to trigger edge cases outside the consideration of PLC design, but such PLC vulnerabilities remain under-studied in the research community. PLC programmers usually place *blind trust* in their assumptions about the peripherals; as a result, safety checks in PLC logic may only account for "assumed inputs" and ignore edge cases that rarely occur in practice, such as out-of-range or conflicting sensor readings. One notorious series of examples is the 2019 Boeing 737 Max crashes, which were caused by a flight control system, the Maneuvering Characteristics Augmentation System, that trusted faulty angle-of-attack (AoA) sensor data—even with redundant sensors installed—and led to 346 fatalities [7]. While it may not be possible for a PLC to detect every erroneous input, it must at least behave safely when confronted with conflicting or unusual inputs.

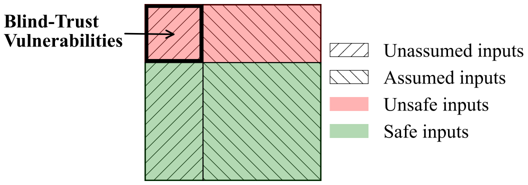


Fig. 1: The relationship between PLC programmers' assumptions on peripheral input and input safety. The intersection of unassumed input and unsafe input leads to BTVs.

In this paper, we refer to these safety problems as *Blind-Trust Vulnerabilities* (BTVs). As Figure 1 shows, PLC programs receive both safe inputs (the green area, which lead to safe behaviors) and unsafe inputs (the red area, which lead to unsafe behaviors). Existing techniques often focus on finding and eliminating unsafe behaviors that are caused by "assumed" inputs [8], [9], which leaves BTVs (unsafe behaviors caused by unassumed, and thus unhandled, input) unaddressed.

Finding BTVs is challenging: Not all unassumed inputs lead to safety violations, which means finding BTVs in PLC programs is more than merely reporting all unassumed inputs. Further, BTVs may exist in "deep" or harder-to-reach parts of PLC programs, which are difficult for fuzzing-based solutions to discover. Model checking has great potential in systematically checking all states and state transitions of a PLC program. Although prior research has proposed safety analysis techniques

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241624
www.ndss-symposium.org

using model checking on PLC programs [10]–[12], they exhibit two main weaknesses:

State explosion. Model checking-based solutions suffer from state explosion. Researchers have shown that automatically analyzing stateful programs (with source code) is challenging, and a key challenge is state explosion [13]–[15]. Analyzing PLC binaries is even more challenging because the state machines are implicit and implemented in multiple variants using arrays and loops [16].

Mandating source code access. Prior PLC safety analysis techniques generally mandate access to source code or designs of PLC programs, which are often unavailable for commercial off-the-shelf (COTS) PLCs.

In this paper, we propose a novel technique that automatically and scalably identifies BTVs in PLC binaries by first extracting the finite state machines (FSMs) from these PLC binaries and then conducting model checking on these FSMs against predefined policies to catch BTVs.

To tackle the state explosion problem, our key insight is that PLC programs always use certain variables (which we call *state variables* and will define in Section V) to represent their current states in their corresponding FSMs. Therefore, we can derive the states of an FSM and *deduplicate* these states based on the values of state variables. To this end, we build a set of empirical heuristics to automatically identify state variables in PLC binaries. Our solution alleviates the state explosion problem without relying on high-level PLC artifacts such as source code.

Based on our insight, we build Ta'veren, a research prototype that automatically and scalably finds BTVs in PLC binaries. Ta'veren takes as input a PLC binary and an environment model describing its interaction with the physical world, and automatically generates an FSM from the starting point of an FSM implementation. Further, Ta'veren formally verifies the recovered FSM against user-specified formal safety policies to find safety policy violations, and many such violations are BTVs. Users can then decide how to handle these violations, e.g., by patching the PLC binary or by using an external PLC anomaly detection solution.

Given the difficulty of obtaining real-world PLC programs [17], we build a dataset of 22 PLC binaries using real-world PLC programs collected from various sources. Our dataset represents a diverse set of PLC development software and toolchains (e.g., Beremiz, OpenPLC, and Simulink), and architectures (x86-64, ARM, MIPS, PowerPC, and AVR). Based on the source code of these PLC programs, we also create reference FSMs for each binary. We believe that this dataset, which includes source code, binary, and reference FSMs, will benefit future research in this area.

We evaluate Ta'veren on our PLC binary dataset of 22 binaries with 20 safety policies on 24 recovered FSMs. We discover a total of 17 BTVs that lead to policy violations in these binaries. We then extend the evaluation of Ta'veren to real-world and more complicated Cyber-Physical System (CPS) programs (rover and copter) and show that Ta'veren successfully recovers their FSMs and discovers three BTVs.

Contributions. This paper makes the following contributions:

- We propose Ta'veren, a novel solution that scalably finds BTVs from PLC binaries without the need for firmware rehosting or PLC simulation.
- We propose a technique that automatically recovers the finite state machines (FSMs) in PLC binaries. To the best of our knowledge, Ta'veren is the first technique that automatically transforms PLC scan cycle implementations to FSMs without using source code or pre-defined models.
- Given the absence of a real-world-scale PLC binary benchmark, we build a dataset of 22 PLC binaries, source code, and the reference FSMs for each binary, which represent a diverse set of PLC development software, toolchains, and architectures. We evaluate the effectiveness and efficiency of Ta'veren on our dataset. Our evaluation shows the high effectiveness and efficiency of Ta'veren.

In the spirit of open science, we have open-sourced all research artifacts, including the source code of Ta'veren and the datasets of our PLC programs, at https://github.com/sefcom/taveren.

II. Background

A. PLC Programs and Binaries

Programmable Logic Controllers (PLCs) are industrial computers that integrate physical components (sensors, actuators) with software components (PLC programs) to control physical processes. For instance, a PLC can control the operation of a conveyor belt by monitoring sensor inputs (e.g., the presence of items on the belt) and adjusting actuators (e.g., the speed of the belt or the position of a gate) based on its control logic.

Many PLC programs use finite state machines (FSMs) to implement the control logic. A *finite state machine* consists of a set of states, which represent sequential stages of a process or different modes of operation, and transitions between states, which are triggered by inputs through internal calculations.

PLC programmers may implement FSMs using a traditional programming language (e.g., C/C++) or PLC-specific languages, such as Ladder Logic (LD), Function Block Diagram (FBD), Structured Text (ST), Sequential Function Chart (SFC) [18], and Stateflow (in Simulink) [19]. PLC-specific languages usually require a specific toolchain to compile the program into binary code that can run on the PLC.

An intrinsic concept to PLC programs is a *scan cycle*, which refers to the code that is repeatedly executed. During a typical scan cycle, the PLC program gathers input from sensors, executes control logic based on the input and global state, updates the global state, and generates output to actuators. Note that while a scan cycle can implement an FSM, the control flow graph of a scan cycle does not reflect the shape or structure of the FSM [16]. We will explain their differences with an example in Section IV-B.

It is worth noting that there are many differences between PLC binaries and typical user-space executables. PLC binaries are often provided as a single monolithic binary and interact with peripherals through ports or memory-mapped I/O (MMIO), whereas user-space executables often load libraries and interact with their environments using library APIs or system calls. Therefore, PLC binaries contain less semantic information than APIs or system calls provide, which is usually available when analyzing user-space executables.

B. Security versus Safety

Safety and security are closely related but different concepts. Security refers to an attacker violating the confidentiality, integrity, and availability of a system, whereas safety refers to the protection of devices from accidents, such as undesired or unplanned behaviors. To breach security, attackers typically exploit software vulnerabilities (e.g., buffer overflows and command injections). Safety problems may be caused by faulty designs, incorrect implementation, or ignored extreme conditions. *Safety policies* of an ICS describe the user-defined intended behaviors with which the system must comply.

In ICSs, where the software components and physical components are tightly integrated, security issues may compromise their safety, and vice versa. In this paper, we focus on finding safety problems in ICSs that are not necessarily caused by security issues. As such, we consider a PLC program safe if it does not violate any given safety policies.

C. BTVs and PLC Anomaly Detection

A Blind-Trust Vulnerability (BTV) is an input validation vulnerability where a PLC's control logic fails to properly handle the full spectrum of possible inputs from its peripherals and leads to potentially unsafe behaviors against the system. BTVs occur because physical sensors can be manipulated by adversaries or can fail in unexpected ways, providing data that violates the PLC programmer's implicit assumptions [7]. Interestingly, this vulnerability class is often overlooked in prior work, which tends to focus on inputs from configuration parameters [9], user controls [20], or even changes to the PLC code itself [8]. This oversight stems from a common assumption that peripherals will always operate within their expected physical bounds. For example, prior work has considered that a potential error state is a false positive if it could only be triggered by a physically "impossible" condition, such as a robot's speed exceeding its maximum [8]. Such reasoning neglects the potential manipulation of peripherals and, therefore, BTVs.

Although prior work has studied input validation bugs in robotic vehicles [21], [22] and robotic operating systems [23], PLCs include unique challenges that fundamentally prevent their application. Specifically, these works require the availability of high-fidelity simulators and software-in-the-loop testing architectures to observe the physical consequences of inputs. However, such simulation environments are rarely available for the diverse, proprietary industrial processes that PLCs control.

Another line of work for ICS security has proposed anomaly detection systems, which monitor the physical process to detect deviations from an expected model of behavior [24], [25]. These systems can, in principle, detect the consequences of a BTV by identifying system states that are physically implausible. However, their effectiveness is often constrained. First, these works aim to detect the attacks at *run-time* rather than *design-time*, and therefore, they can raise an alarm after an anomalous input has already been processed and the ICS may already be transitioning to an unsafe state. Second, creating an accurate model for anomaly detection typically requires access to detailed design specifications, data traces, or even the PLC's source code, which are frequently unavailable for commercial off-the-shelf (COTS) ICS [26]. Lastly, such systems have also been shown to be vulnerable to attacks that subtly manipulate sensor inputs to remain within the bounds of the modeled behavior, thereby evading detection.

III. Research Scope and Threat Model

The intended users of Ta'veren are security analysts who cannot access the source code of a PLC binary program or are interested in vulnerabilities that only manifest in binary code (but invisible in source code). The focus of this paper is finding BTVs in PLC binaries that are loadable and disassembleable using external tools. As such, analyzing and lifting binaries with proprietary instruction set architectures (ISAs) or formats is out of the scope of this paper. Note that many PLC vendors, such as Siemens and Rockwell, use proprietary ISAs and formats for their PLC binaries. Developing loaders and lifters for these proprietary targets is an orthogonal problem.

We mainly consider PLC binaries with FSM implementations. Both benign users and attackers may interact with PLC binaries through their input channels (e.g., by impacting sensor readings so that the readings fall outside normal ranges) and trigger BTVs in the PLC program, which will lead to unsafe PLC behaviors. We consider sensor attacks where attackers must gain physical access to a sensor before manipulating it. According to prior research, generally manipulating multiple sensors is more difficult than one sensor [27]. PLC programs that do not implement FSMs are out of scope.

While attackers may tamper with peripherals to impact the input, we assume that both PLC binaries and the controller hardware are trusted and cannot be tampered with during runtime. For example, flashing the PLC with a new binary, disconnecting the PLC from power, and glitching the PLC by manipulating its power input are out of scope. Finding software vulnerabilities, such as memory corruptions, is also out of scope.

