Reverse Engineering Physical Semantics of PLC Program Variables Using Control Invariants

Zeyu Yang
Zhejiang University
zeyuyang@zju.edu.cn

Liang He
University of Colorado Denver
liang.he@ucdenver.edu

Hua Yu
Zhejiang University
hua.yu@zju.edu.cn

Chengcheng Zhao
Zhejiang University
chengchengzhao@zju.edu.cn

Peng Cheng
Zhejiang University
saodiseng@gmail.com

Jiming Chen
Zhejiang University
cjm@zju.edu.cn

ABSTRACT

Semantic attacks have incurred increasing threats to Industrial Control Systems (ICSs), which manipulate targeted system modules by identifying the physical semantics of variables in Programmable Logic Controllers (PLCs) programs, i.e., the sensing/actuating modules represented by the variables. This is usually (and inefficiently) achieved via manual examination of system documents and long-term observation of system behavior. In this paper, we design ARES, a method that Automatically Reverse Engineers the Semantics of variables in PLC programs without requiring any domain knowledge. ARES is built on the fact that the Supervisory Control And Data Acquisition (SCADA) system monitors the behavior of PLC using a fixed mapping between the variables of program code and data log, and the data log variables are marked with physical semantics. By identifying the mapping between PLC code and SCADA data (i.e., the code-data mapping), ARES reverse engineers the physical semantics of program variables. ARES also sheds light on the preferred practices in implementing control rules that improve the resistance of PLC programs to semantic attacks. We have experimentally evaluated ARES and the recommended implementation practices on two ICS platforms.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

KEYWORDS

Programmable Logic Controller; Physical Semantics; Control Invariants

ACM Reference Format:
Zeyu Yang, Liang He, Hua Yu, Chengcheng Zhao, Peng Cheng, and Jiming Chen. 2022. Reverse Engineering Physical Semantics of PLC Program Variables Using Control Invariants. In *The 20th ACM Conference on Embedded Networked Sensor Systems (SenSys '22), November 6–9, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3560905.3568521

SenSys '22, November 6–9, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9886-2/22/11…$15.00
https://doi.org/10.1145/3560905.3568521

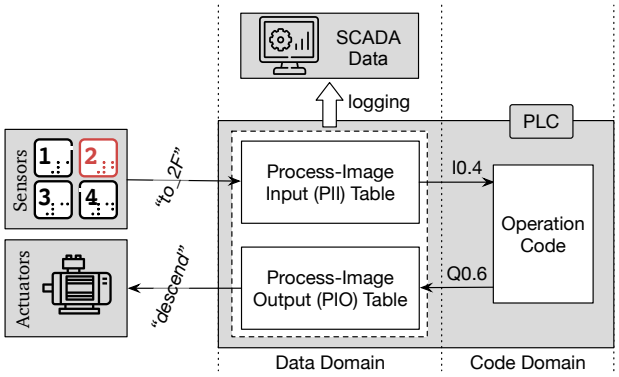


Fig. 1. An exemplary implementation of the control rules of an elevator control system.

1 INTRODUCTION

Programmable Logic Controllers (PLCs) are widely used in Industrial Control Systems (ICSs) to sense and drive physical processes according to defined control rules. A common practice in implementing control rules is to use specific in/output variables in the PLC program to represent the states of given sensing/actuating modules of the system. For example, in the elevator control system shown in Fig. 1, the PLC uses an input variable (e.g., `I0.4`) to accept the instructed destination (e.g., *"to_2F"*) and another output variable (e.g., `Q0.6`) to drive the traction machine (e.g., *"descend"*).

Reverse engineering this one-to-one mapping between program variables and system modules, i.e., identifying physical semantics of program variables, facilitates adversaries to mount *semantic attacks* — attacks that damage targeted system modules by tampering with the PLC program variables with corresponding physical semantics. Semantic attacks are known to be more risky to system operation and attractive to adversaries when compared with attacks that randomly modify PLC states [35, 53]. As a few examples,

- Stuxnet damaged hundreds of centrifuges in a nuclear facility by tampering with the program variable of converter frequency between 2–1410Hz [28];
- BlackEnergy disconnected 27 substations in a power grid by changing the program variables of circuit breakers from 1 to 0 [16];
- an anonymous attacker changed the sodium hydroxide content from 100ppm to 11,100ppm by manipulating the corresponding program variable [22].

However, it is difficult to identify the variable semantics even when adversaries have obtained the program code from the victim PLC, because the variables extracted from PLCs are usually named using memory address (e.g., `I0.4` in Fig. 1) without specifying any semantic information on system modules [5, 25]. As a result, the mounting of semantic attacks usually requires a tedious/manual reconnaissance to identify the variable semantics, e.g., via in-depth examination of the (likely illegitimately accessed) system documents or the long-term observation of system behavior [44]. The lack of automated reverse engineering of variable semantics also impedes the risk assessment of ICSs to semantic attacks.

In this paper, we design ARES, a method that Automatically Reverse Engineers the Semantics of variables in PLC programs without requiring any domain knowledge. ARES is inspired by the fact that the Supervisory Control And Data Acquisition (SCADA) system monitors the behavior of PLC using a fixed mapping between the variables of program code and data log, and the data log variables are marked with physical semantics. By identifying the mapping between PLC code and SCADA data (i.e., the code-data mapping), ARES reverse engineers the physical semantics of program variables. Specifically, ARES identifies this code-data mapping by (i) abstracting the PLC program as a dependency graph in the code domain, (ii) abstracting the SCADA log as a causality graph in the data domain, and (iii) bridging these two cross-domain graphs according to their consistency in describing the control invariants — i.e., how the control commands are generated based on the concomitant sensor readings. Steered by the identified code-data mapping, ARES reverse engineers the physical semantics of program variables as the properties of the mapped data variables. ARES also sheds light on the preferred practices in implementing control rules that improve the resistance of PLC programs to semantic attacks, i.e., the use of commutative variables and decoupling control rules.

We have implemented and evaluated ARES on two ICS platforms, i.e., an Elevator Control System (ECS) and an Ethanol Distillation System (EDS). The experimental results show ARES to reverse engineer the physical semantics of all program variables of the two platforms with 100% accuracy. Atop the thus-identified semantics, we have empirically mounted semantic attacks to 46 (or 10) modules of ECS (or EDS), showing the effectiveness of ARES in damaging the physical processes. We have also validated the effectiveness of the suggested practices in implementing control rules, showing they reduce ARES's ability to reverse engineer the semantics from 100% of the variables to 40.74%.

The contributions of ARES lie in the *integration of PLC code and SCADA data to describe PLC control invariants*, facilitating the reverse engineering of physical semantics of PLC program variables.

- **Cross-domain Characterization of PLC Invariants.** ARES abstracts the control invariants of PLC from the aspects of code domain and data domain, respectively — i.e., as the dependencies of PLC program variables and the causalities of SCADA data variables. These two characterizations of PLC invariants are consistent when describing the same control rules, allowing ARES's identification of code-data mapping. These PLC invariants make ARES pervasively deployable as they exist in all ICSs.
- **Identifying PLC Invariants without Domain Knowledge.** Instead of identifying the cross-domain PLC invariants from the system documents (e.g., electrical control diagrams) or based on the *a priori* domain knowledge (e.g., interactions among sensor readings and control commands), ARES automatically identifies the PLC invariants from SCADA data and PLC code without requiring any human effort — i.e., ARES is easy to deploy.
- **Significantly Reduced Feasible Code-Data Mapping Space.** Essentially, identifying the code-data mapping requires examining all feasible mappings between program and data variables, causing an explosive mapping space. Steered by the customized graphs that characterize PLC invariants, ARES reduces the mapping space by a factor of at least $|O|!/|O|^2$, where $|O|$ denotes the number of output variables in the PLC program.

2 RELATED WORK

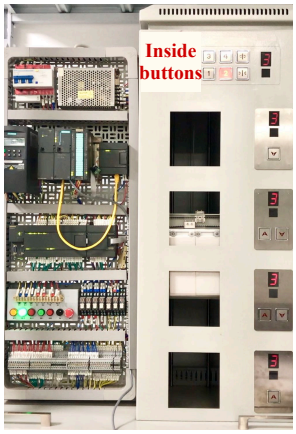
PLCs are well-known to be vulnerable to cyber attacks. Researchers have identified more than 36.7k PLCs that can be directly accessed from the Internet [42]. Symantec also confirmed the risk of intrusions to ICS network [57–59], from which the targeted PLCs can be accessed. The states of these illegitimately accessed PLCs can be tampered with after voiding their weak authentication [14, 24, 48]. For example, by exploiting the vulnerabilities of PLC communication protocols, adversaries can stop the PLC execution [13, 47, 54], manipulate the PLC memory [20, 27, 51, 54], reprogram the PLC control rules [13, 34, 39], hijack the SCADA monitoring traffic [27, 37], etc. Adversaries can even hack the PLC firmware to manipulate PLC states without being detected [1, 11, 31]. The recently proposed PLC binaries de-compilers [38, 41, 49] and fuzzing framework [61] further facilitate the discovery of PLC vulnerabilities.

These attacks/work can be organized into the following two categories according to if they are mounted by exploiting the semantics of PLC programs:

- *Semantic Attacks.* Besides the semantic attacks introduced in Sec. 1 (e.g., the Stuxnet [28] and BlackEnergy [16]), researchers have also measured the impacts of semantic attacks [17, 19, 34, 43, 45, 51, 53, 62] and exploited the behavior of system modules to sneakily damage targeted system modules [27, 31, 32, 52].
- *Non-Semantic Attacks.* Attacks that are mounted without exploiting the semantics of PLC program variables are referred to as non-semantic attacks, such as stopping PLC execution [13, 33, 47, 54], randomly manipulating PLC memory [20, 54], and downloading arbitrary control programs [13, 39].

Semantic attacks allow the manipulation of system operations to intended (and malicious) states, thus being more risky when compared with attacks that randomly modify PLC states [35]. However, the mounting of semantic attacks requires the identification of the physical semantics of PLC program variables. As a pioneer study, SABOT [44] reverse engineered the physical semantics of program variables with the understanding of the manually-obtained behavior of targeted ICSs. MISMO [55] reverse engineered the algorithmic semantics of program variables based on the knowledge of popular control algorithms.

To the best of our knowledge, ARES is the first solution that automatically reverse engineers the physical semantics of program variables without requiring any domain knowledge.



(a) Elevator testbed

```
network
title =
      A(
      A     I     0.4
      A     I     2.6
      O     Q     0.6
      )
      AN    I     2.5
      AN    I     2.2
      =     Q     0.6
network
title =
      A(
      O     I     2.5
      O     Q     1.1
      )
      A     I     2.1
      =     Q     1.1
```

(b) Exemplary PLC control code

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

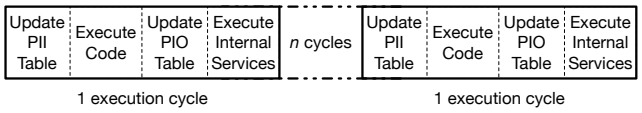


Fig. 3. The typical cyclic procedure of PLC execution.

3 PRELIMINARIES

Below we present the necessary background of PLCs and the basic idea of ARES using our platform of Elevator Control System (ECS), a scaled-down but fully operational four-floor elevator as shown in Fig. 2 and detailed in Appendix A.

3.1 PLCs

PLCs drive the physical processes according to the typical 4-step cyclic procedure, as depicted in Fig. 3.

Update Process-Image Input (PII) Table. At the beginning of each execution cycle, the PLC updates its PII table — which is stored/maintained in the PLC memory — based on the real-time system status (i.e., the measurements of system sensors). For example, the PLC of ECS copies sensor readings (e.g., those describing the status of destination buttons and elevator positions) to the PII table according to the mappings between input variables and sensor modules.

Execute Embedded Control Code. The PLC executes the embedded code with the updated PII table as input to generate the concomitant control commands. Taking Fig. 2(b) as an example, the PLC generates commands of *"descend"* and *"open_door"* using variables `Q0.6` and `Q1.1` as

$$\texttt{Q0.6} \leftarrow ((\texttt{I0.4} \wedge \texttt{I2.6}) \vee \texttt{Q0.6}) \wedge \neg\texttt{I2.5} \wedge \neg\texttt{I2.2}, \quad (1)$$

$$\texttt{Q1.1} \leftarrow (\texttt{I2.5} \vee \texttt{Q1.1}) \wedge \texttt{I2.1}, \quad (2)$$

when the door status of *"door_is_open"* and *"door_is_not_opened"*, the destination button of *"to_2F"*, and elevator positions of *"at_2F"* and *"at_3F"*, are specified in the PII table by variables {`I2.2`, `I2.1`, `I0.4`, `I2.5`, `I2.6`}, respectively.

Update Process-Image Output (PIO) Table. The PLC writes the above-generated control commands to its PIO table and outputs them to the specified actuators at the end of each code execution, e.g., issuing the command of *"descend"* to ECS's traction machine.

Execute Internal Services. After outputting the control commands, the PLC executes the internal services, such as reporting the updated PII/PIO tables to SCADA for system logging/monitoring.

As can be seen, the control rules defined in PLC programs are directly reflected in the PII/PIO tables and then recorded in SCADA logs, laying the foundation of ARES. Also, it is typical for SCADA to record all the sensor readings and control commands in PII/PIO tables to achieve thorough system monitoring.

