SAIN: Improving ICS Attack Detection Sensitivity via State-Aware Invariants

Syed Ghazanfar Abbas, Muslum Ozgur Ozmen, Abdulellah Alsaheel, Arslan Khan, Z. Berkay Celik, and Dongyan Xu, *Purdue University*

https://www.usenix.org/conference/usenixsecurity24/presentation/abbas

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

SAIN: Improving ICS Attack Detection Sensitivity via State-Aware Invariants

Syed Ghazanfar Abbas, Muslum Ozgur Ozmen, Abdulellah Alsaheel, Arslan Khan,
Z. Berkay Celik, Dongyan Xu

Purdue University,
{abbas4, mozmen, aalsahee, khan253, zcelik, dxu}@purdue.edu

Abstract

Industrial Control Systems (ICSs) rely on Programmable Logic Controllers (PLCs) to operate within a set of states. The states are composed of variables that determine how sensor data is interpreted, configuration parameters are applied, and actuator commands are issued. Recent works have shown that attackers can manipulate these variables to compromise ICS safety and security. To detect such attacks, previous approaches have leveraged invariants—a set of rules defining the correct behavior of an ICS. However, these invariants suffer from a critical limitation: they are *state-agnostic*. This means they define variable ranges across all possible ICS states, leading to loosely bounded detection thresholds. Unfortunately, attackers can exploit these loose bounds and launch stealthy attacks that evade detection without violating such invariants.

In this paper, we introduce SAIN, an automated method to derive *state-aware* ICS invariants with tighter bounds and enforce them through a PLC-based monitor. SAIN first generates invariant templates by identifying the PLC program states, state transitions, and the inter-dependencies among sensing, actuation, and configuration variables within each state through program analysis. It then partitions the ICS data traces into state-specific sub-traces and quantifies the invariant templates with concrete, tighter bounds, as system-specific knowledge about the subject ICS. Lastly, it enforces the state-aware invariants through a run-time monitor. We evaluate SAIN on a Fischertechnik manufacturing plant and a chemical plant simulator against 17 attacks. SAIN protects the plants, on average, with a false positive rate of 2% and a run-time overhead of 3%.

1 Introduction

Industrial Control Systems (ICS) play a vital role in automating and managing complex industrial processes. Compared to traditional ICS with non-programmable controllers, ICS with Programmable Logic Controllers (PLCs) achieve increased configurability and automation. The PLC program acts as the central processing unit that monitors and governs the entire control process. The execution of a PLC program is based on three categories of variables. (1) The sensing variables are continuously updated with real-time data acquired from the sensors connected to the PLC's input station. (2) The configuration variables encompass a set of parameters that establish the operational characteristics of the ICS. (3) The actuation variables define the control signals transmitted to the PLC's output station connected to the actuators.

Prior research has shown that ICSs are vulnerable to attacks that manipulate the PLC variables without altering the underlying PLC program [8, 10, 38, 43]. These attacks can have disastrous consequences, as evidenced by real-world incidents. For instance, the Stuxnet worm [13] caused extensive damage to centrifuges in a nuclear facility by altering the converter frequency variable from 2 Hz to 1410 Hz. Similarly, the BlackEnergy attack [8] compromised a power system by manipulating circuit breaker variables from 1 to 0, leading to the disconnection of 27 substations.

To detect such ICS attacks, prior work has explored techniques that leverage invariants, which define the relationship between PLC variables and their value bounds [14, 23, 48]. These invariants are typically derived through data mining techniques applied to ICS execution traces that record sensor readings, actuator values, and configuration parameters during the normal operation of an ICS. Another line of work has incorporated the temporal characteristics of an ICS into invariants [3, 27, 52, 53], enabling them to integrate the timing relations between variables to improve attack detection.

Despite their success, we notice that state-of-the-art techniques define an invariant as a property/behavior that must be true *throughout the entire operation* of an ICS and, as a result, some invariants may set loose bounds of variable values to hold throughout the ICS operation. In practice, however, an ICS process is naturally partitioned into a sequence of distinct states (operational phases) to control the physical process in smaller tasks, ensuring that the operation of the ICS in each state is successfully completed before transitioning to the next state. Our key observation is that the

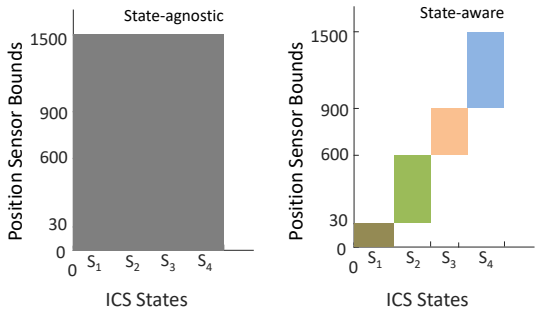


Figure 1: Bounds in state-agnostic vs. state-aware invariants.

value bounds for an invariant may vary depending on the specific ICS state. Figure 1 illustrates this limitation by comparing state-agnostic and state-aware invariants derived from a manufacturing testbed [40]. The state-agnostic approach sets a single, global bound on invariants that is satisfied by all collected ICS traces. Such looser bounds in state-agnostic invariants allow attackers to launch stealthy attacks in certain states. In contrast, the state-aware approach establishes invariants with tighter bounds specific to each ICS state.

To address this problem, we introduce SAIN[1], a framework for deriving and enforcing state-aware invariants with tighter state-aware PLC variable bounds. SAIN consists of four modules: ICS state identification, code-level invariant derivation, invariant quantification, and runtime monitoring. SAIN first conducts static analysis on the PLC program to extract ICS states, state transitions, and the relations among PLC variables (i.e., sensing, configuration, and actuation). It then generates invariant templates that involve the relevant variables for each state, and partitions benign ICS traces into state-specific sub-traces based on the state transitions. Lastly, it leverages an association mining algorithm to numerically instantiate the invariant templates by mining the state-specific sub-traces.

During ICS operation, SAIN enforces the state-aware invariants in the PLC runtime, which executes the process control code and interacts with the I/O modules [33]. Among the PLC runtime functions, only those that are subject to (potentially malicious) remote update of PLC variables are instrumented with the SAIN runtime monitor. As such, the runtime monitor, as a detection agent, is only triggered when the PLC receives a program variable update request from an external entity (e.g., the human-machine interface). This ensures that SAIN runtime monitor guards against external attacks, with minimal impact on the PLC scan cycle time.

We evaluate SAIN on Fischertechnik (FT) [15] (a real miniature manufacturing plant testbed) and a simulated chemical plant [17] against 17 attacks, with real PLC code running on both plants. Our evaluation shows that SAIN detects and prevents all attacks on average with a false positive rate of 2% and run-time overhead of 3%. We compare SAIN's attack detection effectiveness with state-of-the-art ICS invariant-based techniques, Feng et al. [14] and VetPLC [52]. The invariants derived from Feng et al. detect 5/17 attacks, and the temporal invariants derived following VetPLC detect 6/17 attacks.

In summary, we make the following contributions.

- We introduce SAIN, an automated approach for deriving state-aware invariants, as system-specific knowledge, by combining PLC program analysis and trace mining. Compared to their state-agnostic counterparts, state-aware invariants set tighter bounds that enable the detection of stealthy attacks.
- We develop a SAIN runtime monitor, as a detection agent, that leverages state identification signatures to determine the current ICS state and enforces state-aware invariants.
- We demonstrate the effectiveness of SAIN on two ICS testbeds through 17 attacks, including false configuration injection, sensor data manipulation, malicious actuation signals, and physics-aware attacks. SAIN detects all attacks with low runtime overhead.

SAIN source code, attack videos, and traces are available at https://github.com/purseclab/SAIN.

2 Background

Programmable Logic Controller (PLC). The ICS monitors and controls an industrial or manufacturing process. The PLC is the central controller of the ICS, executing the control logic that governs the sensing-control-actuation loop of the process.

The PLC has two key components: the variables and program. PLC program variables fall into three categories: (1) sensing variables that reflect real-time sensor measurements, (2) configuration variables that define operational parameters of the ICS, and (3) actuation variables that carry control commands to physical actuators within the plant. The PLC program defines the control logic, issuing control commands to the physical plant based on real-time sensing variables.

State-driven PLC Program. The nature of ICS leads to the fact that its PLC program reflects and implements the ICS's states and transitions, modeled with a Finite State Machine (FSM) [18,30,53]. Each state in the FSM represents a specific operational phase of the ICS and the corresponding actions it performs. Transitions between states occur based on new events or changes in process variables (e.g., sensor readings or actuator states). The state-driven programming of PLCs naturally aligns with industrial process control. By decomposing the ICS operation into distinct states, the PLC program monitors and executes control logic in a state-specific manner. The FSM model has been widely used for the automated generation, validation, and transformation of PLC code [39,45,54].

PLC Runtime. The PLC runtime is a privileged layer of software that executes the PLC program. Modern PLC runtime

[1]The acronym SAIN stands for "state-aware invariants", with the word *sain* in French meaning "healthy".

```
int main(int argc,char **argv){
    //Creating PLC Variables
    glueVars();
    //Initiating Server Applications
    pthread_create(&interactive_thread, NULL, appThread, NULL);
    //Scan Cycle
    while(run_openplc){
            //Updating Input Variables with Sensor Data
            updateBuffersIn_MB();
            //Execute Control Program
            config_run__(__tick++);
            //Write Output  Variables to Output devices
            updateBuffersOut_MB();}
}
```

Listing 1: Overview of OpenPLC runtime execution flow.

