CoToRu: Automatic Generation of Network Intrusion Detection Rules from Code

Heng Chuan Tan*§, Carmen Cheh†§, Binbin Chen†*
*Advanced Digital Sciences Center, Singapore †Singapore University of Technology and Design, Singapore
Email: hc.tan@adsc-create.edu.sg, {carmen_cheh,binbin_chen}@sutd.edu.sg

***Abstract*—Programmable Logic Controllers (PLCs) are the brains of Industrial Control Systems (ICSes), and thus, are often targeted by attackers. While many intrusion detection systems (IDSes) have been adapted to monitor ICS, they cannot detect malicious network packets from a compromised PLC that conform to the network protocol. A domain expert needs to manually construct IDS rules to model a PLC's behavior. That approach is time-consuming and error-prone. Alternatively, machine learning can infer a PLC's behavior model from network traces, but that model may be inaccurate due to a lack of high-quality training data. This paper presents CoToRu - a toolchain that takes in the PLC's code to automatically generate a comprehensive set of IDS rules. CoToRu comprises (1) an analyzer that parses PLC code to build a state transition table for modeling the PLC's behavior, and (2) a generator that instantiates IDS rules for detecting deviations in PLC behavior. The generated rules can be imported into Zeek IDS to detect various attacks. We apply CoToRu to a power grid testbed and show that our generated rules provide superior performance compared to existing IDSes, including those based on statistical analysis, invariant-checking, and machine learning. Our prototype with CoToRu's generated rules provide sub-millisecond detection latency, even for complex PLC logic.**

I. Introduction

Industrial control systems (ICSes) are complex systems of cyber and physical devices that play key roles in critical infrastructures such as power grids and water treatment plants. As a result, they have increasingly become the targets of cyber and physical attacks [1]. High-profile incidents such as the Stuxnet attack [2] have targeted Programmable Logic Controllers (PLCs), which represent the brain of the ICS. By gaining a foothold at the PLCs through cyber means and manipulating their logic, an attacker can send a legitimate-looking network messages with malicious payloads to create maximum physical consequences.

As a key security measure, various intrusion detection systems (IDSes) have been designed to monitor ICS for malicious activities [3], [4], [5]. Many IDSes in deployment today base their detection rules on only the communication aspects of the system. Those rules are derived by modeling the behavior of selected ICS network protocols and applying semantic checking or statistical analysis. As a result, they cannot detect malicious messages that conform to the network protocol. Other IDSes design their detection rules based on mining operation logs and can be broadly classified into two categories: invariants-based, i.e., using manual creation of specifications that describe the physical process (e.g., [6], [7]), and machine learning-based, that learns models of the system behavior (e.g., [8], [9]). However, manually creating specifications requires a large amount of effort and domain expertise. The approach is also error-prone and subject to high maintenance cost when the ICS is upgraded. Furthermore, as shown in our experiments, such specifications often need to be coarse-grained in order to tolerate measurement noise and reduce false alarms. Hence, they may miss stealthy attacks. While machine-learning-based approaches require less human effort and expertise, there is no assurance regarding their detection accuracy. Our experiments show that although an LSTM-based IDS solution provides good accuracy for several types of attacks, it still gives sufficient opportunity for an advanced attacker to bypass detection and launch a successful attack. It can also cause many false alarms for a large system.

In this work, we propose CoToRu (Code-To-Rule) — a toolchain that takes PLC code as input and automatically generates IDS rules to detect deviations from the PLC operational behavior. CoToRu's design is feasible because (1) the PLC code accurately and comprehensively specifies its operational behavior, and (2) over the years, the ICS industry has developed standardized approaches to codify the controller logic as well as their deployment configurations. Our CoToRu implementation generates rules that can be directly used by the open-source Zeek IDS [10]. As shown in Fig. 1, CoToRu consists of two main components:

State Transition Table Generator: This component converts a given PLC code into a state transition table. Each row of the table is a $(\text{key}, \text{next state})$ pair. The table describes the updating logic of all the PLC variables (i.e., the next state) when the PLC's current state satisfies a set of conditions (i.e., the key). The feasibility of such a modeling approach is based on the observation that a complex PLC logic can be decomposed into smaller function-level logic, which can be represented by a symbolic execution tree with few conditions. That observation allows us to use a small state transition table to enumerate all possible combinations of conditions and their corresponding next state. For larger tables, our framework allows the IDS engine to invoke an external symbolic execution tree emulator to determine the next state on the fly, which is left as future work.

Rules Generator: This component automatically instantiates a set of comprehensive IDS rules that use the state transition table as a reference to check for deviation in PLC

§These authors contributed equally to the paper.

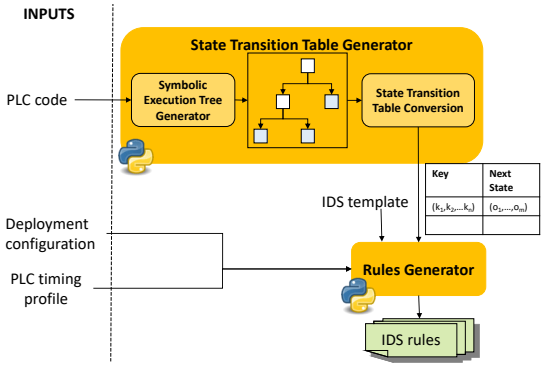


Fig. 1. An overview of CoToRu

behavior. It also needs two additional inputs for the automation: (1) the deployment configuration that describes how PLC local variables are mapped to a PLC's incoming and outgoing messages, and (2) the PLC timing profile that describes the time domain behavior for each type of message. CoToRu toolchain provides scripts to automatically retrieve both inputs. Our IDS is then able to detect various types of deviations caused by the compromised PLC, including: (1) the injection of unexpected packets, (2) the modification of packets, and (3) the deletion/delaying of packets. The generated rules can be directly imported into the open-source Zeek IDS.

We apply CoToRu to a high-fidelity power grid testbed and show that using CoToRu generated IDS rules can provide superior performance compared to existing IDS approaches, including those based on statistical analysis, invariants, and machine learning.

II. Related Work

In this section, we review various IDS solutions for ICS and discuss their shortcomings.

Network-based IDS (NIDS). Most NIDSes used for ICS today can be divided into two categories: protocol-based and protocol-agnostic. Protocol-based IDSes focus on designing IDS rules that follow the specifications of ICS protocols (e.g., [11], [12], [13]). Protocol-agnostic approaches, on the other hand, are independent of the underlying protocol and relies on analysis of network characteristics for deviations. For example, Zhang et al. [14] proposed an IDS that analyzes the periodicity and telemetry features of SCADA traffic.

More recently, Ren et al. [13] proposed an Edge-Based Multi-Level Anomaly Detection (EDMAND) solution that combines both protocol specification modeling and statistical analysis. (More detail will be covered in Section VI-A1). NIDSes in this category, however, do not explicitly model the operational behavior of the ICS. Hence, they cannot detect malicious network packets from a compromised PLC if the attack packets conform to the protocol specification and the traffic statistical pattern.