3.2 Threat Model

We consider adversaries who aim to mount semantic attacks against targeted ICS modules, by identifying program variables' semantics and modifying the concomitant control rules. Adversaries mount semantic attacks with the following abilities that are commonly adopted in the literature and observed in real life.

1) Obtaining the code of PLC program. Most commercial PLCs allow the runtime extraction of their control code after passing the weak authentication [3, 24], facilitating adversaries to obtain the control code once connected to the victim PLC from the public Internet [42] or the intruded Intranet [57–59]. The thus-obtained PLC binary code can be easily translated to PLC programming languages [5, 9], where the variables are named using memory address without specifying any information on system modules (see Fig. 2(b)).

2) Obtaining the SCADA log. SCADA is well-known to be insecure due to the lack of preventive measures such as firewalls, virtual private networks, strong authentication and encryption [2, 4, 10, 18], making the historical SCADA log illegitimately accessible from the Internet [26, 56]. SCADA log is usually labeled with module-relevant information (e.g., *"descend"*) to ensure their interpretability to system operators.

3) Modifying the states of variables in PLC program. This is because most commercial PLCs support online reprogramming and debugging via insecure communication protocols [14, 24, 48], e.g., the weak authentication in Rockwell's PCCC protocol allows adversaries to identify cryptographic keys and further reprogram the control rules [24].

Empirical Corroboration of Adversary Abilities. We have experimentally validated the above abilities on our ECS platform and PLCs from multiple vendors (e.g., Siemens, Allen Bradley, Schneider, Wago), using a similar approach as with BlackEnergy [16], i.e., sending from the public Internet a spear-phishing email with a malicious Excel file as the attachment. Examples of the obtained PLC program code and SCADA log are shown in Fig. 2(b) and Table 1, respectively.

3.3 Motivational Example

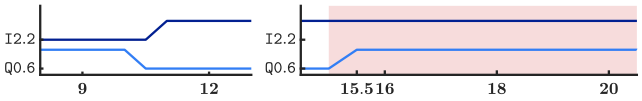
Let us consider a semantic attack against ECS running the exemplary PLC code in Fig. 2(b), in which the adversaries, with the abilities listed in Sec. 3.2, aim to cause a life-threatening accident, i.e., driving the traction machine descending when the door is open.

To complete this attack, adversaries need to identify which program variables represent the command of *"descend"* and the sensor

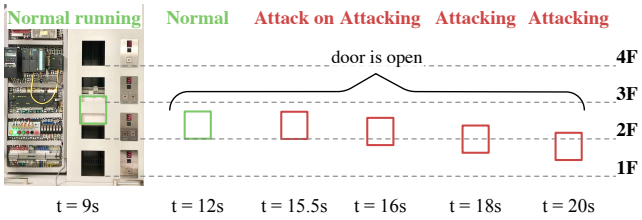
Table 1: SCADA log of a simplified elevator system running the code in Fig. 2(b).

Id.	ValueName	Timestamp†	RealValues
v1	*"descend"*	14:10 – 14:30	[1,1,1,0,0,0,0, . . .]
v2	*"open_door"*	14:10 – 14:30	[0,0,0,1,1,1,1, . . .]
v3	*"to_2F"*	14:10 – 14:30	[0,1,1,0,0,0,0, . . .]
v4	*"at_2F"*	14:10 – 14:30	[0,0,0,1,1,1,1, . . .]
v5	*"at_3F"*	14:10 – 14:30	[0,0,0,0,0,0,0, . . .]
v6	*"door_is_open"*	14:10 – 14:30	[0,0,0,0,0,1,1, . . .]
v7	*"door_is_not_opened"*	14:10 – 14:30	[1,1,1,1,1,1,1, . . .]

† Timestamp is abbreviated from "21.12.2021 14:10 – 21.12.2021 14:30".



(a) SCADA logs of *"descend"* (i.e., `Q0.6`) and *"door_is_open"* (i.e., `I2.2`)



(b) sequential snapshots of the attack consequence

Fig. 4. An exemplary semantic attack against ECS.

reading of *"door_is_open"*. By cross-checking the PLC program (i.e., Fig. 2(b)) with the obtained SCADA data (i.e., the RealValues in Table 1), the adversary finds only when data variables *"descend"*/*"door_is_open"* are mapped to code variables `Q0.6`/`I2.2`, the SCADA data agrees with the control rules defined by the PLC program. This way, the adversary concludes that code variables of `Q0.6`/`I2.2` are the attack vectors to mount the intended semantic attack. Note that the executability of the obtained PLC program allows the adversary to complete the above consistency checking without being required to explicitly interpret the control rules as Eqs. (1) and (2). The adversary then mounts the attack by bit-flipping the state of `Q0.6` from 0 to 1 whenever the state of `I2.2` is 1. Fig. 4 shows an example of this attack, in which the adversary modifies `Q0.6`'s state to 1 at t=15.5s when the door is open. The elevator was then driven away from 2F with an open door starting from t=16s — the attack takes effect within as short as 0.5s.

The above successful attack inspires ARES to reverse engineer the physical semantics of PLC variables by constructing the code-data mapping, based on the consistency of PLC code and SCADA data in representing the control logic. However, the code-data mapping has an explosive consistency checking space, which ARES reduces by only considering the code-data mapping of the same control logic using the control invariants.

4 CONTROL INVARIANTS

Given a PLC, its control invariants — i.e., how to generate control commands based on the concomitant sensor readings — can be captured using three customized graphs.

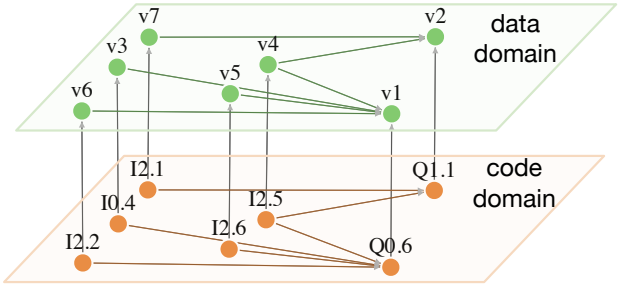


Fig. 5. The $\{G^{code}, G^{data}, G^{cross}\}$ of a simplified elevator system running the code in Fig. 2(b).

- Graph $G^{code}(\bar{V}, \bar{E})$ abstracts the dependencies between input and output variables of the PLC program, describing the control rules in the code domain.
- Graph $G^{data}(V, E, W)$ abstracts the causalities between sensor readings and control commands in the SCADA log, describing the control rules in the data domain.
- Graph $G^{cross}(\tilde{V}, \tilde{E}, \tilde{W})$ crossing the domains captures the supervisory relationship between SCADA and PLC, describing the consistency between G^{code} and G^{data}.

Fig. 5 visualizes the three graphs of a simplified elevator system running the PLC code in Fig. 2(b).

4.1 Code Graph $G^{code}(\bar{V}, \bar{E})$

As an assembly-like language defined by the IEC 61131-3 standard, the instruction list (IL) is an intermediate language to which other languages can be compiled automatically [5, 25]. The control invariants in the IL code obtained (or further translated) from a running PLC can be captured using a dependency graph $G^{code}(\bar{V}, \bar{E})$, where:

- $\bar{V} = \{\mathtt{I}, \mathtt{O}\}$ consists of an input node set `I` and an output node set `O`, representing all the in/output variables.
- $\bar{E} = \{e_i^o\}$ is the set of directed edges connecting nodes in `I` to nodes in `O`, representing the dependencies from input to output variables — i.e., an edge $e_{i_p}^{o_q}$ exists in $\bar{E}$ when the output node o_q is determined by input node i_p.

Edges Set $\bar{E}$. Although different vendors may implement IL differently, they follow a similar syntax according to IEC 61131-3 for generality [12, 36]. We use the statement list (STL), an implementation of IL by Siemens, to explain the edge set $\bar{E}$. An STL program consists of a sequence of statements, each of which is comprised of an *operator* and an *operand*, representing one specific operation of the PLC. Specifically, the *operator* describes the operation (e.g., "A" for AND) that the PLC executes, and the *operand* provides the information (e.g., variable `I2.2`) that needs to be processed. The assignment operator (e.g., "=") assigns the processed result to the corresponding operand (e.g., variable `Q0.6`). This way, the dependencies among program variables are defined.

Note that instead of capturing explicitly/exhaustively the dependencies among all program variables (including the dependencies among output variables) [46, 62], $\bar{E}$ only keeps the edges that connect the input variables to output variables. This way, the output nodes in G^{code} will not be reachable from each other, facilitating the construction of G^{cross} (as we explain in Sec. 5.3). Also note the

above abstraction of dependencies (and hence identification of $\bar{\mathtt{E}}$) is also applicable to other IL implementations, e.g., by replacing the customized identifiers thereof.

4.2 Data Graph $\mathrm{G}^{\mathrm{data}}(\mathtt{V},\mathtt{E},\mathtt{W})$

The control invariants in the SCADA log can be captured using a causality graph $\mathrm{G}^{\mathrm{data}}(\mathtt{V},\mathtt{E},\mathtt{W})$, where:

- $\mathtt{V}=\{\mathtt{S},\mathtt{U}\}$ consists of a sensor node set $\mathtt{S}$ and a command node set $\mathtt{U}$, representing all data variables in the SCADA log. The necessary monitoring of SCADA makes $|\mathtt{S}|\geq|\mathtt{I}|$ and $|\mathtt{U}|\geq|\mathtt{O}|$.
- $\mathtt{E}=\{e_s^u\}$ is the set of directed edges connecting nodes in $\mathtt{S}$ to nodes in $\mathtt{U}$, representing the causalities between sensor readings and control commands — i.e., an edge $e_{s_k}^{u_j}$ exists in $\mathtt{E}$ when the PLC generates command u_j based on the states of s_k.
- $\mathtt{W}=\{w_s^u\}$ is the set of edge weights representing the causality strength of node pairs $\{s_k,u_j\}$s in $\mathtt{E}$.

Weight Set $\mathtt{W}$. PLCs generate control commands based on the concomitant sensor readings according to the defined control rules, making the control commands correlate strongly with the corresponding sensor readings. However, one control command (e.g., *"descend"*) is usually determined by multiple sensor readings (e.g., *"to_2F"*, *"at_3F"*, *"door_is_open"*), as well as the historical states of the command itself. To decouple these causalities when generating the control command u_j, the mutual information between sensor reading s_k and u_j conditioned on all other factors can be utilized to define the weight of edge $e_{s_k}^{u_j}$ in $\mathtt{E}$. Formally, given the node u_j's neighbor set $\{s_1,\ldots,s_k,s_m\}$, the weight $w_{s_k}^{u_j}$ is defined as

$$w_{s_k}^{u_j}=I(\mathcal{X};\mathcal{Y}\mid\mathcal{Z})/I(\widetilde{\mathcal{X}};\mathcal{Y}\mid\mathcal{Y}^a),\tag{3}$$

where (i) $\mathcal{X}$ is the distribution of sensor reading $\mathbf{s}_k$; (ii) $\mathcal{Y}$ is the distribution of control command $\mathbf{u}_j$; (iii) $\mathcal{Z}$ denotes the joint distribution of all other factors $\mathbf{z}:=[\mathbf{s}_1,\ldots,\mathbf{s}_m,\mathbf{u}'_j]$ and $\mathbf{u}'_j$ is the history of $\mathbf{u}_j$ with one sample lag; (iv) $\widetilde{\mathcal{X}}$ denotes the joint distribution of all sensor readings $[\mathbf{s}_1,\ldots,\mathbf{s}_{|\mathtt{S}|}]$; (v) $\mathcal{Y}^a$ denotes the distribution of historical control command $\mathbf{u}'_j$. The conditional mutual information $I(\mathcal{X};\mathcal{Y}\mid\mathcal{Z})$ is calculated as $H(\mathcal{X},\mathcal{Z})+H(\mathcal{Y},\mathcal{Z})-H(\mathcal{Z})-H(\mathcal{X},\mathcal{Y},\mathcal{Z})$, where $H(\cdot)$ denotes the entropy function.

Observing the fact that historical sensor readings will also be used to generate the control commands, such as the control rules of the Proportional-Integral-Derivative (PID) controller [8], we further extend the weight in Eq. (3) by replacing the distribution of $\mathcal{X}/\mathcal{Z}/\widetilde{\mathcal{X}}$ with $\mathcal{X}^h/\mathcal{Z}^h/\widetilde{\mathcal{X}}^h$, respectively, i.e.,

$$w_{s_k}^{u_j}=I(\mathcal{X}^h;\mathcal{Y}\mid\mathcal{Z}^h)/I(\widetilde{\mathcal{X}}^h;\mathcal{Y}\mid\mathcal{Y}^a),\tag{4}$$

where (i) $\mathcal{X}^h$ denotes the joint distribution of sensor readings $[\mathbf{s}_k^{[1:L-L^h]},\ldots,\mathbf{s}_k^{[1+L^h:L]}]$, and $\{L,L^h\}$ are the duration of SCADA logs and the number of s_k's historical samples used to generate control command u_j, respectively; (ii) $\mathcal{Z}^h$ denotes the joint distribution of $[\mathbf{s}_1^{[1:L-L^h]},\ldots,\mathbf{s}_1^{[1+L^h:L]},\ldots,\mathbf{s}_m^{[1:L-L^h]},\ldots,\mathbf{s}_m^{[1+L^h:L]},\mathbf{u}'_j]$; (iii) $\widetilde{\mathcal{X}}^h$ denotes the joint distribution of all sensor readings $[\mathbf{s}_1^{[1:L-L^h]},\ldots,\mathbf{s}_1^{[1+L^h:L]},\ldots,\mathbf{s}_{|\mathtt{S}|}^{[1:L-L^h]},\ldots,\mathbf{s}_{|\mathtt{S}|}^{[1+L^h:L]}]$.