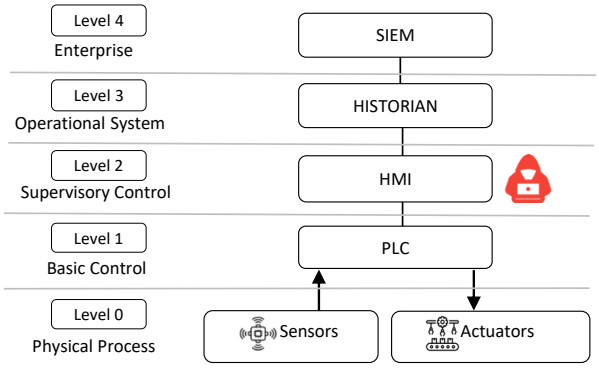


Figure 2: The illustration of ICS operation based on Purdue model and the threat model of SAIN.

also supports the execution of server applications that allow other devices/machines to communicate with the PLC. For example, the Siemens PLC runtime executes a web server [5], while the OpenPLC runtime supports applications to which devices can be connected [2].

Listing 1 shows the control flow of an OpenPLC runtime. The initialization phase establishes the PLC variables and server applications. These variables can be accessed by both the server applications and the PLC control program. The main loop continuously updates the PLC variables with sensor readings, executes the control program, and subsequently sends control commands to the actuators.

3 Threat Model

We describe our threat model using the Purdue Model [50] for ICS security. Purdue Model is a hierarchical model for ICS, as shown in Figure 2. Level 0 implements a network of sensors and actuators interfaced to Level 1, which consists of multiple PLCs. Level 2 implements supervisory entities, such as SCADA and HMI, to control PLCs by sending messages. The messages from Level 2 are served by different server applications in the PLCs. These applications can allow the manipulation of PLC variables, including sensing, configuration, and actuation variables.

We assume the entities above Level 1 can be remotely compromised by the attacker to disrupt ICS operations. The attacker can compromise entities at Level 2 and above using existing mechanisms (e.g., spearphishing to compromise the HMI [31] and by exploiting disabled security functionalities in server applications [11]). Specifically, attackers can remotely access these entities to inject malicious messages into the Level 1 network and manipulate the sensing, configuration, and actuation variables in the PLC. We assume that the Level 0 and 1 network is air-gapped from the Level 2 network. Therefore, entities at Level 2 cannot spoof messages within Level 0 and Level 1 networks. We assume that the attacker cannot manipulate the PLC firmware. Additionally, we assume that the attacker does not have physical access to ICS devices such as sensors, actuators, and PLCs. Consequently, the attacker cannot spoof physical sensor and actuator values. We also assume that the PLC control program correctly implements the ICS states and state transitions, ensuring that the FSM derived from the PLC program is correct. Our threat model aligns with those in the prior work that leverage invariants for ICS security [24, 29, 30].

4 Overview of ICS Attacks and Invariant-Based Defenses

A wide range of attacks are caused by the fact that PLCs are becoming increasingly connected by running server applications (e.g., Web server, MQTT, and OPC-UA). Through these applications, remote operators can access and modify PLC program variables as an operation convenience. Unfortunately, such remote connectivity also enables attacks that remotely manipulate PLC program variables [8, 10, 38, 43]. As a defense, invariant-based techniques have been proposed to detect attacks that violate the invariants.

However, previous approaches define state-agnostic invariants. The lack of ICS state-awareness leads to invariants with loose bounds, within which stealthy attackers can manipulate the variables "under the radar" and evade invariant-based detection. We present a motivating example of a stealthy evasion attack that exploits the loose bounds of the PLC variable values in state-agnostic invariants.

4.1 Motivating Example

We present an attack against the Fischertechnik (FT) manufacturing plant, a PLC-controlled production factory [40] (described in Appendix A). The plant's delivery process is controlled by a PLC program that implements the FSM shown in Figure 3. Each state (S_i) is associated with a set of semantically close actions to be completed. The state transitions (T_i) represent the conditions that must hold for the PLC to advance to the next state.

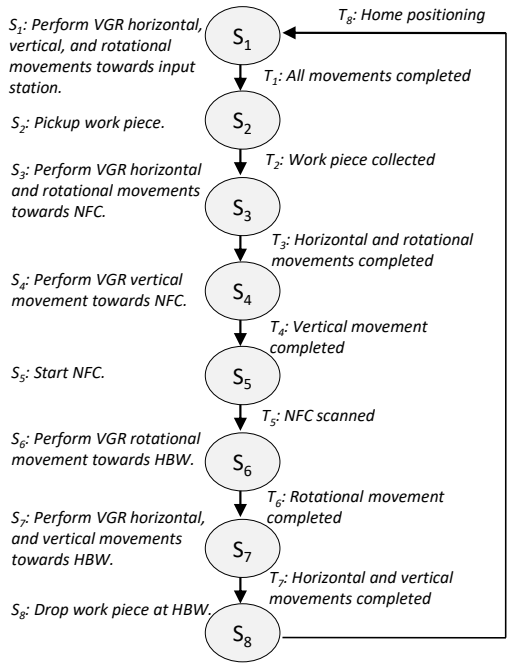


Figure 3: PLC workflow in the FT manufacturing plant. S_i denotes the states, and T_i denotes the transition between states.

Attack Objective. In the delivery process, the attacker's objective is to initiate the vacuum gripper robot (VGR) horizontal movement during state S_6. Yet, the PLC control logic is designed to trigger only the rotational movement in S_6, with the intent of preventing any collision between the VGR and high-bay warehouse (HBW) storage site. Thus, if the attacker can cause the VGR to perform a horizontal movement during S_6, this could potentially damage both the VGR and HBW.

PLC Logic for Controlling VGR Horizontal Movement. The PLC configuration variable "H_TargetPosition" is set based on the VGR's target position, as 600 for HBW and 900 for NFC. To initiate the horizontal movement, the PLC activates the VGR's horizontal actuator "H_StartPositioning." During the horizontal movement, the PLC continuously receives the VGR's current position through sensing variable "H_ActualPositioning." The PLC monitors the value of that variable and compares it to the target position and an offset. Specifically, when the value of "H_ActualPositioning" becomes greater than or equal to [H_TargetPosition - Offset] and less than or equal to [H_TargetPosition + Offset], the PLC stops the VGR's horizontal movement.

Limitations of Previous Defenses. Previous invariant-based approaches [12, 14] usually begin by defining predicates for actuation variables and then deriving the value ranges of the sensing variables for each predicate. For instance, the invariant (I_1) listed below is derived following Feng et al. [14] using Fischertechnik's data traces. I_1 establishes the benign variable value ranges based on traces from the entire plant operation.

Table 1: State-aware invariants to prevent VGR-HBW attack.

State	Invariant
S_1	H_Target = 30 & H_Actual[0-14] → H_Start = ON
S_2	H_Start = OFF
S_3	H_Target = 600 & H_Actual[0-584] → H_Start = ON
S_4	H_Start = OFF
S_5	H_Start = OFF
S_6	H_Start = OFF
S_7	H_Target = 900 & H_Actual[0-884] → H_Start = ON
S_8	H_Start = OFF

However, such ranges turn out to be too loose, allowing a stealthy attacker to carry out the VGR-HBW collision attack. Specifically, the attacker is able to initiate the VGR's horizontal movement during a specific ICS state (S_6) without violating the invariant (I_1).

- I_1: if H_TargetPos[30-900] and H_ActualPos[0-884] then H_Start = ON

The VGR-HBW attack remains undetectable even with temporal invariants due to the absence of a tighter time bound. Invariant I_2 below is extracted following the temporal invariant derivation method in VetPLC [52]. This invariant sets an upper time limit of 7 seconds without considering the specific timing constraints under individual states.

- I_2: if H_TargetPos[30-900] and H_ActualPos[0-884] then H_Start = [ON, 7 sec]

Our Approach. SAIN detects stealthy attacks by generating state-aware invariants with tighter bounds and enforcing them when the PLC receives an external variable update request from the HMI. Table 1 presents the PLC variable bounds for an invariant under different states. Thus, when the attacker initiates the VGR's horizontal movement in S_6, a violation is detected because the VGR's horizontal movement is not allowed in this state.

4.2 Design Goals and Challenges

Our main goal is to defend ICS against stealthy attacks that evade state-agnostic invariants. To this end, we define our design goals as follows:

G1: Generating state-aware invariants with tighter bounds. As detailed in Section 4.1, prior works overlook the ICS states and derive invariants from the entire set of benign ICS traces, resulting in loose PLC variable bounds. However, generating state-aware invariants as system-specific, quantifiable knowledge is challenging because it requires (1) identifying the specific states from a large PLC code that controls multiple interacting components, (2) partitioning the data traces based on the states, and (3) generating and quantitatively concretizing invariants that define the relations between PLC variables under each state separately. Addressing these challenges requires a combination of PLC program analysis and ICS data trace mining.

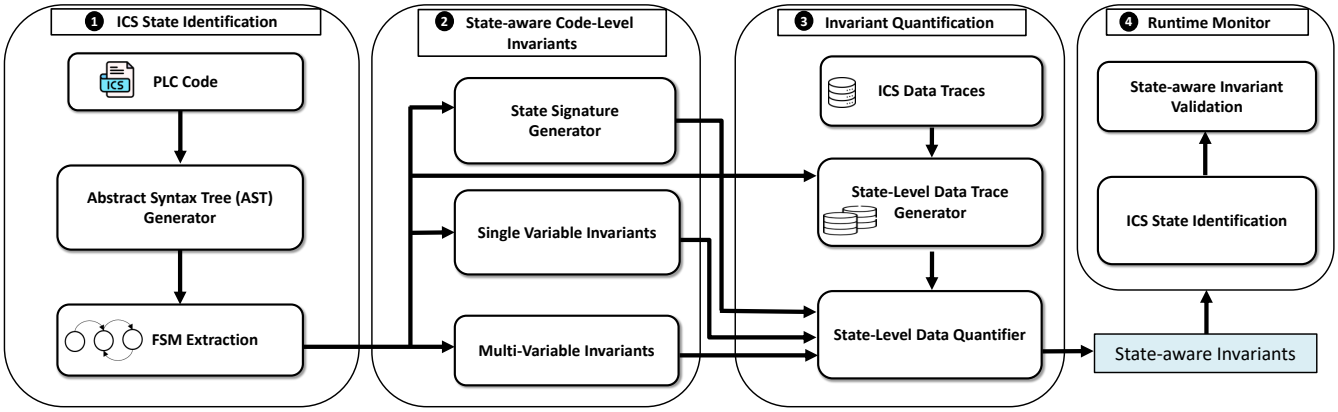


Figure 4: Overview of SAIN architecture.