Operational-aware NIDS. As an enhancement, operational-aware NIDS uses IDS rules that capture the operational aspect of an ICS as well. One main approach is to use machine-learning techniques [8], [15], [16], [17] to derive models of the operational aspect from normal physical data logs. For example, the authors in [15] used a Long Short Term Memory Recurrent Neural Network (LSTM-RNN) to predict the temporal behavior of time-series data. They then used CUSUM to identify deviations between the actual sensor data and the outputs of the trained model to detect anomalies. Similarly, the authors in [17] used LSTM-RNN to detect command delay attacks in cyber-physical systems. While machine-learning techniques can detect some known and unknown attacks, there is no guarantee of the accuracy, which is critical for security applications. Furthermore, this approach is prone to prediction errors due to the quality of data and the limited number of samples needed to build the baseline model. Also, it is hard to obtain or generate comprehensive data (under both normal and attack scenarios) to train the IDSes for real-world settings.

Another way to model the operational aspect is to rely on expert knowledge of the physical process. An example model is physical invariants, which refer to the physical status of the field devices that must be satisfied in a given state of the system. In this direction, Adepu and Mathur [18] proposed an IDS that checks for sensor and actuator readings that deviate from the physical specifications. Although an automated framework is described to generate the invariants, the authors created invariants manually for a water treatment plant case study. In some other efforts, Umer et al. [19] and Feng et al. [20] use historical data logs to mine association rules, which are then used as invariants. However, useful invariants are hard to mine due to the complexity of certain event triggers. Since invariants rely on physical system measurements, which are prone to noise and delay, careful tuning of the parameters in invariants (e.g., the threshold values) is required to reduce the rate of false alarms. As a result, stealthy attacks may go undetected due to the coarse-grained definitions of the invariants.

III. Problem Statement

This section presents the threat model and formulates the design considerations that CoToRu aims to satisfy.

Threat Model. PLCs are deployed in ICSes to monitor and control physical devices. Each PLC contains control logic that governs how the PLC should behave under specific input conditions. A PLC also includes firmware that provides driver support for the control logic.

Our threat model assumes that the attacker manipulates the PLC control logic or firmware to change the PLC's runtime behavior and as a result, disrupts the physical operations. A remote attacker can use social engineering tactics to gain access to the engineering workstation. The engineering workstation runs PLC management software (e.g., CoDeSys) and has a dedicated programming interface that allows users to configure and update the PLC logic. Once a foothold is established, the attacker can exploit that software's unauthenticated command-line interface to upload arbitrary codes or firmware and change the PLC's runtime behaviors [21], [22].

The PLC can also be attacked locally if the attacker has physical access. The on-site attacker can download malicious code to the PLC via the serial port or use the JTAG interface to

manipulate the PLC firmware, as demonstrated in [23]. Once the firmware is corrupted, it can intercept and modify sensor readings in the incoming messages, and/or the control logic values before sending them to the actuators. Unlike control logic attacks, firmware attacks are harder to detect because the control logic looks legitimate to the workstation operators.

The sensors, as well as the communication channels between the sensors and the PLC, may also be compromised (e.g., [24], [5]). We assume various techniques [25], [26], [27] exist to protect the sensor readings. We also do not explicitly address attacks on control center of the SCADA system [28]. Many security solutions and standards [29], [30] have been established to mitigate the associated risks. We assume that some of these existing defence mechanisms have been deployed to handle the problem of compromised sensor readings and control center attacks. Hence, our work will focus on detecting compromised PLCs. Our solution can be used as an orthogonal technology to complement those techniques against compromised sensors or control centers.

Design Considerations. Based on the threat model, we aim to develop an IDS solution that achieves three key objectives:

- O1: It can detect malicious behavior of compromised PLC in real-time and with very high accuracy.
- O2: It requires minimal manual effort to configure and maintain.
- O3: It can be deployed to an ICS in a non-invasive manner and introduces minimal attack surface.

The survey of existing IDS solutions (in Section II) and our experimental evaluation (see Section VI) show that none of the existing solutions can achieve all three objectives. Specifically, mainstream NIDSes cannot detect advanced attacks by compromised PLC (failing to meet O1) because it can issue malicious commands that conform to network protocols. However, they can be configured, deployed, and operated relatively easily (meeting O2 and O3). Machine learning-based NIDSes are unable to achieve very high accuracy (failing to meet O1) and also require large efforts to collect normal and attack data for training the model (failing to meet O2). Existing invariant-based IDSes require significant manual effort from experts (failing to meet O2), can be error-prone, and may not detect attacks in real-time (failing to meet O1).

Motivated by the gaps of existing solutions, our approach is to enhance an NIDS solution with a companion toolchain that can automatically analyze the targeted PLC's code to generate IDS rules. Our approach is based on the observations that (1) the PLC's embedded logic is an accurate, fine-grained, and comprehensive specification of the control operations over the physical process, compared to other sources of information (e.g., collected network traces or operational logs), and (2) many ICS sectors (e.g., the smart power grid) have standardized the way PLC logic is coded as well as their deployment configurations. Furthermore, PLC code contains deterministic logic and, typically, a small number of legitimate states. As we will show, this allows the PLC code to be represented by a state transition table with a small number of rows.

IV. CoToRu: A Toolchain for IDS Rule Generation

In this section, we introduce CoToRu, our toolchain that generates IDS rules using PLC code as the main input. As shown in Fig. 1, CoToRu comprises two components: a state transition table generator and a rules generator. CoToRu's state transition table generator first transforms PLC code into a symbolic execution tree for modeling the conditions that trigger the different state changes. Then, the symbolic execution tree is converted into a state transition table. Using the state transition table and some additional inputs, the rules generator instantiates a set of specific IDS rules based on a built-in set of generic IDS rule templates. The generated IDS rules can be directly imported to the Zeek IDS for deployment.

A. State Transition Table Generation

We briefly overview the syntax of PLC programs and the definition of symbolic execution trees for modeling PLC logic.

1) PLC Languages: The IEC 61131-3 standard [31] defines five programming languages for PLC. Among these programming languages, Structured Text (ST) is the most popular due to its similarity to C language and its expressiveness to handle various logic. The other text-based languages and graphical-based languages can be converted into ST, as shown in [32]. For this reason, we focus on ST to explain the conversion process. ST code is a series of statements formed using *variables*, *operators*, *conditions*, and *loops*.

2) Symbolic Execution Tree: One way to generate IDS rules is to directly convert the PLC code into a syntax that is understood by the IDS engine. However, that approach implies that the IDS rules will have the same running time complexity as the PLC code. More importantly, putting operational code into an IDS increases its complexity and makes it more difficult to ensure the security and correctness of the IDS itself. Additionally, any change in the PLC code will affect a large portion of the IDS rules. This increases the cost of maintenance and introduces more room for error. Thus, our approach serves to analyze the PLC code written in ST and abstract the logic into a form that can be easily integrated into an IDS engine. Our goal is to reduce the running time complexity to a simple table look-up operation at the IDS.

