ADIS: Detecting and Identifying Manipulated PLC Program Variables Using State-Aware Dependency Graph

Zeyu Yang, Liang He, Yujiao Hu, Peng Cheng, Jiming Chen, and Jianying Zhou

***Abstract*—The increasing network integration of industrial control systems amplifies the risk of cyberattacks on Programmable Logic Controllers (PLCs). In particular, the weak authentication of industrial communication protocols makes PLC program variables vulnerable to manipulation. Current defensive methods cannot reliably identify manipulated variables, even after PLC program manipulations have been detected. To bridge this gap, we present ADIS, a cross-domain Attack Detection and Identification System designed to detect and identify manipulated PLC program variables. Building on a novel state-aware graph representation of the PLC program, ADIS detects variable manipulations by comparing SCADA monitoring data with the control logic defined by the PLC program. ADIS further identifies suspiciously manipulated program variables by excluding cascading failures from the detected anomalies and tracking suspicious variables based on the edges of the state-aware dependency graph. We have implemented and evaluated ADIS on two platforms. The results demonstrate that ADIS detects attacks with a true positive rate exceeding 99% and a false positive rate of less than 0.04‰. Furthermore, it successfully identifies manipulated program variables with up to a 71.3% reduction in suspicious variables compared to a baseline method.**

***Index Terms*—Programmable Logic Controller, Attack Detection, Attack Identification, State-Aware Dependency**

I. Introduction

Programmable Logic Controllers (PLCs) drive the operation of Industrial Control Systems (ICSs) based on the control logic defined in the embedded PLC program. A nearly universal practice in defining PLC control logic is mapping the states of physical devices to the values of specific program variables. Ensuring the correct values of these program variables is crucial for the proper operation of ICSs.

This work was supported in part by the National Natural Science Foundation of China under Grant 62303411 and 62293510/62293511, in part by the National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure: Phase II" (Award No: NRF-NCR25-NSOE05-0001), and in part by the Key Research and Development Program of Zhejiang Province (Grant No: 2025C01061). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. *(Corresponding author: Peng Cheng.)*

Zeyu Yang and Jianying Zhou are with iTrust, Singapore University of Technology and Design, Singapore (e-mail: zeyu_yang@sutd.edu.sg, jianying_zhou@sutd.edu.sg).

Liang He is with the School of Computing, the University of Nebraska-Lincoln, Lincoln, NE, 68588 USA (e-mail: lhe20@unl.edu).

Yujiao Hu, Peng Cheng and Jiming Chen are with the College of Control Science and Engineering, Zhejiang University, Hangzhou, China (e-mail: yujiaohu@zju.edu.cn; lunarheart@zju.edu.cn, jmchen@ieee.org).

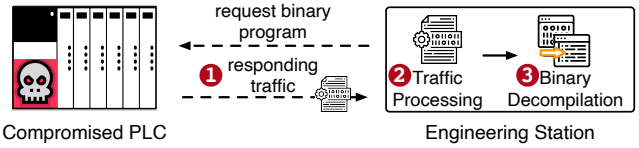


Fig. 1. The typical procedure of obtaining the source code of a manipulated program from compromised PLCs.

The correctness of program variable values has become increasingly jeopardized due to the heightened cyber risks associated with PLCs [1], [2], [3], [4], [5]. As a few examples,

- The program variable values of a (suspected to be) Siemens S7-400 PLC were manipulated to maliciously dump molten steel, causing a massive fire in 2022 [6].
- The program variable value specifying the chlorine content was manipulated to disturb water treatment, threatening public health in 2020 [7].
- The program variable values associated with safety actuators were perturbed by the Triton malware, causing the shutdown of a petrochemical facility in 2019 [8].
- The program variable value defining the converter frequency was periodically manipulated by Stuxnet, damaging hundreds of centrifuges in 2010 [9].

All these cyber incidents were carried out by manipulating the values of PLC program variables, achieved either by injecting malicious program code through the industrial control network or by sending network requests to remotely modify program variables. Although these variable manipulations were eventually detected through the resulting physical consequences, the specific program variables being manipulated (i.e., the root causes) were not identified until post-incident investigations were completed. Until then, the entire system had to remain shut down to prevent further damage.

System operators would have greater flexibility in restoring the PLC's control logic and mitigating the negative impact of cyber incidents if their root causes could be swiftly identified. For instance, in the cyber attack against a Florida water plant, identifying that the program variable controlling the NaOH content was being manipulated allowed the operator to promptly restore the normal NaOH setting, preventing any physical consequences [10].

Identifying manipulated PLC program variables is a challenging task, typically carried out by the following four steps: (i) requesting the executed binary program from the compromised PLC and collecting the response traffic, (ii) extracting the binary program from the received response traffic, (iii) decompiling the binary program into source code, and (iv) comparing the decompiled source code with the original PLC

program source. However, this approach becomes ineffective if the variable manipulation is achieved by writing variable values directly through network requests, as these writing requests do not alter the executed binary program [11], [12], [13]. Additionally, even when variable manipulation is carried out by injecting malicious binary code, attackers can undermine the trustworthiness of the retrieved binary program or render its decompilation impossible through the following methods.

- *Manipulate the responding network traffic.* Attackers have successfully replaced the responding traffic of binary program requests (as shown in step ❶ of Fig. 1) with the legitimate one by exploiting the authentication weakness in PLC communication [14].
- *Manipulate the traffic processing.* The Stuxnet malware successfully replaced the binary program extracted from the received traffic (as shown in step ❷ of Fig. 1) with the legitimate one using a crafted .dll file for the PLC programming software [9].
- *Disable the decompilation of extracted binary program.* Obfuscating the malicious binary program has prevented forensic tools (e.g., programming software) from decompiling the binary program to source code (step ❸ of Fig. 1) [15], [16].

This paper presents a novel cross-domain Attack Detection and Identification System, called ADIS, against PLC program variable manipulations. Unlike the above-explained conventional approaches that rely solely on code domain analysis to identify the manipulated variables, ADIS jointly analyzes the program code and the monitoring data collected by the Supervisory Control and Data Acquisition (SCADA) system, leveraging the fact that SCADA data reflects the same control logic defined by the program code. Specifically, ADIS detects manipulated program variables by verifying whether the SCADA data aligns with the expected control logic. When inconsistencies are detected, ADIS further identifies the manipulated program variables by mapping the abnormal data variables to their corresponding program variables.

Although seemingly straightforward, implementing the above design idea is challenging due to the complex interactions among individual modules of an ICS. First, manipulating a single program variable can result in multiple abnormal data variables because of their cascading behaviors. When mapped back to the code, these abnormalities may implicate multiple program variables as suspicious. Narrowing this set of suspicious program variables — ideally to only the truly manipulated ones — is critical for effective root cause analysis. Additionally, it is common for SCADA systems to record only a subset of program variables, meaning that the manipulated variables may not be recorded in the data domain. As a result, the set of suspicious variables may include only those dependent on the manipulated variables, rather than the manipulated variables themselves. Although it is possible to extend the suspicious set to include unrecorded (and potentially manipulated) program variables by tracking the dependencies among variables [17], such an extension is likely to incorporate other legitimate variables into the suspicious set. As a result, this inclusion introduces additional uncertainty in root cause analysis, making the precise identification of manipulated variables even more challenging.

ADIS addresses these challenges by capturing the interactions among system modules through a novel *state-aware dependency graph*. This graph is used to identify potentially cascading abnormal variables, thereby reducing the set of suspicious variables. Additionally, the graph helps ADIS verify the legitimacy of program variables on which suspicious variables depend but are not recorded by SCADA, facilitating the identification of manipulated variables even if they are not included in the suspicious set. It is important to note that the state-awareness of ADIS's dependency graph enables the precise description of program variables' dependencies that exist only under specific conditions (e.g., those defined with *if* statements). This state-awareness is a key differentiator of ADIS compared to the existing state-agnostic dependency graphs for the PLC program (e.g., the data flow graph [17], [18], [19]).

We have evaluated ADIS on an elevator control system and an ethanol distillation system. The experimental results show ADIS to detect program variable manipulation with an average true positive rate of above 99% and false positive rate of below 0.04‰. More importantly, ADIS successfully identifies all manipulated program variables for the detected PLC program manipulations and reduces the suspicious variables by up to 71.3% compared to a baseline method proposed in [17].

II. Related Work

In this section, we review the literature on detecting and identifying manipulated PLC program variables, highlighting the gap that motivates the design of ADIS.

Detecting Manipulated PLC Program Variables. Various methods have been proposed to detect PLC program manipulations by characterizing the runtime behaviors of program execution and comparing them with their expected norms.

Code-domain detectors verify whether the execution of PLC program satisfies the specified ICS constraints (e.g., safety conditions). TSV [20] models the program as a symbolic execution graph and checks the graph against ICS constraints defined by linear temporal logic. Adiego et al. model the program execution as an automata of variable states [21] and check the expected constraints with multiple model-checking techniques. This method is integrated into the PLCverif tool [22]. UBIS [23] transforms the aforementioned automata to an attribute graph, variations in which signify program manipulation. Abbasi et al. construct a control flow graph and compare the graph with the legitimate one [24].

Data-domain detectors verify the consistency of SCADA monitoring data with their expected legitimate behaviors. Any discrepancies between the expected and observed SCADA data flag the program manipulations. In addition to modeling expected SCADA data using domain knowledge [25], researchers have also derived expected data behaviors from historical monitoring data. For example, Feng et al. [26] and Yang et al. [27] identified the invariant causalities between sensor readings and control commands. Chen et al. [28] model the calculation among SCADA data variables as a black-box neural network.

Cross-domain detectors allow more accurate modeling of the program behavior — and hence better attack detection — by jointly considering the PLC program and SCADA data. On the one hand, the physics-aware constraints (e.g., timed event causalities) in SCADA data can be associated with the pairs of dependent variables derived from the PLC program [29]. On the other hand, the variable dependency conditions in the PLC program can also refine the relationships among SCADA data variables [30]. With redundant hardware and/or software-defined PLCs [31], [32], [33], the PLC program can serve as an accurate prediction model for control commands of SCADA data [34], [35], [36].

None of these detectors can reliably identify manipulated PLC program variables when multiple abnormal variables are detected, especially considering that the truly manipulated variables may not even be monitored by SCADA.

Identifying Manipulated PLC Program Variables. The typical procedure for identifying manipulated PLC program variables involves retrieving the infected binary program from the compromised PLC and decompiling it to source code [14], [37], [38], [39]. By comparing the control flow of the decompiled source code with that of the original program, the manipulated variables and/or functions can be identified [20], [24]. However, such program analysis procedure fails when the retrieved binary program is untrustworthy or cannot be decompiled [9], [14], [15], [16], [40], as explained in Sec. I.