IV. Motivation

We first motivate the analysis of PLC binaries (instead of source code) (Section IV-A), then present a running example that we use throughout this paper (Section IV-B), and briefly introduce the technical challenges we face when analyzing PLC binaries (Section IV-C).

A. Analyzing PLC Binaries

While many existing PLC safety analysis techniques require source code, in reality, the source code of commercial-off-the-shelf (COTS) PLC programs is often unavailable to security

TABLE I: A comparison of Ta'veren and relevant techniques that analyze PLC programs.

Type	Tool	Analysis Input	Generated Model	Technique	Input Channel
PLC	TSV [10]	Instruction List (IL)	Temporal Execution Tree	Model Checking	Peripherals
	VetPLC [8]	Structured Text (ST)	Timed Event Sequences		Timing
	SAIN [24]	Structured Text (ST)	Program Dependency Graph		N/A
	Ta'veren	Binary	Finite State Machine		Peripherals
CPS	MISMO [28]	Binary	Control Algorithms	Pattern Matching	PID Parameters
	DISPATCH [29]	Binary	PID Controller		PID Parameters
RV	RVFUZZER [22]	RV Source code	N/A	Fuzzing	Configurations
	PGFUZZ [21]	RV Source code	N/A		User Commands, Configurations, Peripherals

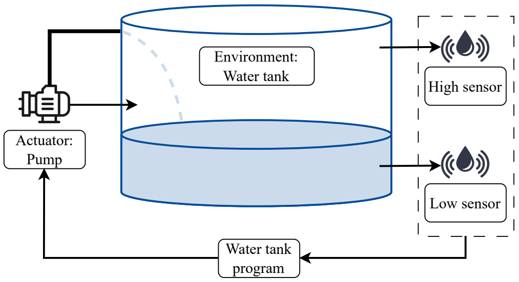


Fig. 2: An example PLC controlling a water tank system. The PLC program takes inputs from two sensors (low and high) to perceive the water level in the tank and, based on those inputs, controls the actuator pump to add water to the tank.

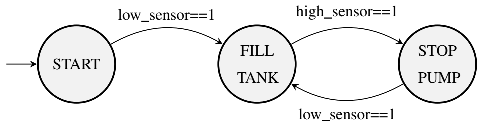


Fig. 3: Water tank system FSM with BTV. The pump turns on and off depending on the signals it received from low and high sensors.

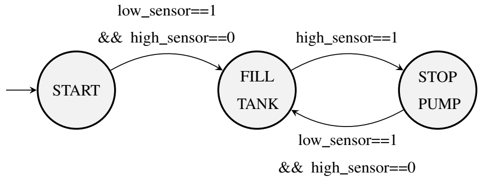


Fig. 4: Fixed water tank system FSM. It checks both low and high sensors before turning on the pump.

analysts due to copyright protection. However, the binaries of PLC programs are usually available either as downloadable firmware images (from, e.g., vendor websites) or on the actual PLC device (and can be extracted). Even when the source code of PLC programs is available, due to the *What You See Is Not What You eXecute* phenomenon [30], security analysts may still need to analyze the actual binary code that runs on PLCs to avoid unknowingly neglecting bugs that can be introduced by misimplementation [7] or toolchains [2], [5].

Table I shows the design comparison of Ta'veren against existing PLC program analysis work. Existing static techniques work on high-level source code (e.g., IL, ST, or C code) and derive execution or event sequences. They rely on parsing explicitly defined states (e.g., enum data types) and event-relevant facts (e.g., event handlers, sequences, and intervals) in the source code. Such information is PLC source-specific and does not exist in PLC binaries, so these techniques do not apply in our setting. Dynamic techniques, such as MISMO [28] and DISPATCH [29], focus on specific algorithms or models and thus are not generic to PLC programs. Fuzzing approaches [21], [22] rely on the availability of execution, which requires PLC program simulation or emulation, either of which lacks generic solutions over different architectures, plants, and environments. Therefore, none of the existing works are generic for PLC binary programs.

B. Running Example

Figure 2 illustrates a simplified water tank control system with two sensors: The high (water level) sensor and the low (water level) sensor. Upon start, the pump receives a low signal (that the low sensor emits) and starts to pump water and fill the tank. Once the pump receives a high signal (that the high sensor emits), it turns off to prevent water from overflowing. When the water level is low again, the pump turns on and repeats the process. Figure 3 shows the FSM of this system that a programmer implements in Sequential Function Chart (SFC) in OpenPLC.

From SFC to the PLC binary. When building the PLC binary, OpenPLC first converts the SFC program to C++ code and then compiles it to a binary program. Listing 1 shows a simplified C++ snippet of the function that implements the scan cycle logic. Each state and transition uses boolean flags to set and track the activation of the associated state and transition (Lines 5 and 11). In each scan cycle, the program checks the water level in the tank by reading the two sensor values (Lines 5 and 7). Based on sensor readings, it activates the transitions accordingly. Then, the PLC program activates or resets the steps according to the transitions (Lines 10–17). The status of these steps determines the state-specific control logic to execute (Lines 19–21) and updates the actuator (pump) state accordingly (Line 23).

The BTV. An attacker with physical access to the low sensor may manipulate it (or its environment) to emit a low signal, which causes the pump to keep pumping water and potentially

```
void WATER_TANK_SFC_body() {
  // Calculate elapsed_time, initialization
  ...
  // Set transitions
  if (START.X) { transition[0] = LOW_LEVEL_SENSOR; }
  else { transition[0] = 0; }
  if (FILL_TANK.X) { transition[1] = HIGH_LEVEL_SENSOR;  }
  else { transition[1] = 0; }
  ...
  // Transitions reset steps
  if (transition[0]) { START.X = 0; }
  if (transition[1]) { FILL_TANK.X = 0; }
  ...
  // Transitions set steps
  if (transition[0]) { FILL_TANK.X = 1; }
  if (transition[1]) { STOP_PUMP.X = 1; }
  ...
  // Action associations
  char active = START.X;
  char activated = active && !START.prev_state;
  char deactivated = !active && START.prev_state;
  ...
  if (deactivated) { PUMP = 0; }
  ...
} // WATER_TANK_SFC_body()
```

Listing 1: Snippet of scan cycle in OpenPLC generated C++ code for the water tank program (simplified for readability).

overflow the tank. This is because the PLC program does not check the high sensor before re-activating the water pump (i.e., the transition from STOP_PUMP to FILL_TANK is only conditioned on the low sensor). This BTV could have been fixed by adding a check for the high sensor before activating the pump, as Figure 4 shows.

Finding the BTV. Ta'veren analyzes the PLC binary, recovers a complete FSM in Figure 3[1], and then examine if there exist any states and paths that violate a set of pre-defined safety policies. This BTV violates the safety policy that states *when the water level is high, the water pump must stop* to ensure that the water does not overflow.

C. Challenges

Conducting bounded model checking to identify BTVs in PLC binaries requires addressing two major challenges:

Challenge 1. No source code access. Source code, whether written in IL, ST, SFC, or C/C++, usually contains high-level semantic information that allows direct extraction of event sequences or state machines. For example, VetPLC extracts events, event intervals, and their guarding conditions from PLC ST source code [8]. A relevant technique, StateLifter (which generates FSMs for format parsers by analyzing their C implementations) extracts guarding conditions of events by parsing the C source [16]. However, PLC binaries do not contain such high-level semantic information. Therefore, Ta'veren must recover the necessary information from analyzing the binary code.

Challenge 2. State explosion. The control flow graph (CFG, shown in Figure 13 in the appendix) of the C++ program in Listing 1 and the FSM (Figure 3) do not resemble each other. A key difference is that the FSM has only a few (abstract) states, while its C++ implementation contains logic for *all* states, which potentially represent an *infinite* number of (concrete) program states. Suppose S_0 is the state before invoking the `WATER_TANK_SFC_body` function for the first time, and S_1 is the state after invoking the function. S_0 and S_1 are different concrete program states because the data in memory and registers have changed. In fact, during the execution of the function, we may observe one unique concrete state after executing every instruction in the binary. However, when the water level is not low, and the low water level sensor is not activated, both S_0 and S_1 (as well as all other concrete states that we may observe) correspond to the same abstract state, START, in the FSM.

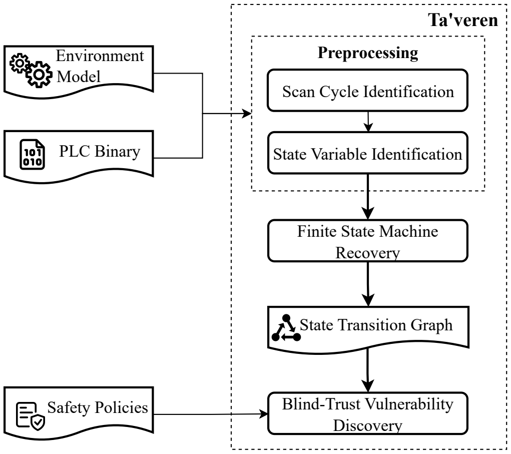

Ta'veren
Preprocessing
Environment Model
PLC Binary
Scan Cycle Identification
State Variable Identification
Finite State Machine Recovery
State Transition Graph
Safety Policies
Blind-Trust Vulnerability Discovery

Fig. 5: Overview of Ta'veren. The user provides as input a PLC binary, an environment model, and safety policies to verify.

Without high-level knowledge of the PLC program, we cannot derive the correspondence between concrete program states and abstract states in the FSM. Therefore, there are potentially an *infinite* number of concrete program states, which causes state explosion [31]. Any static or dynamic analysis technique that directly works on a CFG will encounter this problem. Existing techniques, such as TSV, aggressively limit the number of explored paths during dynamic symbolic execution, which alleviates state explosion by sacrificing completeness (i.e., missing states) and may cause false negatives during the discovery of safety policy violations.

V. Overview of Ta'veren

Figure 5 shows the overview of Ta'veren. At a high level, Ta'veren takes as input a PLC binary, specified inputs and outputs, and the safety policies to verify. Next, Ta'veren automatically identifies the location of the scan cycle and state variables (which we will define next) in the PLC binary. It then automatically generates an FSM represented in a state transition graph and uses this graph to verify each safety policy and report any discovered violations. We define three concepts: state variable, environment model, and state transition graph.

[1] The FSM that Ta'veren recovers is actually different from Figure 3 because it also recovers conditions of the high sensor. For simplicity, we will explain how Ta'veren works in the later sections.

State variables and state IDs. State variables are what a PLC program implementing an FSM uses to record and track current states. The values that state variables hold are *state IDs*, which are concrete values (e.g., integer values or boolean flags) that correspond to abstract states in the FSM of the PLC program. Both state variables and state IDs are artifacts resulting from the implementation of an FSM in the PLC program. An FSM implementation may contain multiple state variables, in which case we consider the union of these state variables for tracking abstract states. In a typical Kripke structure where each node in the graph is associated with a unique symbol, its state variables will hold the values of these symbols.