In particular, we utilize algorithms and concepts from the mature area of symbolic execution to model the PLC program as a symbolic execution tree that captures the conditions necessary to produce a certain network message. Symbolic execution has traditionally been used to test programming code by generating test data that explores all control flow paths through the program [33], [34], [35]. It has also been used in tandem with formal methods to verify the safety of PLC programs in an offline fashion [36], [37], [38].

Unlike the existing approaches, our purpose is to use symbolic execution to extract a succinct model of the PLC's normal behavior and use that model to instantiate NIDS rules that detect non-conformance to PLC logic. More formally, symbolic execution is represented as a tree that consists of nodes and edges where:

- each node maintains a state $(stmt, \sigma, \pi)$ where $stmt$ is the next expression to evaluate, σ represents a value store that assigns expressions over concrete or symbolic values to program variables, and π is the path constraints, which is a set of first order Boolean formulas representing the conditions on the variables that need to be satisfied to reach this node, and
- each edge represents a possible transition between states, i.e., where $stmt$ is either an assignment expression or a branching condition.

The path constraints in each leaf node of the tree will then contain the input variables conditions that result in an assignment of some expression to the outgoing network message from the PLC. Such a tree allows for easy conversion to a state transition table.

The traditional algorithm for symbolic execution is to step through the code line by line. At any time, the symbolic state $(stmt, \pi, \sigma)$ is maintained. If the $stmt$ (i.e., code line) is an assignment, e.g., $x = a$, then the value store σ is updated with the expression. If the code line is a conditional statement based on an expression e, then the symbolic execution is forked into two separate symbolic states, where each symbolic state represents the state of the code when e is evaluated to be True or False. More specifically, the path constraint for the forked states would be $\pi \wedge e$ and $\pi \wedge \sim e$ respectively.

3) State Transition Table: We use the resulting symbolic states, which are the leaf nodes of the tree, to derive a state transition table that maps the current PLC state to the next PLC state, which includes the expected value of the outgoing network message from the PLC.

First, we identify all the conditions, expressed as a function of the PLC variables, that impact the PLC's next state. Specifically, we extract the path constraints, π, of all the leaf nodes. Those Boolean formulas in the path constraint form the Key in the table, which will then be used by the IDS to do a table look-up. The result of the table look-up is NEXTSTATE, the value expression of all the PLC variables at the end of the PLC program's scan cycle. So we use the expressions in the value store, σ, to derive the NEXTSTATE. Those expressions consist of calculation of values for (1) outgoing message variables and (2) intermediate variables. For example, using the PLC logic from our case study (Fig. 3), CoToRu will generate the state transition table shown in Table II.

B. IDS Rules Generation

As the second component of CoToRu, the rules generator automatically generates rules that can be directly deployed to the Zeek IDS. The rules generator works by using the information extracted from the PLC to instantiate an IDS rules template that can detect the injection, modification, deletion, and delay of network messages.

Zeek is an open-source network security monitoring tool and its architecture consists of two components: event engine and policy script interpreter [10], [39]. The event engine handles the processing and parsing of network packets. The policy script interpreter contains the IDS script which executes a set of rules to monitor the parsed information for anomalies. We chose Zeek because of its ability to retain state through the use of global variables when processing the last network message value received, intermediate variables, and state of messages received. The Zeek scripting language is event-driven. All events are placed in a FIFO queue.

Besides the state transition table, the rules generator needs the deployment configuration and the PLC timing profile, which can be retrieved automatically from the PLC software project. The deployment configuration defines (1) the list of all PLC variables, and (2) the mapping between the different network packets and the corresponding PLC variables. The PLC timing profile defines the time-domain behavior for each type of message.

Mapping of PLC variables to network messages. The mapping between PLC variables and network messages are specified in the deployment configuration file. Finding the mapping depends on (1) the type of communication protocol used, (2) the encapsulation level of PLC variables in the communication protocol, and (3) the support from the PLC programming tool (e.g., CoDeSys [40]). In our case study, $VSD1\_Start$, $SCADA\_Sync\_Activate$, and In_Sync are Manufacturing Message Specification (MMS) messages and $VSD1\_Command$ is a Modbus write command. To extract the MMS-to-local PLC variable mappings, CoToRu uses the Substation Configuration Language (SCL) file defined by IEC 61850 [41]. Similarly, there is a configuration file that specifies the Modbus-to-local variable mapping. For CoDeSys, both configuration files can be obtained via the export function of its configuration tool.

The SCL file uses the logical device (LD), logical node (LN), functional constraint (FC), data object (DO), and data attribute (DA) defined in the IEC 61850 standard to describe the data model used in communication by a physical device. The CoDeSys software allows embedding of the PLC variable name as part of a private block in the SCL file. Specifically, the PLC variable can be found embedded in one of the DA in the MMS object hierarchy. We used that information to parse the SCL file and extract the mappings using a Java program from our previous work [42]. Starting from the PLC variable name (e.g., In_Sync) at the DA level, the program reverses the path from DA level upward to construct the corresponding network mapping that will uniquely identify the packet field used for communication, including: physical device ID (IP address), and $LD/LN\$FC\$DO\$DA$.

On the other hand, Modbus messages do not contain complex hierarchical data structures. So for the Modbus-to-local variable mapping, we only need to know the device IP address, the function code x, and register index y used to access the Modbus registers. We can retrieve that information by parsing the corresponding Modbus deployment file.

Given the mapping information, CoToRu toolchain contains scripts to automatically define event processing logic in the event engine level. In our case study, for the In_Sync variable, the derived MMS mapping of the form (device IP, $LD/LN\$FC\$DO\$DA$) will be used to filter its value before

sending it to the policy script interpreter. Similarly, Modbus device IP, function code x and register index y, is used to create event handler function to extract the value at that particular register address location to the script for processing. Correspondingly, three matching event handler functions with the same input arguments as those specified in the event engine level will be defined in the script (.bro) level to complete the association. Those event handlers at the script level will be instantiated using the IDS rule template.

Detection logic in the CoToRu IDS template. The key idea behind the IDS rules defined in the template is to identify the violation of conditions that result in the sending of a message and the values contained in that message. Intuitively, our template rules check whether a message is expected (i.e., the conditions are true), and that the contents of the message are valid (i.e., values are correct). We define a function DO_WE_EXPECT in our template that uses the value of the incoming messages to the PLC and the state kept in the IDS to infer the value of the outgoing network messages from the PLC. The state kept in the IDS refers to the values of the PLC variables used for intermediate computation. Then the IDS does a table lookup using the key value to obtain the next state. We update the new state values and store the output value into the corresponding global variables, which include the expected values of messages to be sent by the PLC.