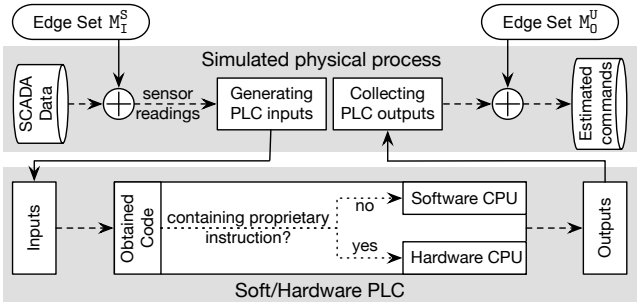


Fig. 6. Overview of PLC-Twin, a hardware-in-the-loop system that reconstructs a control process with the obtained PLC program and SCADA log.

Note that the weight defined in Eq. (4) applies to both binary and continuous SCADA data, by calculating the corresponding distributions via counting (for binary data) or binning (for continuous data) [40].

4.3 Cross-Domain Mapping $\mathrm{G}^{\mathrm{cross}}(\tilde{\mathtt{V}},\tilde{\mathtt{E}},\tilde{\mathtt{W}})$

Given graphs $\mathrm{G}^{\mathrm{code}}(\bar{\mathtt{V}},\bar{\mathtt{E}})$ and $\mathrm{G}^{\mathrm{data}}(\mathtt{V},\mathtt{E},\mathtt{W})$, their consistency in describing the PLC control invariants can be captured using a code-data mapping $\mathrm{G}^{\mathrm{cross}}(\tilde{\mathtt{V}},\tilde{\mathtt{E}},\tilde{\mathtt{W}})$, where:

- $\tilde{\mathtt{V}}=\{\bar{\mathtt{V}},\mathtt{V}\}$ consists of a node set $\bar{\mathtt{V}}=\{\mathtt{I},\mathtt{O}\}$ in the code domain and a node set $\mathtt{V}=\{\mathtt{S},\mathtt{U}\}$ in the data domain, as explained in Sec. 4.1 and 4.2;
- $\tilde{\mathtt{E}}=\{\mathtt{M}_\mathtt{O}^\mathtt{U},\mathtt{M}_\mathtt{I}^\mathtt{S}\}$, where $\mathtt{M}_\mathtt{O}^\mathtt{U}=\{m_o^u\}$ is the set of cross-domain edges connecting each output node $o\in\mathtt{O}$ to a command node $u\in\mathtt{U}$, and $\mathtt{M}_\mathtt{I}^\mathtt{S}=\{m_i^s\}$ is the set of cross-domain edges connecting each input node $i\in\mathtt{I}$ to a sensor node $s\in\mathtt{S}$. An edge $m_{o_q}^{u_j}$ (or $m_{i_p}^{s_k}$) exists in $\mathtt{M}_\mathtt{O}^\mathtt{U}$ (or $\mathtt{M}_\mathtt{I}^\mathtt{S}$) when the SCADA records the states of o_q (or i_p) as data variable u_j (or s_k). The one-to-one logging scheme — using one SCADA data to record one code variable — makes both the out-degree of nodes in $\bar{\mathtt{V}}$ and the in-degree of nodes in $\mathtt{V}$ equal to 1.
- $\tilde{\mathtt{W}}=\{w_o^u\}$ is the set of weights for each edge $m_{o_q}^{u_j}$ in $\mathtt{M}_\mathtt{O}^\mathtt{U}$, representing the consistency level of $\mathrm{G}^{\mathrm{code}}$ and $\mathrm{G}^{\mathrm{data}}$ in describing the control invariants of o_q, i.e., the accuracy of recovering states evolution of code variable o_q using data variable u_j.

Weight Set $\tilde{\mathtt{W}}$. By examining the PLC code with sensor readings $\{\mathbf{s}_1,\ldots,\mathbf{s}_k\}$ according to the mapping edges $m_{i_p}^{s_k}$s in $\mathtt{M}_\mathtt{I}^\mathtt{S}$, the concomitant control commands estimation $\{\hat{\mathbf{u}}_1,\ldots,\hat{\mathbf{u}}_j\}$ can be obtained according to the mapping edges $m_{o_q}^{u_j}$s in $\mathtt{M}_\mathtt{O}^\mathtt{U}$. By cross-checking the control commands of $\{\mathbf{u}_1,\ldots,\mathbf{u}_j\}$ with $\{\hat{\mathbf{u}}_1,\ldots,\hat{\mathbf{u}}_j\}$, the consistency level of $\mathrm{G}^{\mathrm{code}}$ and $\mathrm{G}^{\mathrm{data}}$ in describing the control invariants of o_q is calculated as

$$w_{o_q}^{u_j}=\rho(\mathbf{u}_j,\hat{\mathbf{u}}_j)/(\left\|\mathbf{u}_j-\hat{\mathbf{u}}_j\right\|_1/|\mathbf{u}_j|),\tag{5}$$

where $\rho(\cdot)$ denotes the *correlation coefficient* of two time series, $\|\cdot\|_1$ denotes the L1-norm of a vector, and $|\cdot|$ denotes the length of a vector.

We design PLC-Twin, a hardware-in-the-loop PLC control system to realize the above cross-checking without requiring adversaries to understand the explicit programming rules (which may

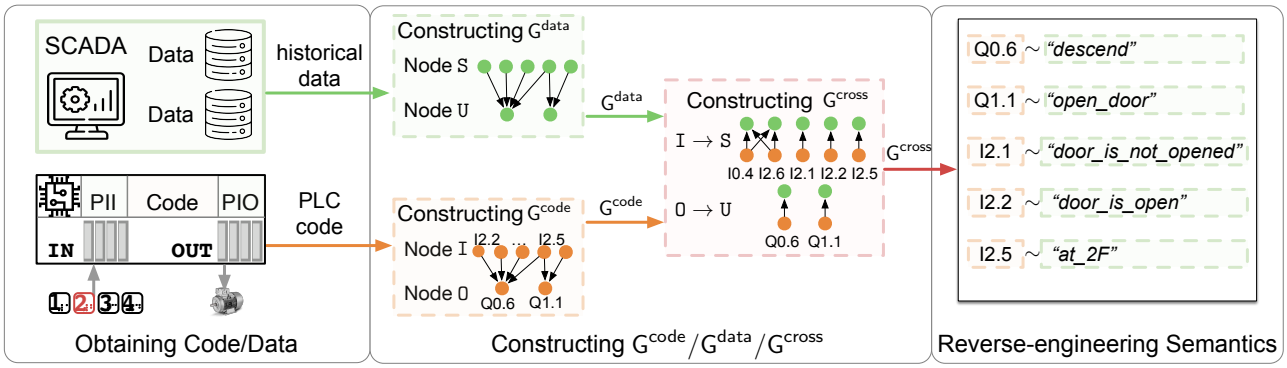


Fig. 7. Overview of ARES.

differ among different PLC vendors). PLC-Twin allows the regeneration of control commands by executing the PLC program with the sensor readings in the SCADA log and mapping edges in $[\mathsf{M}_0^{\mathsf{U}}, \mathsf{M}_{\mathsf{I}}^{\mathsf{S}}]$ as inputs. Specifically, PLC-Twin consists of a simulated physical process and a soft/hardware PLC code execution, as shown in Fig. 6.

- *Simulated physical process.* The simulated process reconstructs a physical process. It selects the sensor readings $\{\mathbf{s}_1(t), \ldots, \mathbf{s}_k(t)\}$ from the SCADA data, and sends them at time t to the input variables of PLC-Twin according to the mapping defined in $\mathsf{M}_{\mathsf{I}}^{\mathsf{S}}$. The estimated control commands $\{\hat{\mathbf{u}}_1(t), \ldots, \hat{\mathbf{u}}_j(t)\}$ are then obtained from the output variables of PLC-Twin according to the mapping defined in $\mathsf{M}_0^{\mathsf{U}}$.
- *Soft/Hardware PLC.* PLC-Twin reconstructs the control system by executing the obtained PLC program with the above inputs using either software- or hardware-based PLCs. Specifically, the software-based PLCs are implemented based on Awlsim [15] and OpenPLC [7] to support the quick reconstruction of control systems programed by non-proprietary instructions, and the hardware PLCs support the execution of all instructions but need a longer time to complete the control system reconstruction. Note that the hardware PLCs should not be connected to any input modules to ensure their PII table is only written by the simulated physical process.

5 DESIGN OF ARES

Fig. 7 depicts an overview of ARES: taking the obtained PLC program and SCADA log of a given ICS as inputs, ARES constructs $\{\mathsf{G}^{\mathsf{code}}, \mathsf{G}^{\mathsf{data}}\}$, identifies their mapping $\mathsf{G}^{\mathsf{cross}}$, and reverse engineers the semantics of program variables according to their uniquely mapped data variables in $\mathsf{G}^{\mathsf{cross}}$. In the following, we elaborate the design of ARES using the platform of ECS as an example.

5.1 Constructing $\mathsf{G}^{\mathsf{code}}(\bar{\mathsf{V}}, \bar{\mathsf{E}})$

ARES constructs $\mathsf{G}^{\mathsf{code}}(\bar{\mathsf{V}}, \bar{\mathsf{E}})$ based on the program code. Note that instead of parsing the PLC program to explicit control rules, identifying related input variables for each output variable is sufficient to construct $\mathsf{G}^{\mathsf{code}}$. Let us take the Siemens STL, which is programmed in ECS, as an example to explain the construction of $\mathsf{G}^{\mathsf{code}}$.

5.1.1 Identifying Node Set $\bar{\mathsf{V}}$

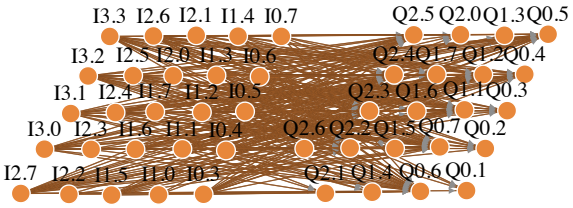


Fig. 8. Constructed $\mathsf{G}^{\mathsf{code}}$ of ECS.

Variables in the unannotated STL are declared in a pre-defined format, consisting of an address identifier (including a memory identifier "I"/"Q"/"M"/"T"/"C"/"DB" and a size prefix "B"/"W"/"D") and an absolute location (e.g., 0.1 for byte 0, bit 1), such as variables I2.2 and Q0.6 in Fig. 2(b). ARES identifies set I (or O) by finding variables with address identifier "I" (or "Q").

5.1.2 Identifying Edge Set $\bar{\mathsf{E}}$

For each output node $o_q \in \mathsf{O}$, ARES identifies its dependent input nodes from I. ARES first segments the given STL code as basic programming blocks, including organization blocks, function blocks, and functions. These blocks begin with the keyword "BEGIN" and end with the keyword "END_ORGANIZATION_BLOCK", "END_FUNCTION_BLOCK" and "END_FUNCTION". Each block is an individual control unit and can be divided into programming networks, which begins with the keyword "NETWORK" followed by "TITLE=" in the next line (see Fig. 2(b)) and consists of several statements. Within a programming network, statements with the assignment operator (i.e., "=", "S", "R", "T", "TAR1", "TAR2") define dependencies of the operand thereof — i.e., the operand for an assignment operator depends on all the previous non-assignment operands. For example, the variable Q1.1 in Fig. 2(b) depends on variables I2.1, I2.5 and Q1.1. This way, dependencies in each network are extracted. Given the fact that PLCs update the output variables based on the states of input variables, ARES recursively substitutes o_q's dependent nodes $o_{q'}$ (if exists) with dependent nodes of $o_{q'}$, where $q' \neq q$, until all dependent nodes of o_q are from set I or itself. Self-loops are discarded in $\mathsf{G}^{\mathsf{code}}$. Fig. 8 visualizes the constructed $\mathsf{G}^{\mathsf{code}}$ of ECS.