In the FSM of the water tank (Figure 3), each of the three states is associated with a unique ID (`START`, `FILL_TANK`, and `STOP_PUMP`). Each state is also associated with a set of actions (e.g., turn on the pump). In the implementation of this FSM (Listing 1), `FILL_TANK.X` (Line 12) is a state variable, and `FILL_TANK.X == 1` represents that the water tank is in the `FILL_TANK` state.

Environment model. The environment model specifies how the PLC program interacts with the physical world. An environment model comprises two components:

1) Input variables, their locations, and types. Input variables store sensor readings and are used in state transition conditions. For example, in the water tank example (Listing 1), `LOW_LEVEL_SENSOR` is an input variable.
2) Output variables, their locations, and types. Output variables are linked to actuators and control the behaviors of actuators (e.g., turning on and off pumps), which influence the physical world (e.g., pumping water into the tank and raising the water level) and may impact input variables.

As Section II-A discusses, PLC binaries usually do not contain any semantics or location information about input or output variables. Typically, security analysts get such information from reading the PLC documentation (which usually indicates the mapping between memory addresses and physical ports of a PLC), analyzing artifacts (e.g., source code) of a closely related PLC program [8], or reverse engineering the PLC binary itself with domain-specific knowledge [28], [32]. Because automatically locating input or output variables is not the focus of our research, Ta'veren takes environment models as input from users.

State transition graph. A state transition graph is a quadruple $(\mathsf{A}, \mathsf{a}_0, \Sigma, \delta)$, where

- A is a set of *abstract states* where each abstract state a_i is represented by a unique tuple of state variables and output variables that are extracted from the actual program state. Any state a_i can be represented by $[s_0, \cdots, s_n, u_0, \cdots, u_m]$, where $s_i, 0 \leq i \leq n$ are state variables and $u_j, 0 \leq j \leq m$ are output variables.
- $\mathsf{a}_0 \in \mathsf{A}$ is the initial abstract state.
- Σ are the constraints on input variables that can be evaluated to a boolean value.
- $\delta : \mathsf{A} \times \Sigma \rightarrow \mathsf{A}$ is the state transition function. The transition is taken when the condition $\sigma \in \Sigma$ is true.

State deduplication. We consider any concrete states with the same values of state variables and output variables as the same state. This allows us to aggressively deduplicate concrete states and create a finite number of abstract states.

In a program, *input variables* do not have control or data dependencies on any other variables. The updates of *state variables* and *output variables* are determined by previous state variables and input variables. We consider the remaining variables in the program as *intermediate variables*.

Theorem. A recovered FSM fully reflects all the behaviors of the original FSM (as implemented in the source program) if the set K of selected variables used to represent a state (i.e., state variables and output variables) is *complete* (i.e., no variable in K exhibits control or data dependencies on intermediate variables).

For interested readers, we provide a formal proof of the theorem in Appendix A. This theorem shows that an FSM that Ta'veren recovers via state deduplication is sound (i.e., all recovered states and transitions must exist in the FSM implemented in the source) if Ta'veren fully discovers all state variables [33].

Ta'veren's analysis has four main steps (Section VI):

Step 1. Scan cycle identification. Ta'veren identifies the locations in the binary that implement the scan cycle, as well as any pre-requisite initialization functions.

Step 2. State variable identification. Ta'veren automatically identifies all state variables in the target binary.

Step 3. FSM recovery. Ta'veren automatically recovers the state transition graph for the FSM from the binary. The core technique takes input, the scan cycle functions, state variables, and the environment model that specifies information about the input and output variables of the PLC program. It uses dynamic symbolic execution (DSE) to analyze the functions in the PLC binary and recover abstract states, state transitions, and state transition conditions in a scalable manner. This step outputs an FSM represented in a state transition graph.

Step 4. BTV discovery. Given safety policies as input, Ta'veren verifies each user-specified safety policy on the recovered FSM. For each reported violation, Ta'veren provides a counterexample trace, including the specific sequence of peripheral inputs that triggers the unsafe state. This allows identifying when a safety violation is the direct result of a BTV.

The theoretical underpinning of FSM recovery. Upon a quick glance, a PLC binary always appears to be a Turing machine program (consider that it has stack, memory, and instructions with loops, which corresponds to an infinite tape that instructs its execution). In general, because a Turing machine is more expressive than an FSM, converting a Turing machine into an FSM cannot be sound. However, when a Turing machine program strictly implements an FSM (and nothing more), we can represent the program as an FSM without loss of expressivity. The FSM recovery step in Ta'veren can abstract the FSM from a PLC binary soundly when the PLC binary (or the part of it that Ta'veren targets) strictly implements an FSM.

When the PLC binary (or a part of it that Ta'veren targets) implements logic that requires more expressivity than an FSM can provide (e.g., it requires a finite pushdown automata or a Turing machine), sound abstraction of the FSM from the PLC binary is no longer possible.

VI. Design of Ta'veren

In this section, we discuss the design of key components of Ta'veren.

A. Scan Cycle Identification

Given the PLC binary, Ta'veren first identifies the locations of the scan cycle function and initialization functions.

Based on our observations, a scan cycle function in PLC binaries exhibits the following patterns: (a) The scan cycle function is invoked inside a loop, such as `while(1)` or `while(power_on)`. (b) The scan cycle function has at least one switch-case construct or many consecutive if-conditions. This is because the scan cycle function dispatches the execution to state-specific logic depending on the current state. We identify such constructs in binary code by observing the existence of jump tables (which are usually the result of switch-cases in C) or cascading if-else branches.

Initialization functions initialize persistent data structures and state variables. We observe that initialization functions exhibit the following patterns: (a) They are reachable from the program entry point and are executed before the scan cycle function. (b) They perform multiple memory writes to non-volatile regions (e.g., global sections or heap regions). We identify initialization functions by counting the number of memory writes to global or heap regions in each candidate function and its callees and pick the functions with the highest number of writes.

B. State Variable Identification

Once Ta'veren identifies the scan cycle function, the next step is to identify the state variables. PLC programs initialize state variables before entering the scan cycle and keep state variables alive across multiple scan cycles. Control dependencies may exist between state variables. For example, in Listing 1, the state variable `FILL_TANK.X` depends on the value of the local variable `transition[0]` (Line 15), which in turn depends on another state variable `START.X` (Line 5). Performing DSE on a single scan cycle without considering the initial or prior values of state variables will ignore the control dependencies between state variables, which leads to redundant abstract states and state transitions in the recovered FSM. Therefore, Ta'veren must identify all state variables.

Identifying state variables is challenging because they are not special regarding their storage locations or their types (or sizes). Even worse, the semantics of variables, including variable names and types, do not exist in PLC binaries that Ta'veren analyzes, so we cannot determine if a variable is a state variable by inspecting its name or type. These challenges render previous methods for state variable identification inapplicable [13]–[15], [24].

We theorize that state variables in PLC programs must follow these rules:

1) State variables stay alive across scan cycles, so they are stored in non-volatile regions, e.g., global sections or the heap. State variables cannot be stored in function stack frames.
2) State variables are used in branch conditionals so that they impact the internal state by determining which branches to take and which code path to execute.
3) During each scan cycle, state variables must be read first before being written to (i.e., assigned new values), whereas local variables must be assigned before use.

These rules allow Ta'veren to automatically identify state variables using static data-flow analysis on binary code. First, Ta'veren extracts conditions that guard each branch in the scan cycle functions. Then, Ta'veren filters all variables used in conditions to identify the ones that are located in global sections or heap regions. Next, Ta'veren builds a data-dependency graph across all scan cycle functions, including all candidate variables. Lastly, Ta'veren considers all candidate variables as state variables if they are read before being updated and are not input variables.

In all PLC binaries in our dataset, we confirm that the identified state variables align with the ones that are used in the source code.

C. Finite State Machine Recovery

The FSM recovery algorithm in Ta'veren begins by initializing an abstract state and then symbolically explores the scan cycle functions until reaching a fixed point. To prevent state explosion, Ta'veren (1) aggressively deduplicates abstract states and (2) discovers new values of input variables that are meaningful to the PLC program (e.g., time and temperature).

1) Initialization: We initialize the analysis by executing identified initialization functions, such as `init`, to set up the initial program state s_0. After initialization, values for state variables in s_0 are set to their corresponding start values.

2) Scan Cycle Execution: As Algorithm 1 shows, the FSM recovery algorithm is a worklist algorithm. Each work item in the worklist has four elements (s, Δ, C, a), where s is the (concrete) program state (with registers and memory), Δ is the new input values for the next state, C is the path constraints that cause state transition and a is the abstract state. The `GenAbstractState` function maps a concrete program state s to an abstract state a by extracting values of the state variables and output variables from s. A directed graph G stores the recovered FSM.

To start, Ta'veren derives an abstract state a_0 from s_0, initializes the worklist, and adds a_0 to G. Then Ta'veren symbolically explores scan cycle functions through the following steps.

Step 1. Getting abstract states (Lines 5–14). We get the current program state s, new values for input variables Δ, path constraints C, and the previous abstract state $prev\_a$ from the worklist. We symbolically execute s through the scan cycle functions to obtain the end program state s_1. Then, we extract

Algorithm 1 Finite state machine recovery

```
Input: Initial program state $s_0$
Input: Scan cycle functions
Input: State variables
Input: Inputs and outputs
Output: State transition graph
 1: Abstract state $a_0 \leftarrow$ GenAbstractState($s_0$)
 2: State queue $Q \leftarrow \{(s_0, \varnothing, \varnothing, a_0)\}$
 3: State transition graph $G \leftarrow$ MultiDiGraph($a_0$)
 4: while $Q \neq \emptyset$ do
 5:     $(s, \Delta, C, prev\_a) \leftarrow$ pop($Q$)
 6:     # First step, concolic
 7:     $s_1 \leftarrow$ ScanCycleExecution($s, input = \Delta$)
 8:     $a \leftarrow$ GenAbstractState($s_1$)
 9:     if $edge(prev\_a, a, cond = C) \notin G$ then
10:         $G$.add_node($a$)
11:         $G$.add_edge($prev\_a, a, cond = C$)
12:     else
13:         continue
14:     end if
15:     # Second step, symbolic
16:     $s_2 \leftarrow$ ScanCycleExecution($s_1, input = SymVar$)
17:     for each $s_{2.i} \in s_2$ do
18:         analyze path constraints on transitional program states
19:         converge path constraints $C_{new}$
20:         generate new inputs $\Delta_{new}$
21:         $Q \leftarrow Q \cup (s_1, \Delta_{new}, C_{new}, a)$
22:     end for
23: end while
```

values for state and output variables from s_1 to derive the corresponding abstract state a and add a transition edge from $prev\_a$ to a in G.

Step 2. Discovering new values for input variables (Lines 17–22). Updating the values of input variables by fixed deltas (e.g., increasing `time` by 0.01 seconds each time) leads to redundant states, because the PLC program only transitions to a new abstract state when conditions on input variables are satisfied (e.g., checking if at least 0.5 seconds have passed since the last execution).

Ta'veren discovers *meaningful* values for each input variable that will cause transitions to the next abstract states. It sets each input variable to a symbolic value and performs DSE on scan cycle functions using s_1 as the input state to discover conditions that each input variable must satisfy to cause state transitions.

We observe two types of conditions: (a) comparisons between concrete values and arithmetic expressions that contain input variables (e.g., `if (sensor > 4500)`); (b) the conjunction of comparison expressions, e.g., `if (sensor < 4500 && sensor > -9000) {state_id = 5;}`, where `sensor` is an input variable[2].