The detection logic is described in Table I. We assume that the PLC sends an outgoing message periodically, with the message inter-arrival time falling between T_{Min} and T_{Max}. These timings are defined for each type of message and can be retrieved from the PLC timing profile. There are three different events in the template that are triggered by either the receipt of network messages or the expiration of timers:

- RECV_MSG_TO_PLC: This event handler is triggered upon receiving network messages that are sent to the PLC. When this function executes, the relevant values in the received message are saved in the corresponding global variables. A call is then made to the DO_WE_EXPECT function.
- RECV_MSG_FRM_PLC: This event handler is triggered upon receiving networks messages sent from the PLC. If outgoing messages from the PLC are sent periodically, it increments a message counter and sets a timer which will be called after a duration of T_{Max}. It also compares the timestamp of the message with the previous message of the same type. If the time difference is smaller than T_{Min}, it raises an injection alarm. Otherwise it proceeds to check whether the contents of the message match the expected variable (as calculated by the DO_WE_EXPECT function). If the contents do not match, it raises a modification alarm.
- PERIODIC_TIMER_FINISH: This function is triggered when the timer (set in RECV_MSG_FRM_PLC) expires. When this function executes, it compares the current message counter with the recorded value of the message counter when the timer was started. If the difference is more than some threshold, it raises an injection alarm. If the difference between the two counters is zero, it raises a deletion alarm (i.e., some packet is deleted or delayed).

TABLE I
DETECTION LOGIC OF COTORU'S IDS RULES TEMPLATE

Events	Logic
Incoming message to PLC RECV_MSG_TO_PLC	Record message values Call DO_WE_EXPECT()
Outgoing message from PLC RECV_MSG_FRM_PLC	Increase message counter Start periodic timer If message time difference $< T_{Min}$ raise Injection Alarm If message value != expected value raise Modification Alarm
Periodic timer PERIODIC_TIMER_FINISH	If counter difference $>$ threshold: raise Injection Alarm If counter difference $==$ 0: raise Deletion Alarm

Generation of IDS rules. To streamline the process of generating IDS rules automatically, we have created template functions that serve as placeholders for instantiating the system-specific information (e.g., event handlers for network messages, global variables replicating the PLC variables). In particular, the global variable placeholders in the template are:

- K_1 - K_n: Each K_x stores Boolean value corresponding to the evaluation of the expression in some column of the Key.
- I_1 - I_p: Each I_x stores the value corresponding to some incoming message to the PLC.
- O_1 - O_m: Each O_x stores the calculated value of the corresponding variable in the next state.

First, we instantiate global variables in the template that are related to the incoming and outgoing messages. In particular, we replace the $I_1 \ldots I_p$ variable placeholders with the PLC variable names (and initial values) representing the incoming message variables. For each outgoing message type X from the PLC, we use its corresponding PLC timing profile to define the following global variables, which will be used for detecting attacks that affect its behavior over the time domain:

- timer_period_X: The maximum inter-arrival time between two consecutive messages of type X,
- timer_min_X: The minimum inter-arrival time between two consecutive messages of type X,
- prev_time_X: The timestamp of the previous message of type X, and
- inject_counter_X: The number of messages of type X that has been received.

In our template, the RECV_MSG_TO_PLC and RECV_MSG_FRM_PLC events are placeholder functions that represents the event handlers for processing messages. So a RECV_MSG_TO_PLC event is generated for each type of incoming message to the PLC, and a RECV_MSG_FRM_PLC event is generated for each type of outgoing message from the PLC. The RECV_MSG_TO_PLC and RECV_MSG_FRM_PLC event name will be replaced with the corresponding event handler names defined in the event engine level. Finally, the PERIODIC_TIMER_FINISH event is a placeholder function that is used to keep track of the periodic sending of the outgoing message from the PLC. We duplicate the event (e.g., PERIODIC_TIMER_FINISH_X) for each outgoing message X from the PLC.

Next, we use the state transition table to instantiate the

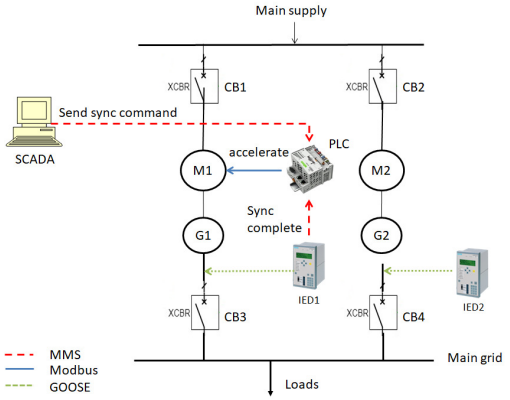


Fig. 2. Data flow of synchronizing generator G1 to the grid

relevant global variables. We replace the $O_1 \dots O_m$ variable placeholders with the PLC variable names that appear in NEXTSTATE in the state transition table. For each column x of Key in the state transition table, we create a global variable K_x. In the template, we also define the global variable STATE_TRANSITION_TABLE which maps the different value combinations of $K_1 \dots K_n$ (representing the value of the corresponding path constraints) to some integer index that points to the corresponding updating logic. We replicate the key entries in the global variable STATE_TRANSITION_TABLE. The NEXTSTATE entries, i.e., expressions, are placed into a switch-case block in the DO_WE_EXPECT function template. The indices into that block are filled into the STATE_TRANSITION_TABLE accordingly. Finally, the Boolean expressions of each Key in the state transition table is replicated in the DO_WE_EXPECT function. Note that we only need to instantiate the DO_WE_EXPECT function and STATE_TRANSITION_TABLE once.

V. A POWER GRID CASE STUDY

We apply CoToRu to a high-fidelity power grid testbed to demonstrate its applicability and performance in a practical setting. The power grid testbed consists of real-world equipment for power generation, transmission, distribution, and consumption. Specifically, the testbed consists of multiple generators rated at 10kW each. The generated electricity is distributed to programmable load banks to emulate different loading profiles in the power grid. Each stage of the testbed has its own set of switches, PLC, and IEDs to support different grid operations. In this case study, we focus on the PLC deployed in the distribution stage of the testbed, particularly its logic for controlling a generator synchronization process.

PLC Logic: Generator Synchronization. Synchronization is a process of connecting two or more generators to the grid to supply a larger load. The generators are synchronized when their frequency and phase angle match. Otherwise, an out-of-sync condition may damage critical equipment as in [43] and the 2007 Aurora generator test [44].

As shown in Fig. 2, the synchronization process involves SCADA workstation, PLC, and IEDs. The two generators are connected to two VSD-driven motors that are both controlled by the PLC using the Modbus protocol. The PLC uses the MMS protocol in IEC 61850 standard [41] to communicate with the SCADA and IEDs. The IEDs communicate with one another using the Generic Object Oriented Substation Events (GOOSE) protocol specified in the IEC 61850 standard.

```
1 IF VSD1_Start = TRUE THEN
2     MBCFG_ModbusVSD1.VSD1_Command := 16#1D4C;
3 END_IF;
4 IF SCADA_Sync_Activate = TRUE THEN
5     MBCFG_ModbusVSD1.VSD1_Command :=16#1D4E;
6 END_IF;
7 IF In_Sync = TRUE THEN
8     MBCFG_ModbusVSD1.VSD1_Command :=16#1D4C;
9 END_IF;
```

Fig. 3. Synchronization logic in ST language.