G2: Monitoring and enforcing invariants at runtime. The state-aware invariants need to be enforced during ICS operations. Here, challenges include dynamically determining the ICS state at runtime and enforcing the associated invariants accordingly. Moreover, the invariant enforcement should incur a low performance overhead to satisfy the ICS's real-time requirements. To address these challenges, we extract the state transition conditions from the PLC code as unique signatures of each state, and use them at runtime to determine the current ICS state. We also develop a PLC runtime monitor that incurs minimal overhead on the PLC scan cycle timing.

5 SAIN Design

We introduce SAIN, an automated framework that generates and enforces state-aware invariants with tighter PLC variable value bounds, derived as system-specific knowledge from both PLC program and traces. Figure 4 presents the overview of SAIN, which is composed of four key stages: (1) ICS state identification, (2) state-specific code-level invariant derivation, (3) invariant quantification, and (4) SAIN runtime.

SAIN first leverages the PLC program to identify the states that are defined and implemented to control ICS operations (❶). SAIN determines the conditions, which include the sensing, configuration, and actuation variables in the PLC, that must hold under each ICS state and converts them into invariant templates (❷). For example, an invariant template defines the relation between specific sensing, configuration, and actuation variables that trigger the VGR's horizontal movement. SAIN then quantifies the invariant templates by leveraging ICS data traces – partitioned by state – to set concrete bound values in the templates (❸). Lastly, to enforce state-aware invariants, SAIN has a runtime monitor that validates PLC invariants upon receiving external requests through the HMI. The runtime monitor identifies the current ICS state and enforces invariants for that state (❹).

5.1 ICS State Identification

SAIN extracts the ICS states from the PLC code through program analysis. These states then guide SAIN's code-level, state-specific invariant derivation (Section 5.2) and invariant quantification (Section 5.3) stages.

Our key observation is that the PLC functions as an FSM, running among predefined states in accordance with the control flow of the PLC program. In fact, PLC programming languages (IEC 61131-3 standard [22]) inherently emphasize state-based control. For example, Structured Control Language (SCL) and Structured Text (ST) allow the creation of FSMs and state-driven logic using a text-based syntax. Sequential Function Chart (SFC) is specifically designed to define and execute state-based control logic, offering a visual representation of states and transitions [54].

Following this observation, to extract ICS states from a PLC program, SAIN first generates an Abstract Syntax Tree (AST) as an intermediate representation of the PLC program. It then builds an FSM model of the PLC code, mapping out the behaviors and transitions of the ICS within and across states.

5.1.1 Abstract Syntax Tree (AST) Generation

SAIN performs parsing to break down the PLC code into individual tokens, and constructs an AST through the Parsing Expression Grammar (PEG) [16]. This grammar acts as a set of rules that guide the parsing process and reflect the control logic structure. The AST is a generalized representation of the PLC program that includes state identifiers, variable dependencies, and control flow relations, enabling SAIN to extract language-independent FSMs.

5.1.2 FSM Extraction

Algorithm 1 outlines SAIN's approach to extracting an FSM from the AST of the PLC code. The algorithm first examines the functional blocks of the PLC control logic (Line 3-4).

Algorithm 1 Finite State Machine (FSM) Construction

Input: PLC Program
Output: Finite State Machine (FSM)

1: Initialize an empty FSM
2: Convert PLC program to an Abstract Syntax Tree (AST)
3: Extract function blocks from AST
4: **for** each function block **do**
5: **for** each case block in the function block **do**
6: Define a state in the FSM for case block
7: Create control flow graph for case block
8: Create data flow graph for case block
9: Combine control and data flow graphs to form PDG
10: Identify transition conditions from PDG
11: Include the transition conditions as an edge in the FSM
12: **end for**
13: **end for**
14: Return the FSM

Table 2: Illustration of state transition conditions for the delivery process in the FT manufacturing plant.

SS*	DS†	Transition Condition‡
S_1	S_2	H = INPUT, V = INPUT, R = INPUT
S_2	S_3	W = $\surd$
S_3	S_4	H = NFC, R = NFC
S_4	S_5	V = NFC
S_5	S_6	S = $\surd$
S_6	S_7	R = HBW
S_7	S_8	H = HBW, V = HBW

*SS is the source state. †DS is the destination state. ‡H is for horizontal sensor, V is for vertical sensor, R is for rotational sensor, W is for workpiece collected, S is for NFC scan done.

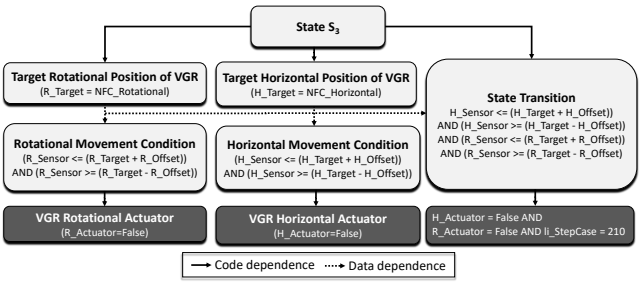
For each functional block, SAIN iterates the case statement blocks (Line 5), constructing a control flow graph for each case (Line 7). The control flow graph represents individual statements as nodes and reflects the program's control flow along its edges. SAIN then constructs a data flow graph by tracing the data dependencies between statements (Line 8). For each case statement, SAIN combines the control and data flow graphs to create the Program Dependency Graph (PDG), including both control and data dependencies (Line 9). Each case block represents a distinct state in the FSM, allowing us to perform state-level program analysis.

For each state, SAIN stores the PDG and a state transition table. The transition table has two fields. The "toState" to record the next state and "condition" to record state transition conditions. SAIN determines the "toState" field by identifying the specific variable(s) that cause the switches between the case statements. It then finds the variables that have control and data dependencies on the "toState" field from the PDG, identifying the state transition conditions (Line 10). SAIN then includes the state in the FSM as a node, and incorporates the transition conditions to the outgoing edges from that node (Line 11). Figure 5 shows the PDG and transition table for state S_3 extracted from Listing 2.

Table 2 shows the state transition conditions that correspond to the FSM depicted in Figure 3. The Source State

```
1  //Horizontal (H) movement towards NFC
2  H_Target := NFC_Horizontal;
3  H_Actuator := TRUE ;
4  IF (H_Sensor <= (H_Target + H_Offset))
5    AND (H_Sensor >= (H_Target - H_Offset)) THEN
6    H_Actuator := FALSE;
7  END_IF;
8
9  //Rotational (R) movement towards NFC
10 R_Target := NFC_Rotational;
11 R_Actuator := TRUE ;
12 IF (R_Sensor <= (R_Target + R_Offset))
13   AND (R_Sensor >= (R_Target - R_Offset)) THEN
14   R_Actuator := FALSE;
15 END_IF;
16
17 IF (H_Sensor <= (H_Target + H_Offset))
18   AND (H_Sensor >= (H_Target - H_Offset))
19   AND (R_Sensor <= (R_Target + R_Offset))
20   AND (R_Sensor >= (R_Target - R_Offset)) THEN
21   H_Actuator := FALSE;
22   R_Actuator := FALSE;
23   #li_StepCase := 210;
24 END_IF;
```

Listing 2: The PLC code of state S_3 shown in Figure 3.



(a) Program Dependency Graph (PDG)

Condition	toState
H_Sensor <= (H_Target + H_Offset)) AND (H_Sensor >= (H_Target - H_Offset)) AND (R_Sensor <= (R_Target + R_Offset)) AND (R_Sensor >= (R_Target - R_Offset) AND H_Actuator = FALSE AND R_Actuator = FALSE	li_StepCase = 210

(b) Transition Table

Figure 5: The program dependency graph and transition table for the state S_3, extracted from Listing 2.

(SS) indicates the state from which the transition condition is verified to move the ICS to the Destination State (DS). For instance, if the ICS is in state S_3, and the horizontal and rotational sensors values indicate the VGR has reached the NFC position, the ICS moves to state S_4.

5.2 State-aware Code-level Invariants

SAIN leverages the FSM of the PLC program to extract each state's distinct properties, which become invariants of that state. From the inter-state characteristics (state transition conditions), we define (1) *state identification signatures* that enable identifying the current ICS state at runtime and (2) *inter-state invariants* that allow checking the validity of state transitions. From the intra-state characteristics (the PLC variables in a specific ICS state), we generate *intra-state invariants* that define the properties/behaviors intrinsic to that state.

Algorithm 2 State Signature Generation

Input: Finite State Machine (FSM)
Output: State Signature Table

1: Initialize an empty State Signatures Table
2: Identify all paths in FSM using Depth-First Search (DFS)
3: **for** each path in the FSM **do**
4: **for** each state in the path **do**
5: Extract transition conditions leading to the state
6: Accumulate conditions into signature
7: **end for**
8: Remove redundant conditions from signature
9: Store the signature in State Signature Table
10: **end for**
11: **return** State Signature Table

Table 3: State signatures of the delivery process states.

State	H	V	R	W	S
S_1	0	0	0		
S_2	Input	Input	Input		
S_3	Input	Input	Input	√	
S_4	NFC	Input	NFC	√	
S_5	NFC	NFC	NFC	√	
S_6	NFC	NFC	NFC	√	√
S_7	NFC	NFC	HBW	√	√
S_8	HBW	HBW	HBW	√	√

H: Horizontal sensor, **V**: Vertical sensor, **R**: Rotation sensor, **W**: Workpiece collected, **S**: NFC scanned.