Ta'veren handles the first type of conditions by normalizing and parsing them. Handling the second type is more complex, and we detail our prior approach using an example. After DSE, the final set of program states contains three states: $s_{2.0}$, $s_{2.1}$, and $s_{2.2}$, each with its own path constraints as shown in Table II. The abstract states of $s_{2.0}$ and $s_{2.1}$ are both `state_id = 4`, which is the same as the input state s_1. This indicates that $s_{2.0}$, $s_{2.1}$, and s_1 are the same state, and state transitions did not occur. In $s_{2.2}$ we find two path constraints `sensor < 4500` and `sensor > -9000`, and the value of `state_id` is updated to `5`, which indicates that a state transition from s_1 to $s_{2.2}$ has happened. We then invoke an SMT solver on the path constraints to generate a concrete value Δ_1 for the input variable `sensor` that satisfies the path constraints. Next, we create a new worklist item containing the previous program state s_1, the new value for the input variable Δ_1, the corresponding path constraints, and the abstract program state a_1, and we push this new item to the worklist.

TABLE II: The resulting states, their state IDs, and their path conditions after DSE of the example (b). "T?" indicates if a state transition occurs. State s_1 is the input state whereas States $s_{2.0}$, $s_{2.1}$ and $s_{2.2}$ are output states.

State	State ID	Path Constraints	T?
s_1	4	N/A	Original
$s_{2.0}$	4	`sensor >= 4500`	No
$s_{2.1}$	4	`sensor < 4500 && sensor <= -9000`	No
$s_{2.2}$	5	`sensor < 4500 && sensor > -9000`	Yes

Step 3. Deduplicating abstract states. After getting a new abstract state a_2, we check if a_2 and an edge from a_1 to a_2 with constraint C_1 exist in G. If not, the abstract state a_2 and the edge from a_1 to a_2 with constraint C_1 are added to G, indicating that when condition C_1 is satisfied, a state transition from a_1 to a_2 will occur.

Step 4. Reaching a fix point. The analysis stops when the worklist is empty, and we get a complete FSM from G. According to the proof in Appendix A, G preserves the behavior of the original FSM.

D. Blind-Trust Vulnerability Discovery

In PLC programs, a set of safety policies serves as a bug oracle to determine if a program's behavior is a failure. Such failure is called a safety policy violation.

Ta'veren uses model checking to find BTVs of PLC programs that result in safety policy violations. It takes as input a set of safety policies derived from expert domain knowledge or system specifications, as well as the FSM that Ta'veren previously recovered. During model checking, Ta'veren traverses the FSM and verifies if each state and transition complies with the safety policy. Ta'veren repeats this process until the verification is completed or a violation is found.

For each safety policy violation, Ta'veren outputs the state in the FSM where the violation occurs, a path in the FSM from the entry node to the violated state, an input sequence that triggers a BTV resulting in the violation, as well as the exact address in the binary code for the violating state transition condition.

Table III shows a list of safety policies that Ta'veren currently implements for verification. Ta'veren translates each safety policy into boolean expressions in Linear Temporal Logic. Users may implement more safety policies as needed. For

[2] The exact conditions in ArduCopter program can be found here: https://github.com/ArduPilot/ardupilot/blob/Copter-4.5/ArduCopter/mode_flip.cpp#L144

TABLE III: Safety policies that we created for the PLC binaries described in Table IV.

Policy ID	Safety Policy Description	Consequences of Violation	Policy Type
P.Lift.1	Conveyor motor and lift motor cannot be activated at the same time.	Packages fall to the warehouse floor, damaging the equipment and potentially injuring warehouse workers.	IllegalNode
P.Lift.2	When the height reaches the assigned pallet rack, lift up motor should be off.	Too much height will cause damage to the lifter and package to fall.	IllegalTransition
P.WT.1	When water level is high, pump must be off.	Water in the tank will overflow.	IllegalTransition
P.WT.2	The low level and high level water sensors should never be true at the same time.	Conflicting sensor readings will create inconsistent control.	IllegalEdge
P.Pack.1	The conveyor belt and product valve should never be active at the same time.	It will result in product spillage.	IllegalNode
P.Pack.2	The product valve should not be active for more than 15 seconds a time.	It may cause the product overflow from the packaging box.	MaxDelay
P.CarW.1	No two sprinklers should be active at the same time.	It increases the operation costs.	IllegalNode
P.CarW.2	The conveyor motor should not be active for more than 10 seconds a time.	It may lead to vehicle collision on the conveyor belt.	MaxDelay
P.TL.1	The yellow light should be on for at least 2 seconds.	Not enough time for cars to exit intersection, could cause a collision.	MinDelay
P.TL.2	The pedestrian green light should be on for at least 40 seconds.	Not enough time for pedestrians to exit intersection.	MinDelay
P.TL.3	The pedestrian green light and car green light should not be on at the same time.	The cars can hit the pedestrians.	IllegalNode
P.Abort.1	When there is no anomaly, the rocket should not release anything.	Releasing them during normal launch will cause the rocket to break the launch sequence and fail the mission.	IllegalTransition
P.Abort.2	If anomaly occurs in early stage, the rocket should dump fuel, release rocket booster then external tank in order.	Releasing them in the incorrect order will cause imbalance and uncontrollability.	IllegalNode
P.Oven.1	The cook time of the oven should not be over 9000 ms.	Over-cooking will damage the board and create a fire risk.	MaxDelay
P.Oven.2	The oven should be turned off after cooking time reaches 9000 ms.	Over-heating will damage the oven and create a fire risk.	IllegalTransition
P.Oven.3	The oven should be turned off after temperature reaches 170 degrees.	Over-heating will damage board and create a fire risk.	IllegalTransition
P.Vend.1	When inserted amount is more than the price, the vending machine has to give change.	Customers will lose money.	IllegalTransition
P.Elev.1	When the elevator is at the top floor, it cannot go up.	Elevator may hit the ceiling and cause damage.	IllegalTransition
P.Elev.2	When the elevator is at the bottom floor, it cannot go down.	Elevator may hit the pit and cause damage.	IllegalTransition
P.Elev.3	If the button on high floor is pressed when the elevator is in the lower floors, it must go up.	It may cause long waiting time and confusion for passengers.	IllegalTransition

more complex safety policies, users may export the FSM that Ta'veren generates into a specialized model checking tool, such as Spin [34] or NuSMV [35]. We leave it as future work.

The following is the list of the categories of safety policies that Ta'veren currently supports:

IllegalNode Policies check if an abstract state has abnormal behaviors. An example policy for a traffic light is that *the red light and the green light cannot be on simultaneously*. In this case, we use the IllegalNode policy to define that `node.red==1 and node.green==1` is an illegal node.

IllegalTransition Policies check if a transition from one abstract state to another is illegal. An example policy for the oven program is that *when the oven finishes cooking for a certain time (9 seconds), the oven should turn off*. In this case, we use the IllegalTransition policy to define that when `node.time==9000`, `node.successor.status==ON` is an illegal transition.

MinDelay and MaxDelay Policies check if a time delay between two abstract states is too short or too long.

VII. Evaluation

We first evaluate the effectiveness (Section VII-B) and efficiency (Section VII-C) of Ta'veren for BTV discovery in PLC binaries. We then focus on the core step of Ta'veren and measure the correctness of the recovered FSMs from PLC binaries (Section VII-D) and identification of scan cycle functions and initialization functions (Section VII-E). Next, we evaluate the impact of inaccurate environment models on Ta'veren's FSM recovery and BTV discovery (Section VII-F). Lastly, we report our attempts to apply Ta'veren to more complex targets beyond PLCs (Section VII-G).

We cannot compare Ta'veren against existing solutions (TSV [10] and VetPLC [8]) because they do not generate FSMs or work on binaries. They also rely on artifacts like pre-defined events and enumeration of states for model generation, which do not exist in PLC binaries. Instead, we show the efficiency and measure the correctness of Ta'veren on our dataset.

TABLE IV: PLC binaries in our dataset. "Real example" indicates that the binary is built using example programs that toolchains or development environments provide. "Synthesized" indicates that the original program is created based on real-world PLC logic.

	Category	Source	Target ID	Software	Arch
DS_G	Warehouse Lifter	Synthesized	Lift.1	OpenPLC	x86-64
	Water Tank	Synthesized	WT.1	OpenPLC	x86-64
			WT.2	OpenPLC	x86-64
			WT.3	OpenPLC	x86-64
	Packaging	Synthesized	Pack.1	OpenPLC	x86-64
			Pack.2	OpenPLC	MIPS
			Pack.3	OpenPLC	PPC32
	Car Wash	Synthesized	CarW.1	OpenPLC	x86-64
			CarW.2	OpenPLC	ARM
	Traffic Light	Real example	TL.4	Beremiz	x86-64
			TL.5	Beremiz	x86-64
			TL.6	Beremiz	ARM
			TL.7	Beremiz	x86-64
			TL.8	Beremiz	x86-64
			TL.9	Beremiz	x86-64
			TL.10	Beremiz	ARM
		Synthesized	TL.11	Simulink	ARM
	Launch Abort System	Real example	Abort.1	Simulink	ARM
			Abort.2	Simulink	ARM
			Abort.3	Simulink	ARM
DS_T	Oven	Real example	Oven.1	Arduino	ARM
	Vending Machine	Real example	Vend.1	Arduino	ARM
	Elevator	Real example	Elev.1	Arduino	ARM
			Elev.2	Arduino	AVR8

Implementation and the evaluation environment. We implement Ta'veren atop the binary analysis framework angr [36]. We lift binary code to angr Intermediate Language (AIL) for static analysis during scan cycle and state variable identification and perform all other analyses on the VEX IR. We conducted all experiments on an Ubuntu 22.04 LTS workstation with Intel i7-10750H CPU and 32 GB of RAM under Python 3.12.12.

A. Dataset

A major challenge in evaluating Ta'veren is the lack of *publicly available* datasets containing PLC programs with nontrivial FSMs due to proprietary constraints. Existing datasets rarely include PLC binaries that implement meaningful stateful logic, and our observation is echoed by prior work [17]. To address this gap, we construct our own dataset as presented in the following paragraphs.

Building the dataset. We build two datasets, DS_G and DS_T, of PLC binaries that represent a diverse range of real-world PLC programs. The first dataset, DS_G (Dataset-Graphical), contains binaries that we built using PLC programs written in three graphical languages (SFC, FBD, and Stateflow) that inherently support FSM programming. This dataset encompasses six categories of PLC programs, including water tanks, warehouse conveyor lifter, and Launch Abort System [37]. We generate a total of 18 binaries with three popular PLC programming software: OpenPLC, Beremiz, and Simulink, and four architectures: x86-64, ARM, MIPS, and PowerPC. For these binaries, the source code serves as the ground truth FSMs.

The second dataset, DS_T (Dataset-C/C++-Variants), contains PLC binaries that we built using collected PLC programs that are in traditional, imperative programming languages (variants of C and C++). By analyzing the code structure and comments, we believe these programs are written by (human) engineers with FSMs in mind instead of being automatically generated from another graphical PLC language. DS_T encompasses three categories, including oven, vending machine, and elevator, and two architectures: ARM and AVR8. Because there is no coding convention for creating FSMs in C or C++, we do not have ground truth FSMs for these binaries in DS_T. Therefore, we had to manually create reference FSMs for these programs by analyzing their source code.