TABLE II
STATE TRANSITION TABLE FOR SYNCHRONIZATION LOGIC

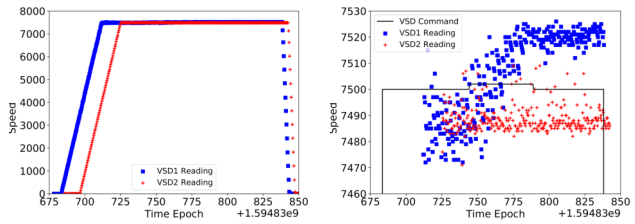
Key			NextState
VSD1_Start	SCADA_Sync_Activate	In_Sync	VSD1_Command
0	0	0	*VSD1_Command*
0	0	1	7500
0	1	0	7502
0	1	1	7500
1	0	0	7500
1	0	1	7500
1	1	0	7502
1	1	1	7500

Fig. 3 is a code snippet of the synchronization logic inside the PLC program. The assumption is that $G2$ is already connected to the grid and is spinning at 7500. The goal is to connect $G1$ to supply power in parallel. In lines 1-3 of the code, if "VSD1_Start" in the MMS message sent by SCADA is true, the PLC will start the VSD-driven generator $G1$ by commanding $M1$ (VSD1) to rotate at a speed of $1D4C_{16}$ (or 7500). Then in lines 4-6, when variable "SCADA_Sync_Activate" in the MMS message is true, the PLC will start the synchronization process by sending a Modbus message to accelerate $M1$ at a speed of $1D4E_{16}$ (or 7502)[1]. The changes in motor speed will cause the phase angle of $G1$ to catch up with the reference generator $G2$. The phase angle of $G1$ is monitored by IED1. When the phase angle difference between $G1$ and $G2$ reaches approximately zero, the IED will close the circuit breaker via GOOSE and send an MMS message to inform the PLC that synchronization is complete. So, in lines 7-9, if variable "In_Sync" in the MMS message sent by IED1 is true, the PLC will send a Modbus message to reset VSD1 to nominal speed, i.e., 7500, to complete the synchronization process.

To apply CoToRu to this case study, the state transition table generator first converts the ST code into a symbolic execution tree that models the causal relationship between the control commands to be sent and the various input signals and internal control logic. That symbolic execution tree is then used to generate a state transition table, shown in Table II.

Analysis of Network Traces. While the testbed provides network traces in the public domain, the available traces do

[1]We note that 7500 and 7502 are speed parameters used by the PLC for communication and do not represent the actual speed values for the generators to spin. For 7500 (the nominal value), the generator is instructed to output AC power at a frequency of 50 Hz.



(a) (b)

Fig. 4. (a) Speed readings from startup, synchronization, to shutdown. (b) Speed readings and command during synchronization.

TABLE III
GENERATED ATTACK TRACES

Attack Tactic	Packet Operation	Description
WF1	Modify	Modify Modbus command to VSD1 to 7500
WF2	Delay	Delay VSD1 Modbus command by 75 s
WF3	Delete	Delete VSD1 Modbus command
DS1	Modify	Modify Modbus command to VSD1 to 7577 and VSD2 to 7425
DT1	Inject	Vary VSD1 speed value between 0 and 8000
DT2	Modify	Alternate VSD1 speed value between 0 and 7500
FC1	Inject	Modify VSD1 speed value to 8200 just before In_Sync command
FC2	Modify	Modify VSD1 speed value to 0 just before In_Sync command

not contain all the messages sent to the PLC during the synchronization process. Hence, we conducted experiments on the testbed to collect the required traces for analysis. We repeated the synchronization process from the starting of the motors to the synchronization of the generators and then the shutdown of the motors. Fig. 4 shows the motor speed measurements in one trace. During startup, the speed of the motor will increase linearly from 0 to the specified speed. During normal operation and synchronization, the speed measurement fluctuates by $\pm 1\%$. During shutdown, the motor speed decreases linearly back to 0.

Attack Strategies. To attack the synchronization process, we have identified four attack strategies: (1) preventing the incoming generator from synchronizing (abbreviated as **Wait Forever (WF)**), (2) delaying the synchronization of incoming generator (abbreviated as **Delayed Sync (DS)**), (3) allowing the synchronization to complete but with frequent speed changes to cause short-term damage to the motors (abbreviated as **Damaging Transient (DT)**), and (4) causing unbalanced load upon completion of synchronization (abbreviated as **Final Change (FC)**). These four strategies target the speed value of the Modbus command sent to the motors (VSDs) through *injection*, *modification*, *delay*, and *deletion* of messages. The details of each attack tactic are summarized in Table III.

Generation of Attack Traces. We use the existing traces from the testbed (in public domain) and our captured traces as templates for the structures of the MMS and Modbus packets. Our Scapy packet generator reads in the packets from the two capture files and sends out modified packets based on the attack tactics described in Table III. We generated approximately 500 traces for each attack tactic. We also generated 5000 traces of normal behavior.

VI. EVALUATION

We implemented the IDS rules generated by CoToRu in a Zeek script that runs on the script interpreter layer. Zeek's standard distribution already contains a Modbus analyzer. To parse MMS traffic, we used the MMS analyzer proposed in [42]. To test our approach, we deployed a client to replay our attack traces to the Hirschmann industrial switch. The switch then forwards the MMS and Modbus packets to Zeek for analysis. Zeek event engine decodes the packets and triggers the corresponding event handler in the Zeek script to execute the IDS rules. The test environment is shown in Fig. 5. We evaluate the performance of CoToRu in terms of detection accuracy and compare it with EDMAND [13], LSTM [15], and an invariant-based IDS. We chose LSTM because it is capable of learning temporal dependencies involving time-evolving variables, e.g., motor speed.

A. Baseline Approaches

1) Statistical-based IDS: EDMAND allows the definition of statistical-based features at different levels. We identified five features and incorporated them into EDMAND's statistical framework for detecting synchronization attacks.

- **Synchronization duration (f1):** We use simple thresholding to detect anomalies. As the duration can vary from 0 to 75 seconds, we set the duration threshold at 100 seconds to allow for network and processing delays.
- **Modbus speed command (f2):** We use DenStream clustering algorithm [45] to group data points into clusters. An alarm will be raised when a data point cannot be classified into existing clusters. We set the cluster radius to 2 so that 7500 and 7502 will be grouped into one cluster, and value 0 will form the second cluster.
- **Reported motor speed (f3):** We characterize the steady-state speed readings as a unimodal distribution and use the exponential moving average (EMA) and standard deviation (SD) with a smoothing factor (α) of 0.01 to model the speed readings. The smoothing factor accounts for decreasing weights as the data evolves over time.
- **Interarrival time (f4):** Since the commands to the motor are sent periodically, we use the EMA and SD with a smoothing factor of 0.01 to detect anomalies.
- **Phase angle difference (f5):** Since the phase angle difference changes constantly, we use the exponential mean and SD to characterize that data series.