5.2 Constructing $\mathsf{G}^{\mathsf{data}}(\mathsf{V}, \mathsf{E}, \mathsf{W})$

ARES constructs the data-domain graph $\mathsf{G}^{\mathsf{data}}(\mathsf{V}, \mathsf{E}, \mathsf{W})$ using the system's historical SCADA log, which is readily available in ICSs, e.g., the WinCC-based SCADA stores more than one day's data log by

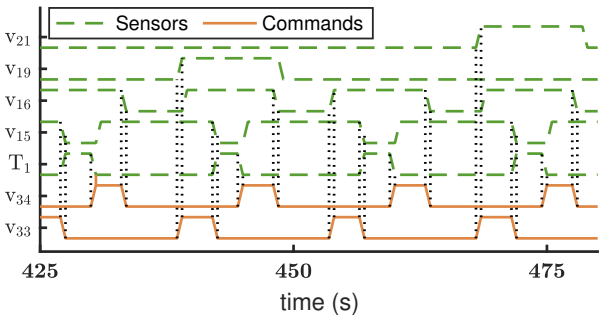
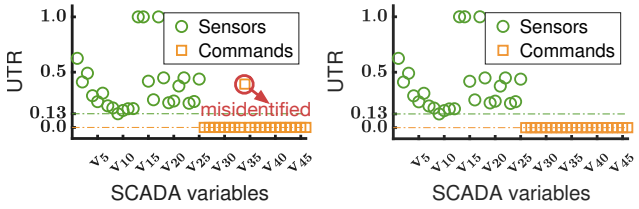


Fig. 9. STs of a control command always occur simultaneously with STs in sensor readings.



(a) UTR distribution (b) UTR distribution

Fig. 10. All the UTRs of sensor readings in ECS are larger than 0.13, while those of control commands are 0.

default [6]. We will experimentally explore ARES's robustness to the duration of the available SCADA log in Sec. 6. Note that different from existing approaches that abstract the causalities among SCADA data based on the *a priori* knowledge on if a given data variable represents a sensor reading or control command [21, 29, 50, 62], ARES constructs G^{data} without requiring any domain knowledge.

5.2.1 Identifying Node Set V

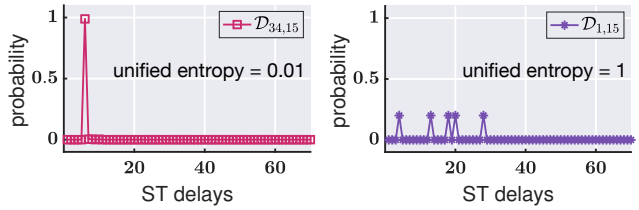
Basic Principle. ARES automatically identifies the sensor readings and control commands, which cannot be directly differentiated in the SCADA log, based on the following two observations on their *state transition* (ST), i.e., the change of their states.

- The STs of control commands always have simultaneous STs in sensor readings (see Fig. 9), because of the sequential execution of the PLC procedure illustrated in Fig. 3: (i) the updated PII table remains unchanged for one cycle; (ii) the concomitantly issued commands are stored in the PIO table; (iii) the SCADA reads/records both the PII and PIO tables at the same time.
- The STs of sensor readings do not always have simultaneous STs in control commands, because: (i) a sensor determines the corresponding control command conditioned on other system states, e.g., the STs of sensor *"to_2F"* only determine STs of command *"descend"* when the status of *"door_is_open"* is *false*; (ii) the inevitable mechanical inertia causes a lag between the sensor's ST and that of the concomitant command, e.g., the inertia of traction machines makes STs of *"at_2F"* occur after STs of *"descend"*.

Define the *unique-transition ratio* (UTR) of a data variable in the SCADA log as:

$$UST_r = ST_r \setminus (ST_1 \cup \cdots \cup ST_{r'} \cup \cdots \cup ST_{|\mathsf{S}|+|\mathsf{U}|}), \quad (6)$$

$$UTR_r = |UST_r|/|ST_r|, \quad (7)$$



(a) ST delay of command *"close_door"* (b) ST delay of sensor *"to_1F"*

Fig. 11. ST delays of timer-related commands are centrally distributed, while those of sensors are randomly distributed.

where ST_r denotes the set of timestamp t of STs for variable v_r (i.e., the state of v_r at time t changes from that at time $(t-1)$), and UST_r denotes the unique STs of v_r that do not have simultaneous STs in all other variables. ARES identifies sensor readings and control commands based on their UTRs: as empirically corroborated in Fig. 10(a), the UTRs of all sensor readings are larger than 0.13, while most of those for control commands are 0. We use a threshold of $\alpha = 0.1$ to differentiate the sensors and commands in our implementation of ARES. We will elaborate on the misidentified command in Fig. 10(a) later.

Sensor Node Set S. It is typical that a control system will deploy redundant sensors to monitor the key modules to ensure system safety/robustness. To avoid capturing the invalid simultaneous STs in sensor readings caused by this redundancy, ARES first identifies the redundant sensor pairs — variables having large correlation coefficients (e.g., >0.9) are viewed as mutually redundant, and captures them in set $\acute{\mathsf{S}}$. ARES then calculates the unique-transition ratio UTR_r for each variable v_r in the SCADA log. The redundant sensors of v_r will be discarded when calculating the corresponding UTR_r. For variables without STs, ARES assigns their UTR as 1 and treats them as barely changed modules (e.g., the sensor of *"on_fire"*) in constant set $\mathsf{C} \subset \mathsf{S}$. After calculating the UTR of all variables as Eq. (7) for SCADA data, ARES concludes the sensor set S as the variables whose UTR are larger than the pre-defined threshold α.

Command Node Set U. The remaining variables are concluded as command nodes in U. However, this may misidentify the commands whose states change following a preset timer, which is widely used in the PLC control logic. Taking the traces of ECS in Fig. 5 as an example, the command of *"close_door"* (v_{34}) will only be triggered when the sensor reading *"door_is_not_opened"* (v_{15}) becomes *false* and lasts for 3 seconds (T_1)[1]. The delayed STs of a timer-related command will induce a higher UTR, and further lead to the misidentification (e.g., the outlier in Fig. 10(a)). As mitigation, ARES further identifies the timer-related command node from the identified S based on the fact that the ST delay caused by timers remains constant, with the following three steps.

a) **Calculating ST Delay Distribution $\mathcal{D}$.** For each sensor $s_k \in \mathsf{S}$, ARES calculates its ST delay distributions $\mathcal{D}_{k,r}$ by comparing each ST in the set UST_k (i.e., $UST_k(n)$, $n \in \mathbb{R}^+$) with STs of all other nodes $v_r \in \mathsf{S}$ as

$$\mathcal{D}_{k,r}(n) = \min(UST_k(n) - ST_r)^+, \quad (8)$$

[1]The timer T1 is not recorded by the SCADA.

Table 2: Weighting ECS's causalities using different metrics.

Control Commands	Weight Metric	Sensor Readings v_{15}	v_{16}	v_{18}	v_{19}	v_{20}	v_{21}
		●	○	●	●	●	●
v_{33}	Eq.(3)	0.471	0.160	0.096	0.167	0.200	0.120
	Eq.(4)	0.347	0.003	0.116	0.168	0.203	0.127
		●	●	○	○	○	○
v_{34}	Eq.(3)	0.156	0.572	0.009	0.003	0.010	0.002
	Eq.(4)	0.398	0.355	0.003	0.001	0.004	0.001

● (or ○) denotes the existence (or non-existence) of causal relation.

where $\min(\cdot)^+$ returns the minimum positive value. The constant ST delay of a timer-related command induces a centrally distributed $\mathcal{D}_{k,r}$, as shown by $\mathcal{D}_{34,15}$ (i.e., the ST delay of *"close_door"* since *"door_is_not_opened"*) in Fig. 11(a). On the contrary, the delay distribution of sensor readings has a randomly distributed $\mathcal{D}_{k,r}$, as corroborated in Fig. 11(b).

b) **Calculating Unified Entropy.** The difference between the centrally and randomly distributed $\mathcal{D}_{k,r}$ can be captured by their entropy. Specifically, ARES calculates the unified entropy of $\mathcal{D}_{k,r}$ as

$$UE_{k,r} = H(\mathcal{D}_{k,r})/\log_2(|\mathcal{D}_{k,r}|), \tag{9}$$

where $\log_2(|\mathcal{D}_{k,r}|)$ is the maximum entropy of a distribution containing $|\mathcal{D}_{k,r}|$ instances. As shown in Figs. 11(a) and 11(b), the centrally distributed $\mathcal{D}_{34,15}$ has a much smaller unified entropy (i.e., 0.01) than randomly distributed $\mathcal{D}_{1,15}$ (i.e., 1).

c) **Identifying Timer-Related Commands.** ARES identifies the timer-induced delay distributions as the one with small $UE_{k,r}$ (e.g., 0.1) and the corresponding $\{v_r, s_k\}$s as possible sensor-command pairs — i.e., s_k is a possible timer-related command that is (temporarily) misidentified as sensor node and v_r is the sensor node that determines the states of s_k. For each candidate of timer-related command s_k, ARES (i) identifies the preset delay of s_k from v_r as the delay instance with the highest probability, denoted as $\mathcal{D}_{k,r}^{\text{mode}}$ (e.g., $\mathcal{D}_{34,13}^{\text{mode}} = 6$ in Fig. 11(a)); (ii) updates the UTR of s_k as UTR'_k using Eq. (6) and Eq. (7) by replacing the ST_r thereof with $UST_k - \mathcal{D}_{k,r}^{\text{mode}}$; (iii) corrects the (misidentified) sensor node s_k as a timer-related command node in U if $UTR'_{s_k} < \alpha$.

This way, ARES identifies sensor set S and command set U of a given ICS, as shown in Fig. 10(b) as an example, where the misidentified timer-related command in Fig. 10(a) is corrected.

5.2.2 Identifying Edge Set E

As corroborated in Table 2, the weight defined in Eq. (4) quantifies the causalities among data variables more accurately than that in Eq. (3), because in ECS: (i) command v_{33} has no causal relationship with v_{16}; (ii) command v_{34} has closer causality with v_{15} when compared with v_{16}. ARES uses a structure learning method with the weight defined in Eq. (4) to identify the edge set E connecting nodes in S and nodes in U. Control rules are usually designed incrementally (e.g., as in PID algorithm) due to PLC's limited computation capacity, steering ARES to use $L^h = 1$ in Eq. (4).

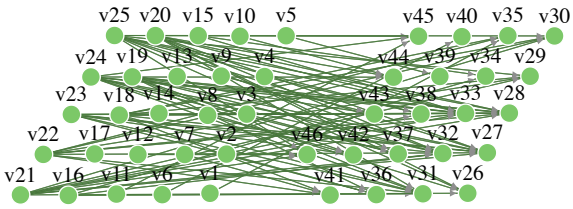


Fig. 12. Constructed G^{data} of ECS with edge confidence θ=0.1.

Starting from an empty edge set E, ARES recursively identifies the edge set E^{u_j} for each command u_j as those with weight (as defined Eq. (4)) larger than a pre-defined confidence level θ (e.g., $\theta = 0.1$). Specifically, ARES identifies the edge set E^{u_j} by:

(1) *Updating Edge Weights.* Based on existing edges in E^{u_j} (initialized as ∅), ARES abstracts the identified causal sensor nodes as S′, calculates the conditional joint distribution $\mathcal{Z}^h$ based on the readings of sensors in S′ according to Eq. (4), and then updates all the weights of potential edges $e^{u_j}_{s_k \in S \setminus S'}$ using Eq. (4).

(2) *Expanding Edge Set* E^{u_j}. ARES (i) identifies the edge $e^{u_j}_{s_k^*}$ which has the maximum weight among all the potential edges $e^{u_j}_{s_k \in S \setminus S'}$, (ii) examines if the maximum weight $w^{u_j}_{s_k^*}$ is larger than a given confidence level θ, and (iii) adds $e^{u_j}_{s_k^*}$ to E^{u_j} and s_k^* to S′ if yes. If s_k^* has redundant sensor $\acute{s}_k^*$ in $\acute{S}$, the corresponding edges $e^{u_j}_{\acute{s}_k^*}$ and node $\acute{s}_k^*$ will also be added.

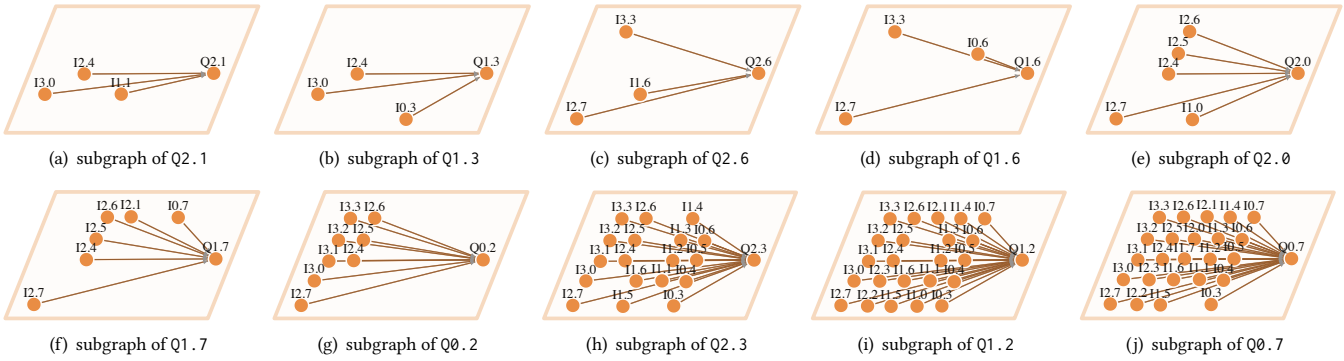
(3) *Verifying Expanded* E^{u_j}. After adding new edges/nodes to E^{u_j}/S′, ARES further verifies the weights of previously identified edges in E^{u_j}, because their weights (as defined in Eq. (4)) may decrease due to the newly added sensor s_k^*. ARES updates the weight of each edge $e^{u_j}_{s_k} \in E^{u_j}$ by removing s_k (and the corresponding $\acute{s}_k$ if any) from $\mathcal{Z}^h$, which is calculated based on all readings of sensors in the newly appended S′. The edges whose updated weight is smaller than θ are removed from E^{u_j}.