Table IV shows the PLC program categories, the toolchain we use to build each binary, and the architecture of each binary. As part of the dataset, we also study each PLC program's code descriptions to manually create safety policies to verify the corresponding binaries. We provide a brief description of each program category in Appendix B.

B. *Effectiveness of BTV Discovery*

Table V shows the safety policy violations Ta'veren found in the PLC programs. T(rue) indicates that the target program (e.g., Lift.1) complies with the safety policy (e.g., P.Lift.1), and F(alse) indicates that Ta'veren finds a violation in target program that is caused by a BTV (e.g., P.Lift.2). We then manually validate Ta'veren's safety verification results. To confirm the violations, we run the source code with the inputs that Ta'veren identifies and check whether the policy is violated. For instance, for OpenPLC-based programs, we leverage OpenPLC's simulator. To confirm the cases where policies hold, we check whether the policy holds on the reference graph as well.

The result shows that Ta'veren finds a total of 23 violations in the dataset. We investigate the root cause of each violation and discover that 17 of them are caused by BTVs.

TABLE V: Safety policy verification results. T(rue) indicates that the target program (e.g. Lift.1) complies with the safety policy (e.g. P.Lift.1), and F(alse) indicates Ta'veren finds a violation in the target program that is caused by a BTV (e.g. P.Lift.2).

#	**P.Lift.1**	**P.Lift.2**		**Cause***
Lift.1	T	F		C.1
#	**P.WT.1**	**P.WT.2**		
WT.1	F	F		C.3
WT.2	T	T		-
WT.3	F	F		C.3
#	**P.Pack.1**	**P.Pack.2**		
Pack.1,2,3	T	F		C.2
#	**P.CarW.1**	**P.CarW.2**		
CarW.1,2	T	F		C.2
#	**P.TL.1**	**P.TL.2**	**P.TL.3**	
TL.4	T	F	T	C.2
TL.5,6	T	F	T	C.2
TL.7,10	T	F	F	C.5
TL.8	F	F	T	C.2
TL.9	T	T	T	-
TL.11	T	T	T	-
#	**P.Abort.1**	**P.Abort.2**		
Abort.3	F	T		C.4
#	**P.Oven.1**	**P.Oven.2**	**P.Oven.3**	
Oven.1	T	F	F	C.5
#	**P.Vend.1**			
Vend.1	F			C.3
#	**P.Elev.1**	**P.Elev.2**	**P.Elev.3**	
Elev.1,2	T	T	T	-

* C.1: Incomplete Range Handling, C.2: Incorrect Input Check, C.3: Unhandled Input Combination, C.4: Unchecked Input Acceptance, C.5: Wrong Action.

Incomplete range handling (C.1). Ta'veren finds a BTV in the warehouse lifter program (Lift.1). In a state where the lifter is moving up, the lifter does not stop going up if the input (*current_rack*) is larger than the *assigned_pallet_rack*, which is a violation of P.Lift.2.

```
_TMP_EQ12_OUT := EQ(assigned_pallet_rack, current_rack) ;
TRANSITION FROM Go_Up TO Deliver_Box
    := _TMP_EQ12_OUT;
END_TRANSITION
```

Listing 2: Snippet of incomplete input (*current_rack*) check in Lift.1 causes its violation of P.Lift.2.

Listing 2 shows that Lift.1 compares if the input *current_rack* is *equal* to *assigned_pallet_rack* to determine the lifter's current position. It does not handle the case where the input (*current_rack*) is larger than *assigned_pallet_rack*. Ta'veren finds that if the program receives any input larger than *assigned_pallet_rack*, it will return to the *Go_up* state. To test this BTV, we used OpenPLC's simulator. After assigning *current_rack* a large input (e.g., 7), the lifter becomes stuck indefinitely in the *Go_up* state, which may cause the lifter to collide with its physical limits, risking equipment failure and endangering users.

Incorrect input check (C.2). Ta'veren finds a BTV in the packaging program (Pack.1): in a state where it starts packaging and the product valve is turned on, the valve fails to turn off in time after the duration specified in P.Pack.2. This violation of P.Pack.2 is caused by an incorrect check of the *packaging* state duration in the program. We tested this BTV in OpenPLC's simulator. We enter the input sequence generated by Ta'veren to set it to the *packaging* state; then, after 15 seconds of duration, the program fails to transition to the next state, which turns off the product valve. This BTV would cause the product valve to remain on, which may create an uncontrolled product spill.

The car wash programs that violate P.CarW.2 and the traffic light programs that violate P.TL.1 and P.TL.2 are also caused by incorrect input checks.

Unhandled input combination (C.3). Ta'veren finds a BTV in the vending machine program (Vend.1) that causes violation of P.Vend.1: When a user, who is getting a 75-cent can, inserts a quarter, then a dollar, the vending machine fails to drop the can and give change back to the user. The cause of this BTV is that when a quarter is inserted, the program only checks if two more quarters are inserted, and does not handle the case where a dollar is inserted after a quarter.

The policy violations in water tank programs (WT.1 and WT.3) used in the running example (Section IV-B) are also caused by unhandled input combinations.

Unchecked input acceptance (C.4). Ta'veren finds a BTV in the launch abort system (Abort.3) that causes violation of P.Abort.1. Abort.3 is implemented with embedded FSMs, where the control logic of one state (*Abort*) is another FSM (*AbortLogic*). The program did not implement constraints on what *anomaly* signals it could accept and blindly accepted *anomaly* signals at any time, even after the *AbortLogic* is triggered. Consequently, *AbortLogic* actions keep executing, even if the outer FSM is no longer in the *Abort* state. In particular, the program fails to properly coordinate state changes between these two FSMs, which leads to a safety violation.

Wrong action (C.5). Ta'veren finds states in TL.7 and TL.10 that violate P.TL.3. This violation is caused by the wrong action in the pedestrian green state, where the program incorrectly turns on the pedestrian red light instead of the pedestrian green light. This wrong action also causes the violation of P.TL.2 in later states. The two violations that Ta'veren finds in Oven.1 are caused by misimplementation in COOL state. Ta'veren can find such wrong actions even though we do not consider such misimplementations as BTVs.

C. Efficiency of BTV Discovery

We run Ta'veren on all binaries in DS_G and DS_T to recover FSMs and find BTVs. Ta'veren works on all 22 test binaries without errors, and analysis completes within minutes on all but CarW.1 and CarW.2. FSM recovery dominates the overall analysis time. Two factors determine the speed of FSM recovery: the time that each iteration needs, which is roughly linear in the size of the scan cycle function, and the number of iterations that the analysis requires before convergence, which is roughly linear in the size of the FSM (Figure 6).

Once the FSM is recovered, verifying it against safety policies to find BTVs costs less than 0.5 seconds. Here, we note that FSM recovery is a one-time cost, i.e., the recovered FSM can be used to verify multiple safety policies. We present the detailed results of the efficiency evaluation in Table VI.

D. Evaluation of Finite State Machine Recovery

Because there are minor differences in the definitions of states between Ta'veren (abstract states) and the source programs (steps of actions), we make transformations on the recovered FSMs before evaluating their correctness, which includes discarding start state and self-loops, combining multiple transitions between two states into one transition, and collapsing semantically equivalent nodes.

TABLE VI: CFGs of the scan cycle functions that Ta'veren analyzed for FSM recovery (the numbers of nodes and edges in CFG approximately represent the complexity of the control logic) and the time spent (seconds) during BTV discovery.

Target ID	Target Description	CFG		Time Taken
		Nodes	Edges	
Lift.1	-	255	471	280.81
WT.1	two sensors - FBD	24	34	4.69
WT.2	one sensor - SFC	142	257	20.99
WT.3	two sensors - SFC	142	257	26.69
Pack.1	-	276	512	41.81
Pack.2	-	279	517	40.69
Pack.3	-	279	518	42.35
CarW.1	-	2,137	4,025	35,163.25
CarW.2	-	2,152	4,037	36,416.51
TL.4	add sensor	522	961	43.22
TL.5	short ped	500	929	32.26
TL.6	short ped	499	915	32.06
TL.7	both green	494	911	45.65
TL.8	short yellow	494	911	32.65
TL.9	Original	494	911	35.01
TL.10	both green	499	915	44.88
TL.11	-	24	30	2.29
Abort.1	ModeLogic	64	95	1.66
Abort.2	AbortLogic	64	95	2.01
Abort.3	whole system	64	95	140.41
Oven.1	-	74	101	24.76
Vend.1	-	26	37	4.13
Elev.1	-	61	100	62.3
Elev.2	-	63	102	169.28

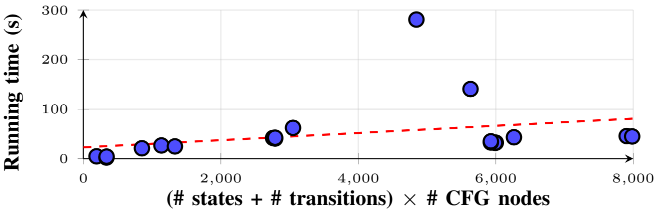


Fig. 6: The time for BTV discovery is roughly linear in the size of FSM (# of states + # of transitions) times the size of the scan cycle function (# of nodes in the CFG).

1) Correctness on DS_G: For DS_G, we have ground-truth FSMs from the source code. We evaluate the correctness of all recovered FSMs from five aspects: Topology, states, output values, transitions, and conditions by comparing the recovered graphs against the ground-truth FSMs.

Ta'veren-recovered FSMs are *sound* (i.e., all recovered states exist in the original PLC binary) (Appendix A shows the proof). The recovered FSM is complete only when (a) the scan cycle functions in the PLC binary do not exhibit any non-well-formed behaviors (e.g., memory corruptions) that can only trigger under certain conditions, and (b) all conditions involving input and state variables are correctly parsed.

Topology. We measure the similarity of topology between

TABLE VII: Comparing recovered FSMs against reference FSMs. "State & Outputs" refers to nodes and node attributes in both graphs, while "Transition & Condition" refers to edges and edge attributes in both graphs. The columns labeled "ref", "rec", and "match" refer to the number of nodes or edges in the reference graph, recovered graph, and the number of matched nodes or edges between the two graphs, respectively. GED = 0.0 means the two graphs are fully matched.

	Topology	**State & Outputs**			**Transition & Condition**		
Target	**GED**	**ref**	**rec**	**match**	**ref**	**rec**	**match**
Lift.1	4.0	5	5	5	5	9	5
WT.1,3	2.0	3	3	3	3	5	3
WT.2	0.0	3	3	3	3	3	3
Pack.1,2,3	0.0	5	5	5	5	5	5
CarW.1,2	0.0	37	37	37	39	39	39
TL.4,5, TL.6,8,9	0.0	6	6	6	6	6	6
TL.7,10	6.0	6	8	6	6	8	5
TL.11	2.0	6	7	6	6	7	6
Abort.1	0.0	5	5	5	4	4	4
Abort.2	0.0	5	5	5	4	4	4
Abort.3	57.0	16	35	16	15	53	15
Oven.1	2.0	9	8	8	11	10	10
Vend.1	2.0	5	5	5	6	8	6
Elev.1,2	0.0	10	10	10	40	40	40

the reference graph and the generated FSM using graph edit distance (GED). We do not consider properties on nodes and edges.