5.2.1 ICS State Signatures

An ICS state signature consists of a set of PLC variables and the unique values they take for the ICS to transition to that specific state. In the FSM, the transition conditions are designed to create a unique path to a state, guaranteeing that no two states can be reached using the same combination of transition conditions. Therefore, the combination of the transition conditions generates a unique signature for a state.

From the FSM, SAIN collects PLC program variables from the transitions leading to each state and specifies their value or range during that state. By checking these variables during runtime, SAIN determines the current ICS state. Algorithm 2 outlines state signature generation from the FSM. The algorithm first traverses the FSM using the depth-first search method to identify all distinct paths (Line 2). A path is a sequential series of states within the FSM. For each state, SAIN combines the state transition conditions leading to that specific state (Line 5-6). This combination forms a signature that encapsulates the sequence of conditions leading to a state. If there are multiple paths leading to the same state, it generates multiple signatures for that state. SAIN eliminates any repetitive conditions found in the signature, ensuring that each signature is concise and unique (Line 8).

Table 3 presents the state signatures for the delivery process, extracted from the transition conditions in Table 2. Each signature is unique to the corresponding state, e.g., when SAIN observes the signature "H=NFC, V=NFC, R=HBW, W=√, S=√", it determines that the current state is S_7.

5.2.2 Intra-state Invariants

Intra-state invariants are rules that apply to a specific ICS state to maintain its integrity. Each ICS state is programmed to execute specific operations using the sensing, configuration, and actuation variables. If an ICS state deviates from its designated operations at runtime, intra-state invariants are violated.

SAIN derives two types of intra-state invariants: single-variable invariants and multi-variable invariants. A single-variable invariant specifies the allowable value range for an individual variable within a state. For instance, within a state, a configuration variable is restricted to hold only the value "900". A multi-variable invariant involves a set of inter-related sensing, configuration, and actuation variables, along with their respective value ranges, in a state. A multi-variable invariant is composed of two sides. The right side has the actuation variable, and the left side has variables that can impact the actuation variable's value. To determine the actuation variables, SAIN checks if a variable has only incoming edges in the PDG (defined in Section 5.1.2). To determine the variables on the left side of the invariant, SAIN finds all variables with outgoing edges directed toward the actuation variable.

Example. Listing 2 represents a code snippet for the state S_3 of Figure 3. The code segment first initializes the target positions for the VGR's horizontal and rotational movements, represented by variables H_Target and R_Target, respectively. It then sends control signals to the respective actuators. After sending the control signals, it checks the readings from both the horizontal sensor (H_Sensor) and the rotational sensor (R_Sensor). If both the horizontal and rotational sensor readings fall within their respective target ranges, it indicates that the VGR has reached its desired positions for both horizontal and rotational movements. In this case, the code segment sets the corresponding actuators, H_Actuator and R_Actuator, to false. This action deactivates the actuators, stopping further movement and indicating that the desired movements have been performed successfully. From Listing 2, SAIN derives the following intra-state invariants:

- IntraS3$_1$ (For the Horizontal (H) Actuator Start): *If the "H_Sensor" is within the range [H_Sensor_Min, H_Sensor_Max], then "H_Actuator" is TRUE.*
- IntraS3$_2$ (For the Rotational (R) Actuator Start): *If the "R_Sensor" is within the range [R_Sensor_Min, R_Sensor_Max], then "R_Actuator" is TRUE.*
- IntraS3$_3$ (For the Horizontal (H) Actuator Stop): *If the "H_Sensor" is within the range [H_Target - H_Offset, H_Target + H_Offset], then "H_Actuator" is FALSE.*
- IntraS3$_4$ (For the Rotational (R) Actuator Stop): *If the "R_Sensor" is within the range [R_Target - R_Offset, R_Target + R_Offset], then "R_Actuator" is FALSE.*

5.2.3 Inter-state Invariants

Inter-state invariants are rules that indicate valid state transitions. Each state in an ICS is programmed to transition to another state only when certain conditions are satisfied. If a state changes without meeting such conditions, the corresponding inter-state invariant is violated.

SAIN leverages the state transition conditions extracted from the PDG in Section 5.1.2 to derive the inter-state invariants. In contrast to the intra-state invariants that define the ICS properties/behaviors that must be satisfied within a state, inter-state invariants determine the conditions necessary before the ICS can transition to the next state. For instance, in Listing 2, which shows a code snippet for state S_3 from Figure 3, SAIN derives the inter-state invariant defining the transition condition from state S_3 to S_4.

- InterS3$_1$ (Transition from S3 to S4): *If both "H_Sensor" and "R_Sensor" are within their respective target ranges [H_Target-H_Offset, H_Target+H_Offset] and [R_Target-R_Offset, R_Target+R_Offset] and both "H_Actuator" and "R_Actuator" are FALSE, then li_StepCase = 210.*

Intra-state and inter-state invariants complement each other to support state-aware ICS integrity enforcement. Intra-state invariants pertain to the correctness of operations within a state to maintain ICS integrity, whereas inter-state invariants ensure the legitimacy of transitions between ICS states.

5.3 Invariant Quantification

Invariant quantification involves assigning concrete numerical bounds to variables in an invariant template. Invariant templates extracted from the PLC code as described in Section 5.2 may leave their bounds unresolved. SAIN quantifies them using data traces partitioned by state to ensure the bounds are state-aware and hence tighter compared to those in state-agnostic invariants. To this end, SAIN partitions the entire ICS operation traces into state-specific sub-traces, which are then mined to derive concrete variable value bounds.

The system-level ICS data traces encompass all benign operations of the ICS, under all ICS states. SAIN partitions the data traces according to the state transition conditions (derived from the PLC code). The algorithm to partition system-level data traces into state-specific sub-traces involves traversing the traces and comparing the variable values with the state transition conditions. Particularly, the algorithm detects state transitions by identifying the points where the conditions are met and records the corresponding indices in the data traces. From these detected state transition points, the algorithm then partitions the data traces into multiple sub-traces, each corresponding to a specific state.

SAIN then quantifies the "unresolved" invariant templates with state-specific sub-traces using the data mining tool Quantminer [41]. Incorporating the invariant templates (i.e., invariants with unresolved bounds) and the partitioned sub-traces

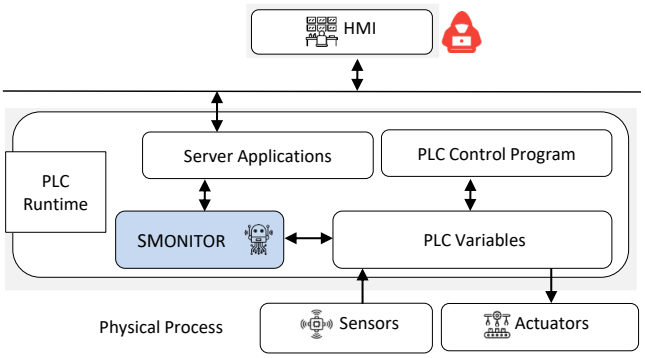


Figure 6: The deployment of SMONITOR in an ICS.

into the data mining process leads to state-aware invariants with concrete, tight PLC variable value bounds. An invariant template is a predicate. For example, a single-variable invariant template may have the form $\# < V < \#$, where V is a variable and # is a numeric value. For a multi-variable invariant, its template defines the relation between sensing, actuation, and configuration variables. The data mining tool then sets the numerical bounds in the invariant templates using the state-specific sub-traces. For instance, the quantified intra-state invariants from Listing 2 are:

- IntraS3$_1$ (For the Horizontal (H) Actuator Start): *If the "H_Sensor" is within the range [0-885], then "H_Actuator" is TRUE.*
- IntraS3$_2$ (For the Rotational (R) Actuator Start): *If the "R_Sensor" is within the range [40-691], then "R_Actuator" is TRUE.*
- IntraS3$_3$ (For the Horizontal (H) Actuator Stop): *If the "H_Sensor" is within the range [890,910], then "H_Actuator" is FALSE.*
- IntraS3$_4$ (For the Rotational (R) Actuator Stop): *If the "R_Sensor" is within the range [695,705], then "R_Actuator" is FALSE.*
- InterS3$_1$ (Common Condition for Both Actuators): *If both "H_Sensor" and "R_Sensor" are within their respective target ranges [890,910] and [695,705] and both "H_Actuator" and "R_Actuator" are FALSE, then li_StepCase = 210.*

5.4 SAIN Monitor

SAIN enforces the extracted state-aware invariants using a monitoring agent (SMONITOR), implemented within the PLC runtime, which checks external accesses to PLC variables. Figure 6 shows the placement of SMONITOR in an ICS, and Figure 7 illustrates the operation steps of SMONITOR.

For a given server application, SMONITOR first performs use-definition chain analysis to identify the functions that can access PLC variables. After identifying these functions, it instruments the target functions to invoke SMONITOR when the function is called. This allows it to check variable update requests against SAIN's state-aware invariants. To achieve this,

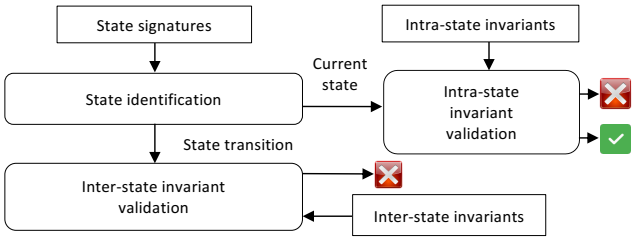


Figure 7: Operation steps of SMONITOR.

Algorithm 3 Run-time ICS State Identification

Input: PLC variables
Output: Current state

1: Load `Signatures` at PLC Runtime startup
2: **for** each `signature` in `Signatures` **do**
3: Compare `signature` with the real-time PLC variables values
4: **if** a `signature` matches the PLC variable values **then**
5: Mark the `signature`'s state as `Current state`
6: **end if**
7: **end for**
8: **return** `Current state`

SMONITOR performs two operations: (1) state identification, and (2) intra-state and inter-state invariant validation.