Manipulated program variables can also be identified by detecting root anomalies in SCADA monitoring data [41], [42], [43] and correlating them to program variables through mapping. However, since SCADA typically records only a subset of program variables, the root anomalies detected in the SCADA monitoring data may not correspond to the manipulated program variables.

In contrast, `ADIS` identifies manipulated PLC program variables without needing the infected binary program. Additionally, `ADIS` can identify the manipulated program variables even when they are not recorded by SCADA.

III. Preliminaries

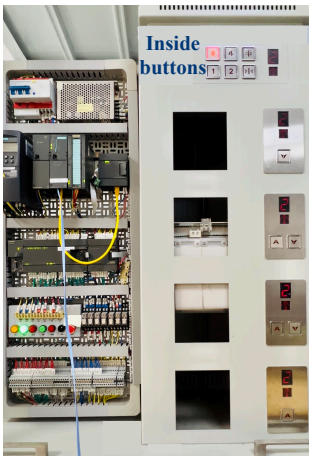
To facilitate understanding, we present in this section the necessary background of PLC control systems using an Elevator Control System (ECS) platform, a scaled-down but fully operational four-floor elevator, as shown in Fig. 2.

A. PLC Program Execution

PLCs execute the embedded problem following a cyclical 4-step procedure.

❶ *Update Input Memory.* At the beginning of each cycle, PLCs update the input memory based on the real-time states of their input modules. The input modules receive sensor readings from the physical process. For example, the PLC of ECS updates the input memory assigned to variable `I2.6` with the readings of sensor *"at_3F"*.

❷ *Execute Control Program.* Taking the updated input memory as input, PLCs execute the embedded program to generate the concomitant control commands. During this process, PLCs



(a) Elevator testbed

```
network
title =
      A(
      O     I     2.6;
      O     Q     1.1;
      )
      A     I     2.1;
      AN    M     0.1;
      =     Q     1.1;
network
title =
      A(
      A     I     0.5;
      A     I     2.5;
      O     Q     0.7;
      )
      AN    I     2.6;
      AN    Q     1.1;
      =     Q     0.7;
```

(b) Exemplary PLC control code

Fig. 2. The Elevator Control System (ECS) platform.

will introduce some *intermediate variables* to temporarily store the outputs of certain program operations. Taking the PLC program in Fig. 2(b) as an example, when sensor readings of the elevator door of *"door_is_not_opened"*, the destination button of *"to_3F"*, and elevator positions of *"at_2F"* and *"at_3F"* are updated in the input memory allocated for variables $\{$`I2.1`, `I0.5`, `I2.5`, `I2.6`$\}$ and the intermediate variable `M0.1` is calculated, the PLC generates commands of *"open_door"* and *"ascend"* using two variables stored at the output memory, namely, `Q1.1` and `Q0.7`, as

$$\texttt{Q1.1} = (\texttt{I2.6} \vee \texttt{Q1.1}) \wedge \texttt{I2.1} \wedge \neg \texttt{M0.1}, \quad (1)$$

$$\texttt{Q0.7} = (\texttt{I0.5} \wedge \texttt{I2.5} \vee \texttt{Q0.7}) \wedge \neg \texttt{I2.6} \wedge \neg \texttt{Q1.1}. \quad (2)$$

Note that the control program source code (e.g., Fig. 2(b)) will be compiled into binary when downloading to PLCs for execution.

❸ *Update Output Memory.* At the end of each program execution, PLCs copy the control commands stored in the output memory to their output modules. The output modules drive the operation of physical processes. For example, the value of output memory assigned to variable `Q0.7` is sent to the *"ascend"* mode of the traction machine.

❹ *Execute Internal Services.* After updating the output memory, PLCs execute internal services of the operation system, such as reporting the states of program variables stored in the in/output memory to SCADA. Note that PLCs commonly do not report intermediate variables — which usually do not have any physical meaning — to SCADA systems [44].

B. SCADA Monitoring

For each program variable reported by PLCs, SCADA records its value using a uniquely mapped data variable as exampled in Table I. Using these data variables, SCADA may be further integrated with monitoring/diagnostic modules for the physical process and the underlying control program.

- **Process monitoring.** SCADA visualizes the physical devices and their state interactions according to the design documents, and displays their values, at runtime, with the

TABLE I
THE DATA VARIABLES RECORDED BY THE SCADA OF THE SIMPLIFIED ELEVATOR CONTROL SYSTEM RUNNING THE CODE IN FIG. 2(B). NOTE THAT THIS IS ONLY A SUBSET OF VARIABLES USED BY ECS.

Program Variable	Data Variable	Time Series of Variable Values
I0.5	*"to_3F"*	[0,0,0,1,0,0,0,0, ...]
I2.5	*"at_2F"*	[1,1,1,1,1,1,0,0, ...]
I2.6	*"at_3F"*	[0,0,0,0,0,0,0,0, ...]
I2.1	*"door_is_not_opened"*	[1,1,1,1,1,1,1,1, ...]
Q0.7	*"ascend"*	[0,0,0,1,1,1,1,1, ...]
Q1.1	*"open_door"*	[0,0,0,0,0,0,0,0, ...]

recorded data variables. Specifically, based on the mappings between program variables and physical process modules, SCADA configures a set of monitoring data variables, each corresponding to a specific program variable. Using ICS communication protocols (e.g., the "read_var" service of Siemens S7comm), SCADA collects the real-time values of the monitored PLC variables and logs them in the corresponding monitoring data variables.

- **System diagnostics.** By comparing the recorded data variables with specified thresholds, SCADA alarms ICS operators with abnormal status of ICSs. In addition, augmented with the detection rules that should always be satisfied during normal operation, such as the invariant relationship among different variables [25], [26], [27], [28], SCADA can detect cyber attacks by examining if these rules are violated.

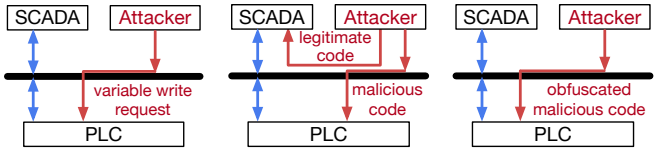
IV. THREAT MODEL

We consider adversaries who aim to damage the physical process in the shortest time by manipulating values of PLC program variables. Adversaries mount such attacks using the following two abilities that are commonly observed in real-life ICS attacks and adopted by related research in the literature.

1) Modify the values of PLC program variables. Most PLCs support online reprogramming and debugging but via insecure communication protocols [45], [46], [47], allowing adversaries to modify the values of program variables by sending the variable writing requests or control program updating requests.

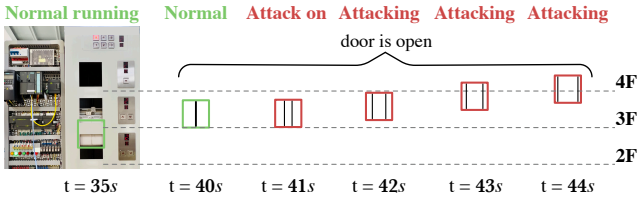
2) Hide the infected program of compromised PLCs. To obstruct post-attack forensics, adversaries can hide the infected malicious control program (if exists) by (i) responding to binary program extraction requests with a legitimate version [9], [14], or (ii) preventing the binary program from being decompiled by forensic tools [15], [16], or (iii) presenting legitimate source code to forensic tools [40].

With the above abilities, adversaries can achieve the commonly observed attacks that manipulate values of PLC program variables and hide the infected PLC program (if exists). As illustrated in Fig. 3, adversaries can (a) send variable writing requests with malicious values when PLC is in running mode [6], [48], which will not infect the executed PLC program; (b) send program updating requests with malicious control program when PLC is in maintenance [7], [49], and respond to SCADA's program extraction requests with the legitimate program, which will present norm to SCADA [14]; (c) send program updating requests with the obfuscated malicious control program when PLC is in maintenance [15], [16], which will prevent SCADA from decompiling the binary program.

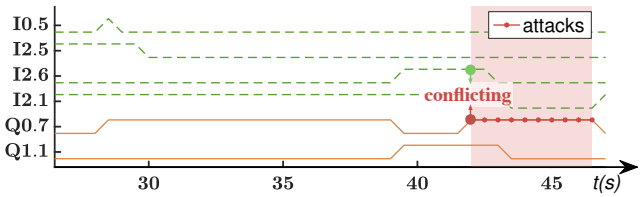


(a) send variable writing requests (b) send malicious program & spoof SCADA (c) send obfuscated malicious program

Fig. 3. Typical attacks scenarios to manipulate program variables and hide the infected program (if exists) from SCADA.



(a) sequential snapshots of the attack consequence



(b) SCADA logs (visualized using corresponding program variables)

Fig. 4. An exemplary attack against ECS: driving the elevator ascending with an opening door.

We have experimentally validated the above adversary abilities on ECS using the similar approach as BlackEnergy [48], [50], [51]. Specifically, based on the reverse-engineering results of Siemens S7comm protocol [40], an attack malware that manipulates values of program variables was crafted. By sending from the public Internet a spear-phishing email with malicious Excel files as the attachment, the malware was transferred to the computer that connects the S7-317 PLC in ECS. The transferred malware automatically forced the program variable of Q0.7 from 0 to 1 — driving the elevator abnormally ascending — within as short as 0.5s, as shown in Fig. 4(a) and demonstrated in the video [52].

V. BASIC IDEA

The dependencies among SCADA data variables are consistent with the control logic defined by the program variables. This consistency lays the foundation of ADIS to identify the manipulated program variables without requiring the infected binary program.

Let us take the attack shown in Fig. 4(a) as an example — in which the program variable Q0.7 is manipulated — to illustrate the basic idea of identifying manipulated program variables from SCADA monitoring data. According to the control logic defined in Eq. (2), Q0.7 is expected to be 0 when I2.6 equals to 1. However, as shown in Fig. 4(b), when ECS ran at $t = 42$s, the logs of *"ascend"* (corresponding to

Q0.7) and *"at_3F"* (corresponding to I2.6) do not agree with this expected behavior. Based on the variable mappings listed in Table I, this abnormal *"ascend"* could be caused due to manipulations in Q0.7 and all antecedent variables that Q0.7 depends on (i.e., {I0.5, I2.5, I2.6, Q1.1}). Since manipulating input variables will not cause inconsistencies between SCADA monitoring data and the PLC program, the set of potentially manipulated program variables can be reduced to {Q0.7, Q1.1}. The legitimacy of Q1.1 can be validated based on the recorded SCADA data variables — the related monitoring data agrees with the generation rules in Eq. (1). This way, we can correctly identify Q0.7 as the manipulated program variable.