States and state transitions. Each node in an FSM represents an abstract state. The node in the recovered graph is correct if all of its state and output variables match the ones in the corresponding node in the reference graph.

Each edge in the recovered FSM corresponds to a state transition and the corresponding conditions. The edge is correct if its source node, destination node, and condition match the corresponding ones in the reference graph.

Table VII shows the result of the correctness evaluation of recovered FSMs. Out of the 20 FSMs that Ta'veren recovers in DS_G, 13 of them completely match the original PLC code. We investigate the remaining seven targets and report our findings in Appendix D.

2) Correctness on DS_T: Ta'veren can recover FSMs without relying on specific patterns of states and transitions that code generation tools may generate, and we apply Ta'veren to DS_T to demonstrate this point. Our results show that Ta'veren successfully recovers FSMs from binaries in DS_T. We manually build the reference graphs in the best effort and measure the correctness of Ta'veren-generated FSMs in the same way as evaluating on DS_G: Comparing against the reference graphs. Table VII shows the result of the correctness evaluation of recovered FSMs. Out of the four targets in DS_T, two of the Ta'veren-recovered FSMs completely matches the state machines implemented in the original PLC code.

E. Scan Cycle Identification Results

Ta'veren successfully identified scan cycle functions for 19 out of 22 binaries. Ta'veren fails to identify the scan cycle function in TL.11 and Abort.3 because their scan cycle functions are invoked using event callbacks, and no direct control flow exists between the loop and the scan cycle function. Additionally, Ta'veren successfully identified initialization functions for 21 out of 22 binaries with Elev.2 being the only failure. Both scan cycle and initialization function identifications fail on Elev.2 because of limitations in angr for AVR8 support.

F. Impact of Inaccurate Environment Models

TABLE VIII: Impact of inaccurate environment models on FSM recovery and the correctness of safety policy verification (Pol.V?).

Scenario	**FSM Impact**	**Pol.V?** **
Extra input variable	Extra incorrect transitions	Yes
Extra output variable	Correct	No
Missing input variable	Missing states and transitions	Yes
Missing output variable	States lack the missing variable values	Yes
Incorrect input size: 2 bytes	Correct	No
Incorrect input size: 4 bytes	Redundant condition values	Yes
Incorrect output size: 2 bytes	Correct	No
Incorrect output size: 4 bytes	Redundant extra states & output values	Yes
Incorrect input type: float*	Unparsable transition constraints	Yes
Incorrect output type: float*	Redundant extra states & output values	Yes

* Changing variable type to float changes both size and type.

** *Yes* indicates this scenario *can potentially* produce incorrect verification results. If the specified policies do not involve the incorrect states and transitions, verification results may still be correct.

We analyze and measure the impact of inaccurate environment models under three main scenarios of incorrectness: an extra variable, a missing variable, and incorrect variable configurations (size or type) on both input and output variables (where the desired variable is one-byte long). Table VIII lists the impact of each scenario. We verify each scenario on Lift.1 using at least one environment model.

Ta'veren is robust against output variable inaccuracies. Missing or having extra output variables in environment models do not impact the correctness of FSM recovery. However, missing output variables may impact verification because policies involving the missing ones can no longer exist. Extra output variables do not affect verification.

Ta'veren is sensitive to input variable inaccuracies. An extra input variable (e.g., incorrectly marking an intermediate variable as input) causes extra transitions in recovered FSM. A missing input variable causes incomplete FSMs. Both cases impact the reliability of safety policy verification.

Ta'veren tolerates variable configuration errors, but not out-of-bound accesses. The input and output variables in Lift.1 are one-byte. When we mark variable sizes as four bytes, accesses to these variables cause out-of-bound (OOB) accesses into adjacent variables, leading to incorrect FSMs. No OOB access occurs when the variable sizes are marked as two bytes, and the FSM and safety policy verification results remain correct.

G. Evaluating Ta'veren on Robotic Vehicle Binaries

We evaluate Ta'veren's capability of BTV discovery on more complex binaries for Robotic Vehicle control software, a total of three control software for copters and rovers: We build an x64 copter binary using ArduPilot [38] and extract two binaries from the controllers of another copter (also based on ArduPilot, in ARM32) and a rover (custom control software in ARM32), which are both real devices. Our goal is to explore the applicability of Ta'veren beyond PLC binaries, which are simpler than Robotic Vehicle binaries in comparison.

Copter programs contain many operating modes. Ta'veren recovers an FSM for ModeFlip [39] from these binaries. Because there are many FSMs in each binary, and Ta'veren only recovers one FSM during its analysis, we specify the function of ModeFlip as the scan cycle function for Ta'veren. The input variable for ModeFlip is `roll_sensor`, which returns roll angle in centidegrees. In our experiments, Ta'veren successfully recovers FSMs for ModeFlip for copter binaries, which demonstrates the applicability of our approach.

We create safety policies by examining the ArduPilot documentation and referring to FLIP-related policies used in PGFUZZ [21]. Specifically, the policy states that the copter should have sufficient altitude before entering the mode flip to avoid crashing to the ground [39]. Ta'veren checks this policy against the generated FSMs and finds a violation in both copter binaries. We then verify this violation in ArduPilot's SITL (software in the loop) simulation: We position the copter at a low altitude (three meters), trigger the flip mode, and observe that the copter crashes into the ground during the flip action. After examining the program, we have realized the root cause is a missing handling (C.6) of altitude, allowing the copter to flip at any altitude.

Rover program receives user commands (*turn_left* or *turn_right*), and follows a predefined mission. The rover program heavily relies on state variables in different threads (e.g., the main thread and motor thread) to perform this mission. Ta'veren successfully recovers its FSM, with some customization to overcome multithreading. Ta'veren checks the rover's specified mission against the recovered FSM, and finds a violation caused by blindly accepting user commands. After the mission starts, the rover still accepts user commands as valid input (C.4), causing the rover to keep restarting its mission; thus, significantly deviating from its planned path.

VIII. Discussion

In this section, we discuss the limitations and threats to the validity of our work.

Manual environment modeling. Ta'veren uses user-provided I/O information for environment modeling. While our experiments show that Ta'veren can handle some inaccuracies in environment models, guaranteeing the correctness of environment models is still tedious and error-prone. Future work may automatically recover environment models.

Support for binaries from proprietary software. Ta'veren uses angr to lift binary code for static analysis and symbolic execution. While we support common architectures such as x86-64 and ARM, many PLCs rely on proprietary software to develop control software. For example, Siemens uses Siemens TIA Portal to program Siemens S7-series PLCs. These tools generate binaries with customized and proprietary architectures and formats. For Ta'veren to analyze these binaries, it needs a dedicated lifter. However, this is not our focus. We leave integrating other lifters into Ta'veren as future work.

Representativeness of the PLC program dataset. While we attempt to cover as many variants of FSM implementations in PLC development software, toolchains, and human-written code as possible, there may be more types of state machine implementations that we missed. However, our best-effort dataset covers multiple toolchains and PLC programs controlling a diverse set of physical processes with different complexities, demonstrating the potential generalizability of Ta'veren.

Support of complex or non-linear state transition conditions. We currently present state transition conditions as constraints on input variables, and rely on an SMT solver to generate these constraints to provide new input values. If there are unsolvable constraints and non-linear functions, they need to be manually modeled. However, empirically, we observe that such unsolvable and non-linear constraints are rare in PLC programs.

IX. Related Work

Safety validation and verification on PLC programs. Prior work has analyzed the problem of safety policy verification on PLC programs. Prior work [40] uses model checking techniques to detect violations of safety policies or specifications for PLC programs. Further research has focused on extracting and recovering improved models for PLC programs and then applying model checkers to verify safety policies using static analysis. TSV [10] uses symbolic execution on scan cycle logic and generates a Temporal Execution Tree until it reaches a predefined bound or termination time. VETPLC [8] performs static program analysis on PLC Structured Text code to construct timed event sequences to represent the temporal dependencies of cross-device events. SAIN [24] extracts a model that combines the control and data flow graphs of PLC programs. Orpheus [41] detects data-oriented attacks by constructing an event-aware finite-state automaton (eFSA) from dependencies of system calls. Choi et al. [42] manually derive a hybrid model for robotic vehicles from their operation specifications. It performs program (dependency) analysis on countermeasure functions/statements to extract program constraints.

Besides static approaches, dynamic techniques are used to find safety policy violations of PLC programs. C^2 [43] instruments PLC code to enforce safety policies at runtime. PGFUZZ [21] uses fuzzing to mutate input space around policies to find the input sequences resulting in a policy violation. However, these works assume the availability of PLC source code or detailed documentation/design. To the best of our knowledge, Ta'veren is the first solution that directly recovers FSMs from PLC binaries for safety policy verification.

Analyzing stateful programs. State machines (or similar constructs) are widely used in programs. Researchers have analyzed non-PLC stateful programs [11], [44], [45], such as network protocols, protocol parsers, and device drivers.

Pacheco et al. [46] use natural language processing (NLP) to extract finite state machines from the documentation of protocol specifications. SEmu [47] uses NLP to model microcontroller firmware from state-diagram peripheral specifications.

StateLifter [16] extracts the formats of regular protocols as state machines by analyzing the parsing loop. StateLifter differentiates between states using path sets. However, PLC programs heavily use state transition tables and action lists, so a path set may correspond to multiple abstract states, which StateLifter cannot differentiate. StateInspector [48] presents a grey-box protocol state machine learning method using dynamic analysis. It requires access to the runtime memory of the system under test, which Ta'veren does not need.

Security analysis of PLC programs. Compared to traditional software, analyzing and understanding PLC binaries is more challenging. Prior work has focused on reverse engineering and analyzing different types, such as ICSREF [32], which aims to reverse engineer low-level information in PLC binaries, such as subroutines and symbol tables, to reconstruct the CFG. ICSFuzz [49] and ICSQuartz [50] aim to fuzz PLC applications for memory corruption vulnerabilities. MISMO [28] and DISPATCH [29] identify control algorithms such as PID controllers in firmware. Ta'veren takes one step further by abstracting high-level logic—state machine semantics—from PLC binaries.

X. Conclusion

We introduce Ta'veren, which statically extracts finite state machines from PLC binaries to discover blind-trust vulnerabilities in PLCs that cause safety policy violations. Our evaluation with 22 PLC binaries collected from real-world PLC programs shows that Ta'veren finds BTVs effectively.

XI. Ethics Considerations

The same bug we discovered in ArduCopter ModeFlip has been found and reported to the ArduPilot development team by other researchers in 2020. We have disclosed the vulnerabilities in the Rover program to the vendor.

Acknowledgment

We thank the anonymous shepherd and reviewers for their feedback and guidance. We thank MITRE for their initial sponsorship of this project, and Dr. Roshan K. Thomas and Gananand G. Kini from MITRE for their unique technical insights and the initial building of the PLC dataset. This material is based upon work supported by the Department of Defense, the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074, the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-22-C-4026, the Department of the Interior Grant No. D22AP00145-00, and the National Science Foundation under Contract No. 2232915, 2146568, 2247954, and 2442984. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] I. Group, "Programmable logic controller (plc) market size and report," 2025.