For features f3, f4, and f5, if a new value of the feature differs by more than one SD from its mean, it is labeled as an anomaly. We adapted the EDMAND code [46] and implemented the various detection mechanisms in Python. We carefully tuned the parameters in EDMAND's statistical models and configured the overall detection threshold of EDMAND to 0.95, which gives the best performance that balances false positive and false negative.

2) Machine learning-based IDS: We formulate the detection problem as a binary classification problem, and use the Keras Python library and TensorFlow framework to build a stacked bidirectional LSTM model. Our model consists of three hidden stacked layers, where the output of one layer is used as the input to the next layer. For each layer, we use the bidirectional LSTM to process the time-series data in both

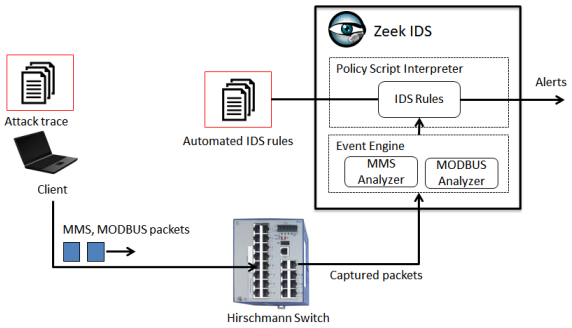


Fig. 5. Test environment for evaluating CoToRu.

the forward and backward directions. Finally, we define two dense layers and use the sigmoid activation function to make a prediction i.e., class 0 (normal) or class 1 (attack). To prevent overfitting, a dropout layer is added after each hidden layer to improve the robustness of training.

We generated 5000 normal and 4000 attack traces, a total of 9,000 samples, and divided them into a 90-10% split between training and validation. The attack traces are composed of the attack tactics described in Table III, with 500 samples for each tactic. We used a vector of six features to train a classifier: synchronization duration, phase angle difference between the two VSDs, and for each VSD, the speed command, and the recorded speed values. Since this is a classification problem, we use the binary cross entropy loss function and the Adam optimization algorithm to train the model. We trained the model for 100 epochs using a batch size of 64 and a learning rate of 0.0001. Once the model is trained, we generate a test set consisting of 500 normal traces and 500 attack traces (for each attack tactic) to evaluate the accuracy of the predictions made by the trained model. When generating new attack traces, we used different speed settings without changing the attack strategies to build more diverse test data. Our goal is to evaluate how well the LSTM model generalizes to new attacks.

3) Invariant-based IDS: We defined a set of invariants, based on expert guidance, that reflect the physical synchronization process. Those invariants check that the state of the system during synchronization satisfies the physical changes undergone by the motors. In particular, the phase angle between the two motors should close at a rate relative to the difference in the set motor speed and the two motors will then synchronize within an upper bound time limit. The speed of the two motors should also remain within a certain range. Specifically, we define three stateful invariants (the bounds in the invariants cater for measurement noise):

- **RPM:** During synchronization, the speed recorded by the VSD must be within 60 of the default 7500 value.
- **Phase Angle Variation:** During synchronization, the difference between the rate of change of phase angle for VSD1 and VSD2 must be between 3.2° and 6°.
- **Synchronization Duration:** After synchronization is started, the difference in phase angle of the two motors must be below 5° after 100 seconds has elapsed, i.e., synchronization has been achieved.

We implement the invariant-based IDS as a Python program that takes the network traces (attack and normal) as input. The IDS processes each packet and will only start checking the invariants after receipt of a "SCADA_Sync_Activate" message. If any invariant is unsatisfied, an alarm is raised.

B. Detection Performance Results

Metrics. To compare CoToRu with the baseline approaches, we use the precision and recall metrics. Precision calculates how many attack predictions made by the classifier are actual attacks, whereas recall measures how many of the actual attacks are classified as true positives. For features that respond slower to the effects of attacks (e.g., synchronization duration, reported motor speed, and phase angle difference), we evaluate the precision and recall based on a sequence of packet events instead of a per-packet basis. Specifically, we consider the packet sequence between the first attack packet and the last attack packet. If a classifier raises an alarm in response to any attack during that packet sequence, then both recall and precision are 100%. If the classifier cannot detect any attacks, recall is 0% and precision will be undefined ($undef$) due to TP and FP being 0. For all the approaches, we mark the use of a packet sequence for detection (with an *) in Table IV. We also measure the detection delay, $DDelay$, which is the time delay for a classifier to raise an alarm from the onset of an attack. In situations where there is no detection i.e., recall is 0, the evaluation of detection delay is not applicable (denoted as N.A.). A lower detection delay is desirable.

EDMAND: Table IV shows that none of the features can detect all attacks. Synchronization duration (f1) can detect delay, deletion, and variants of modification attacks that prevent synchronization from occurring. Modbus speed commands (f2) and speed readings (f3) can detect modification attacks whose values are far away from the mean. Interarrival time (f4) can detect delay and injection attacks. Phase angle difference (f5) can detect modification or injection attacks that result in a phase difference that is far from the mean of 4.8°. Even for those detected attack tactics, EDMAND suffers from both false positives and false negatives. Despite careful tuning of the threshold, its performance is inferior to LSTM which we will present next. Also, the detection delay depends on the set thresholds and the packet sending rate. So most detections are not immediate except for Modbus speed commands (f2).

LSTM: Our results show that LSTM can achieve an average precision and recall of about 98% and 95% respectively, if we look at all the attacks together. More specifically, it can achieve 100% recall for all the *Wait Forever* attack strategies, but can miss 15% of attacks for FC1. Its high precision (about 98%) can be explained by the distinct difference in pattern between the normal data and the more identifiable attack data, which allows LSTM to tell them apart. However, for more stealthy attacks (i.e., FC1) which look similar to normal data, the recall is $< 90\%$, which means it can miss a substantial percentage of attacks which could lead to cascading failures and damage to the generators. That makes it unsuitable for protecting critical ICS. From an operational aspect, even a 98% precision may not be sufficient as it could result in high

TABLE IV
DETECTION PERFORMANCE OF THE THREE BASELINE APPROACHES AND COTORU FOR DIFFERENT ATTACKS.