The above successful identification of manipulated program variables inspires ADIS to (i) detect program manipulations by checking the consistency between SCADA monitoring data and legitimate PLC program, (ii) identify manipulated program variables by mapping abnormal data variables to program variables and tracking all relevant antecedent program variables, and (iii) exclude the program variables whose legitimacy can be validated from the consideration of root cause analysis.

However, the fact that SCADA records only a subset of program variables renders not all variables' legitimacy can be validated. ADIS addresses this challenge using a *state-aware dependency graph* of the PLC program, which we will explain in the next section.

VI. State-aware Dependency Graph

PLCs calculate the value of each program variable based on the real-time values of other relevant variables. Typically, the influence of different program variables on a given variable is conditional on the states of each other. As a result, the dependency between two relevant program variables may not exist consistently across their entire state space. For instance, in the control logic defined by Eq. (1), variable M0.1 can only determine the value of variable Q1.1 when the conditions of $(\texttt{I2.6} \vee \texttt{Q1.1}) = 1$ and $\texttt{I2.1} = 1$ are satisfied. On the other hand, certain states of a given program variable are necessarily dependent on some specific states of related program variables. For example, in the control logic of Eq. (1), the state of $\texttt{Q1.1} = 1$ must depend on the state of $\texttt{M0.1} = 0$.

We capture such state-dependent relationship among program variables using a dependency graph $\mathsf{G}(\mathtt{V}, \mathtt{E}, \mathtt{W}(t))$, which uses, in turn, a time-variant weight set $\mathtt{W}(t)$ to represent if and how two related variables are dependent in the context of the given system states at time t. Specifically,

- $\mathtt{V} = \{\mathtt{I}, \mathtt{M}, \mathtt{O}\}$ consists of an input node set I, an intermediate node set M, and an output node set O, representing all input, intermediate, and output variables of the PLC program.
- $\mathtt{E} = \{e_{v_j}^{v_i}\}$ denotes a set of directed edges connecting node $v_j \in \mathtt{V}$ to node $v_i \in \mathtt{V}$, representing all dependencies of the descendant variable v_i on its antecedent variable v_j. The antecedent variable set of v_i $\mathcal{N}^{v_i} := [v_j, \ldots, v_n]$ consists of all program variables with incoming edges to v_i.
- $\mathtt{W}(t) = \{w_{v_j}^{v_i}(t)\}$ consists of a set of time-variant weights for all edges in E, which is calculated based on PLC control logic and the system states at time t. First, we define the state generation of variable v_i as a function of its antecedent variables in $\mathcal{N}^{v_i}$:

$$v_i = f_i(\mathcal{N}^{v_i}). \tag{3}$$

Then, the weight of edge $e_{v_j}^{v_i}$ at time t is calculated as

$$w_{v_j}^{v_i}(t) = \begin{cases} 0, & \text{if } v_j \perp\!\!\!\perp f_i(\mathcal{N}^{v_i}) \mid \mathcal{N}_{v_j}^{v_i} = \mathcal{N}_{v_j}^{v_i}(t), \\ 1, & \text{if } \exists! v_j \in \mathbb{R},\ f_i(\mathcal{N}^{v_i}) = v_i(t) \mid \mathcal{N}_{v_j}^{v_i} = \mathcal{N}_{v_j}^{v_i}(t), \\ -1, & \text{otherwise,} \end{cases} \tag{4}$$

where $\mathcal{N}_{v_j}^{v_i} := \{\mathcal{N}^{v_i} \backslash v_j\}$ denotes all antecedent variables of v_i excluding v_j, symbols $\perp\!\!\!\perp$ and $\mid$ denote the relationships of independence and conditioning, respectively, and symbol $\exists!$ represents the existence and uniqueness of a solution. The weights of 0, 1, and -1 represent that the state $v_i(t)$ has no dependency, a unique dependency, and a non-unique dependency on the state $v_j(t)$, respectively, in the context of the given system states $\mathcal{N}_{v_j}^{v_i} = \mathcal{N}_{v_j}^{v_i}(t)$. Note that an edge weight $w_{v_j}^{v_i}(t) = 0$ only means the states $v_j(t)$ and $v_i(t)$ have no dependency, but not that variables v_j and v_j have no dependencies at other states.

This state-aware dependency graph enhances ADIS's ability to identify manipulated program variables in two ways. First, this graph enables ADIS to identify manipulated variables with fewer suspicious candidates by excluding those whose state-aware dependencies to suspicious variables have a weight of 0. Second, this graph allows ADIS to verify the legitimacy of additional program variables — specifically, those whose state-aware dependencies to legitimacy-verified variables have a weight of 1 — even if these variables are not recorded by the SCADA (see Theorem 6.1 below). In comparison, traditional state-agnostic graphs cannot precisely identify manipulated program variables, because they cannot exclude from consideration those variables that suspicious variables do not depend on under the given system states, nor verify the legitimacy of program variables that are not recorded by SCADA.

***Theorem** 6.1:* For any dependency edge $e_{v_j}^{v_i} \in \mathtt{E} \in \mathsf{G}$ with a time-variant weight of $w_{v_j}^{v_i}(t) = 1$, if the descendant variable state $v_i(t)$ is legitimate, the antecedent variable state $v_j(t)$ is also legitimate.

Proof: The theorem is proved using proof by contradiction and is detailed in Appendix A of the supplementary materials. ■

VII. Design of ADIS

Fig. 5 shows an overview of ADIS, consisting of three steps: construct the state-aware dependency graph, detect the manipulation of program variables, and if manipulations have been detected, identify the manipulated program variables.

A. Construct State-Aware Dependency Graph

Different from traditional state-agnostic dependency graphs that only capture the existence of dependencies among PLC program variables [17], [18], [19], ADIS's state-aware graph also weights each dependency edge based on the states of relevant variables and the corresponding control logic. ADIS automatically extracts control logic expressions from the original legitimate PLC program stored on the engineering station.

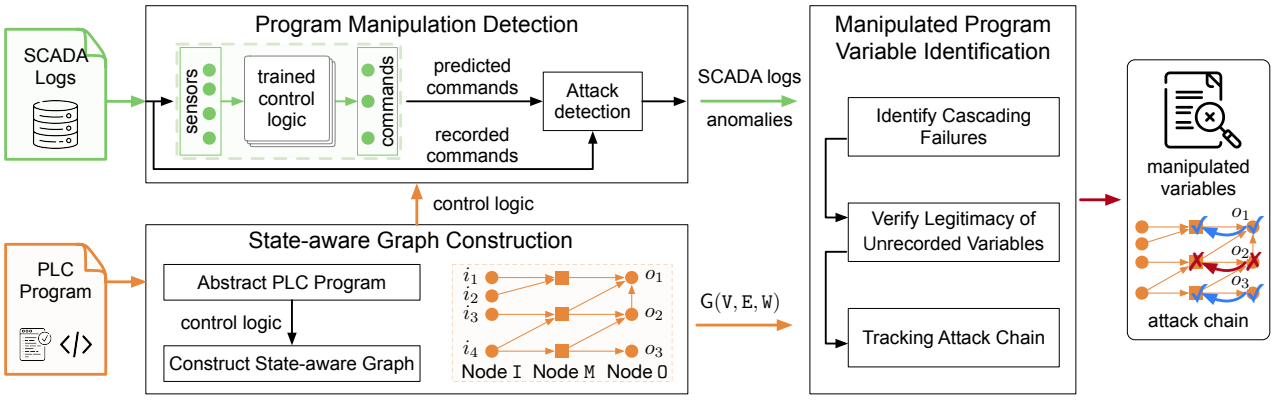


Fig. 5. Overview of ADIS: taking PLC program and SCADA logs as inputs, ADIS automatically detects attacks and identifies the manipulated variables.

1) Control Logic Extraction: As specified by the IEC 61131-3 standard, PLC control logic can be programmed using two textual languages, namely *Instruction List* (IL) and *Structured Text* (ST), and another three graphical languages. Different from ST, which is an advanced programming language, the low-level and assembly-liked IL also acts as an intermediate language to which the other four languages can be automatically translated using the programming software comes with commercial PLCs [11], [12]. To ensure universal applicability, ADIS takes the source code (or translated code) in IL as input to generate the state-aware dependency graph.

Due to the absence of program parsers for extracting control logic expressions from IL programs, we first abstract programming grammar according to the IEC 61131-3 standard and construct a language parser for the assembly-liked IL. Note that even when PLC vendors tailor their own IL, their grammar has a similar program structure per the IEC 61131-3 standard. In the following, we take Statement List (STL), the IL tailored by Siemens, as an example to illustrate the programming grammar of PLC.

An STL program consists of a series of program blocks, including code blocks (i.e., Organization Blocks, Function Blocks, and Functions) and data blocks. Each code block comprises a header specifying the block properties, a declaration section specifying the local variables, and a program section specifying the control logic. The program section within a code block consists of a sequence of statements, which can be grouped into different programming networks according to the calculation tasks. Each statement is structured as

Label (optional): *operator operand*; // comments (optional),

where the *operator* describes what the operation should do and *operand* provides the information (e.g., variables) needed to execute the operation. Statements can be categorized into calculation operation and output operation according to their functionalities. Note that following a sequence of calculation operations, there will be at least one output operation, causing the dependencies of the output operand with all preceding calculation operations. Similar to the code block, the data block also has a structured format. Instead of specifying control logic, the data block uses an initialization section to determine the initial values of program variables.

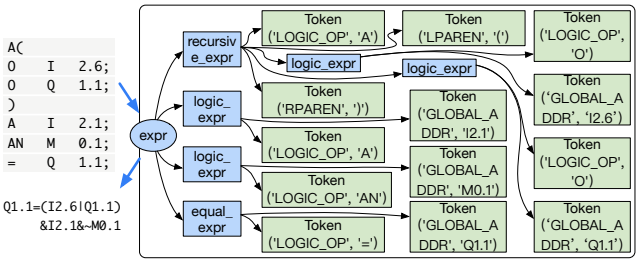


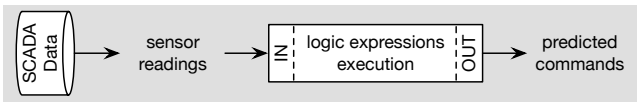
Fig. 6. An exemplary process of ADIS in parsing PLC program.