State Identification. SMONITOR determines the current ICS state by comparing state signatures with current variable values. Algorithm 3 shows the process of identifying an ICS state. State signatures formatted as strings are loaded from the PLC runtime database at the start of the PLC runtime (Line 1). SMONITOR then interprets them through the expression parsing technique [35] and compares them with the real-time PLC variable values (Line 3) to identify the current state of the ICS (Line 5). After identifying the initial state of the ICS, instead of using the state signatures, SMONITOR keeps track of the state transitions to determine the subsequent ICS states. This approach allows it to verify only the specific intra-state and inter-state invariants related to the current state, reducing the number of invariants that need to be checked. Thus, this approach ensures that SMONITOR introduces minimal delays and does not affect the operation of the ICS at runtime.

Intra-State and Inter-State Invariant Validation. For a remote PLC variable update request, SMONITOR leverages the current state to check the intra-state invariants under that state. It first verifies the request against single-variable intra-state invariants. If the request passes this initial check, it then evaluates the variable update request against the multi-variable invariants. For inter-state invariants, SMONITOR checks whether the variable update request causes a state transition by checking whether the updated variable appears in the state transition conditions in the FSM. It then verifies the variable update requests that cause a state transition against the inter-state invariants. If an invariant violation is detected, the monitor denies the variable update request and alerts the operator.

6 Implementation

We implement SAIN in C# with the following components. (1) A static analysis tool that takes the PLC program (ST or SCL) and extracts state identification signatures and intra-/inter-state invariant templates. (2) A data partitioning tool to split system-level data traces into state-specific sub-traces. (3) To quantify invariant templates, we leverage a data mining tool Quantminer [41]. SAIN enhances the data mining process by providing invariant structures and integrating the state-specific sub-traces for mining, allowing us to derive state-aware invariants with tight bounds. (4) A PLC-based monitor, developed in C, compatible with the OpenPLC runtime to check and enforce the state-aware invariants.

7 Evaluation

7.1 Experimental Setup

We evaluate SAIN on two ICS testbeds. The first is the FT manufacturing plant [15], managed by the Unipi PLC [46] equipped with OpenPLC runtime. FT is a suitable evaluation platform as it includes diverse software components, sensors, and actuators with close fidelity to real-world plant operations. The second testbed is the Tennessee Eastman chemical plant simulator [17]. This simulator integrates OpenPLC with an ST application, a Human-Machine Interface (HMI), and the Modbus/TCP protocol. We run SAIN's offline components (i.e., ICS state identification and state-aware invariant generation) on a laptop running Windows 10 (64-bit) and equipped with 16 GB of memory and an Intel(R) Core(TM) i7-8550U processor operating at 3.79 GHz.

Data Trace Collection. We collect data traces (sensing, actuation and configuration variable values) over a 12-day period from ICS plants operating 24/7, through PLC runtime logging mechanisms. For the first ten days, we operated the plants under normal conditions without attacks. We use the traces from the first two days to quantify the invariants and the last eight days to evaluate whether SAIN causes false alarms (i.e., false positives). During the last two days, we perform attacks to evaluate SAIN's attack detection performance. Throughout this period, we constantly log and transfer PLC variable data to a historian server[2].

ICS Attacks Performed. Table 4 presents the 17 attacks we perform on the two ICS testbeds. Our attacks align with our threat model; we exploit the HMI and send malicious variable update requests to the PLC. Our attacks encompass sensor variable manipulation (S), false configuration injection (C), and malicious actuator commands (A). Particularly, in 9 attacks, we manipulate sensor variables; in 11 attacks, we

[2]We make the data traces publicly available to foster future work. https://github.com/purseclab/SAIN.

Table 4: Overview of attacks for SAIN evaluation.

ID	Description	Manipulated* S	A	C	Type†	P
A_1	Start horizontal movement while VGR can just rotate; VGR & HBW collide.		√	√	Intra-State	√
A_2	Rotate the VGR towards the HBW instead of the NFC; the VGR collides with the HBW during NFC operations.			√	Intra-State	√
A_3	Place the MPO robot in a position where only the VGR can work; the VGR and MPO robots collide.	√	√	√	Intra-State	√
A_4	The VGR vacuum turned off during delivery; the workpiece falls at a random location.		√		Intra-State	
A_5	Spoof the VGR rotation sensor to avoid its benign stop; VGR stops at no man's land.	√			Intra-State	√
A_6	VGR relocates to an invalid home position after order completion; a collision occurs for the next order.		√	√	Intra-State	
A_7	Move the MPO robot to an invalid position; safety hazard.	√	√	√	Intra-State	
A_8	Halt the NFC scan operation; the delivery process halts.	√	√	√	Intra-State	
A_9	Send a false signal that the container is available while the VGR waits to drop an item on the conveyor; collision		√		Intra-State	
A_{10}	Continuously manipulate the VGR rotation speed to swing the workpiece; wear and tear of components	√	√	√	Intra-State	
A_{11}	Activate the MPO heating station; safety hazard.	√	√	√	Intra-State	
A_{12}	Rotate VGR to its maximum limit, exceeding the allowable range for normal plant operation; safety hazard.	√	√	√	Intra-State	
A_{13}	Move the HBW robot to a point that is not allowed in normal plant operation; safety hazard.	√	√	√	Intra-State	
A_{14}	Start a new state before finishing the current state; interrupting the planned sequence of activities.			√	Inter-State	
A_{15}	Transition to an unspecified state; stopping the plant operation.			√	Inter-State	
A_{16}	Raise the boiler pressure over the established safety threshold; explosion.			√	Intra-State	
A_{17}	Halt the product development process; decreasing output	√			Intra-State	

*S: sensor, A: actuator, C: configuration, †Intra: Intra-State attack, Inter: inter-state attack. $A_1 - A_{15}$ are attacks against the Fischertechnik manufacturing plant, and $A_{16} - A_{17}$ are attacks against the chemical plant.

manipulate actuator variables; in 13 attacks, we manipulate configuration variables; and in 9 attacks, we manipulate multiple PLC variables (e.g., sensor and actuator variables). These attacks show that stealthy attackers can bypass the loosely bounded invariants and cause damage to the ICS.

We also conduct 4 physics-aware attacks, in which we follow the methodologies outlined in the prior work [19, 28, 47] to minimally manipulate the PLC sensing, configuration, and actuation variables in each scan cycle by closely following the expected physical behavior of the ICS. Our attacks also align with the MITRE ATT&CK techniques for ICS [32], such as Control Manipulation (T831), Unauthorized Command Message (T1047), and Modify Parameter (T817). The MITRE ATT&CK framework for ICS defines adversary tactics, techniques, and procedures used against ICS in the real world. Since our attacks align with the MITRE framework, they can be generalized to other ICS platforms. Our attack vectors are similar to those adopted in prior ICS security efforts [49, 52].

We categorize the attacks performed into two types: intra-state and inter-state. In an intra-state attack, the attacker keeps the ICS in a correct state while manipulating the PLC sensing, configuration, and actuation variables in a way that could result in an unintended action within that current state. In an inter-state attack, an attacker manipulates a PLC program variable to cause either a *premature* or *invalid* state transition within the ICS. A premature state transition occurs when an ICS transitions to the next state before the current state is complete. An invalid transition occurs when an ICS transitions to a state that is not logically sequenced according to the PLC control program's FSM. We implement both types of inter-state attacks to show the generality of SAIN in detecting attacks that cause premature and invalid state transitions.

Research Questions. We evaluate SAIN by focusing on the following research questions:

RQ1 What is the attack detection performance of SAIN?

RQ2 What is the comparison of attack detection performance between SAIN and state-of-the-art methods?

RQ3 How many false positives do SAIN and state-of-the-art methods cause?

RQ4 What is SAIN's state-aware invariant generation time?

RQ5 What is the comparison of the number of validated invariants between SAIN and the state-of-the-art?

RQ6 What is SAIN's runtime performance overhead?

7.2 Effectiveness of SAIN

7.2.1 Attack Detection Performance (RQ1)

SAIN leverages state-aware invariants with tight bounds to detect and mitigate all 17 attacks. These attacks cover intra-state, inter-state, and physics-aware attacks.

In intra-state attacks, the attacker performs an operation that is allowed when considering the overall ICS environment but not in a specific ICS state. For instance, A_1 is an intra-state attack in which the attacker starts the horizontal movement of the VGR. During this attack, the horizontal movement of the VGR remains within the valid system-level bounds but causes the VGR and HBW to collide when it is triggered in a specific ICS state. In the ICS states where this movement is not allowed, SAIN detects this attack as the following invariant is violated: *H_Actuator = False*. As another example, A_{16} is an intra-state attack in which the attacker manipulates the boiler threshold to a level that is prohibited in the overall chemical process. SAIN detects this attack through the single variable invariant *Pressure[55295-55302]*, since the threshold exceeds the limit during the attack.

Table 5: Attack detection comparison.

Attack ID	Feng et al. [14]	VetPLC [52]	SAIN
A_1			√
A_2			√
A_3			√
A_4			√
A_5			√
A_6			√
A_7	√	√	√
A_8		√	√
A_9			√
A_{10}			√
A_{11}			√
A_{12}	√	√	√
A_{13}	√	√	√
A_{14}			√
A_{15}	√	√	√
A_{16}	√	√	√
A_{17}			√

In inter-state attacks, the attacker manipulates PLC variables to follow an unexpected control-flow path. As an example of a premature state transition attack, A_{14} is an inter-state attack in which the attacker initiates a state transition before the current state, S_3, is completed. As an example of an invalid state transition attack, A_{15} is an inter-state attack in which the attacker initiates a state transition to S_{10}, which is an invalid transition from S_6. Since SAIN derives inter-state invariants from state transition conditions that need to be fulfilled before the transitions to the next state, SAIN detects these attacks.

We present two case studies in Section 7.4 to further detail how SAIN detects an intra-state and an inter-state attack.