Attack Tactic	Packet Operation	EDMAND														
		f1*			f2			f3*			f4			f5*		
		Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay
WF1	Modify	100	100	100.02	undef	0	N.A.	undef	0	N.A.	0	0	N.A.	100	100	1.04
WF2	Delay	100	100	100.16	undef	0	N.A.	undef	0	N.A.	8.33	100	50.94	100	100	1.42
WF3	Delete	100	100	100.09	undef	0	N.A.	undef	0	N.A.	25	100	53.59	100	100	1.23
DS1	Modify	undef	0	N.A.	100	0.28	≈0	undef	0	N.A.	0	0	N.A.	undef	0	N.A.
DT1	Inject	undef	0	N.A.	100	30	≈0	100	100	0.33	52.63	100	≈0	100	100	2.63
DT2	Modify	undef	0	N.A.	undef	0	N.A.	100	100	0.39	0	0	N.A.	100	100	2.50
FC1	Inject	undef	0	N.A.	100	0.21	≈0	100	100	0.43	50	100	≈0	undef	0	N.A.
FC2	Modify	undef	0	N.A.	undef	0	N.A.	100	100	0.41	0	0	N.A.	undef	0	N.A.

Attack Tactic	Invariant									Stacked Bi-LSTM*			Stacked Bi-LSTM* (using reduced training set)			CoToRu		
	RPM*			PAV*			SD*											
	Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay	Prec	Recall	DDelay
WF1	undef	0	N.A.	100	100	1.048	100	100	99.84	98.6	100	320.60	99.6	97.6	330.95	100	100	≈0
WF2	undef	0	N.A.	100	100	0.02	100	100	100.9	98.0	100	247.58	99.4	100	239.41	100	100	≈0
WF3	undef	0	N.A.	100	100	1.23	100	100	102	99.0	100	320.94	99	100	327.71	100	100	≈0
DS1	100	100	0.01	undef	0	N.A.	undef	0	N.A.	99.0	95.6	332.08	99	97	335.56	100	100	≈0
DT1	100	100	0.33	100	100	2.62	undef	0	N.A.	98.4	99.8	320.62	99.8	96.2	327.98	100	100	≈0
DT2	100	100	0.38	100	100	2.49	undef	0	N.A.	98.4	99.4	321.47	100	98.4	323.08	100	100	≈0
FC1	undef	0	N.A.	undef	0	N.A.	undef	0	N.A.	97.3	85.0	272.63	96.5	33.4	280.17	100	100	≈0
FC2	undef	0	N.A.	undef	0	N.A.	undef	0	N.A.	99.4	99.8	295.87	98.2	67.2	282.77	100	100	≈0

Note: Features marked with * are evaluated over a packet sequence. Prec=$TP/(TP+FP)$ and recall=$TP/(TP+FN)$
Values in Prec, and Recall columns are in percentage (%). 'undef' in Prec columns denotes undefined, meaning the IDS solution cannot detect any attack.
Detection delay ($DDelay$) is measured in seconds.'N.A.' in $DDelay$ columns denotes not applicable since there is no detection at all.

number of false alarms when deployed to a large system. Another drawback of the approach is that it needs to look at a packet sequence over a sufficiently long time period in order to make a decision, resulting in an average delay of around 300 seconds in our experiments.

In addition, if LSTM is not trained on examples of unseen attacks, then LSTM may fail in detecting those attacks when it occurs. To demonstrate LSTM's performance on unseen attacks, we retrained the LSTM model using a reduced training set by excluding one attack strategy at a time during the training phase. The excluded trace is then used to evaluate the model's accuracy. The third column in Table IV shows a further drop in recall for most cases, especially for the *Final Change* attack strategy. The low recall indicates that a large proportion of attacks were not identified by LSTM.

Invariant-based IDS: The *Final Change* attack strategies are undetectable by all the invariants. That is because that attack strategy relies on synchronization having been achieved before performing the attack. Thus, the three invariants will not be violated. Individually, the RPM invariant (RPM) is able to detect all the other attack strategies except the *Wait Forever* attack strategies because under those WF attacks the speed is set to revolve around the motor's normal operating conditions. The Phase Angle Variation invariant (PAV) cannot detect the *Delayed Sync* attack strategy because the phase angle invariant is carefully preserved by the attacker. Finally, the Sync Duration (SD) invariant performs the worst in accuracy. That invariant only detects the *Wait Forever* attack strategy because synchronization is never achieved. Also, it only detects that attack after the pre-set threshold of 100 seconds.

CoToRu: Table IV shows that CoToRu achieves 100% precision and recall in detecting all the attack scenarios with minimum detection delay. That is because the IDS rules generated by CoToRu exactly capture the PLC logic that describes the operational behavior of the PLC at a fine-grained packet level. None of the other approaches can achieve this. In addition, compared with all the other approaches, we do not need to calibrate any threshold to make a trade-off between detection accuracy and false positive rate, nor does CoToRu need any training data.

VII. CONCLUSION

This paper presents CoToRu for automating the generation of IDS rules to secure ICS against compromised PLC. CoToRu analyzes the PLC code to automatically generate a state transition table that models the operational behavior, and it uses an IDS rule template to automatically generate rules that can directly be used by Zeek IDS to detect injection, modification, deletion, and delay of command messages. Using a power grid testbed as a case study, we showed CoToRu's IDS rules can achieve 100% detection accuracy with sub-millisecond detection delay against a range of advanced attacks, which cannot be achieved by other existing approaches. Overall, CoToRu achieves three desirable objectives for protecting ICS, i.e., accurate and real-time detection of compromised PLC, minimal manual effort to configure and maintain, and easy/secure deployment to a standard NIDS solution. Our plans for future work is to extend the CoToRu toolchain to a water treatment plant and look at building a state transition table for modeling the behaviors of multiple PLCs to help detect more complicated, multi-stage attacks.

ACKNOWLEDGMENT

This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under the Energy Programme and administrated by the Energy Market Authority (EP Award No. NRF2017EWT-EP003-047), in part by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, and in part by the SUTD Start-up Research Grant (SRG Award No: SRG ISTD 2020 157).

REFERENCES

[1] J. Giraldo, E. Sarkar, A. A. Cardenas, M. Maniatakos, and M. Kantarcioglu, "Security and privacy in cyber-physical systems: A survey of surveys," *IEEE Design & Test*, vol. 34, no. 4, pp. 7–17, 2017.

[2] A. Nourian and S. Madnick, "A systems theoretic approach to the security threats in cyber physical systems applied to stuxnet," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 2–13, 2015.

[3] Y. Hu, A. Yang, H. Li, Y. Sun, and L. Sun, "A survey of intrusion detection on industrial control systems," *International Journal of Distributed Sensor Networks*, vol. 14, no. 8, 2018.

[4] H. C. Tan, C. Cheh, B. Chen, and D. Mashima, "Tabulating cybersecurity solutions for substations: Towards pragmatic design and planning," in *IEEE Innovative Smart Grid Technologies - Asia*, 2019, pp. 1018–1023.

[5] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A survey of physics-based attack detection in cyber-physical systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 76, 2018.

[6] H. Lin, A. Slagell, Z. T. Kalbarczyk, P. W. Sauer, and R. K. Iyer, "Runtime semantic security analysis to detect and mitigate control-related attacks in power grids," *IEEE Transactions on Smart Grid*, vol. 9, no. 1, pp. 163–178, 2016.