We formalize the above-described programming rules using extended Backus–Naur form syntax, as given in Appendix B of the supplementary materials (i.e., Fig. 14 and Fig. 15). With the constructed IL program parser, ADIS automatically generates an abstract syntax tree (AST) of the given STL program. By replacing STL operators with common mathematical operators, ADIS generates specific calculation expressions for each output operation of the AST. Fig. 6 visualizes ADIS's process in parsing the exemplary PLC control code in Fig. 2(b). When encountering a function call with return values, ADIS will assign the calculation expressions of the function to the in/output variables of the calling function.

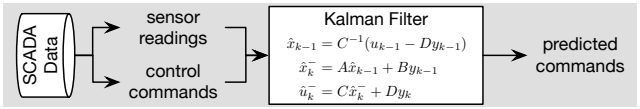
Note that the STL control logic can be programmed using variables of data blocks, which are defined in the declaration of data blocks. However, data block declarations specify variables only by their symbolic names and data types, without assigning specific memory addresses. These memory addresses are, however, essential to restore the control logic after program manipulations have been identified. ADIS recovers the memory addresses of symbolic data block variables when parsing data blocks, based on their specified data types and corresponding memory usage, replicating the process performed by PLCs. Appendix C of the supplementary materials provides more detailed procedures to recover the memory addresses of data block variables.

2) Graph Construction: With the abstracted AST and calculated expressions of a given PLC program, ADIS constructs the state-aware dependency graph $\mathtt{G}(\mathtt{V}, \mathtt{E}, \mathtt{W}(t))$ as follows.

Construct node set V. ADIS constructs input node set I and output node set O by collecting program variables from AST that are named using memory identifiers I and O, respectively. The remaining program variables in AST form



(a) discrete boolean logic



(b) continuous control logic

Fig. 7. The workflow of ADIS in predicting control commands.

the intermediate node set M, including global variables with the memory identifier M and variables declared in data blocks.

Construct edge set E. ADIS constructs edges among nodes in V by identifying all dependencies in the abstracted calculation expressions. Specifically, each edge is directed to the assignment variable from corresponding antecedent variable. Edge set E makes $\mathtt{G}(\mathtt{V}, \mathtt{E}, \mathtt{W}(t))$ a directed connected graph.

Construct time-variant weight set $\mathtt{W}(t)$. With the states of program variables recorded by SCADA, ADIS calculates the edge weights according to Eq. (4). ADIS first symbolizes each abstracted calculation expression $v_i = f_i([v_j, \ldots, v_n])$ to a symbolic execution format. Then, for each dependent variable pair, e.g., $\{v_i, v_j\}$, ADIS substitutes the variable v_i and its dependent variables excluding v_j with states recorded by SCADA at time t, deriving the equation:

$$v_i(t) = f_i([v_j, \ldots, v_n(t)]). \tag{5}$$

If variable v_j is eliminated from Eq. (5), such as when v_j is ANDed with False, ADIS will set the weight $w_{v_j}^{v_i}(t)$ as 0 because state $v_j(t)$ has no impacts on state $v_i(t)$. Otherwise, ADIS will further check whether variable v_j has a unique solution to Eq. (5) by symbolically solving the equation. If and only if v_j has one unique solution, ADIS will set the weight $w_{v_j}^{v_i}(t)$ as 1. Otherwise, $w_{v_j}^{v_i}(t)$ will be assigned as -1. However, the equation solving in symbolic computation is more time-consuming than assigning values to symbolic variables. We derived the following remark of boolean logic to facilitate ADIS calculating edge weights without relying on solving equations.

***Remark** 7.1:* For any boolean logic in a conjunctive (or disjunctive) normal form, if it outputs a value of True (or False), all clauses thereof should be True (or False), and all edges that connect the variable-as-clause to the output variable can be weighted as 1.

Note that ADIS does not require real-time calculation or updating of the time-variant edge weights; these weights only need to be updated when program manipulations are detected.

B. Detect PLC Program Manipulations

ADIS detects PLC program manipulation by symbolically executing the abstracted control logic expressions with SCADA-recorded sensor readings and verifying the resulting control commands against the SCADA-recorded commands. As shown in Fig. 7(a), by substituting the program input variables with corresponding sensor readings, ADIS will generate the expected control commands from corresponding output program variables.

Note that PLC control logic, such as the Proportional-Integral-Derivative (PID) control, may make use of historical states of PLC variables. However, the SCADA logs the states of program variables at a much lower frequency (e.g., 2Hz) than the PLC execution frequency (e.g., 50Hz), making a lot of intermittent states not logged [44]. Such mismatched frequencies between PLC execution and SCADA logging cause errors in predicting the control commands whose states change continuously within one logging cycle. For the control logic that relies on historical system states, ADIS fine-tunes the identified control logic with Kalman Filter. Specifically, the continuous control logic of PLC (such as PID) can be characterized as

$$x(k) = Ax(k-1) + By(k-1) + \omega(k-1), \tag{6}$$
$$u(k) = Cx(k) + Dy(k), \tag{7}$$

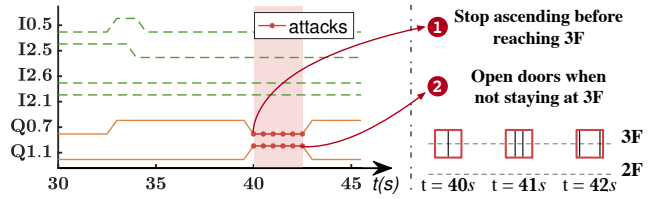
where $y(k)$, $u(k)$, and $x(k)$ denote the inputs, outputs, and intermediate states of the corresponding logic at the k-th logging step; $\omega(k)$ denotes the errors of SCADA for logging the intermediate states $x(k)$; A, B, C and D denote the equivalent control parameters. Given a period of SCADA logs, ADIS identifiers the model parameters A, B, C and D using the least-square system identification [53], and then predicts the control command $\hat{u}^-(k)$ as:

$$\hat{x}(k-1) = C^{-1}(u(k-1) - Dy(k-1)), \tag{8}$$
$$\hat{x}^-(k) = A\hat{x}(k-1) + By(k-1), \tag{9}$$
$$\hat{u}^-(k) = C\hat{x}^-(k) + Dy(k). \tag{10}$$

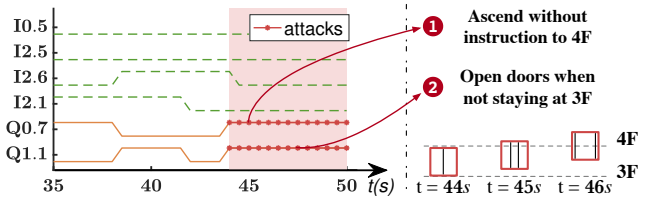
Specifically, Eq. (8) updates the estimated states $\hat{x}(k-1)$ with the truly logged $u(k-1)$ and $y(k-1)$; Eq. (9) estimates the *a priori* states estimation utilizing the identified model; Eq. (10) predicts the control commands by utilizing identified model. Fig. 7(b) visualizes the workflow of ADIS in predicting the control command of continuous control logic. Note that the state estimation of the Kalman Filter works independently of the initial states x_0. ADIS thus can be deployed to predict control commands at any time without knowing the system's initial values in advance.

Taking each of the logged control command u and its predicted value $\hat{u}^-$ as inputs, ADIS detects manipulations of program variables at each logging cycle k using non-parametric CUmulative SUM (CuSUM).

1) Update Command Prediction Error: $z(k) = u(k) - \hat{u}^-(k)$.
2) Update Cumulative Prediction Error: $S(k) = \max(0, S(k-1) + |z(k)| - \delta)$, where $S_0 = 0$, and δ denotes the abnormality confidence, preventing S from increasing persistently when PLC operates normally. In our implementation, δ is empirically set as $3\sigma(z)$, where $\sigma(z)$ denotes the standard deviation of the prediction error z for the corresponding legitimate command u; the δ for the binary control command is set as 0.5.
3) Trigger Alarms: An alarm will be triggered whenever $S(k)$ is larger than the given threshold μ, after which the detector will be reset with $S(k) = 0$. The threshold μ for continuous



(a) manipulate Q1.1 from 0 to 1: the induced anomalies satisfy the causal relationship defined by Eqs. (1)(2)



(b) manipulate both Q0.7 and Q1.1 from 0 to 1: two induced anomalies conflict with the causal relationship defined by Eqs. (1)(2)

Fig. 8. Attack traces of single and multivariate manipulations.

control commands is empirically set as $\mu = 12\sigma(z)$, and the μ for the binary control command is set as 1.

We will experimentally evaluate the impact of abnormality confidence δ and alarming threshold μ on ADIS's attack detection in Sec. VIII.

C. Identify Manipulated Program Variables

Once abnormal SCADA data variables are detected, ADIS will automatically identify the manipulated program variables using the state-aware dependency graph $\mathsf{G}(\mathtt{V}, \mathtt{E}, \mathtt{W}(t))$.

1) Detect Cascading Failures: Manipulating one program variable may cause cascading failures in PLC control logic due to the interactions among individual ICS modules and the tight coupling of corresponding PLC control logic. For example, as shown in Fig. 8(a), manipulating Q1.1 from 0 to 1 caused the elevator to maliciously open its doors and stop ascending at the same time.

The multiple anomalies induced by cascading failures can be distinguished by verifying whether the causal relationships among anomalies still hold. As corroborated in Fig. 8(a), the causal relationship between anomalies in *"ascend"* (Q0.7) and *"open_door"* (Q1.1) satisfied the causal relationship defined by control logic of Eq. (2) — i.e., the elevator should not ascend (Q0.7 = 0) if the door is opening (Q1.1 = 1). In comparison, the anomalies induced by separately manipulating Q1.1 and Q0.7 from 0 to 1 conflicted with such causal relationship, as shown in Fig. 8(b).

Based on the causal relationships among program variables, ADIS identifies cascading failures within detected anomalies and removes the cascaded anomalies from consideration of root cause analysis.