(4) *Stop Condition.* ARES repeats the above steps until the maximum weight of all potential edges $e^{u_j}_{s_k \in S \setminus S'}$ is smaller than θ.

Note that ARES replaces $L^h = 1$ with $L^h = 1 + \mathcal{D}_{k,r}^{\text{mode}}$ in Eq. (4) when calculating the weights of timer-related sensor-command edges. After identifying the E^{u_j}s for all nodes $u_j \in U$, ARES constructs the edge set E by aggregating all E^{u_j}s. Fig. 12 visualizes the thus-constructed G^{data} of ECS with an edge confidence θ of 0.1. We will experimentally evaluate the impact of θ on ARES in Sec. 6.

5.3 Constructing G^{cross}

ARES constructs a code-data mapping $G^{\text{cross}}(\tilde{V}, \tilde{E}, \tilde{W})$ that bridges $G^{\text{code}}(\bar{V}, \bar{E})$ and $G^{\text{data}}(V, E, W)$, where $\tilde{V} = \{\bar{V}, V\}$. Note that the structural differences between G^{code} and G^{data} (e.g., the edge weights) limit the classic graph matching algorithms to construct the accurate cross-domain mapping because they mainly utilize the isomorphism of structure [23, 30]. In comparison, ARES identifies the cross-domain edge set $\tilde{E}$ by finding the mapping between G^{code}



(a) subgraph of Q2.1 (b) subgraph of Q1.3 (c) subgraph of Q2.6 (d) subgraph of Q1.6 (e) subgraph of Q2.0

(f) subgraph of Q1.7 (g) subgraph of Q0.2 (h) subgraph of Q2.3 (i) subgraph of Q1.2 (j) subgraph of Q0.7

Fig. 13. Exemplary subgraphs of ECS's G^{code} in Fig 8 (isolated nodes in each subgraph are not shown for figure clarity).

and G^{data} that maximizes their consistency in describing the control invariants, i.e., having the maximum cross-domain weight as defined in Eq. (5).

Essentially, ARES needs to examine all feasible code-data mapping and compare their cross-domain weights. For ease of description, a set $\mathcal{M}^{[U,S]}_{[O,I]} := \{[\mathtt{M}^{U}_{O}, \mathtt{M}^{S}_{I}]\}$ is used to denote all the potential code-data mappings. There are $|\mathtt{I}|!$ number of input-to-sensor mapping $\mathtt{M}^{S}_{I}$s that should be checked for each of the $|\mathtt{O}|!$ output-to-command mapping $\mathtt{M}^{U}_{O}$s. Instead of naive enumeration, the following properties of G^{code} can be exploited to expedite the construction of G^{cross}.

- The output nodes in O are not reachable from each other in G^{code}, making G^{code} dividable to a set of isolated subgraphs, as shown by the subgraphs in Fig. 13 divided from ECS's G^{code} in Fig 8. Hence, instead of checking $[\mathtt{M}^{S}_{I}, \mathtt{M}^{U}_{O}]$ together, ARES checks each of the $\mathtt{M}^{S}_{I}$ for $|\mathtt{O}|$ independent m^{u}_{o}s, where $u \in \mathtt{U}$, thus reducing the mapping space from $|\mathtt{I}|! \times |\mathtt{O}|!$ to $|\mathtt{I}|! \times |\mathtt{O}|^2$.
- One input node in I may connect to multiple output nodes in O, i.e., the above divided subgraphs may have shared input nodes, such as the I2.4 and I3.0 between Fig. 13(a) and Fig. 13(b). Once a set of input nodes have been mapped, denoted as $\tilde{\mathtt{I}}$, ARES can reuse the corresponding cross-domain edges to further reduce the input-to-sensor mapping space from $|\mathtt{I}|!$ to $|\mathtt{I}\setminus\tilde{\mathtt{I}}|!$.

These properties allow ARES to identify the mapping between G^{code} and G^{data} — i.e., construct G^{cross} — by recursively mapping the subgraphs of G^{code} to G^{data}. Also, within each subgraph of node o, only nodes in $\mathtt{I}_o$ decide o's dependency, where $\mathtt{I}_o$ denotes the neighbors of o in G^{code}, reducing the corresponding input-to-sensor mapping space to $|\mathtt{I}_o\setminus\tilde{\mathtt{I}}|!$. These two observations inspire ARES to identify the mapping between G^{code} and G^{data} using an approach similar to dynamic programming. Specifically, by initializing all the mapped node sets $\{\tilde{\mathtt{O}}, \tilde{\mathtt{I}}, \tilde{\mathtt{U}}, \tilde{\mathtt{S}}\}$ and the corresponding identified mapping set $\mathcal{M}^{[\tilde{U},\tilde{S}]}_{[\tilde{O},\tilde{I}]}$ as $\varnothing$, and starting from the subgraph of node o which has the minimum node degree in G^{code}, ARES constructs the G^{cross} as follows.

Step-I: Reusing Identified Mappings. Given an output node o, ARES first identifies its neighbor set $\mathtt{I}_o$ from G^{code}, and its mapped neighbors as $\tilde{\mathtt{I}}_o := \mathtt{I}_o \cap \tilde{\mathtt{I}}$. ARES then selects all the identified input-to-sensor mappings from $\mathcal{M}^{[\tilde{U},\tilde{S}]}_{[\tilde{O},\tilde{I}]}$ as $\mathcal{M}^{\tilde{S}}_{\tilde{I}_o} := \{\mathtt{M}^{\tilde{S}}_{\tilde{I}_o}\}$. Besides, together with each $\mathtt{M}^{\tilde{S}}_{\tilde{I}_o}$, the corresponding identified output-to-command mappings $\mathtt{M}^{\tilde{U}}_{\tilde{O}}$ in $\mathcal{M}^{[\tilde{U},\tilde{S}]}_{[\tilde{O},\tilde{I}]}$ is also selected as $\mathcal{M}^{[\tilde{U},\tilde{S}]}_{[\tilde{O},\tilde{I}_o]} := \{[\mathtt{M}^{\tilde{U}}_{\tilde{O}}, \mathtt{M}^{\tilde{S}}_{\tilde{I}_o}]\}$.

Step-II: Enumerating Feasible Mappings. ARES enumerates all feasible code-data mappings between the subgraph of node o and G^{data}, i.e., the feasible output-to-command mappings $\mathcal{M}^{U}_{o} := \{\mathtt{M}^{u}_{o}\}$ and the corresponding feasible input-to-sensor mappings $\mathcal{M}^{S_u}_{\mathtt{I}_o} := \{\mathtt{M}^{S_u}_{\mathtt{I}_o}\}$. For each of the identified mapping $\mathtt{M}^{\tilde{S}^*}_{\tilde{I}_o}$ in $\mathcal{M}^{\tilde{S}}_{\tilde{I}_o}$ (or $\mathcal{M}^{[\tilde{U},\tilde{S}]}_{[\tilde{O},\tilde{I}_o]}$), ARES extracts the unmapped neighbors of output o as $\bar{\mathtt{I}}_o := \mathtt{I}_o \setminus \tilde{\mathtt{I}}_o$, and constructs the $\mathcal{M}^{U}_{o}$ using the following properties of G^{data}:

- *Node's neighbor.* For each node $u \in \mathtt{U}$, ARES extracts its unmapped neighbors $\bar{\mathtt{S}}_u := \mathtt{S}_u \setminus \tilde{\mathtt{S}}^*$, where $\mathtt{S}_u$ denotes the neighbors of u in G^{data}. Note that PLCs generate the command u using an output variable o by addressing related sensor readings in S using corresponding input variables in I, implying $|\bar{\mathtt{S}}_u| = |\bar{\mathtt{I}}_o|$. Hence, ARES only abstracts the feasible mappings $\mathcal{M}^{U}_{o} := \{\mathtt{M}^{u}_{o}\}$ and nodes $\bar{\mathtt{U}} := \{u\}$ when u satisfies $|\bar{\mathtt{I}}_o| - |\bar{\mathtt{C}}| \le |\bar{\mathtt{S}}_u| \le |\bar{\mathtt{I}}_o|$, where $\bar{\mathtt{C}}$ denotes the unmapped constant sensors in G^{data}.
- *Value range.* The value range of code variables is indicated by its identifier $[5, 25]$, as shown in Table 3 in Appendix B. ARES therefore refines $\bar{\mathtt{U}}$ and $\mathcal{M}^{U}_{o}$ by removing u and $\mathtt{M}^{u}_{o}$, respectively, if values of u fall out of the range specified by o.
- *Node degree.* The identified mapping of $[\mathtt{M}^{\tilde{U}}_{\tilde{O}}, \mathtt{M}^{\tilde{S}}_{\tilde{I}_o}]$ in $\mathcal{M}^{[\tilde{U},\tilde{S}]}_{[\tilde{O},\tilde{I}_o]}$ also helps ARES reduce $\mathcal{M}^{U}_{o}$ by removing $\mathtt{M}^{u}_{o}$, where $u \in \tilde{\mathtt{U}}$. However, this reduction will introduce repeated checking on $\mathtt{M}^{\tilde{S}}_{\tilde{I}_o}$ if the corresponding $\mathtt{M}^{\tilde{U}}_{\tilde{O}}$ is not unique. Therefore, for the identified mapping $\mathtt{M}^{\tilde{S}^*}_{\tilde{I}_o}$, ARES only applies this reduction if $|\bar{\mathtt{U}} \setminus \tilde{\mathtt{U}}| \times |\{[\mathtt{M}^{\tilde{U}}_{\tilde{O}}, \mathtt{M}^{\tilde{S}^*}_{\tilde{I}_o}]\}| < |\bar{\mathtt{U}}| \times |\{\mathtt{M}^{\tilde{S}^*}_{\tilde{I}_o}\}|$.

Then for each output-command mapping $\mathtt{M}^{u}_{o} \in \mathcal{M}^{U}_{o}$, ARES constructs the corresponding $\mathcal{M}^{S_u}_{\mathtt{I}_o}$ as:

- *Node's neighbor.* ARES enumerates the feasible input-sensor mappings between $\bar{\mathtt{I}}_o$ and $\bar{\mathtt{S}}_u$ as $\mathcal{M}^{\bar{S}_u}_{\bar{I}_o} := \{\mathtt{M}^{\bar{S}_u}_{\bar{I}_o}\}$. If $|\bar{\mathtt{S}}_u| < |\bar{\mathtt{I}}_o|$, ARES will further enumerates the unmapped input variables with nodes in $\bar{\mathtt{C}}$. ARES then constructs $\mathcal{M}^{S_u}_{\mathtt{I}_o}$ by appending the identified mapping $\mathtt{M}^{\tilde{S}^*}_{\tilde{I}_o}$ to each $\mathtt{M}^{\bar{S}_u}_{\bar{I}_o} \in \mathcal{M}^{\bar{S}_u}_{\bar{I}_o}$.

- *Value range.* ARES refines the $\mathcal{M}_{\mathtt{I}_o}^{\mathtt{S}_u}$ by removing the mapping $\mathtt{M}_{\mathtt{I}_o}^{\mathtt{S}_u}$, which pairs sensor nodes in $\mathtt{S}_u$ to input nodes in $\mathtt{I}_o$ with conflicting value ranges.

ARES will end this step when all identified input-sensor mappings in $\mathcal{M}_{\tilde{\mathtt{I}}_o}^{\tilde{\mathtt{S}}}$ (or $\mathcal{M}_{[\tilde{\mathtt{O}},\tilde{\mathtt{I}}_o]}^{[\tilde{\mathtt{U}},\tilde{\mathtt{S}}]}$) have been reused.

Step-III: Identifying Cross-Domain Mappings. ARES identifies the mappings of the subgraph o by examining all feasible code-data mappings enumerated above using PLC-Twin. Specifically, taking each mapping $[\mathtt{M}_o^u, \mathtt{M}_{\mathtt{I}_o}^{\mathtt{S}_u}]$ in $\{\mathcal{M}_o^{\mathtt{U}}, \mathcal{M}_{\mathtt{I}_o}^{\mathtt{S}_u}\}$ as input, ARES generates the estimated control commands $\hat{\mathbf{u}}$ by executing PLC-Twin, and calculates the corresponding cross-domain weight w_o^u according to Eq. (5). Denote the maximum weight as $\tilde{w}_o^u$. ARES identifies the code-data mapping $[\mathtt{M}_o^{u^*}, \mathtt{M}_{\mathtt{I}_o}^{\mathtt{S}_u^*}]$ which has a cross-domain weight larger than $\tilde{w}_o^u \times \eta$, and abstracts them as $\mathcal{M}_{[o,\mathtt{I}_o]}^{[\mathtt{U},\mathtt{S}_u]} := \{[\mathtt{M}_o^{u^*}, \mathtt{M}_{\mathtt{I}_o}^{\mathtt{S}_u^*}]\}$, where $\eta \in (0, 1]$ defines the mapping confidence, and will be evaluated in Sec. 6.