[2] "Ariane flight v88," https://en.wikipedia.org/wiki/Ariane_flight_V88, 2023.

[3] TrendMicro, "German Steel Plant Suffers Significant Damage from Targeted Attack," 2015, https://www.trendmicro.com/vinfo/fr/security/news/cyber-attacks/german-steel-plant-suffers-significant-damage-from-targeted-attack.

[4] N. Falliere, L. O. Murchu, E. Chien *et al.*, "W32. stuxnet dossier," *White paper, symantec corp., security response*, vol. 5, no. 6, p. 29, 2011.

[5] "Therac-25," https://en.wikipedia.org/wiki/Therac-25, 2023.

[6] The Guardian, "Robot kills worker at Volkswagen plant in Germany," 2015, https://www.theguardian.com/world/2015/jul/02/robot-kills-worker-at-volkswagen-plant-in-germany.

[7] The Airplane Accident Investigation Bureau of Ethiopia, "Investigation Report on accident to the B737-MAX8 operated by Ethiopian Airlines," 2019, https://bea.aero/fileadmin/user_upload/ET_302__B737-8MAX_ACCIDENT_FINAL_REPORT.pdf.

[8] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne, and Z. M. Mao, "Towards Automated Safety Vetting of PLC Code in Real-World Plants," in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 522–538.

[9] Q. Zhang, X. Zhu, M. Zhang, and Z. M. Mao, "Automated runtime mitigation for misconfiguration vulnerabilities in industrial control systems," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 333–349.

[10] S. McLaughlin, S. Zonouz, D. Pohly, and P. McDaniel, "A Trusted Safety Verifier for Process Controller Code," in *Proceedings 2014 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2014.

[11] E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li, "Analyzing Operational Behavior of Stateful Protocol Implementations for Detecting Semantic Bugs," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Denver, CO, USA, 2017, pp. 627–638.

[12] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT Safety and Security Analysis," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA,USA, 2018, pp. 147–158.

[13] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, "Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 4365–4382.

[14] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful Greybox Fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, USA, 2022, pp. 3255–3272.

[15] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 3273–3289.

[16] Q. Shi, X. Xu, and X. Zhang, "Extracting Protocol Format as State Machine via Controlled Static Loop Analysis," in *32st USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, USA, 2023.

[17] E. López-Morales, U. Planta, C. Rubio-Medrano, A. Abbasi, and A. A. Cárdenas, "SoK: Security of Programmable Logic Controllers," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[18] PLCopen, "Sequential function chart (sfc) textual," https://plcopen.org/sites/default/files/downloads/sfc_textual.pdf, 2023.

[19] MathWorks, "Stateflow," https://www.mathworks.com/products/stateflow.html, 2023.

[20] Z. Yang, L. He, P. Cheng, J. Chen, D. K. Yau, and L. Du, "PLC-Sleuth: Detecting and localizing PLC intrusions using control invariants," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 333–348.

[21] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles," in *Network and Distributed Systems Security (NDSS) Symposium 2021*, Feb. 2021.

[22] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, "RVFuzzer: Finding Input Validation Bugs in

Robotic Vehicles through Control-Guided Testing," in *USENIX Security Symposium*, Santa Clara, CA, USA, aug 2019, pp. 425–442.

[23] S. Kim and T. Kim, "Robofuzz: fuzzing robotic systems over robot operating system (ros) for finding correctness bugs," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 447–458.

[24] S. G. Abbas, M. O. Ozmen, A. Alsaheel, A. Khan, Z. B. Celik, and D. Xu, "SAIN: Improving ICS attack detection sensitivity via State-Aware invariants," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 6597–6613.

[25] C. Feng, V. R. Palleti, A. Mathur, and D. Chana, "A systematic framework to generate invariants for anomaly detection in industrial control systems." in *NDSS*, 2019, pp. 1–15.

[26] K. Stouffer, J. Falco, K. Scarfone *et al.*, "Guide to industrial control systems (ics) security," *NIST special publication*, 2011.

[27] H. Kim, R. Bandyopadhyay, M. O. Ozmen, Z. B. Celik, A. Bianchi, Y. Kim, and D. Xu, "A systematic study of physical sensor attack hardness," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 2328–2347.

[28] P. Sun, L. Garcia, and S. Zonouz, "Tell me more than just assembly! reversing cyber-physical execution semantics of embedded iot controller software binaries," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 349–361.

[29] T. Kim, A. Ding, S. Etigowni, P. Sun, J. Chen, L. Garcia, S. Zonouz, D. Xu, and D. J. Tian, "Reverse Engineering and Retrofitting Robotic Aerial Vehicle Control Firmware Using DisPatch," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 69–83.

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

[31] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, *Model Checking and the State Explosion Problem*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30.

[32] A. Keliris and M. Maniatakos, "ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries," in *Proceedings 2019 Network and Distributed System Security Symposium*, 2019, arXiv:1812.03478 [cs].

[33] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, "Model projection: simplifying models in response to restricting the environment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 291–300.

[34] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[35] NuSMV, "NuSMV: a new symbolic model checker," https://nusmv.fbk.eu/, 2023.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 138–157.

[37] MathWorks Inc, "Launch Abort System Simulink example," https://www.mathworks.com/help/stateflow/ug/modeling-a-launch-abort-system.html.

[38] ArduPilot development team, "ArduPilot Project," https://github.com/ArduPilot/ardupilot, 2023.

[39] ——, "Flip Mode," https://ardupilot.org/copter/docs/flip-mode.html, 2023.

[40] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in Instruction List," in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0)*, vol. 4, 2000, pp. 2449–2454 vol.4.

[41] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. Orlando, FL, USA: ACM, Dec. 2017, pp. 315–326.

[42] H. Choi, S. Kate, Y. Aafer, X. Zhang, and D. Xu, "Cyber-Physical Inconsistency Vulnerability Identification for Safety Checks in Robotic Vehicles," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event USA: ACM, Oct. 2020, pp. 263–278.

[43] S. McLaughlin, "CPS:Stateful Policy Enforcement for Control System Device Usage," in *Proceedings of the 29th Annual Computer Security Applications Conference*. New Orleans, Louisiana, USA: ACM, Dec. 2013, pp. 109–118.

[44] Y. Chen, L. Song, X. Xing, F. Xu, and W. Wu, "Automated Finite State Machine Extraction," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. London, United Kingdom: ACM, Nov. 2019, pp. 9–15.

[45] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 523–538.

[46] M. L. Pacheco, M. v. Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, "Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents," in *2022 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2022, pp. 51–68.

[47] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, "What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. Association for Computing Machinery, 2022, pp. 3269–3283.

[48] C. McMahon Stone, S. L. Thomas, M. Vanhoef, J. Henderson, N. Bailluet, and T. Chothia, "The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles CA USA: ACM, Nov. 2022, pp. 2265–2278.

[49] D. Tychalas, H. Benkraouda, and M. Maniatakos, "ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2847–2862.

[50] C. Villa, C. Doumanidis, H. Lamri, P. H. N. Rajput, and M. Maniatakos, "Icsquartz: Scan cycle-aware and vendor-agnostic fuzzing for industrial control systems," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[51] Factory IO, "3. filling tank (timers) - factory i/o," 2024. [Online]. Available: https://docs.factoryio.com/manual/scenes/filling-tank/

[52] Logic Design Inc., "Plclogix - 3d worlds," 2024. [Online]. Available: https://www.plclogix.com/3d-worlds.php

[53] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *Proceedings of the 29th USENIX Conference on Security Symposium*, August 2020, pp. 1237–1254.

[54] Jacques Bellavance, "Vending Machine Arduino example," https://github.com/j-bellavance/Tutorials/tree/master/State%20machines%20Tutorial/Part%202/VendingMachine, 2017.

[55] Ulas Dikme, "Elevator Arduino example," https://ulasdikme.com/2014/05/20/arduino-simple-elevator/, 2014.

[56] ArduPilot development team, "Copter SITL/MAVProxy Tutorial," https://ardupilot.org/dev/docs/copter-sitl-mavproxy-tutorial.html, 2023.

[57] "Intel-hex-file-parser," https://github.com/sfyip/Intel-HEX-file-parser, 2020.

Appendix

A. Formal Proof

We provide a formal proof that the FSM we recovered through state deduplication is sound [33] by proving the soundness of the theorem in Section V.

Theorem. A recovered FSM fully reflects all the behaviors of the original FSM (as implemented in the source program) if the set K of selected variables used to represent a state (i.e., state variables and output variables) is *complete* (i.e., no variable in K exhibits control or data dependencies on intermediate variables).

Proof. Suppose A_f is the full state space of the target program with FSM implemented, where each concrete state can be represented by a $p+q$ dimensional vector $[v_1, ..., v_p, v_{p+1}, ..., v_{p+q}]$, where $v_i \in R$, the set of real numbers, and $v_1, ..., v_p$ are variables in the set K. And $v_{p+1}, ..., v_{p+q}$ are intermediate variables. Σ is the finite input set. $\delta_f : A_f \times \Sigma \rightarrow A_f$ is the transition function.

Let π be a mapping $\pi : R^{p+q} \rightarrow R^p$. For any concrete state $a_f = [v_1, ..., v_{p+q}]$, $\pi(a_f) = [v_1, ..., v_p]$. Let A_r be the reduced state space after the mapping. On the reduced state space, we have a transition $\delta_r : A_r \times \Sigma \rightarrow A_r$. $\delta_r(a_r, \sigma) = \pi(\delta_f(a_f, \sigma))$, where $a_r \in A_r$, $a_f \in A_f$ and $a_r = \pi(a_f)$.

To prove the behavior equivalence of the reduced FSM and the original one, we need to prove:

a) $\forall \sigma \in \Sigma$ and $\forall a_f, a'_f \in A_f$ if $\delta_f(a_f, \sigma) = a'_f$, then $\exists a_r, a'_r \in A_r$ such that $a_r = \pi(a_f)$, $a'_r = \pi(a'_f)$ and $\delta_r(a_r, \sigma) = a'_r$.

b) $\forall \sigma \in \Sigma$ and $\forall a_r, a'_r \in A_r$, if $\delta_r(a_r, \sigma) = a'_r$ then $\exists a_f, a'_f \in A_f$ such that $a_f = [a_r, \cdot]$, $a'_f = [a'_r, \cdot]$ and $\delta_f(a_f, \sigma) = a'_f$.

These two are self evident because of how we defined the transition δ_r in the reduced FSM. Note that in the original FSM, the last q variables in the state vector a_f can be arbitrary values, and they do not affect the calculation of the first p variables in the transition function δ_f. Thus, we have $\pi(\delta_f(a_f, \sigma)) = a'_f[: p] = \delta_r(a_r, \sigma) = \delta_r(\pi(a_f), \sigma)$. This proves a) and b).

B. Descriptions of the Targets

We tried our best to ensure the realism and representativeness of our dataset. We invited collaborators with PLC expertise to implement programs for our dataset. To avoid biases, the collaborators do not know the implementation of Ta'veren, and Ta'veren authors do not influence the PLC implementation. We describe each target program in detail below.