- Offline Causal Relationships Abstraction. For each output variable $o_i \in \mathtt{O}$ of the PLC program, ADIS identifies its antecedent variables based on edge set $\mathtt{E} \in \mathsf{G}(\mathtt{V}, \mathtt{E})$, and further abstracts the causal relationships set $\mathrm{R}_{o_i}^{o_j}$, where $o_j \in \mathtt{O}$. Although the calculation expressions abstracted in Sec. VII-A1 encompass all theoretical causal relationships between variables o_i and o_j, some of these relationships are physically infeasible and cannot occur in practice due to the process's physical constraints. Instead, steered by the edges of the state-aware dependency graph, ADIS abstracts the concise and physics-constrained causal relationships set, $\mathrm{R}_{o_i}^{o_j}$, from historical SCADA records. Specifically, because the relationship between $o_i(k)$ and $o_j(k)$ is conditional on the historical states of $o_i(k-1)$ and $o_j(k-1)$, ADIS constructs $\mathrm{R}_{o_i}^{o_j}$ as a set of state transition array $[o_i'(k), o_j'(k)]$, where $o_i'(k)$ equals $[o_i(k-1), o_i(k)]$ if o_i is binary and $o_i(k-1) - o_i(k)$ if o_i is continuous, and $o_j'(k)$ is calculated in the same way as $o_i'(k)$. Because the relationships between $o_i(k)$ and $o_j(k)$ are also conditional on other relevant variables in $\mathcal{N}^{o_i}$, each element in $\mathrm{R}_{o_i}^{o_j}$ will be attached with $C_{o_i}^{o_j}(k)$ as the condition, where $C_{o_i}^{o_j}(k)$ denotes the state transition array of all variables in $\mathcal{N}^{o_i}$ excluding o_j.
- Online Causal Relationships Checking. If more than one anomalies are detected in SCADA monitoring data, ADIS will map all anomalies to the corresponding program variables, forming the abnormal program variable set O^a. For each abnormal output variable $o_i^a \in O^a$, ADIS will (i) check whether o_i^a has an antecedent variable o_j^a in O^a according to $\mathsf{G}(\mathtt{V}, \mathtt{E})$, and if yes, (ii) check whether their causal relationships exist in the identified $\mathrm{R}_{o_i}^{o_j}$, and if yes again, (iii) update O^a by removing o_i^a.

2) Verify the Legitimacy of Unrecorded Program Variables: Once the abnormal program variables set O^a is collected, ADIS will further verify the legitimacy of the unrecorded program variables according to Theorem 6.1. Given the states of legitimacy-verified program variables (obtained in Sec. VII-B) and their calculation expressions (obtained in Sec. VII-A), ADIS verifies the legitimacy of unrecorded program variables as follows.

Step-1. Construct Weighting Equations. For each abstracted control logic, ADIS constructs the weighting equation by substituting the legitimate antecedent variables with their real-time states.

Step-2. Calculate Edge Weights & Verify Variable Legitimacy. For each constructed weighting equation, whenever an antecedent program variable is eliminated, ADIS will weight the corresponding dependency edge to 0. For antecedent program variables that remain in the weighting equations, ADIS will weight their dependencies to legitimate output variables. Specifically, ADIS first identifies the datatype of a legitimate output variable based on its identifier.

- For the boolean output variable v_i with the memory identifier X, ADIS weights each dependency edge that connects antecedent variable v_j to v_i according to Remark 7.1. If the state of v_i equals True/False, ADIS will transfer the corresponding weighting equation to the normal conjunctive/disjunctive form, and conclude all clauses equal True/False. ADIS will set the weight $w_{v_j}^{v_i}(t) = 1$ if the antecedent variable v_j makes one single clause, else set the weight $w_{v_j}^{v_i}(t) = -1$.
- For the continuous output variable v_i with the memory identifier B/W/D, ADIS weights each dependency edge that connects antecedent variable v_j to v_i by solving the corresponding Eq. (5).

According to Theorem 6.1, ADIS concludes program variable v_j is legitimate if it has a dependency edge to a legitimate program variable in $\mathsf{G}(\mathtt{V},\mathtt{E},\mathtt{W}(t))$ and is weighted as 1.

Step-3. Recursively Updating. Because different control logic expressions will share the same intermediate program variables, ADIS can utilize the newly verified variables/clauses from one expression to verify additional variables/clauses in other expressions. ADIS thus recursively repeats Step-1/-2 until no more clauses/variables can be verified as legitimate.

3) Track the Attack Chain: By tracking the abnormal variables in O^a in the weighted dependency graph $\mathsf{G}(\mathtt{V},\mathtt{E},\mathtt{W}(t))$, ADIS identifies all suspiciously manipulated program variables and the corresponding attack chains causing these anomalies. Specifically, for each abnormal program variable $o_i \in O^a$, ADIS initializes a subgraph $\mathsf{G}^a_{o_i}(\mathtt{V}^a_{o_i},\mathtt{E}^a_{o_i})$ to represent the attack chain. ADIS identifies the manipulated variables set $\mathtt{V}^a_{o_i}$ as

$$\mathtt{V}^a_{o_i} = \{o_i, \mathcal{N}^{o_i}\}, \tag{11}$$

where $\mathcal{N}^{o_i}$ denotes the one-hop antecedent neighbors of o_i who connect o_i with non-zero weights in $\mathsf{G}(\mathtt{V},\mathtt{E},\mathtt{W}(t))$. Then, for the variable $v_j \in \mathcal{N}^{o_i}$ whose legitimacy is not proved, ADIS replaces v_j with its one-hop antecedent neighbors who connect v_j with non-zero weights in $\mathsf{G}(\mathtt{V},\mathtt{E},\mathtt{W}(t))$. ADIS keeps updating $\mathtt{V}^a_{o_i}$ until all elements therein have been proven legitimate, or all relevant antecedent variables in $\mathsf{G}(\mathtt{V},\mathtt{E},\mathtt{W}(t))$ are included. ADIS, at last, will remove all legitimate variables from $\mathtt{V}^a_{o_i}$. The edges among variables in $\mathtt{V}^a_{o_i}$ are automatically connected when identifying the corresponding antecedent neighbors, forming the attack chain set $\mathtt{E}^a_{o_i}$.

VIII. Evaluation

A. Evaluation Platforms

We have implemented ADIS in Python and evaluated it on two real-world platforms: an Elevator Control System (ECS) and an Ethanol Distillation System (EDS).

- ECS, as illustrated in Fig. 2, encompasses tasks such as driving the traction machine, opening and closing doors, and indicating operational states to passengers. All control logic in ECS is defined based on boolean operations and is implemented using a Siemens S7-317 PLC with 25 input variables, 21 output variables, and 21 intermediate variables. All program variables in ECS are boolean.
- EDS, as shown in Appendix D of the supplementary materials (see Fig. 17), involves tasks such as water-ethanol mixture heating (Loop-I), ethanol vapor condensing (Loop-II), and tower liquid level maintaining (Loop-III), separating ethanol from the mixture with over 90% purity. The control logic of these tasks is defined based on a combination of boolean and arithmetic operations, and is implemented using a Siemens S7-315 PLC with 40 input variables, 28 output variables, and 179 intermediate variables.

More details of ECS and EDS can be found in Appendix D of the supplementary materials.

B. Evaluation Methodology

In line with the adversary abilities outlined in Sec. IV, we manipulated the values of program variables in ECS and EDS by sending malicious communication packets. Demo videos of these variable manipulations can be found in [52], [54].

Evaluation Setup. Against ECS, we manipulated 42 program variables, including SCADA-recorded output program variables and non-recorded intermediate program variables (as listed in Table II and declared in Table V of the supplementary materials). Since all 42 program variables are boolean, they were manipulated by modifying their values between 0 and 1, and each variable was restored to its original value after 5 seconds. This process was repeated 10 times, with a 30-second interval of normal operation between each manipulation. Against EDS, we also modified the value of the boolean output program variable (i.e., Q1.2) between 0 and 1 for 10 times. For the continuous output program variables (i.e., QW70 and QW74), which are calculated by arithmetic operation, we manipulated their values in two ways: (i) injecting the manipulation $n^a(k)$ into the SCADA-monitored QW70 and QW74, with $n^a(k)$ updated at every monitoring cycle; (ii) resetting the non-recorded PID controller's parameters (i.e., DB13.DBD14 and DB13.DBD18 for QW70, DB10.DBD14 and DB10.DBD18 for QW74). Both manipulation strategies resulted in gradually varying values. Each of QW70 and QW74 was tampered with 60 times.

In total, we have conducted 420 attacks against ECS and 130 attacks against EDS, covering both univariate manipulations and cases in which multiple variables are manipulated simultaneously. Note that manipulating the output and intermediate program variables directly and indirectly sends malicious control commands to the connected actuators, leading to actuator malfunctions and degradation of the physical process.

Evaluation Metrics. We evaluate ADIS's attack detection using true positive alarm rate (TPR), false positive alarm rate (FPR), and detection latency.

$$\begin{aligned}\text{TPR} &:= \frac{\text{number of detected attacks}}{\text{number of mounted attacks}},\\ \text{FPR} &:= \frac{\text{number of false alarms}}{\text{number of samples without attacks}},\\ \text{Latency} &:= \text{ time to detect attacks since launched.}\end{aligned}$$

Note that FPR also indicates the average time that ADIS will trigger a false alarm when no program variable is manipulated.

We evaluate ADIS's attack identification performance using the ratio of correctly identified attacks to all detected attacks. An attack is considered correctly identified if the concluded suspicious variable set includes the truly manipulated program variables. Additionally, we assess the effectiveness of ADIS's attack identification based on the size of the concluded suspicious variable set. A smaller suspicious variable set signifies better identification effectiveness.

C. Evaluation Results with ECS

The 420 attacks we launched on ECS have caused different anomalies, including misleading the traction machine, destroying the bidirectional motor, displaying wrong operation states to passengers, and frying fuses to cut off the power supply.

TABLE II
DETECTING ATTACKS MOUNTED ON ECS. EACH PROGRAM VARIABLE WAS MANIPULATED 10 TIMES (420 MANIPULATIONS IN TOTAL).