Step-IV: Updating Identified Mapping Set. ARES appends the previously identified mappings in $\mathcal{M}_{[\tilde{\mathtt{O}},\tilde{\mathtt{I}}]}^{[\tilde{\mathtt{U}},\tilde{\mathtt{S}}]}$ with the newly identified mappings in $\mathcal{M}_{[o,\mathtt{I}_o]}^{[\mathtt{U},\mathtt{S}_u]}$. As both the out-degree of nodes in $\{\mathtt{O}, \mathtt{I}\}$ and the in-degree of nodes in $\{\mathtt{U}, \mathtt{S}\}$ equal to 1 in $\mathrm{G}^{\mathrm{cross}}$, ARES then removes all conflicting mappings $[\mathtt{M}_{\tilde{\mathtt{O}}}^{\tilde{\mathtt{U}}}, \mathtt{M}_{\tilde{\mathtt{I}}}^{\tilde{\mathtt{S}}}]$s from $\mathcal{M}_{[\tilde{\mathtt{O}},\tilde{\mathtt{I}}]}^{[\tilde{\mathtt{U}},\tilde{\mathtt{S}}]}$.

Step-V: Selecting The Subgraph To Be Checked Next. Among all the unchecked subgraphs of os in $\mathtt{O} \setminus \tilde{\mathtt{O}}$, ARES chooses the subgraph of o which has the minimum $|\mathtt{I}_o \setminus \tilde{\mathtt{I}}_o| \times |\mathcal{M}_{\tilde{\mathtt{I}}_o}^{\tilde{\mathtt{S}}}|$ to check next, where $\tilde{\mathtt{I}}_o$ denotes the mapped neighbors of o, and $\mathcal{M}_{\tilde{\mathtt{I}}_o}^{\tilde{\mathtt{S}}}$ denotes the corresponding input-to-sensor mappings in the identified mapping set $\mathcal{M}_{[\tilde{\mathtt{O}},\tilde{\mathtt{I}}]}^{[\tilde{\mathtt{U}},\tilde{\mathtt{S}}]}$.

ARES repeats the above process until $\mathtt{O} = \tilde{\mathtt{O}}$. ARES, in turn, identifies the cross-domain edges set $\tilde{\mathtt{E}}$ by aggregating all mappings in $\mathcal{M}_{[\tilde{\mathtt{O}},\tilde{\mathtt{I}}]}^{[\tilde{\mathtt{U}},\tilde{\mathtt{S}}]}$ — i.e., paralleling all cross-domain edges of node $o \in \tilde{\mathtt{O}}$ and $i \in \tilde{\mathtt{I}}$ according to $[\mathtt{M}_{\tilde{\mathtt{O}}}^{\tilde{\mathtt{U}}}, \mathtt{M}_{\tilde{\mathtt{I}}}^{\tilde{\mathtt{S}}}]$s. If there are multiple mappings in $\mathcal{M}_{[\tilde{\mathtt{O}},\tilde{\mathtt{I}}]}^{[\tilde{\mathtt{U}},\tilde{\mathtt{S}}]}$, ARES will keep those with the maximum sum of cross-domain weights in $\tilde{\mathtt{W}}$. Fig. 14 visualizes the $\mathrm{G}^{\mathrm{cross}}$ of ECS constructed with a mapping confidence η of 1.

Note that the dynamic programming mapping approach of ARES reduces the mapping space by a factor of at least $|\mathtt{O}|!/|\mathtt{O}|^2$. The detailed mapping complexity of ARES depends on the implementation of control logic, such as the shared input variables among the control logic for different output variables $o \in \mathtt{O}$.

5.4 Inferring Variable Semantics

ARES reverse engineers the variable semantics using $\mathrm{G}^{\mathrm{cross}}$. Specifically, for a given in/output variable i/o in the PLC program, ARES finds the data variable that is connected to i/o in $\mathrm{G}^{\mathrm{cross}}$, and identifies the physical semantics of i/o as the physical property of this data variable. Note that ARES's reverse engineering will not be conclusive if a program variable is connected to multiple data variables

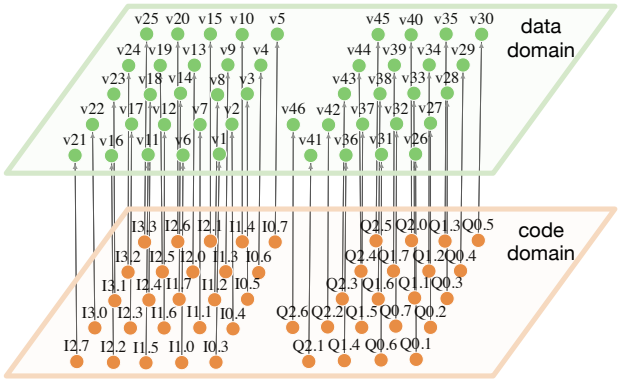


Fig. 14. Constructed $\mathrm{G}^{\mathrm{cross}}$ of ECS: all program variables are uniquely/correctly mapped to data variables in SCADA log.

in $\mathrm{G}^{\mathrm{cross}}$, driving us to the preferred practices in implementing control rules that improve the system's resistance to semantic attacks, as we elaborate in Sec. 7. The constructed $\mathrm{G}^{\mathrm{cross}}$ also grants the opportunities for the risk assessment of ICSs to semantic attacks.

6 EVALUATION

We have evaluated ARES on two ICS platforms, i.e., an Elevator Control System (ECS) and an Ethanol Distillation System (EDS). Note that although ECS and EDS are scaled down, they have fully operational and representative control logic, representing the two basic control scenarios — i.e., ECS for the discrete scenario and EDS for the continuous scenario.

6.1 Methodology

We obtain the PLC program and SCADA log of the two platforms using the Ability-1) and 2) specified in Sec. 3.2 via a similar approach as with BlackEnergy, and then apply ARES to reverse engineer the physical semantics of program variables thereof. Using the thus-inferred variable semantics, we empirically mount semantic attacks to the two platforms with the following two approaches (i.e., implementations of Ability-3) described in Sec. 3.2).

- *Approach-1.* Exploiting the vulnerabilities of PLC communication protocols (e.g., the "Force" debugging function of Siemens S7comm), we send malicious payloads to manipulate the states of PII/PIO tables. For example, the exemplary semantic attack in Sec. 3.3 — i.e., driving the elevator descending with an open door — is mounted by setting the memory field of the protocol as `Q0.6` and the expected value field of the protocol as 1.
- *Approach-2.* Exploiting the vulnerabilities of PLC programming software (e.g., the "s7otbxdx.dll" of Siemens Step7), we download the malicious program to the victim PLC. For example, we use a modified "s7otbxdx.dll" to concatenate the malicious logic of *"assigning* `Q0.6` *'s states as* 1" at the end of legitimate PLC code.

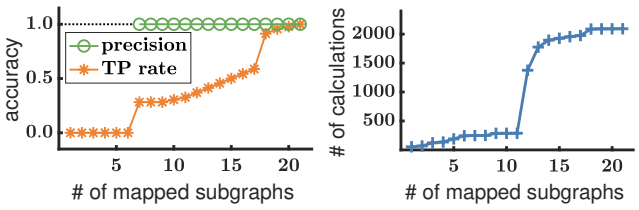
The edge confidence θ in $\mathrm{G}^{\mathrm{data}}$ and the mapping confidence η in $\mathrm{G}^{\mathrm{cross}}$ are set by default as 0.1 and 1, respectively.

6.2 Evaluation with ECS

We evaluate ARES on ECS, which operates a four-floor elevator using 25 sensors and 21 commands. The control rules are programmed

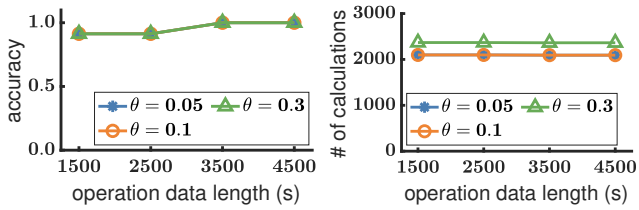
Inferred Semantics	Code Var.	Correct or Not
"1st_digital_light"	Q0.1	✓
"2nd_digital_light"	Q0.2	✓
"3rd_digital_light"	Q0.3	✓
"indicate_ascending"	Q0.4	✓
"indicate_descending"	Q0.5	✓
"descend"	Q0.6	✓
"ascend"	Q0.7	✓
"open_door"	Q1.1	✓
"close_door"	Q1.2	✓
"indicate_to_1F"	Q1.3	✓
"indicate_to_2F"	Q1.4	✓
"indicate_to_3F"	Q1.5	✓
"indicate_to_4F"	Q1.6	✓
"indicate_openning_door"	Q1.7	✓
"indicate_closing_door"	Q2.0	✓
"indicate_go_upstairs_1F"	Q2.1	✓
"indicate_go_upstairs_2F"	Q2.2	✓
"indicate_go_downstairs_2F"	Q2.3	✓
"indicate_go_upstairs_3F"	Q2.4	✓
"indicate_go_downstairs_3F"	Q2.5	✓
"indicate_go_downstairs_4F"	Q2.6	✓
"to_1F"	I0.3	✓
"to_2F"	I0.4	✓
"to_3F"	I0.5	✓
"to_4F"	I0.6	✓
"open_door_request"	I0.7	✓
"close_door_request"	I1.0	✓
"go_upstairs_1F"	I1.1	✓
"go_upstairs_2F"	I1.2	✓
"go_downstairs_2F"	I1.3	✓
"go_upstairs_3F"	I1.4	✓
"go_downstairs_3F"	I1.5	✓
"go_downstairs_4F"	I1.6	✓
"max_height_limitation"	I1.7	✓
"min_height_limitation"	I2.0	✓
"door_is_not_opened"	I2.1	✓
"door_is_open"	I2.2	✓
"door_blocked_by_obstacle"	I2.3	✓
"at_1F"	I2.4	✓
"at_2F"	I2.5	✓
"at_3F"	I2.6	✓
"at_4F"	I2.7	✓
"at_1F_backup"	I3.0	✓
"at_2F_backup"	I3.1	✓
"at_3F_backup"	I3.2	✓
"at_4F_backup"	I3.3	✓

Fig. 15. The reverse engineered semantics of variables in ECS's PLC program.



(a) reverse engineering accuracy (b) cumulative number of calculations

Fig. 16. Reverse engineering the semantics of ECS's variables with a 3,500s SCADA data and edge confidence $\theta = 0.1$.



(a) reverse engineering accuracy (b) total number of calculations

Fig. 17. Reverse engineering the semantics of ECS's variables with varying data duration and edge confidence θ.

into a Siemens S7-317 PLC and monitored by a WinCC SCADA at a frequency of 2Hz.

Reverse Engineering Variable Semantics. ARES constructs ECS's G^{code} using the program extracted from its PLC, and G^{data} using a SCADA log of 3,500s, as shown in Fig. 8 and Fig. 12, respectively. After checking 2,093 (i.e., $\ll 25!\times 21!$) feasible mappings in $\mathcal{M}^{[U,S]}_{[O,I]}$, ARES constructs G^{cross} as shown in Fig. 14, bridging the program variables in the code domain with data variables in the data domain. All the 46 cross-domain edges between program and data variables are identified uniquely and correctly.

Based on these cross-domain edges, ARES correctly reverse engineered the semantics of all program variables according to the data variables they map to, as summarized in Fig. 15. Note that ARES does not need to complete the checking of all feasible mappings before the variable semantics can be inferred, indicating the scalability of ARES for large-scale ICSs: as shown in Fig. 16, ARES correctly reverse engineers the semantics of 13 (out of 46) program variables of ECS after only the first 248 (out of 2,093) checkings.

We further evaluate the robustness of ARES to the duration of available SCADA data, G^{data}'s edge confidence θ, and G^{cross}'s mapping confidence η. As shown in Fig. 17(a), ARES reverse engineers ECS's semantics more accurately when more SCADA data is available, because more operation data capture more distinctive system states, facilitating ARES to distinguish better the behavior of program variables. Fig. 17(a) also corroborates the robustness of ARES to the edge confidence θ when constructing G^{data}, although a larger confidence θ will discard more causality edges for command nodes (see Fig. 18). This is because the shared input neighbors among

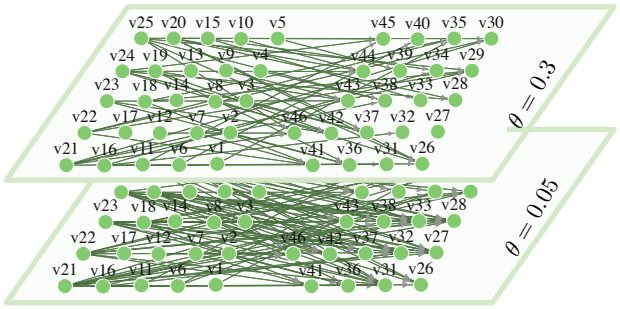


Fig. 18. Constructed G^{data} of ECS with an edge confidence θ of 0.3 and 0.05, respectively.

different output nodes in G^{code} allow ARES to identify the input-to-sensor mappings without relying on the discarded causality edges in G^{data}. Taking the mapping of nodes Q0.7 with v_{32} as an example, the discarded causality sensors of v_{32} are already mapped to corresponding input nodes after checking the mappings of subgraphs of {Q2.1, Q1.3, Q2.6, Q1.6, Q2.0, Q1.7, Q0.2, Q2.3, Q1.2}, which have shared input neighbors with Q0.7 (see Fig. 13(j) vs. Figs. 13(a)-13(i)). The incomplete E of G^{data} requires more examinations on the feasible mappings in $\mathcal{M}^{[U,S]}_{[O,I]}$ when constructing G^{cross}, as shown in Fig. 17(b). Moreover, we also evaluate ARES with various mapping confidence η, and find ARES constructs G^{cross} with the same accuracy as the case when $\eta = 1$. This is because $\eta < 1$ allows ARES to identify more code-data mappings, including the correct one which has the maximum weight to be distinguished.