In physics-aware attacks, the attacker manipulates the PLC variables with minimal deviation from the benign values. For example, in the attack A_5, the attacker manipulates the VGR's rotational movement sensing variable (*R_Sensor*) close to its benign values, eventually causing a collision between the VGR and HBW. SAIN defines tight upper and lower bounds for the VGR's rotational movement sensing variable for each state (S) as follows.

$$0 \leq S_1 \leq 30 < S_2 \leq 40 < S_3 \leq 705 < S_4, S_5 \leq 720 < S_6 \leq 5350$$

These tight bounds are sensitive even to stealthy manipulation of variable values. Specifically, SAIN detects this attack through the invariant bounds *R_Sensor[0-30]* for state S_1, as the variable's manipulations exceed the upper bounds. The attacker increases the *R_Sensor* value in the benign range of 1.5 to 2.1 at each scan cycle, and SAIN detects this attack when it becomes 30.1 before any damage to the plant.

As another physics-aware attack example, in A_1, the attacker stealthily manipulates the actuator and configuration variables to initiate the VGR's horizontal movement during state S_6. Particularly, the attacker increases the *H_Target* variable from its benign value 900 by 0.1-0.3 in each scan cycle until it reaches 960 and sets the *H_Actuator* variable to True, to cause a collision between the VGR and HBW. For state S_6, SAIN's state-aware invariants specify that *H_Target* must be within the [890-910] range and *H_Actuator* must be False and, therefore, SAIN detects this attack.

Table 6: False positive comparison.

Testbed	Feng et al. [14]	VetPLC [52]	SAIN
FT	51%	25%	2%
CP	2%	1%	1%

7.2.2 Attack Detection Comparison (RQ2)

We compare SAIN with two state-of-the-art ICS invariant generation methods, Feng et al. and VetPLC because they demonstrate better performance in generating comprehensive ICS invariants compared to prior approaches. We reimplemented both of these approaches. To ensure the correctness of our implementations, we extracted invariants from data collected from our testbeds and integrated their invariants within the PLC runtime. We then evaluated their ability to detect ICS attacks in which the attacker performs an operation that is not allowed during the entire ICS operation. Our results indicate that they detect such attacks with high accuracy, consistent with the results reported in their papers.

Table 5 presents the attack detection comparison between SAIN and the state-of-the-art methods. As detailed in Section 7.2.1, SAIN's state-aware invariants detect and mitigate all 17 intra-state and inter-state attacks. In contrast, Feng et al. detects 5/17 (29%) attacks, while VetPLC detects 6/17 (35%) attacks. Both Feng et al. and VetPLC prevent attacks where the invariants derived from system-level data traces are violated. VetPLC, which extracts temporal invariants, is more effective when the attack violates the timing constraints between the PLC variables.

7.2.3 SAIN's False Positive Performance (RQ3)

False positives occur when benign variable updates are flagged as attacks, causing unnecessary alerts. Table 6 presents the false positive rates of SAIN and state-of-the-art invariant derivation methods. For the FT manufacturing plant Feng et al. generates 51% false positives, VetPLC generates 25% false positives, and SAIN generates 2% false positives. For the chemical plant, Feng et al. generates 2% false positives, VetPLC, and SAIN generates 1% false positives.

We observe that false positives in Feng et al. and VetPLC are primarily caused by either the incorrect definition of invariant structures or the underestimation of the invariant data bounds, with the majority resulting from the former issue. For instance, Feng et al. derives its invariant structures solely from data traces, causing it to derive invariants that represent the *correlations* between PLC variables. However, we observe that, especially in the FT manufacturing plant, the correlations between the PLC variables vary based on ICS

Table 7: Average time (hh:mm:ss) to analyze the PLC program of Fischertechnik (FT) and chemical (CP) plants to extract state-aware invariants.

ICS Testbed	AST Construction	FSM Extraction	State Signatures	Invariant Templates	Trace Partitioning	State-aware Invariant Quantification
FT	00:00:17	00:01:30	00:00:45	00:02:25	02:10:00	00:17:00
CP	00:00:01	00:00:09	00:00:04	00:00:11	00:24:00	00:05:00

states. Such variations in correlations across ICS states cause Feng et al. to have high false positives in certain states. For example, FT manufacturing plant traces show a correlation between the VGR horizontal sensor, vertical sensor, and vertical actuator. Feng et al. derives an invariant based on this correlation. However, in state S_8, only the vertical sensor is correlated with the vertical actuator. This causes Feng et al.'s invariant to generate false positives in state S_8.

Additionally, state-agnostic analysis generates conflicting invariants when the occurrence of events varies across different states. For instance, if the order of events is sequential in one state but parallel in another state, an invariant designed for sequential events generates false positives in the other state. In contrast, SAIN does not generate false positives due to incorrect invariant structures because it derives state-aware invariant structures directly from the PLC code that controls the ICS process. Thus, it only generates false positives when it derives invariants with overly tight bounds.

7.3 Performance Evaluation

7.3.1 Invariant Generation Overhead (RQ4)

Table 7 presents the processing time for each stage of SAIN's state-aware invariant generation. We found that SAIN's processing overhead increases with the number of ICS states and the amount of data traces used for invariant quantification. For instance, FSM extraction takes 90 seconds for the FT manufacturing plant and 9 seconds for the chemical plant. As another example, partitioning the data traces takes 2 hours and 25 mins for FT and 17 mins for the chemical plant.

7.3.2 Number of Validated Invariants (RQ5)

Table 8 shows a comparison between SAIN and state-of-the-art methods in terms of the number of invariants validated. For instance, the "Average" column indicates the average number of invariants that are checked in response to a variable update request. SMONITOR, upon receiving a variable update request, only validates the specific set of invariants that are relevant to the updated variable. On average, it validates 5 invariants for the FT manufacturing plant and 3 for the chemical plant. In comparison, Feng et al. validates 134 invariants for the FT manufacturing plant and 11 for the chemical plant, while VetPLC validates 35 invariants for the FT manufacturing plant and 7 for the chemical plant. SAIN validates a smaller number of invariants because it selectively validates invariants according to the current ICS state. Since each ICS state is designed to execute a limited set of operations, it reduces the number of invariant validations.

Table 8: Number of invariants validated for variable updates in the Fischertechnik (FT) and Chemical (CP) plants.

Testbed	Method	Min.	Average	Max.
FT	Feng et al.	43	134	315
	VETPLC	11	35	83
	SAIN	1	5	11
CP	Feng et al.	7	11	27
	VETPLC	3	7	12
	SAIN	1	3	5

Table 9: Scan cycle time (ms) of PLCs controlling Fischertechnik (FT) and Chemical (CP) plants.

ICS Testbed	Normal Cycle Time	Time Limit	Method	Min.	Average	Max.
FT	36	150	Feng et al.	44.46	57.48	84.3
			VETPLC	38.42	43.7	49.26
			SAIN	36.22	37.1	38.42
CP	5	25	Feng et al.	6.54	7.42	10.94
			VETPLC	5.66	6.54	7.64
			SAIN	5.22	5.56	6.10

7.3.3 Runtime Performance Overhead (RQ6)

To assess the impact of SAIN's state-aware monitoring on the Fischertechnik and chemical plants, we measured the variations in PLC scan cycle times in two scenarios.

- Normal Operation: When SMONITOR is inactive.
- SMONITOR: When SMONITOR remains active and checks invariants whenever the PLC receives a variable update request from an external source.

To track the scan cycle time in the PLC runtime, we create variables to record the start and end times of each cycle. We use the CLOCK_MONOTONIC function for precise timing. At the beginning of each cycle, we note the timestamp, let the PLC perform its operations, and then mark the timestamp again at the end. The difference between these two timestamps provides us with the cycle's duration.

Table 9 shows the PLC scan cycle time in both plants during their normal operation, with an active SMONITOR, and the maximum allowed execution time required for the safe operation of the plants. We derive the maximum allowed execution time for each plant from their PLC configurations and developer documentations. Our measurements show that the scan cycle time with SMONITOR for the FT manufacturing plant is 37.1 ms, which slightly differs from the normal operation of 36 ms. The normal operation and SMONITOR monitor scan cycle times for the chemical plant are less than 6 ms. The overall maximum allowed execution time is 150 ms for the FT manufacturing plant and 25 ms for the chemical plant. Therefore, our experiments show that SMONITOR's

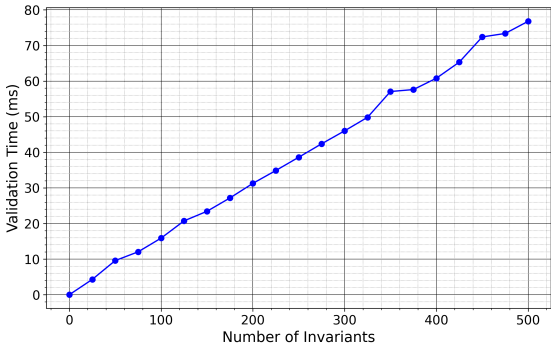


Figure 8: Validation Time vs. No. of Invariants.

runtime performance overhead is minimal and complies with the maximum allowed scan cycle time limits.

To assess the real-world impact of SAIN's runtime overhead, we evaluate the change in invariant validation overhead as the number of invariants increases. Figure 8 shows a linear relationship between the number of invariants and the invariant validation time. Considering that SAIN requires the validation of only a small number of invariants, it remains highly suitable for real-world ICS. Even with 500 invariants, SAIN's invariant validation overhead is below the maximum execution time allowed in the FT manufacturing plant.

7.4 Case Studies

We present two case studies, an intra-state and an inter-state attack, to demonstrate how SAIN's state-aware invariants with tight bounds detect these attacks.

7.4.1 Case Study 1: Intra-State Attack

In attack A_{10}, the attacker constantly increases and decreases the vertical movement speed of the VGR. This speed manipulation over time causes the workpiece hanging from the VGR to swing while the VGR moves.