Manipulated Variables		Q0.1	Q0.2	Q0.3	Q0.4	Q0.5	Q0.6	Q0.7	Q1.1	Q1.2	Q1.3	Q1.4	Q1.5	Q1.6	Q1.7
Detection Results	TPR	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
	FPR	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Latency	1.25s	2s	**1s**	3.05s	3.55s	**1s**	**1s**	**1s**	**1s**	1.15s	**1s**	3.5s	**1s**	**1s**
Manipulated Variables		Q2.0	Q2.1	Q2.2	Q2.3	Q2.4	Q2.5	Q2.6	M1.0	M2.0	M2.2	M2.3	M3.3	M8.0	M8.1
Detection Results	TPR	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
	FPR	0%	0%	0%	0%	0%	0%	0%	0.52‰	0.33‰	0%	0%	0%	0%	0%
	Latency	1.2s	1.35s	1.15s	1.1s	4.8s	1.3s	1.1s	**1s**	**1s**	**1s**	**1s**	5.5s	3.55s	1.5s
Manipulated Variables		M8.2	M8.3	M10.0	M11.0	M12.0	M13.0	M20.0	M20.3	M20.4	M20.5	M21.0	M22.0	M24.0	M30.5
Detection Results	TPR	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
	FPR	0%	0%	0.17‰	0.02‰	0%	0.29‰	0.08‰	0%	0%	0.21‰	0%	0%	0%	0%
	Latency	1.25s	**1s**	**1s**	1.1s	**1s**	**1s**	1.2s	2.75s	**1s**	**1s**	**1s**	**1s**	3.6s	2.4s
Averaged Results		TPR: 100%; FPR: 0.038‰; Latency: 1.65s													

† The abnormality confidence is set as $\delta = 0.5$, and the alarming threshold is set as $\mu = 1$.

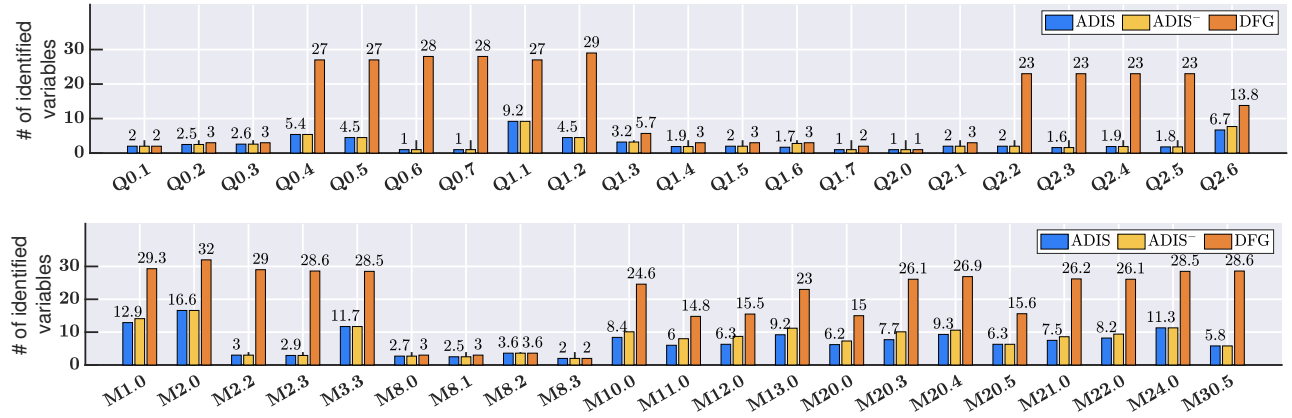


Fig. 9. Identifying suspiciously manipulated program variables for the detected attacks against ECS.

1) Attack Detection: The 420 attacks are detected with an average TPR of 100%, FPR of 0.038‰, and latency of 1.65 seconds.

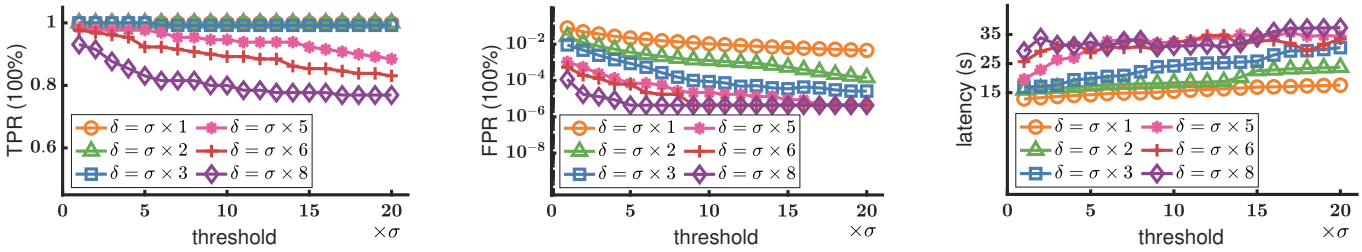
Detection Accuracy. As shown in Table II, ADIS detects all the 10 attacks against each variable of ECS, achieving an overall 100% TPR. ADIS triggers 0 FP alarm for 83% (35/42) of these variable manipulations. For the remaining manipulations, ADIS triggers, at most, 0.52‰ of false positives when no attacks are being mounted.

A deeper analysis of the triggered false alarms reveals that they are all caused by the persistent effects of attacks initiated earlier and terminated before the alarm is triggered. For example, the intermediate program variables M1.0 and M2.0 represent temporary values upon which the commands *"ascend"* (Q0.7) and *"descend"* (Q0.6) depend. Since M1.0 and M2.0 are mutually exclusive in the control logic, manipulating M1.0 from 0 to 1 will also force M2.0 from 1 to 0, effectively stopping the elevator from descending and causing it to start ascending. This unintended ascent triggers further anomalies, such as *"indicate_to_3F"* (Q1.5) changing from 1 to 0. Consequently, even though the manipulation lasts for only 5 seconds, its effects persist, resulting in alarms being triggered after the attack has ceased. These alarms are therefore classified as *false*.

Detection Latency. ADIS detects the 420 attacks with an average latency of 1.65 seconds. Note that with abnormality confidence $\delta = 0.5$ and alarming threshold $\mu = 1$, ADIS concludes an attack is detected only after at least two continuous anomalies have been observed, indicating a minimum detection latency — under these settings of δ and μ — of 1 second for the 2Hz SCADA monitoring. As shown in Table II, among 48% (20/42) of these variable manipulations, ADIS triggered alarms for all the corresponding attacks within 1 second. The manipulations with a detection latency exceeding one second occur because they do not immediately cause abnormal control commands when launched, such as when the manipulated variables were ANDed with conditions of False. To provide more information, the 1-second anomalies in ECS's control logic caused the elevator to only ascend/descend by 1/10 of a floor and to open/close the door by 1/3.

2) Attack Identification: With the above detected attacks and abstracted variable dependencies (as shown in Fig. 18 of Appendix E in supplementary materials), we further evaluated ADIS's attack identification. The evaluation results showed that ADIS successfully included the manipulated variables into the suspicious variable sets for all 420 attacks, demonstrating ADIS's 100% correctness in attack identification.

As shown in Fig. 9, ADIS identified 4.99 suspicious vari-



(a) TPR vs. μ under different δ (b) FPR vs. μ under different δ (c) latency vs. μ under different δ

Fig. 10. Detecting the attacks mounted against EDS with different abnormality confidence δ and alarming threshold μ.

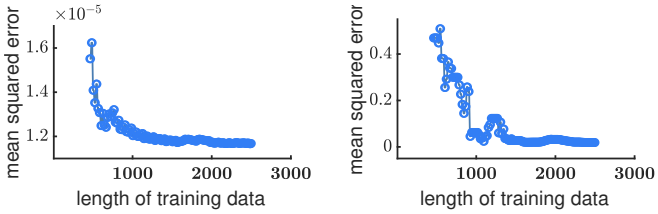
TABLE III
DETECTING ATTACKS MOUNTED ON EDS, WHERE PROGRAM VARIABLES WERE MANIPULATED 130 TIMES IN TOTAL.

Manipulation	Q1.2		QW70		QW74	
# of attacks # of normalities	10 101477		60 64500		60 62619	
Detection	ADIS	[27]	ADIS	[27]	ADIS	[27]
TPR	**100%**	100%	**100%**	98.3%	**98.3%**	95%
FPR	**0**	7.9‰	**0.03‰**	8‰	**0.14‰**	13‰
Latency (s)	**1**	9	**30.7**	80.9	**23.7**	65.9
Average	ADIS	TPR: 99.23%; FPR: 0.05‰; Latency: 25.2s				
	[27]	TPR: 96.92%; FPR: 9.46‰; Latency: 68.4s				

† The detection parameters of $\{\delta, \mu\}$ for ADIS are set as $\{0.5, 1\}$ for Q1.2, and $\{3\sigma(\cdot), 12\sigma(\cdot)\}$ for QW70 and QW74.

ables on average, and 16.6 at most, for all the 420 variable manipulations. For all manipulations on program variables Q0.6 and Q0.7, which control elevator's *"ascending"* and *"descending"* operations, ADIS identified only one suspicious variable: the manipulated variable itself. This high identification effectiveness is due to the dependence of Q0.6 and Q0.7 on many *shared variables* — variables on which multiple program variables rely. The legitimacy of these shared variables can be verified by the state-aware dependencies of other legitimate variables, even without SCADA data. Once verified, these shared variables can be excluded from the suspicious set. We also evaluated ADIS's attack identification without enabling the detection of cascading failures, denoted as ADIS$^-$. As shown by the blue and yellow bars in Fig. 9, detecting cascading failures allows ADIS to reduce further the average size of suspicious variable set from 5.43 to 4.99.

To examine the advantages of ADIS's state-aware dependency graph over traditional state-agnostic counterparts, we replaced ADIS's state-aware dependency graph with a state-agnostic graph (i.e., data flow graph) proposed by Castellanos et al. [17], while keeping all other design components unchanged. This modified version of ADIS is referred to as DFG. As shown by the orange bar of Fig. 9, DFG identified 17.39 variables on average (and 32 at most) for the 420 variable manipulations. ADIS outperforms DFG with an average 71.3% reduction in the number of concluded suspicious variables, reaching as high as 96.4% for the attack identifications of Q0.6 and Q0.7.



(a) cooling water loop (Loop-II) (b) tower liquid level loop (Loop-III)

Fig. 11. Training ADIS's attack detection model with different durations of SCADA monitoring data.

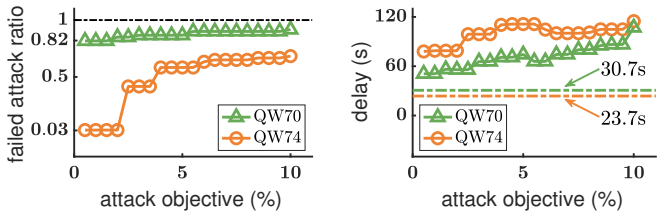
D. Evaluation Results with EDS

The control logic of ECS uses only boolean operations. To examine ADIS's generality, we also deployed and tested ADIS on EDS whose control logic is defined with a mix of boolean and arithmetic operations. The launched attacks have caused furnace overheating, tower flooding, system oscillations, and device damage to EDS.