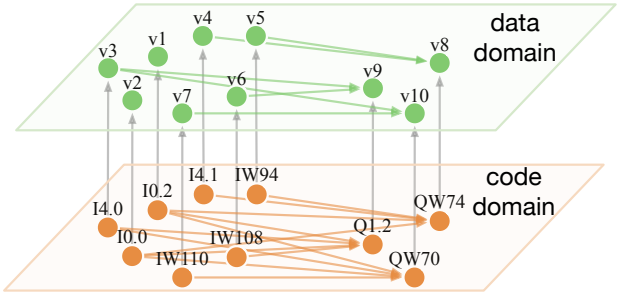


Fig. 19. Constructed {G^{code}, G^{data}, G^{cross}} of EDS.

Inferred Semantics	"heat_material"	"feed_material"	"feed_coolant"	"emergency_status"	"auto_mode_status"	"start_system_status"	"start_cooling_status"	"cooling_water_flow"	"heater_temperature"	"tower_liquid_level"
Code Var.	Q1.2	QW70	QW74	I0.0	I0.2	I4.0	I4.1	IW94	IW108	IW110
Correct or Not	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fig. 20. The reverse engineered semantics of variables in EDS's PLC program.

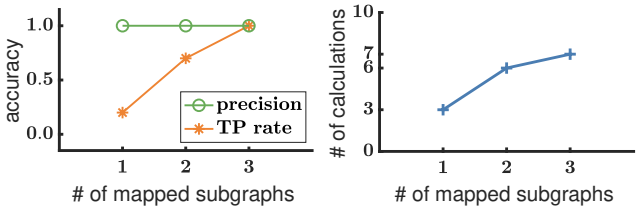
Mounting Semantic Attack to ECS. Using the thus-inferred semantics of program variables, we have empirically mounted semantic attacks against 46 modules of ECS. Compared with attacks that randomly modify PLC states, these semantic attacks manipulate targeted system modules to achieve the intended (and malicious) impacts, including: driving the elevator with an open door; closing the door when it is blocked by an obstacle; stopping the elevator before it arrives at the destination; driving the elevator without missions, etc. See [60] for a demo video of one such semantic attack that drives the elevator with an open door.

6.3 Evaluation with EDS

To further evaluate ARES with systems running proprietary PLC instructions (e.g., the PID function), we deploy ARES to an Ethanol Distillation System (EDS), which operates based on 3 control commands and 7 sensor readings, reports system states to SCADA at a frequency of 2Hz, and produces the alcohol with a purity of 90%. See Appendix A for details of EDS.

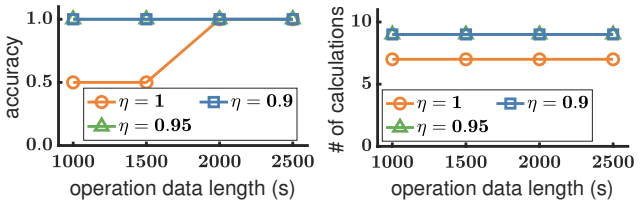
Reverse Engineering Variable Semantics. ARES constructs EDS's G^{code} using the program extracted from its PLC, and G^{data} using a SCADA data of 2,000s. After examining 7 (i.e., $\ll 7!\times 3!$) feasible mappings in $\mathcal{M}_{[0,1]}^{[U,S]}$, ARES constructs G^{cross} to bridge the program and data variables. Fig. 19 visualizes the constructed {G^{code}, G^{data}, G^{cross}} of EDS. All the 10 cross-domain edges in G^{cross} are identified uniquely and correctly.

Using the constructed G^{cross}, ARES successfully reverse engineers the semantics of all program variables (7 input variables and 3 output variables) according to the connected data variables, as summarized in Fig. 20. Also, as shown in Fig. 21, ARES reverse engineers the semantics of 2 (out of 10) program variables after the first 3 (out of 7) checkings of feasible mappings, demonstrating the scalability of ARES for the large-scale ICSs.



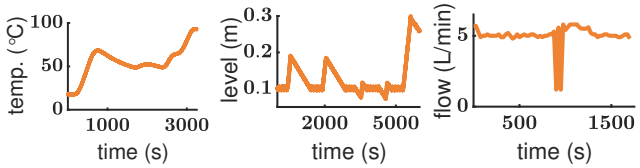
(a) reverse engineering accuracy (b) cumulative number of calculations

Fig. 21. Reverse engineering the semantics of EDS's variables with a 2,000s SCADA data and mapping confidence $\eta = 1$.



(a) reverse engineering accuracy (b) total number of calculations

Fig. 22. Reverse engineering the semantics of EDS's variables with varying data duration and mapping confidence η.



(a) tower temperature (b) cooling water flow (c) tower liquid level

Fig. 23. The impact of semantic attacks against EDS: (a) overheating the tower, (b) overwhelming the tower, (c) destabilizing the cooling water flow.

We then evaluate ARES's robustness to the duration of available SCADA log, mapping confidence η, and edge confidence θ. As shown in Fig. 23, ARES reverse engineers the semantics of EDS more accurately when more SCADA data is available. Also, Fig. 23 shows that a smaller mapping confidence η allows ARES to correctly reverse engineer the variable semantics even when the available SCADA log is limited, but at the cost of increased calculations. This is because a smaller η allows ARES to identify more code-data mappings during the construction of G^{cross}, each of which is examined multiple times due to the shared input neighbors among {Q1.2, QW70, QW74} in G^{code}. Besides, the varying edge confidence θ has little impact on the accuracy of constructed G^{cross}, because the shared input neighbors among {Q1.2, QW70, QW74} help ARES to construct G^{cross} without relying on the discarded edges in G^{data}.

Mounting Semantic Attack to EDS. With the reverse engineered semantics of program variables, we have designed and mounted semantic attacks against 10 physical modules of EDS. All these attacks manipulate the EDS as intended, including i) damaging the system product and ii) destroying physical modules. As shown in Fig. 23,

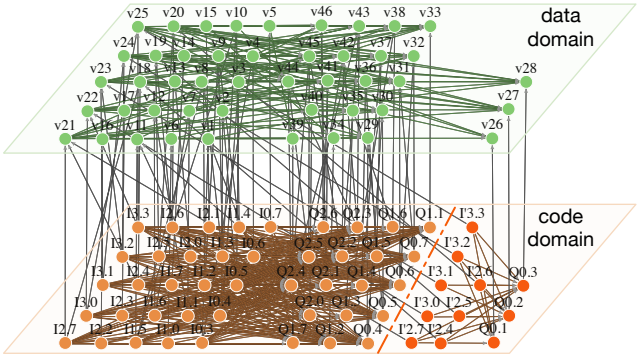


Fig. 24. Constructed {G^{code}, G^{data}, G^{cross}} of ECS after decoupling the control rules: only 40.74% of program variables (i.e., 22 out of 54) are uniquely mapped to data variables.

the mounted attacks drive EDS to be overheated, overwhelmed, and destabilized. Note that the destabilized cooling water flow may also physically damage the measuring device, such as the flywheel whose speed changes with the water flow.

7 LESSONS LEARNED

ARES is built on the uniquely identified code-data mapping, which is induced by the distinctive consistency between PLC program and SCADA data in describing different control rules. This inspires the following two practices in implementing control rules that improve the resistance of PLC programs to semantic attacks.

First, commutative variables — i.e., variables whose assignments can be exchanged without changing the control logic — can be used to void the one-to-one mapping between code and data variables. Considering an elevator system that *ONLY* operates the code in Fig. 2(b), input variables `I0.4` and `I2.6` thereof are commutative, which defines the instruction of *"to_2F"* and the position of "*at_3F*", respectively — the output variable `Q0.6` can be generated correctly even when exchanging the assignments of `I0.4` and `I2.6`. With these commutative variables, the code-data mappings of {`I0.4`~*"to_2F"*, `I2.6`~*"at_3F"*} and {`I0.4`~*"at_3F"*, `I2.6`~*"to_2F"*} have the same consistency in describing the control invariants of `Q0.6`, inducing two cross-domain edges for both `I0.4` and `I2.6`. These multiple cross-domain edges hinder adversaries to correctly reverse engineer the semantics of `I0.4` and `I2.6`, even when they can see through this commutativity from the PLC program — applying ARES to this improved elevator only reverse engineers the semantics of 5 (out of 7) program variables.

Also, we recommend decoupling the control rules, i.e., avoiding using shared input variables to define multiple control rules. This is because the shared input variables will share the uniquely identified input-to-sensor mappings to quantify the consistency of multiple control invariants. For example, the input variables {`I2.4`, `I2.5`, `I2.6`, `I2.7`, `I3.0`, `I3.1`, `I3.2`, `I3.3`} in ECS are used to define the control rules of output variables {`Q0.1`, ..., `Q2.6`}. If we re-implement the control rules of {`Q0.1`, `Q0.2`, `Q0.3`} with another set of input variables {`I'2.4`, ..., `I'3.3`} — which read the system status from sensors of {`I2.4`, ..., `I3.3`} — to avoid the sharing of {`I2.4`, ..., `I3.3`}, the ratio of variables whose semantics can be conclusively inferred by ARES will reduce to 40.74% (22/54) from 100%, as plotted in Fig. 24.

8 CONCLUSION

We have proposed ARES to automatically reverse engineer the physical semantics of PLC program variables. Steered by ARES, we also suggested preferred practices in implementing control rules to improve the resistance of PLC program to semantic attacks. We have experimentally evaluated ARES on two ICS platforms, showing ARES to reverse engineer the semantics of all program variables with 100% accuracy. We have also experimentally corroborated the effectiveness of the proposed implementation practices.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and the shepherd for their insightful feedback. This work was supported in part by the National Natural Science Foundation of China under Grant 61833015, U1911401, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), and an institutional grant from the University of Colorado Denver.

REFERENCES

[1] Ali Abbasi and Majid Hashemi. 2016. Ghost in the PLC Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack. In *Black Hat Europe*. 1–35.
[2] ICS Advisory. 2020. Emerson OpenEnterprise. https://www.cisa.gov/uscert/ics/advisories/icsa-20-238-02. (2020). [Online; Accessed October 2022].
[3] ICS Advisory. 2020. Siemens SIMATIC HMI Products. https://us-cert.cisa.gov/ics/advisories/icsa-20-252-06. (2020). [Online; Accessed October 2022].
[4] ICS Advisory. 2021. SIMATIC WinCC Graphics Designer. https://us-cert.cisa.gov/ics/advisories/icsa-21-040-09. (2021). [Online; Accessed October 2022].
[5] Siemens AG. 2017. Programming with STEP 7. https://cache.industry.siemens.com/dl/files/825/109751825/att_933142/v1/STEP_7_-_Programming_with_STEP_7.pdf. (2017). [Online; Accessed October 2022].
[6] Siemens AG. 2019. WinCC V7.5 SP1: Working with WinCC. https://support.industry.siemens.com/cs/document/109773058/wincc-v7-5-sp1-working-with-wincc?dti=0&lc=en-CZ. (2019). [Online; Accessed October 2022].
[7] Thiago Alves. 2021. OpenPLC Runtime version 3. https://github.com/thiagoralves/OpenPLC_v3. (2021). [Online; Accessed October 2022].
[8] Karl Johan Åström, Tore Hägglund, and Karl J Astrom. 2006. *Advanced PID control*.
[9] Rockwell Automation. 2021. MicroLogix 1400 Programmable Controllers User Manual. https://literature.rockwellautomation.com/idc/groups/literature/documents/um/1766-um001_-en-p.pdf. (2021). [Online; Accessed October 2022].
[10] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying Intrusion-Tolerant SCADA for the Power Grid. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 328–335.
[11] Zachry Basnight, Jonathan Butts, Juan Lopez Jr, and Thomas Dube. 2013. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection* 6, 2 (2013), 76–84.
[12] Hans Berger. 2012. *Automating with STEP7 in STL and SCL: programmable controllers Simatic S7-300/400*.
[13] Eli Biham, Sara Bitan, Aviad Carmel, Alon Dankner, Uriel Malin, and Avishai Wool. 2019. Rogue7: Rogue Engineering Station Attacks on S7 Simatic PLCs. In *BlackHat USA*.
[14] Siemens Security Advisory by Siemens ProductCERT. 2020. 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). [Online; Accessed October 2022].
[15] Michael Büsch. 2020. Awlsim: S7 compatible PLC/SPS. https://bues.ch/cms/automation/awlsim. (2020). [Online; Accessed October 2022].
[16] Defense Use Case. 2016. Analysis of the Cyber Attack on the Ukrainian Power Grid. *Electricity Information Sharing and Analysis Center* (2016).
[17] John H Castellanos, Martin Ochoa, Alvaro A Cardenas, Owen Arden, and Jianying Zhou. 2021. AttkFinder: Discovering Attack Vectors in PLC Programs using Information Flow Analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 235–250.