[7] S.-W. Hsiao, Y. S. Sun, M. C. Chen, and H. Zhang, "Cross-level behavioral analysis for robust early intrusion detection," in *IEEE International Conference on Intelligence and Security Informatics*, 2010, pp. 95–100.

[8] P. Schneider and K. Böttinger, "High-performance unsupervised anomaly detection for cyber-physical system networks," in *Proc. of the ACM Workshop on Cyber-Physical Systems Security and PrivaCy*, 2018, pp. 1–12.

[9] H. Lahza, K. Radke, and E. Foo, "Applying domain-specific knowledge to construct features for detecting distributed denial-of-service attacks on the goose and mms protocols," *International Journal of Critical Infrastructure Protection*, vol. 20, pp. 48–67, 2018.

[10] "The Zeek Network Security Monitor," accessed 2020-06-03. [Online]. Available: https://www.zeek.org/

[11] H. Lin, A. Slagell, C. Di Martino, Z. Kalbarczyk, and R. K. Iyer, "Adapting bro into scada: building a specification-based intrusion detection system for the dnp3 protocol," in *Proc. of the Annual Cyber Security and Information Intelligence Research Workshop*. ACM, 2013, p. 5.

[12] J. Hong, C.-C. Liu, and M. Govindarasu, "Detection of cyber intrusions using network-based multicast messages for substation automation," in *ISGT 2014*. IEEE, 2014, pp. 1–5.

[13] W. Ren, T. Yardley, and K. Nahrstedt, "Edmand: Edge-based multi-level anomaly detection for scada networks," in *IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids*, 2018, pp. 1–7.

[14] J. Zhang, S. Gan, X. Liu, and P. Zhu, "Intrusion detection in scada systems by traffic periodicity and telemetry analysis," in *IEEE Symposium on Computers and Communication*, 2016, pp. 318–325.

[15] J. Goh, S. Adepu, M. Tan, and Z. S. Lee, "Anomaly detection in cyber physical systems using recurrent neural networks," in *IEEE 18th International Symposium on HASE*, 2017, pp. 140–145.

[16] Y. Chen, C. M. Poskitt, and J. Sun, "Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system," in *IEEE Symposium on Security and Privacy*, 2018, pp. 648–660.

[17] X. Lou, C. Tran, D. K. Yau, R. Tan, H. Ng, T. Z. Fu, and M. Winslett, "Learning-based time delay attack characterization for cyber-physical systems," in *IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids*, 2019, pp. 1–6.

[18] S. Adepu and A. Mathur, "Distributed attack detection in a water treatment plant: method and case study," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[19] M. Umer, A. Mathur, K. Junejo, and S. Adepu, "Generating invariants using design and data-centric approaches for distributed attack detection," *International Journal of Critical Infrastructure Protection*, vol. 28, p. 100341, 02 2020.

[20] C. Feng, V. R. Palleti, A. Mathur, and D. Chana, "A systematic framework to generate invariants for anomaly detection in industrial control systems," in *Network and Distributed Systems Security Symposium*, 2019.

[21] N. Govil, A. Agrawal, and N. O. Tippenhauer, "On ladder logic bombs in industrial control systems," in *Proc. of the Workshop on the Security of Industrial Control Systems and of Cyber-Physical Systems*, 2017.

[22] NIST, "CVE-2012-6068 Detail," Dec 2012. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2012-6068

[23] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! attacking plcs with physical model aware rootkit." in *Network and Distributed System Security Symposium*, 2017.

[24] Y. Mo and B. Sinopoli, "Integrity attacks on cyber-physical systems," in *Proceedings of the 1st international conference on High Confidence Networked Systems*, 2012, pp. 47–54.

[25] C. M. Ahmed, M. Ochoa, J. Zhou, A. P. Mathur, R. Qadeer, C. Murguia, and J. Ruths, "Noiseprint: Attack detection using sensor and process noise fingerprint in cyber physical systems," in *Proc. of the ASIACCS*, 2018, pp. 483–497.

[26] C. Murguia and J. Ruths, "Characterization of a cusum model-based sensor attack detector," in *IEEE Conference on Decision and Control*, 2016, pp. 1303–1309.

[27] Y. Hu, H. Li, H. Yang, Y. Sun, L. Sun, and Z. Wang, "Detecting stealthy attacks against industrial control systems based on residual skewness analysis," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 74, 2019.

[28] D. Pliatsios, P. Sarigiannidis, T. Lagkas, and A. G. Sarigiannidis, "A survey on scada systems: Secure protocols, incidents, threats and tactics," *IEEE Communications Surveys & Tutorials*, 2020.

[29] V. Y. Pillitteri and T. L. Brewer, "Guidelines for smart grid cybersecurity," NIST, Tech. Rep., 2014.

[30] K. Stouffer, J. Falco, and K. Scarfone, "Guide to industrial control systems security," *NIST special pub.*, vol. 800, no. 82, p. 16, 2011.

[31] "IEC 61131-3 - Programmable controllers," Jan 2020. [Online]. Available: https://webstore.iec.ch/publication/62427

[32] J. H. Castellanos, M. Ochoa, and J. Zhou, "Finding dependencies between cyber-physical domains for security testing of industrial control systems," in *Proc. of the 34th Annual Computer Security Applications Conference*. ACM, 2018, p. 582–594.

[33] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," *SIGPLAN Not.*, vol. 10, no. 6, p. 234–245, Apr. 1975.

[34] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[35] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2003, p. 553–568.

[36] S. McLaughlin and P. McDaniel, "Sabot: Specification-based payload generation for programmable logic controllers," in *Proc. of the ACM Conference on Computer and Communications Security*, New York, NY, USA, 2012, p. 439–449.

[37] S. Mclaughlin, S. Zonouz, D. Pohly, and P. McDaniel, "A trusted safety verifier for process controller code," in *NDSS*, 01 2014.

[38] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne, and Z. M. Mao, "Towards automated safety vetting of plc code in real-world plants," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 522–538.

[39] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[40] 3S-Smart Software Solutions GmbH, "Codesys." [Online]. Available: https://www.codesys.com

[41] "IEC 61850 - Communication networks and systems for power utility automation," Feb 2020. [Online]. Available: https://webstore.iec.ch/publication/6028

[42] H. C. Tan, V. Mohanraj, B. Chen, D. Mashima, S. K. S. Nan, and A. Yang, "An iec 61850 mms traffic parser for customizable and efficient intrusion detection," in *IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids*. IEEE, 2021, pp. 194–200.

[43] N. K. Kandasamy, "An investigation on feasibility and security for cyber attacks on generator synchronization process," *IEEE Transactions on Industrial Informatics*, 2019.

[44] M. Zeller, "Common questions and answers addressing the aurora vulnerability," in *DistribuTECH Conference*, 2011.

[45] F. Cao, M. Estert, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *Proc. of the SIAM international conference on data mining*. SIAM, 2006, pp. 328–339.

[46] "ITI/EDMAND," Oct 2020. [Online]. Available: https://github.com/ITI/EDMAND?files=1