1) Attack Detection: ADIS detected the mounted 130 attacks with an average TPR of 99.23%, FPR of 0.05‰, and detection latency of 25.2s. ADIS outperforms PLC-Sleuth [27] (a causality-based attack detector) with a higher TPR, a lower FPR, and a shorter detection latency, as compared in Table III.

Detection Accuracy. As summarized in Table III, ADIS detected the attacks against Q1.2, QW70 and QW74 with an average TPR of 100%, 100% and 98.3%, respectively. Among all normalities during the attack experiments, ADIS triggered only 0‰, 0.03‰ and 0.15‰ of false positives for Q1.2, QW70 and QW74, respectively. ADIS outperforms PLC-Sleuth [27] because the control logic expressions abstracted from PLC program provide a more accurate model for predicting PLC program execution than the normal behaviors captured from SCADA data.

We have also evaluated ADIS with different settings of abnormality confidence δ and alarming threshold μ. As shown in Figs. 10(a) and 10(b), ADIS (i) has a robust TPR with threshold μ when δ is small, because even minor prediction errors caused by attacks can accumulate to reach the alarming threshold; and (ii) has a robust FPR with threshold μ when δ is large, because system noise-induced prediction errors are limited and unlikely to accumulate enough to trigger false alarms. Evaluations with $\delta = \sigma \times \{5, 6, 8\}$ indicate that ADIS's TPR is more sensitive to δ than ADIS's FPR, as ADIS's FPR converges to the same value across different δs when $\mu \geq 17$, whereas ADIS's TPR does not.



(a) the ratio of mounted attacks not achieving attack objectives

(b) the average delay in achieving attack objectives

Fig. 12. Performance of mounted attacks. The attack objective is defined as causing the system status to deviate from expected norms by a certain percentage (%).

Lastly, we trained ADIS's attack detection model with different durations of SCADA monitoring data. As shown in Fig. 11, the mean squared errors of ADIS's prediction on legitimate control commands for Loop-II and Loop-III decrease and converge as the training data duration increases. ADIS needs only about $2,000$ SCADA samples (16 minutes) to complete training, demonstrating the robustness of ADIS's attack detection model to the duration of training data.

Detection Latency. Similar to the performance with ECS, ADIS detected the manipulations on boolean logic of Q1.2 within 1 second. For manipulations on the arithmetic logic of QW70 and QW74, ADIS detected attacks with an average latency of 30.7s and 23.7s, respectively, which are sufficiently short when compared to the time needed for the attack to cause physical impact to system operation. As shown in Fig. 12(a), during the attack periods, 82% (49 out of 60) manipulations on QW70 and 3% (2 out of 60) manipulations on QW74 fail to cause even a 0.5% deviation from the normal operation status. Moreover, as shown in Fig. 12(b), manipulations on QW70 and QW74 that have caused 0.5% deviations took an average delay of 50.8s and 78.0s, respectively, which are significantly longer than ADIS's detection latency (i.e., 30.7s and 23.7s). The short detection latency provides ICS operators with sufficient time to implement emergency responses before any physical damages on the ICS occur, particularly with the manipulated program variables identified by ADIS, which we will examine next.

2) Attack Identification: With the abstracted program variable dependencies of PLC in EDS (as shown in Fig. 19 in supplementary materials), ADIS correctly identified the manipulated variables as suspicious for all the detected attacks, demonstrating ADIS's 100% identification correctness.

As shown in Fig. 13, ADIS identified on average 8, 25.9 and 19 suspicious variables for the manipulations against Q1.2, QW70 and QW74, respectively. Unlike the operation of boolean variables in ECS, operating continuous variables in EDS requires converting the signal ranges — e.g., from [0, 27648] to [0, 300] for cooling water flow and from [0, 100] to [0, 27648] for command QW74, thus introducing more intermediate variables than PLC program of ECS. Note that there is no causality among the three control commands in EDS, so the cascading failures identification of ADIS does not help reduce the number of suspicious variables, as shown by ADIS and ADIS$^-$ in Fig. 13.

In addition, the control logic for different commands in EDS shares only one variable (between Q1.2 and QW70), limiting

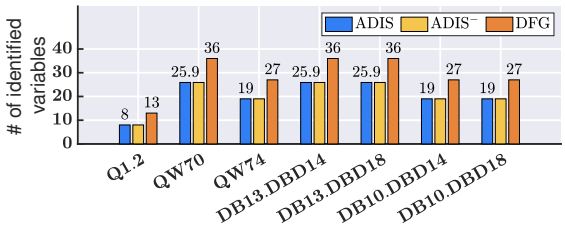


Fig. 13. Identifying suspiciously manipulated program variables for the detected attacks against EDS.

ADIS's improvement in identification effectiveness over DFG. Still, steered by the edges whose state-aware dependency weight is 0, ADIS reduces the size of suspicious variables set by 32.05% on average when compared to DFG (see Fig. 13).

IX. Discussion

Real-World Implementation of ADIS. ADIS can be modularly integrated into real-world ICSs by system owners, leveraging the corresponding PLC programs and SCADA logs. For the modern and well-documented ICSs, these artifacts are readily available. For legacy ICSs with limited documentation, owners can extract the program source code from PLCs using commercial programming tools (e.g., Siemens TIA Portal [55] and Allen-Bradley Studio 5000 [56]) and customize SCADA with ICS communication protocols to collect logs. In cases where the program source code is unavailable to owners, ADIS can incorporate data-driven detection methods and perform cascading failure analysis, as discussed below.

Compatibility of ADIS with State-Of-The-Art Attack Detection. ADIS complements data-driven manipulation detection [25], [26], [27], [28] when training data covering all legitimate operations is unavailable. Conversely, these data-driven methods complement ADIS when the legitimate PLC program is not accessible. On the other hand, ADIS's attack detection is orthogonal to *physics-based attack detection* — i.e., verifying the impact of control commands on specific system states [57], [58], [59]. Together, ADIS and physics-based detection form a closed-loop attack detection framework for ICSs, detecting anomalies both in control logic and in physical processes. Following attack detection, ADIS can utilize the causalities identified from historical operation data to conduct cascading-failure analysis, enabling root-cause identification among multiple anomalies.

ADIS v.s. Monitoring Data Deception Attacks. To conceal program corruptions, attackers may deceive SCADA records of sensor readings and/or control commands to make them appear consistent with the defined control logic. However, deceiving only sensor readings or only control commands will conflict with the legitimate physical impact of control commands on specific system states, and can thus be detected [57], [58], [59]. More sophisticated attackers with full knowledge of the target ICS may jointly deceive sensor readings and control commands to evade detection by both control-logic and physics-based verification [60]. However, the closed-loop attack detection consisting of ADIS and the *physics-based attack detection* will significantly limit the physical impact of

such stealthy attacks [57], [61]. Moreover, researchers have integrated controlled stochasticity into ICS operations, making it unpredictable to attackers and revealing the deceived SCADA records [62], [63]. By excluding the deceived SCADA records revealed by the physics-based detection and the controlled stochasticity, ADIS can identify the manipulated program variables, but with an expanded set of suspicious variables.

Scalability of ADIS**.** ADIS's scalability to large-scale ICSs depends on the complexity of the defined control logic. The syntax of PLC IL programs is context-free, with each instruction being a syntactically independent unit. Consequently, deterministic parsing (e.g., using an LALR(1) parser [64]) and AST construction can be performed in linear time and space. Meanwhile, the control logic of large-scale ICSs is typically designed as a set of control loops (e.g., cascaded loops) for different physical process units [65]. This design inherently limits the number of program variables within each loop, thus constraining the complexity of calculating state-aware weights. Moreover, leveraging specific function properties, such as the monotonicity of linear range transformations, can further reduce the complexity of weight calculation, similar to Remark 7.1.

X. Conclusion

Based on the novel *state-aware dependency graph* of PLC program, this paper presented ADIS to defend against PLC program variable manipulations. The state-aware dependency graph represents whether a time-variant dependency exists between any two relevant program variables under specific system states. Taking the PLC program and real-time SCADA monitoring data as inputs, ADIS automatically constructs the state-aware dependency graph, detects PLC program manipulations, and identifies the manipulated program variables. We have experimentally corroborated ADIS's effectiveness on two ICS platforms.

References

[1] Efrén López-Morales, Ulysse Planta, Carlos Rubio-Medrano, Ali Abbasi, and Alvaro A Cardenas. SoK: Security of Programmable Logic Controllers. In *33th USENIX Security Symposium*, pages 1–20, 2024.

[2] Ryan Pickren, Tohid Shekari, Saman Zonouz, and Raheem Beyah. Compromising Industrial Processes using Web-Based Programmable Logic Controller Malware. In *Network and Distributed System Security Symposium*, pages 1–18, 2024.

[3] Ryan Pickren, Animesh Chhotaray, Frank Li, Saman Zonouz, and Raheem Beyah. Release the Hounds! Automated Inference and Empirical Security Evaluation of Field-Deployed PLCs Using Active Network Data. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3674–3688, 2024.

[4] Zeyu Yang, Liang He, Peng Cheng, and Jiming Chen. Mismatched Control and Monitoring Frequencies: Vulnerability, Attack, and Mitigation. *IEEE Transactions on Dependable and Secure Computing*, 22(1):16–33, 2024.

[5] Jie Meng, Zhenyong Zhang, Hengye Zhu, Zeyu Yang, Ruilong Deng, Peng Cheng, and Jianying Zhou. SSTAF: Security Settings-Based Threat Assessment Framework of Programmable Logic Controllers. *IEEE Transactions on Information Forensics and Security*, 20:7512–7527, 2025.

[6] Paul Smith. The Iran Steel Industry Cyber Attack Explained. https://blog.scadafence.com/the-iran-steel-industry-cyber-attack-explained, 2022.

[7] Eduard Kovacs. Hackers Knew How to Target PLCs in Israel Water Facility Attacks: Sources. https://www.securityweek.com/hackers-knew-how-target-plcs-israel-water-facility-attacks-sources/, 2020.

[8] Martin Giles. Triton is the world's most murderous malware, and it's spreading. https://www.technologyreview.com/2019/03/05/103328/cybersecurity-critical-infrastructure-triton-malware/, 2019.

[9] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. Stuxnet Dossier. *Symantec Security Response*, 5(6):29, 2011.

[10] Kevin Collier. In Florida, a near-miss with a cybersecurity worst-case scenario. https://www.nbcnews.com/tech/security/florida-near-miss-cybersecurity-worst-case-scenario-n1257091, 2021.

[11] Siemens AG. Programming with STEP 7. https://cache.industry.siemens.com/dl/files/825/109751825/att_933142/v1/STEP_7_-_Programming_with_STEP_7.pdf, 2017.