Warehouse Conveyor Lifter. We synthesized the warehouse conveyor lifter program from the real-world implementation used by one of the largest logistics automation companies in the world (the company has decided to stay anonymous) and built it in OpenPLC. This program controls a conveyor lifter that moves boxes vertically to place them in their corresponding pallet rack. The input of the program is the current position of the conveyor. The program has three outputs: a motor that controls the lifter to go up, a motor that controls the lifter to go down, and a motor that controls the conveyor that delivers the box inside the pallet rack. The program starts when the conveyor lifter is on the ground level. The lifter motor is activated to go up and stop at the assigned pallet rack. The motor of the conveyor lifter is activated to deliver the box inside the pallet rack. Finally, the lifter motor is activated to go down to the ground level to get the next box, and the process starts all over again.

Water Tank. We synthesized the water tank program from Factory I/O [51] and built it in OpenPLC with FBD and SFC. Two variants have two sensors. The inputs are the readings of two boolean water level sensors, one for low and one for high. One variant has one sensor. The input is the reading of a continuous value of water level, and the output is the water pump. During the process, if the water level is too low, the water pump is turned on; if the water level is too high, the pump is turned off.

Packaging System. We synthesized the packaging system program from PLCLogix 500 [52] and built it in OpenPLC. The program inputs are box proximity sensors to detect the empty status of the product. The outputs are the conveyor belt motor and product valve. At the start, the conveyor belt moves the box toward the product valve and stops when the proximity sensor detects the box. The product valve is then activated for 25 seconds to fill the box. Finally, the conveyor belt is activated again, and the process starts all over.

Car Wash System. We synthesized the car wash program from PLCLogix 500 [52] and built it in OpenPLC. The inputs of the program are an internal timer, a sensor that detects the car, and a selection button that allows customers to select the service type (wash only, wash and dry, or wash, dry, and wax). The output is the control of washing, rinsing, drying, and waxing functionalities. The process starts when the customer selects a service type, then the conveyor belt is activated to move the car to each one of the car wash stages.

Traffic Light Simulation. This program is based on an example provided by Beremiz. We built the same program in Beremiz, OpenPLC, and Simulink to get the binaries. It represents a traffic light at an intersection. It features three lights (yellow, red, and green) for cars and two lights (red and green) for pedestrians. The input for this program is the internal timer, and the outputs are five boolean signals.

Launch Abort System. This is an example project from Simulink [37]. This program includes the control logic for a rocket to safely initiate an abort sequence if an anomaly occurs. Its inputs include the rocket's altitude and an anomaly signal that indicates whether a fault or attack has occurred during the launch. Its outputs are switches to initiate the abort sequences. When an abort sequence is initiated, the rocket dumps fuel, releases boosters, and releases the external tank based on its altitude.

Oven Controller. We synthesize the oven controller program from the dataset of $\mathrm{P^2IM}$ [53] and build it in Arduino. The inputs of this program are the readings of an internal timer and a temperature sensor. The output is a continuous (analog) signal that controls the power of the heater. When the oven starts, it turns on the heater and preheats the oven. When the temperature reaches a threshold (e.g., 160°C), it will cook for 9 seconds while maintaining the temperature. Then it turns off the heater. The process completes when the oven reaches room temperature.

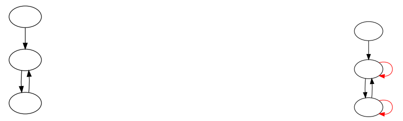
Vending Machine. This is an open-sourced Arduino project [54] that we compiled locally. The vending machine control program takes dollars and quarters as input and outputs a can of soda (sold for 75 cents) and the change if the input exceeds the price.

Elevator. This is another open-sourced Arduino project [55] that we compiled locally. The input of the elevator program is a set of buttons indicating the specific floor the user would like to go to. Based on the selected floor, it outputs a motor value that determines the elevator's vertical movement.

ArduCopter. ArduPilot [38] is an open-sourced AutoPilot Software Suite for unmanned vehicles. We built ArduCopter binaries using the SITL simulator [56], and pulled one off the processor of a physical ArduCopter-based drone. Note that we do not perform FSM recovery on the whole system. The FSM that we recover and evaluate is the FLIP mode for ArduCopter [39]. In this program, the inputs include rolling angle sensor measurements, and the outputs include throttle. After the copter is switched to FLIP mode, (1) it first starts to increase the throttle until the rolling sensor detects that the rolling angle is larger than 45 degrees, (2) it then decreases the throttle while rolling, and (3) when the rolling angle exceeds -90 degrees, it increases the throttle again to gain lost altitude to recover to the original position.

Rover. We received a rover from our funding agency and pulled the binary off the STM32 processor. The rover's CPS program takes, as input, a command and traveling distance, and outputs throttle and servo to cause the rover to move a fixed length, turn in the given direction, continue moving for a fixed length, and then stop.

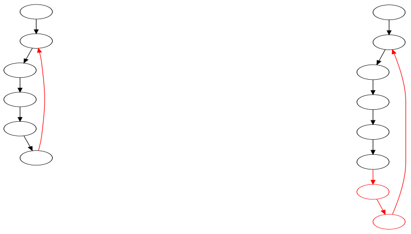
C. Graphs



(a) Reference graph for Water Tank target WT.1 and WT.3

(b) Generated graph for Water Tank target WT.1 and WT.3

Fig. 7: FSMs for Water Tank two sensors FBD (WT.1) and SFC (WT.3).



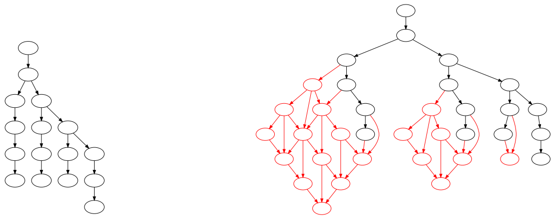
(a) The reference graph.

(b) The recovered graph.

Fig. 8: The FSMs for TL.7. Due to the source code misimplementation, there are two unmatched nodes in (b), which causes four unmatched edges.

D. Investigating Mismatched FSMs

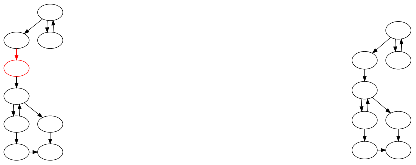
The recovered and reference FSMs of some targets do not fully match. The figures in Appendix C show some unmatched FSMs. We highlight the nodes and edges that do not match between the reference and generated graphs in red. We investigate every semantic difference and summarize the reasons as follows.



(a) Reference graph for abort target Abort.3

(b) Generated graph for abort target Abort.3

Fig. 9: FSMs for Abort (Abort.3).



(a) Reference graph for oven target Oven.1

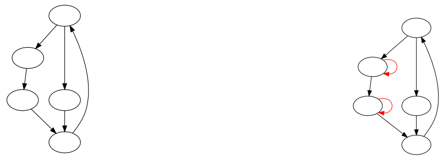
(b) Generated graph for oven target Oven.1

Fig. 10: FSMs for Oven (Oven.1). There is a missing intermediate node in the generated graph.

Missing intermediate states. For Oven.1 (see Figure 10), Ta'veren misses an intermediate state between PREHEAT and COOK states. In this case, the action (heater on in PREHEAT) and the state transition (to COOK) can occur within the same scan cycle. Because Ta'veren recovers an abstract state at the end of the scan cycle, Ta'veren only recovers one abstract state after the state transition, missing the intermediate state.

Unmatched states due to source code misimplementation. For TL.7 (see Figure 8) and TL.10, Ta'veren discovers two more nodes than the source code. This is caused by the misimplementation of the Beremiz program—it fails to reset the output of the inactive actuators, whose values are carried over to the next state. This misimplementation causes two extra unintended states, where Ta'veren later finds policy violations.

Similarly, for Abort.3 (see Figure 9), Ta'veren discovers 19 more states than the source code. The Launch Abort System is implemented with embedded FSMs, where the control logic of one state (Abort) is another FSM (AbortLogic). However, the program did not implement the constraints on what anomalous signal it could take and failed to properly coordinate state changes between the two FSMs. Specifically, once the AbortLogic is triggered, its actions keep executing, even if the outer FSM is no longer in this state, creating extra states.



(a) Reference graph for vending machine target Vend.1

(b) Generated graph for vending machine target Vend.1

Fig. 11: FSMs for Vending Machine (Vend.1).

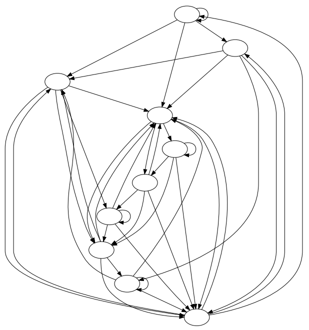


Fig. 12: FSM for Elevator (Elev.1) (matched)

E. Control Flow Graphs

F. The Reduction of State Space

We investigate the effectiveness of Ta'veren in avoiding the state explosion problem and reducing the sizes of state space during FSM recovery.

We first compare Ta'veren against bounded symbolic exploration on TL.9. Figure 14 shows the number of actual program states that bounded symbolic exploration discovers with path depth limits of 1K, 10K, and 100K. Ta'veren terminates and discovers 14 unique states in the recovered FSM, while bounded symbolic exploration discovers hundreds to close to 100k states. This demonstrates that Ta'veren effectively avoids the state explosion problem from which symbolic exploration suffers.

Next, we compare Ta'veren and TSV [10]. Because TSV works on Instruction List (IL) programs and is not public, our comparison is qualitative. The original TSV paper reports TSV-recovered states of a traffic light program similar to our Traffic Light programs. With a bound of 6, TSV recovers 20 states, and with a bound of 14, TSV recovers about 4k states. The number of states recovered by TSV grows exponentially to its bound limit. This is because although TSV aggressively deduplicates program states, it considers *all* actual program states, while Ta'veren only considers abstract states.

G. Protocol test

To explore Ta'veren's capabilities of recovering FSMs on other types of FSM implementations beyond PLC programs, we adopt Ta'veren's methodology on regular parsers. Ta'veren successfully generated an FSM for Intel hex file parser [57], which demonstrates the generalizability of Ta'veren.

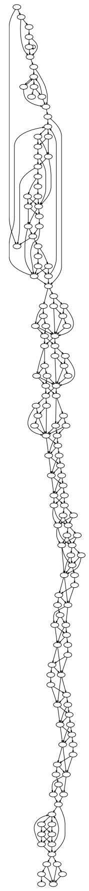


Fig. 13: The control-flow graph of the main logic function of the Water Tank PLC program as shown in Listing 1.

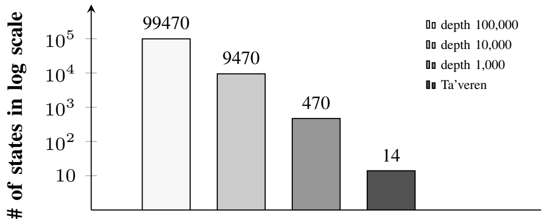


Fig. 14: The total number of recovered states of Ta'veren comparing against those of bounded symbolic exploration with path depth limits of 1,000, 10,000, and 100,000 on binary TL.9. Lower is better.