Stealthiness. Unfortunately, both Feng et al. and VetPLC cannot detect this attack because the attacker remains within the allowed system-level bound ([500-800]) for the VGR's vertical speed, and does not violate the timing between events.

The VGR's vertical speed in normal plant operation varies based on the ICS states. The PLC adjusts the speed to match the specific needs of the ICS process. In some states, the speed increases, while in others it decreases to ensure synchronization within the ICS. Feng et al.'s state-agnostic invariants merge such speed variations into a generic bound, and do not consider that the speed bounds could change when the ICS state changes. Therefore, it fails to detect this attack.

In total, it takes the VGR 10 seconds to reach the HBW with the workpiece during the benign operation of FT. While the attacker modifies the VGR speed, they employ both fast and slow speeds to ensure that the 10-second timing is not violated while the workpiece swings and falls. This ensures that the attack is stealthy against timing-based invariants (e.g., those extracted by VetPLC).

SAIN Detection. To detect this attack, SAIN leverages a tightly bounded invariant regarding the VGR's vertical speed while it carries a workpiece to the HBW.

7.4.2 Case Study 2: Inter-State Attack

In the FT manufacturing plant, the PLC program uses a variable *li_StepCase* to determine which state should be executed. In attack A_{14}, an attacker prematurely sets *li_StepCase* to "210", initiating the next state before the current one is finished. This causes the VGR robot to begin its vertical movement while still completing its horizontal and rotational movements, leading to a collision with the HBW station.

Stealthiness. Unfortunately, both Feng et al. and VetPLC cannot detect this attack due to their state-agnostic invariants. During normal plant operation, the variable *li_StepCase* is set to "210" multiple times in different ICS states. Consequently, the state-agnostic invariant merges the constraints into a single invariant that sets the *li_StepCase* variable to "210". This enables the attacker to manipulate the *li_StepCase* variable before it was supposed to change in its intended state without being detected by the invariants of these works.

SAIN Detection. SAIN detects this attack through a state-aware invariant that ensures *li_StepCase* changes only when the state transition condition is met. As this attack causes the premature state transition, SAIN detects it.

8 Limitations And Discussions

False Positives. SAIN reduces false positive rates compared to state-of-the-art invariant generation methods, yet it still faces the challenge of completely eliminating false positives. As discussed in Section 7.2.3, SAIN false positives occur mainly due to overly tight bounds. This could be addressed by ensuring that the data traces used to quantify invariants are complete. However, collecting complete traces in ICS environments might be challenging due to different environmental conditions and the complexity of capturing diverse ICS processes. We note that this limitation exists in all data-driven approaches in ICS security [14, 30], which may be mitigated by other forms of ICS knowledge/invariants (e.g., formal or natural language specifications by vendors).

Adapting to Code Updates. If the PLC code is updated, SAIN's invariants must be regenerated to adapt to the new control logic. However, in ICS, code updates are infrequent for two main reasons. First, the code running on PLCs is typically stable and well-tested before deployment. Second, the process of updating the PLC code often involves downtime, which could be costly in ICS [7, 42].

Table 10: Comparison of SAIN with relevant systems.

	Network IDS [24]	C2 [29]	Urbina et al. [47]	Orpheus [9]	Aoudi et al. [4]	Ghaeini et al. [19]	Feng et al. [14]	VetPLC [52]	PLC Sleuth [48]	SAIN
System Type										
State-aware	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓
Invariant-Based	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓
Physics-Based	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗
ICS Process Modeling										
Data Traces	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
Physics-Based Models	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗
PLC Code Analysis	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓
Automated	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
Documentation	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
State Identification Method										
Kalman Filter	N/A	N/A	✗	N/A	N/A	✓	N/A	N/A	N/A	✗
Linear Dynamical Statespace	N/A	N/A	✓	N/A	N/A	✗	N/A	N/A	N/A	✗
FSM Transitions	N/A	N/A	✗	N/A	N/A	✗	N/A	N/A	N/A	✓
Invariant Types										
Event Based	✓	✓	N/A	✓	N/A	N/A	✓	✓	✓	✓
Temporal	✗	✓	N/A	✗	N/A	N/A	✗	✓	✗	✗
State-aware	✗	✗	N/A	✗	N/A	N/A	✗	✗	✗	✓
Data Mining and Monitoring System										
State-aware Data Mining	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
PLC Based Monitor	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Potential Race Conditions. Race conditions in PLCs occur when variable updates depend on the timing or order of execution of multiple code paths that manipulate shared variables. SAIN does not introduce any race conditions into the PLC. Particularly, SAIN's invariants in SMONITOR are employed in code blocks intended for external access to PLC variables and do not involve any write operations on the variables; therefore, it avoids introducing new race conditions.

Potential False Negatives. SAIN validates PLC invariants when it receives external variable update requests via the HMI. An attacker who has physical access to ICS devices such as sensors and actuators can spoof physical sensor and actuator values and bypass SAIN's detection. This limitation is common in ICS security approaches that operate on the PLC or HMI [21, 25]. In future work, we will extend the SAIN runtime monitor to validate sensor-to-PLC and PLC-to-actuator communication with state-aware invariants, under a more advanced threat model.

9 Related Work

We compare SAIN with the prior work in Table 10. For ICS security, several static analysis approaches have been proposed to analyze the PLC code and identify vulnerabilities that may lead to violations of predefined security policies [6, 30, 37]. Yet, the lack of runtime information poses challenges for these approaches, resulting in overapproximations that cause wrongly identified vulnerabilities and underapproximations that cause missed vulnerabilities. For instance, when an attacker manipulates runtime temperature sensor readings, static analysis methods may not detect this attack because they lack real-time sensor data. To address the limitations of static analysis, prior work has explored extracting invariants by mining benign execution traces of a plant through statistical learning techniques [4, 14, 23, 25] and machine learning models [9, 26]. These state-agnostic invariants are derived solely from data traces without explicitly considering the specific states defined in the PLC code. This limitation makes them vulnerable to stealthy evasion attacks, as demonstrated in Section 7.2.

Recent works have proposed temporal invariant mining approaches that capture the temporal dependencies between system events and use event timings as safety requirements [3, 34, 52, 53]. These methods prove highly effective in detecting attacks in which adversaries manipulate the order or timing of events. Yet, some attacks may not involve altering event order or timing. For instance, an attacker can manipulate a robot's speed in such a way that it arrives at its destination on time, but the speed changes cause the robot to drop its payload.

Prior research has also proposed state-aware physics-based attack detection (PBAD) techniques for ICS [19, 47]. These techniques first extract ICS states manually or from data traces. They then model the state-aware physical behavior of the ICS with subspace models and compute runtime residuals, which indicate the deviations between model estimations and real sensor readings. PBAD techniques are highly effective against ICS attacks when the residuals exceed a threshold. Unfortunately, they are less effective against stealthy physics-aware attacks that manipulate the actuator and sensor variables *within* the residuals [20, 36, 44]. In contrast, SAIN derives state-aware invariants for ICS through the combination of program analysis and trace data mining. To achieve this, SAIN extracts an FSM representing the states and state transitions in the ICS through PLC program analysis, and it splits the ICS data traces into state-specific sub-traces and quantifies the invariants with concrete and tighter bounds. Therefore, SAIN complements PBAD techniques by enforcing intra-state invariants with tighter bounds on PLC variables and inter-state invariants that check the validity of state transitions.

10 Conclusions

In this paper, we first show that prior work generates invariants with loose bounds, allowing an attacker to conduct stealthy evasion attacks and avoid detection. We then introduce SAIN, an automated method for generating state-aware ICS invariants with tight bounds that detect such stealthy evasion attacks. SAIN reasons about and enforces the state-aware invariants through a novel PLC-based runtime monitor that detects and mitigates ICS attacks with minimal impact on PLC scan cycle timings. We evaluate SAIN on both Fischertechnik and chemical plants to show its effectiveness in detecting various attacks, including PLC variable manipulation and physics-aware attacks, with minimal runtime performance overhead.

Acknowledgments

We thank the anonymous shepherd and the reviewers for their timely, valuable comments and suggestions. This work was supported in part by NSF under Grants IIS-2229876, CNS-2144645, by ONR under Grant N00014-20-1-2128, and by a grant from Cisco. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Wael Alsabbagh and Peter Langendöerfer. A new injection threat on s7-1500 PLCs-disrupting the physical process offline. *IEEE Open Journal of the Industrial Electronics Society*, 2022.

[2] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio De Souza, and Thelma Virginia Rodrigues. OpenPLC: An open source alternative to automation. In *IEEE Global Humanitarian Technology Conference*, 2014.

[3] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *ACM International Symposium on the Foundations of Software Engineering*, 2012.

[4] Wissam Aoudi, Mikel Iturbe, and Magnus Almgren. Truth will out: Departure-based process-level detection of stealthy attacks on control systems. In *ACM Conference on Computer and Communications Security*, 2018.

[5] Dillon Beresford. Exploiting siemens simatic s7 PLCs. *Black Hat USA*, 2011.

[6] Sebastian Biallas. *Verification of programmable logic controller code using model checking and static analysis*. PhD thesis, RWTH Aachen University, 2016.

[7] Alvaro A Cárdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. Attacks against process control systems: risk assessment, detection, and response. In *ACM Symposium on Information, Computer and Communications Security*, 2011.

[8] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 2016.

[9] Long Cheng, Ke Tian, and Danfeng Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Annual Computer Security Applications Conference*, 2017.

[10] Cyberattack on oldsmar water treatment facility. https://www.dragos.com/blog/industry-news/recommendations-following-the-oldsmar-water-\treatment-facility-cyber-attack/, 2023. [Online; accessed March 15, 2024].

[11] Markus Dahlmanns, Johannes Lohmöller, Ina Berenice Fink, Jan Pennekamp, Klaus Wehrle, and Martin Henze. Easing the conscience with opc ua: An internet-wide study on insecure deployments. In *ACM Internet Measurement Conference (IMC)*, 2020.