[12] Schneider Electric. SoMachine Programming Guide. https://download.schneider-electric.com/files?p_enDocType=User+guide&p_File_Name=EIO0000000067.15.pdf&p_Doc_Ref=EIO0000000067, 2018.

[13] Rockwell Automation. MicroLogix 1400 Programmable Controllers User Manual. https://literature.rockwellautomation.com/idc/groups/literature/documents/um/1766-um001_-en-p.pdf, 2021.

[14] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium*, 2019.

[15] Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, and Vassil Roussev. Denial of Engineering Operations Attacks in Industrial Control Systems. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 319–329, 2018.

[16] Nauman Zubair, Adeen Ayub, Hyunguk Yoo, and Irfan Ahmed. Control Logic Obfuscation Attack in Industrial Control Systems. In *2022 IEEE International Conference on Cyber Security and Resilience*, pages 227–232, 2022.

[17] John H Castellanos, Martín Ochoa, and Jianying Zhou. Finding Dependencies between Cyber-Physical Domains for Security Testing of Industrial Control Systems. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 582–594, 2018.

[18] Zeyu Yang, Liang He, Hua Yu, Chengcheng Zhao, Peng Cheng, and Jiming Chen. Reverse Engineering Physical Semantics of PLC Program Variables Using Control Invariants. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pages 548–562, 2022.

[19] John H Castellanos, Martin Ochoa, Alvaro A Cardenas, Owen Arden, and Jianying Zhou. AttkFinder: Discovering Attack Vectors in PLC Programs using Information Flow Analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 235–250, 2021.

[20] Stephen McLaughlin, Saman Zonouz, Devin Pohly, and Patrick McDaniel. A Trusted Safety Verifier for Process Controller Code. In *Network and Distributed Systems Security Symposium*, 2014.

[21] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor Manuel González Suárez. Applying Model Checking to Industrial-Sized PLC Programs. *IEEE Transactions on Industrial Informatics*, 11(6):1400–1410, 2015.

[22] Dániel Darvas, Enrique Blanco, and Switzerland V Molnár. PLCverif Re-engineered: An Open Platform for the Formal Analysis of PLC Programs. In *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 21–27, 2019.

[23] Muluken Hailesellasie and Syed Rafay Hasan. Intrusion Detection in PLC-Based Industrial Control Systems Using Formal Verification Approach in Conjunction with Graphs. *Journal of Hardware and Systems Security*, 2:1–14, 2018.

[24] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 437–448, 2017.

[25] Luis Garcia, Saman Zonouz, Dong Wei, and Leandro Pfleger De Aguiar. Detecting PLC Control Corruption via On-Device Runtime Verification. In *2016 Resilience Week (RWS)*, pages 67–72, 2016.

[26] Cheng Feng, Venkata Reddy Palleti, Aditya Mathur, and Deeph Chana. A Systematic Framework to Generate Invariants for Anomaly Detection in Industrial Control Systems. In *Network and Distributed Systems Security Symposium*, 2019.

[27] 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 *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 333–348, 2020.

[28] Yuqi Chen, Christopher M Poskitt, and Jun Sun. Code Integrity Attestation for PLCs using Black Box Neural Network Predictions. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[29] Mu Zhang, James Moyne, Z Morley Mao, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, et al. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *2019 IEEE Symposium on Security and Privacy*, pages 522–538, 2019.

[30] Syed Ghazanfar Abbas, Muslum Ozgur Ozmen, Abdulellah Alsaheel, Arslan Khan, Z Berkay Celik, and Dongyan Xu. SAIN: Improving ICS Attack Detection Sensitivity via State-Aware Invariants. In *33th USENIX Security Symposium*, 2024.

[31] Thiago Alves. OpenPLC Runtime version 3. https://github.com/thiagoralves/OpenPLC_v3, 2021.

[32] 3S Smart Software Solutions. CODESYS Device Directory. https://devices.codesys.com, 2019.

[33] Michael Büsch. Awlsim: S7 compatible PLC/SPS. https://bues.ch/cms/automation/awlsim, 2020.

[34] Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman Zonouz. Control Behavior Integrity for Distributed Cyber-Physical Systems. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems*, pages 30–40. IEEE, 2020.

[35] Wei Lin, Heng Chuan Tan, Binbin Chen, and Fan Zhang. Dnattest: Digital-twin-based non-intrusive attestation under transient uncertainty. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 376–388. IEEE, 2023.

[36] Zeyu Yang, Liang He, Yucheng Ruan, Peng Cheng, and Jiming Chen. Unveiling Physical Semantics of PLC Variables Using Control Invariants. *IEEE Transactions on Dependable and Secure Computing*, 22(2):1400–1417, 2024.

[37] Anastasis Keliris and Michail Maniatakos. ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries. In *Symposium on Network and Distributed System Security*, 2019.

[38] Hadjer Benkraouda, Anand Agrawal, Dimitrios Tychalas, Marios Sazos, and Michail Maniatakos. Towards PLC-Specific Binary Analysis Tools: An Investigation of Codesys-Compiled PLC Software Applications. In *Proceedings of the 5th Workshop on CPS&IoT Security and Privacy*, pages 83–89, 2023.

[39] Chao Sang, Jun Wu, Jianhua Li, and Mohsen Guizani. From Control Application to Control Logic: PLC Decompile Framework for Industrial Control System. *IEEE Transactions on Information Forensics and Security*, 2024.

[40] Eli Biham, Sara Bitan, Aviad Carmel, Alon Dankner, Uriel Malin, and Avishai Wool. Rogue7: Rogue Engineering Station Attacks on S7 Simatic PLCs. In *BlackHat USA*, pages 1–21, 2019.

[41] Jianfang Jiao, Weiting Zhen, Wenxiang Zhu, and Guang Wang. Quality-Related Root Cause Diagnosis Based on Orthogonal Kernel Principal Component Regression and Transfer Entropy. *IEEE Transactions on Industrial Informatics*, 17(9):6347–6356, 2020.

[42] Chao Liu, Kin Gwn Lore, Zhanhong Jiang, and Soumik Sarkar. Root-cause Analysis for Time-series Anomalies via Spatiotemporal Graphical Modeling in Distributed Complex Systems. *Knowledge-Based Systems*, 211:106527, 2021.

[43] Qingchao Jiang, Wenjing Wang, Shutian Chen, Chunjian Pan, and Weimin Zhong. Hierarchical fault root cause identification in plant-wide processes using distributed direct causality analysis. *IEEE Transactions on Industrial Informatics*, 2023.

[44] ISA. *ANSI/ISA–95.00.03–2005: Enterprise-Control System Integration Part 3: Activity Models of Manufacturing Operations Management*. International Society of Automation, 2005.

[45] Cybersecurity and Infrastructure Security Agency. Rockwell Automation Logix Controllers. https://www.cisa.gov/news-events/ics-advisories/icsa-22-090-05, 2022.

[46] Siemens Security Advisory by Siemens ProductCERT. SSA-381684: Improper Password Protection during Authentication in SIMATIC S7-300 and S7-400 CPUs and Derived Products. https://cert-portal.siemens.com/productcert/pdf/ssa-381684.pdf, 2020.

[47] Schneider Electric Security Notification. Security Notification - Modicon M100/M200/M221 Programmable Logic Controller (V3.0). https://www.se.com/ww/en/download/document/SEVD-2020-315-05/, 2021.

[48] Defense Use Case. Analysis of the Cyber Attack on the Ukrainian Power Grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 388:1–29, 2016.

[49] America's Cyber Defense Agency. Exploitation of Unitronics PLCs used in Water and Wastewater Systems. https://www.cisa.gov/news-events/alerts/2023/11/28/exploitation-unitronics-plcs-used-water-and-wastewater-systems, 2023.

[50] Zhen Qin, Zeyu Yang, Yangyang Geng, Xin Che, Tianyi Wang, Hengye Zhu, Peng Cheng, and Jiming Chen. Reverse engineering industrial protocols driven by control fields. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*, pages 2408–2417. IEEE, 2024.

[51] Jie Meng, Zeyu Yang, Zhenyong Zhang, Yangyang Geng, Ruilong Deng, Peng Cheng, Jiming Chen, and Jianying Zhou. Sepanner: Analyzing semantics of controller variables in industrial control systems based on network traffic. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 310–323, 2023.

[52] Anonymous Anonymous. Demo video of manipulating program variables of Siemens S7-317 PLC in Elevator Control System. https://youtu.be/7lItczh_f2A, 2024.

[53] Lennart Ljung. System Identification. In *Signal analysis and prediction*, pages 163–173. Springer, 1998.

[54] Anonymous Anonymous. Demo video of manipulating program variables of Siemens S7-315 PLC in Ethanol Distillation System. https://youtu.be/FHMkcLJoaN8, 2025.

[55] Siemens AG. STEP 7 and WinCC Engineering V17. https://cache.industry.siemens.com/dl/files/671/109798671/att_1071920/v1/STEP_7_WinCC_V17_enUS_en-US.pdf, 2021.

[56] Tim Wilborne. Upload PLC program from a Compactlogix/Controllogix PLC Studio 5000. https://www.youtube.com/watch?v=yEUHNZrDCUk, 2020.

[57] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1105, 2016.

[58] Alessandro Erba and Nils Ole Tippenhauer. Assessing Model-free Anomaly Detection in Industrial Control Systems Against Generic Concealment Attacks. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 412–426, 2022.

[59] Konrad Wolsing, Eric Wagner, Luisa Lux, Klaus Wehrle, Martin Henze, and FKIE Fraunhofer. GeCos Replacing Experts: Generalizable and Comprehensible Industrial Intrusion Detection. In *34th USENIX Security Symposium*, 2025.

[60] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *Network and Distributed Systems Security Symposium*, 2017.

[61] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants. In *29th USENIX Security Symposium*, pages 895–912, 2020.

[62] Hamid Reza Ghaeini, Matthew Chan, Raad Bahmani, Ferdinand Brasser, Luis Garcia, Jianying Zhou, Ahmad-Reza Sadeghi, Nils Ole Tippenhauer, and Saman Zonouz. PAtt: Physics-based Attestation of Control Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 165–180, 2019.

[63] Zeyu Yang, Hongyi Pu, Liang He, Chengtao Yao, Jianying Zhou, Peng Cheng, and Jiming Chen. Deception-Resistant Stochastic Manufacturing for Automated Production Lines. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 546–560, 2024.

[64] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques & Tools*. Pearson Education, 2007.

[65] Sigurd Skogestad and Ian Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons, 2005.