[18] Joao M Ceron, Justyna J Chromik, Jair Santanna, and Aiko Pras. 2020. Online Discoverability and Vulnerabilities of ICS/SCADA Devices in the Netherlands. *arXiv preprint arXiv:2011.02019* (2020).
[19] Yuqi Chen, Christopher M Poskitt, and Jun Sun. 2021. 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 (ESEC/FSE)*.
[20] Yuqi Chen, Bohan Xuan, Christopher M Poskitt, Jun Sun, and Fan Zhang. 2020. Active Fuzzing for Testing and Securing Cyber-Physical Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 14–26.
[21] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. 2018. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 801–816.
[22] Kevin Collier. 2021. 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). [Online; Accessed October 2022].
[23] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. THIRTY YEARS OF GRAPH MATCHING IN PATTERN RECOGNITION. *International journal of pattern recognition and artificial intelligence* 18, 03 (2004), 265–298.
[24] Cybersecurity and Infrastructure Security Agency. 2020. Rockwell Automation MicroLogix Controllers and RSLogix 500 Software. https://us-cert.cisa.gov/ics/advisories/icsa-20-070-06. (2020). [Online; Accessed October 2022].
[25] Schneider Electric. 2010. User Manual for PLC Programming with CoDeSys 2.3. https://www.ee.pw.edu.pl/~purap/PLC/manuals/m07590333_00000000_1en.pdf. (2010). [Online; Accessed October 2022].
[26] Schneider Electric. 2019. EcoStruxureTM Machine SCADA Expert Technical Reference Manual. https://damrexelprod.blob.core.windows.net/medias/10bdfea1-bba7-490a-94ff-cff4fabf55e2. (2019). [Online; Accessed October 2022].
[27] Alessandro Erba, Riccardo Taormina, Stefano Galelli, Marcello Pogliani, Michele Carminati, Stefano Zanero, and Nils Ole Tippenhauer. 2020. Constrained Concealment Attacks against Reconstruction-based Anomaly Detectors in Industrial Control Systems. In *Annual Computer Security Applications Conference*. 480–495.
[28] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2011. W32. Stuxnet Dossier. *White paper, Symantec Corp., Security Response*. 5, 6 (2011), 29.
[29] Cheng Feng, Venkata Reddy Palleti, Aditya Mathur, and Deeph Chana. 2019. A Systematic Framework to Generate Invariants for Anomaly Detection in Industrial Control Systems. In *Network and Distributed Systems Security (NDSS) Symposium*.
[30] Brian Gallagher. 2006. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*.
[31] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. 2017. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *Network and Distributed Systems Security (NDSS) Symposium*.
[32] Hamid Reza Ghaeini, Nils Ole Tippenhauer, and Jianying Zhou. 2019. Zero Residual Attacks on Industrial Control Systems and Stateful Countermeasures. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*. 1–10.
[33] Martin Giles. 2019. 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). [Online; Accessed October 2022].
[34] Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. 2018. On Ladder Logic Bombs in Industrial Control Systems. In *Computer Security*. 110–126.
[35] Benjamin Green, Marina Krotofil, and Ali Abbasi. 2017. On the Significance of Process Comprehension for Conducting Targeted ICS Attacks. In *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy*. 57–67.
[36] Dag H Hanssen. 2015. *Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS*.
[37] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. 2019. CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*.
[38] Anastasis Keliris and Michail Maniatakos. 2019. ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries. In *Symposium on Network and Distributed System Security (NDSS)*.
[39] Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, and Volker Roth. 2015. Internet-facing PLCs - A New Back Orifice. In *Blackhat USA*.
[40] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating Mutual Information. *Physical review E* 69, 6 (2004), 066138.
[41] Jochen Kühner. 2014. Library to Access Siemens PLCs and Step5/Step7 Project Files. https://github.com/dotnetprojects/DotNetSiemensPLCToolBoxLibrary. (2014). [Online; Accessed October 2022].
[42] Qiang Li, Xuan Feng, Haining Wang, and Limin Sun. 2018. Understanding the Usage of Industrial Control System Devices on the Internet. *IEEE Internet of Things Journal* 5, 3 (2018), 2178–2189.
[43] Aditya P. Mathur and Nils Ole Tippenhauer. 2016. SWaT: A Water Treatment Testbed for Research and Training on ICS Security. In *2016 international workshop on cyber-physical systems for smart water networks (CySWater)*. IEEE, 31–36.
[44] Stephen McLaughlin and Patrick McDaniel. 2012. SABOT: Specification-based Payload Generation for Programmable Logic Controllers. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. 439–449.
[45] Stephen McLaughlin and Saman Zonouz. 2014. Controller-Aware False Data Injection Against Programmable Logic Controllers. In *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 848–853.
[46] Stephen McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. 2014. A Trusted Safety Verifier for Process Controller Code. In *Network and Distributed Systems Security (NDSS) Symposium*.
[47] Matthias Niedermaier, Jan-Ole Malchow, Florian Fischer, Daniel Marzin, Dominik Merli, Volker Roth, and Alexander Von Bodisco. 2018. You Snooze, You Lose: Measuring PLC Cycle Times under Attacks. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.
[48] Schneider Electric Security Notification. 2021. Security Notification - Modicon M100/M200/M221 Programmable Logic Controller (V3.0). https://www.se.com/ww/en/download/document/SEVD-2020-315-05/. (2021). [Online; Accessed October 2022].
[49] Inc PNF Software. 2015. JEB Decompiler for S7 PLC. https://www.pnfsoftware.com/jeb/plc. (2015). [Online; Accessed October 2022].
[50] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. 2020. SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants. In *29th USENIX Security Symposium (USENIX Security 20)*. 895–912.
[51] Andres Robles-Durazno, Naghmeh Moradpoor, James McWhinnie, Gordon Russell, and Inaki Maneru-Marin. 2018. Implementation and Detection of Novel Attacks to the PLC Memory on a Clean Water Supply System. In *International Conference on Technology Trends*. 91–103.
[52] Thomas Roth and Bruce McMillin. 2013. Physical Attestation of Cyber Processes in the Smart Grid. In *International Workshop on Critical Information Infrastructures Security*. 96–107.
[53] Esha Sarkar, Hadjer Benkraouda, and Michail Maniatakos. 2020. I came, I saw, I hacked: Automated Generation of Process-independent Attacks for Industrial Control Systems. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 744–758.
[54] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. 2016. PLC-Blaster: A Worm Living Solely in the PLC. *Black Hat Asia* 16 (2016), 1–16.
[55] Pengfei Sun, Luis Garcia, and Saman Zonouz. 2019. Tell Me More Than Just Assembly! Reversing Cyber-physical Execution Semantics of Embedded IoT Controller Software Binaries. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 349–361.
[56] Siemens Industry Online Support. 2019. Exporting Archived Data from WinCC with the OLE DB Provider. https://cache.industry.siemens.com/dl/files/261/38132261/att_946466/v3/38132261__Application_Reverse_Osmosis_DOC_en.pdf. (2019). [Online; Accessed October 2022].
[57] Symantec DeepSight Adversary Intelligence Team. 2018. Seedworm: Group Compromises Government Agencies, Oil & Gas, NGOs, Telecoms, and IT Firms. https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/seedworm-espionage-group. (2018). [Online; Accessed October 2022].
[58] Symantec Threat Hunter Team. 2017. Dragonfly: Western energy sector targeted by sophisticated attack group. https://www.symantec.com/blogs/threat-intelligence/dragonfly-energy-sector-cyber-attacks. (2017). [Online; Accessed October 2022].
[59] Symantec Threat Hunter Team. 2018. Shamoon: Destructive Threat Re-Emerges with New Sting in its Tail. https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/shamoon-destructive-threat-re-emerges-new-sting-its-tail. (2018). [Online; Accessed October 2022].
[60] A Semantic Attack Against the Elevator Control System. 2022. https://youtu.be/1mksLMRYFtE. (2022).
[61] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. 2021. ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications. In *30th USENIX Security Symposium (USENIX Security 21)*.
[62] 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. 2019. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *2019 IEEE Symposium on Security and Privacy (S&P)*. 522–538.

APPENDIX

A IMPLEMENTATION OF ICS PLATFORMS

We introduce the details of our ECS and EDS platforms below.

Elevator Control System (ECS) Platform. ECS is a scaled-down but fully operational four-floor elevator, including the common functions:

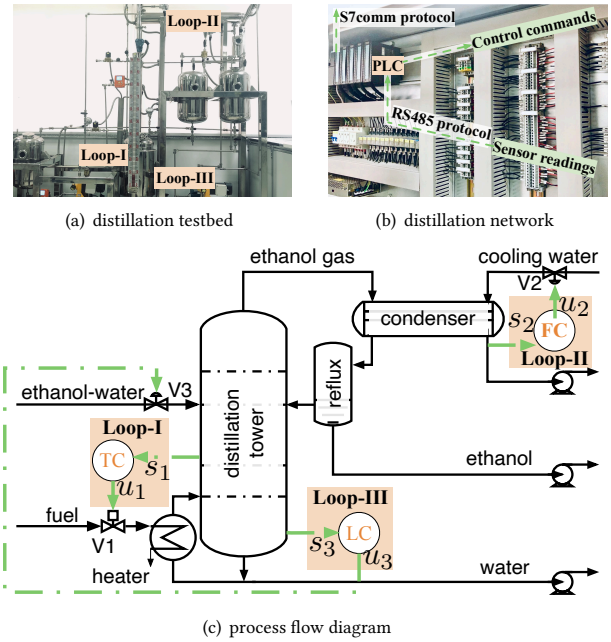
- **Function-I** aims to indicate the status of elevator based on the sensing of 12 destination buttons and 8 elevator positions. For example, the indicator of *"going_to_2F"* will be enabled if the elevator position of *"at_2F"* is *false* and the button of *"to_2F"* is triggered.
- **Function-II** controls the running direction of the elevator, i.e., *"descend"* and *"ascend"* base on the current status of elevator (e.g., *"descending"* and *"going_to_2F"*). ECS is designed to run the shortest distance, where the command of *"descend"* will be enabled if system states of *"going_to_2F"*, *"at_3F"* and *"descending"* are all enabled, regardless of the status of *"going_to_4F"*.
- **Function-III** controls the elevator door, i.e., *"open_door"* and *"close_door"*, based on the preset timers, elevator positions and door positions. For example, the command of *"open_door"* will be enabled when the elevator arrives at the targeted destination.

The above control rules are programmed to a Siemens S7-317 PLC by using 25 sensors to sense 21 elevator statuses and 21 commands to drive 21 actuators, respectively. Besides, a WinCC SCADA is deployed to monitor the states of the above system modules at a frequency of 2Hz.

Ethanol Distillation System (EDS) Platform. Fig. 25 shows an overview of EDS, which is a scaled-down but fully operational distillation process. EDS is able to separate the ethanol from the ethanol-water mixture with a purity of 90%. EDS involves three control loops as shown in Fig. 25(c):

- **Loop-I** aims to keep EDS operating at the pre-defined temperature (i.e., 78°C), at which the ethanol can be purified with a high efficiency. The temperature controller (TC) generates the command u_1 based on the sensor reading of s_1 to drive the opening/closing of valve V1, i.e., EDS will stop heating if $s_1 > 78$°C.
- **Loop-II** controls the cooling water at a pre-defined flow (e.g., $5L/min$) to condense the ethanol from gas to liquid, which is refluxed to the tower to improve ethanol concentration. Based on the measured flow s_2, the flow controller (FC) regulates valve V2 by generating a control command u_2 using the PID algorithm [8]. Note that the unstable cooling water condenses the excessive or deficient ethanol refluxing, which impedes the heat/mass transfer and further degrades the distillation quality.
- **Loop-III** keeps the ethanol-water mixture in the tower at a pre-defined level (e.g., $0.1m$). The liquid controller (LC) also uses PID to generate u_3 based on the measured level s_3. Note that a higher level of the mixture will cause premature flood and thus damage the separation device and distillation efficiency.

The above control rules are programmed to a Siemens S7-315 PLC by using one code variable to define one system module (i.e, sensors/actuators). The PID algorithm is implemented using `CONT_C`, a proprietary function of Siemens. Taking 7 sensor readings as inputs (4 of which are used to describe the control modes), the PLC generates 3 control commands for the three valves. The states of the above system modules are monitored by a WinCC SCADA at a frequency of 2Hz.



(a) distillation testbed (b) distillation network

(c) process flow diagram

Fig. 25. The Ethanol Distillation System (EDS) platform.

B VALUE RANGE OF CODE VARIABLES

The value range of a code variable is decided by its identifier, which is detailed in Table 3. Note that a code variable could be signed and unsigned.

Table 3: Ranges of code variables with different identifiers.

Symbol	X	B	W	DW
Type	Bit	Byte	Word	Double Word
Range†	$\{0, 1\}$	$[-2^7, 2^8 - 1]$	$[-2^{15}, 2^{16} - 1]$	$[-2^{31}, 2^{32} - 1]$

† The value range includes the signed and unsigned data types.