[12] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, 1999.

[13] Nicolas Falliere, Liam O Murchu, Eric Chien, et al. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 2011.

[14] Cheng Feng, Venkata Reddy Palleti, Aditya Mathur, and Deeph Chana. A systematic framework to generate invariants for anomaly detection in industrial control systems. In *NDSS*, 2019.

[15] FischerTechnik. Fischertechnik plant manual. "https://www.fischertechnik.de/-/media/fischertechnik/fite/service/elearning/simulieren/lernfabrik-4-0-24v/fabrik_2020_deutsch_s7-1500_en_korrigiert_final.ashx", 2022. [Online; accessed March 15, 2024].

[16] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM Symposium on Principles of Programming Languages*, 2004.

[17] David Formby, Milad Rad, and Raheem Beyah. Lowering the barriers to industrial control system security with GRFICS. In *USENIX Workshop on Advances in Security Education (ASE)*, 2018.

[18] Luis Garcia, Stefan Mitsch, and André Platzer. Hy-PLC: Hybrid programmable logic controller program translation for verification. In *ACM/IEEE International Conference on Cyber-physical Systems*, 2019.

[19] Hamid Reza Ghaeini, Daniele Antonioli, Ferdinand Brasser, Ahmad-Reza Sadeghi, and Nils Ole Tippenhauer. State-aware anomaly detection for industrial control systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018.

[20] Jairo Giraldo, David Urbina, Alvaro Cardenas, Junia Valente, Mustafa Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candell. A survey of physics-based attack detection in cyber-physical systems. *ACM Computing Surveys (CSUR)*, 2018.

[21] Benjamin Green, Richard Derbyshire, Marina Krotofil, William Knowles, Daniel Prince, and Neeraj Suri. Pcaad:

Towards automated determination and exploitation of industrial systems. *Computers & Security*, 2021.

[22] Peter Gsellmann, Martin Melik-Merkumians, and Georg Schitter. Comparison of code measures of iec 61131–3 and 61499 standards for typical automation applications. In *23rd International Conference on Emerging Technologies and Factory Automation*, 2018.

[23] Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[24] Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H Hartel. Through the eye of the PLC: semantic security monitoring for industrial processes. In *Annual Computer Security Applications Conference*, 2014.

[25] Moses Ike, Kandy Phan, Keaton Sadoski, Romuald Valme, and Wenke Lee. SCAPHY: Detecting modern ICS attacks by correlating behaviors in scada and physical. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[26] Jun Inoue, Yoriyuki Yamagata, Yuqi Chen, Christopher M Poskitt, and Jun Sun. Anomaly detection for a water treatment system using unsupervised machine learning. In *IEEE International Conference on Data Mining Workshops (ICDMW)*, 2017.

[27] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis*, 2013.

[28] Ruggero Lanotte, Massimo Merro, Andrei Munteanu, and Luca Vigan. A formal approach to physics-based attacks in cyber-physical systems. *ACM Transactions on Privacy and Security (TOPS)*, 2020.

[29] Stephen McLaughlin. Blocking unsafe behaviors in control systems through static and dynamic policy enforcement. In *Annual Design Automation Conference*, 2015.

[30] Stephen E McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. A trusted safety verifier for process controller code. In *NDSS*, 2014.

[31] Thomas Miller, Alexander Staves, Sam Maesschalck, Miriam Sturdee, and Benjamin Green. Looking back to look forward: Lessons learnt from cyber-attacks on industrial control systems. *International Journal of Critical Infrastructure Protection*, 2021.

[32] Mitre. ICS attack techniques. https://attack.mitre.org/techniques/ics/, 2023. [Online; accessed March 15, 2024].

[33] Efrén López Morales, Ulysse Planta, Carlos Rubio-Medrano, Ali Abbasi, and Alvaro A Cardenas. Sok: Security of programmable logic controllers. In *USENIX Security*, 2024.

[34] Muslum Ozgur Ozmen, Xuansong Li, Andrew Chu, Z. Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. Discovering physical interaction vulnerabilities in IoT deployments. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[35] Arash Partow. C++ mathematical expression toolkit library (exprtk). "https://github.com/ArashPartow/exprtk", 2000. [Online; accessed March 15, 2024].

[36] Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. Attack detection and identification in cyber-physical systems. *IEEE Transactions on Automatic Control*, 2013.

[37] Prashant Hari Narayan Rajput, Constantine Doumanidis, and Michail Maniatakos. ICSPatch: Automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs. In *USENIX Security*, 2023.

[38] Ransomware attack leads to shutdown of major u.s. pipeline system. https://www.washingtonpost.com/business/2021/05/08/cyber-attack-colonial-pipeline/, 2023. [Online; accessed March 20, 2024].

[39] Krzysztof Sacha. Automatic code generation for PLC controllers. In *International Conference on Computer Safety, Reliability, and Security*, 2005.

[40] Roberto Sala, Fabiana Pirola, and Giuditta Pezzotta. On the development of the digital shadow of the fischertechnik training factory industry 4.0: An educational perspective. *Procedia Computer Science*, 2023.

[41] Ansaf Salleb-Aouissi, Christel Vrain, Cyril Nortet, Xiangrong Kong, Vivek Rathod, and Daniel Cassard. Quantminer for mining quantitative association rules. *Journal of Machine Learning Research*, 2013.

[42] Abraham Serhane, Mohamad Raad, Raad Raad, and Willy Susilo. Programmable logic controllers based systems (PLC-BS): Vulnerabilities and threats. *SN Applied Sciences*, 2019.

[43] W32.stuxnet dossier. https://www.wired.com/images_blogs/threatlevel/2011/02/Symantec-Stuxnet-Update-Feb2011.pdf, 2023. [Online; accessed March 20, 2024].

[44] André Teixeira, Daniel Pérez, Henrik Sandberg, and Karl Henrik Johansson. Attack models and scenarios for networked control systems. In *International Conference on High Confidence Networked Systems*, 2012.

[45] Devinder Thapa, Chang Mok Park, Sang C Park, and Gi-Nam Wang. Auto-generation of iec standard PLC code using t-mpsg. *International Journal of Control, Automation and Systems*, 2009.

[46] Unipi PLC. "https://www.unipi.technology/unipi-neuron-l203-rpi3-p29", 2022. [Online; accessed Mar 7, 2024].

[47] David I Urbina, Jairo A Giraldo, Alvaro A Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *ACM Conference on Computer and Communications Security*, 2016.

[48] Zeyu Yang, Liang He, Peng Cheng, Jiming Chen, David KY Yau, and Linkang Du. PLC-Sleuth: Detecting and localizing PLC intrusions using control invariants. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[49] Zeyu Yang, Liang He, Hua Yu, Chengcheng Zhao, Peng Cheng, and Jiming Chen. Detecting PLC intrusions using control invariants. *IEEE Internet of Things Journal*, 2022.

[50] Alberto Zanutto, K Follis, J Busby, Awais Rashid, et al. The shadow warriors: In the no man's land between industrial control systems and enterprise it systems. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[51] Maximilian Zarte, Jeffrey Wermann, Philipp Heeren, and Agnes Pechmann. Concept, challenges, and learning benefits developing an industry 4.0 learning factory with student projects. In *IEEE International Conference on Industrial Informatics (INDIN)*, 2019.

[52] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James Moyne, and Mao. Towards automated safety vetting of PLC code in real-world plants. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[53] Qingzhao Zhang, Xiao Zhu, Mu Zhang, and Z Morley Mao. Automated runtime mitigation for misconfiguration vulnerabilities in industrial control systems. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022.

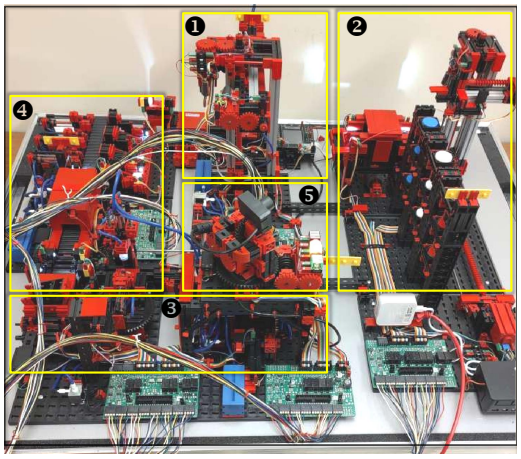


Figure 9: The components of the FT manufacturing plant: ❶ Vacuum Gripper Robot (VGR), ❷ High-bay Warehouse (HBW), ❸ Multi-processing Station (MPO), ❹ Sorting Line (SLD), and ❺ Environmental Station (SSC).

[54] Vladimir E Zyubin, Andrei S Rozov, Igor S Anureev, Natalia O Garanina, and Valeriy Vyatkin. post: A process-oriented extension of the iec 61131-3 structured text language. *IEEE Access*, 2022.

Appendix

A Fischertechnik (FT) Manufacturing Plant

The Fischertechnik (FT) Manufacturing Plant [40] in Figure 9 performs a number of PLC-controlled manufacturing processes. It uses the Siemens S7-1500 PLC and SCL programming language. The same testbed has already been used in other ICS security research efforts [1, 40, 51]. It consists of typical factory components such as a storage and retrieval station, a vacuum gripper robot (VGR), a high-bay warehouse (HBW), a multi-processing station with an oven (MPO), a sorting line with color detection (SLD), an environmental sensor, and a swiveling camera (SSC).

The factory operation is divided into two modes: (1) *Delivery* - A raw workpiece is placed in the input area. When the sensor detects the workpiece, it triggers the VGR to collect it and move it to the NFC reader, where its identification number is read. The VGR then transports the piece to the HBW storage site. (2) *Order* - Users place an order through the dashboard by choosing the desired workpiece. The HBW withdraws the workpiece and positions it for VGR. The VGR transfers the workpiece to the MPO, where it is processed for delivery and finally moved to the delivery location.