SoK: Attacks on Industrial Control Logic and Formal Verification-Based Defenses

Ruimin Sun
Northeastern University
r.sun@northeastern.edu

Alejandro Mera
Northeastern University
mera.a@northeastern.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

David Choffnes
Northeastern University
choffnes@ccs.neu.edu

***Abstract*—Programmable Logic Controllers (PLCs) play a critical role in the industrial control systems. Vulnerabilities in PLC programs might lead to attacks causing devastating consequences to the critical infrastructure, as shown in Stuxnet and similar attacks. In recent years, we have seen an exponential increase in vulnerabilities reported for PLC control logic. Looking back on past research, we found extensive studies explored control logic modification attacks, as well as formal verification-based security solutions.**

We performed systematization on these studies, and found attacks that can compromise a full chain of control and evade detection. However, the majority of the formal verification research investigated ad-hoc techniques targeting PLC programs. We discovered challenges in every aspect of formal verification, rising from (1) the ever-expanding attack surface from evolved system design, (2) the real-time constraint during the program execution, and (3) the barrier in security evaluation given proprietary and vendor-specific dependencies on different techniques. Based on the knowledge systematization, we provide a set of recommendations for future research directions, and we highlight the need of defending security issues besides safety issues.

***Index Terms*—PLC, attack, formal verification**

1. Introduction

Industrial control systems (ICS) are subject to attacks sabotaging the physical processes, as shown in Stuxnet [33], Havex [46], TRITON [31], Black Energy [8], and the German Steel Mill [63]. PLCs are the last line in controlling and defending for these critical ICS systems.

However, in our analysis of Common Vulnerabilities and Exposures (CVE)s related to control logic, we have seen a fast growth of vulnerabilities in recent years [86]. These vulnerabilities are distributed across vendors and domains, and their severeness remains high. A closer look at these vulnerabilities reveals that the weaknesses behind them are not novel. As Figure 1 shows, multiple weaknesses are repeating across different industrial domains, such as stack-based buffer overflow and improper input validation. We want to understand how these weaknesses have been used in different attacks, and how existing solutions defend against the attacks.

Among various attacks, control logic modification attacks cause the most critical damages. Such attacks leverage the flaws in the PLC program to produce undesired states. As a principled approach detecting flaws in programs, formal verification has long been used to defend

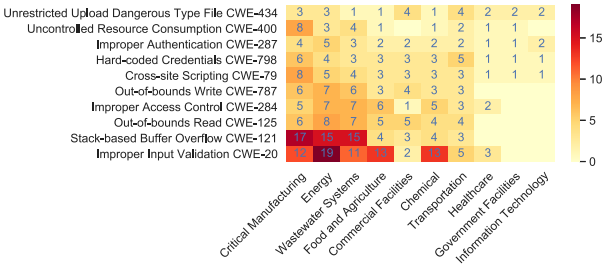

Figure 1: The reported common weaknesses and the affected industrial sectors. The notation denotes the number of CVEs.

control logic modification attacks [24], [26]. It benefits from several advantages that other security solutions fail to provide. First, PLCs have to strictly meet the real-time constraints in controlling the physical processes. This makes it impractical for heavyweight solutions to perform a large amount of dynamic analysis. Second, the physical processes are often safety-critical, meaning false positives are intolerable. Formal verification is lightweight, accurate, and suitable for graphical languages, which are commonly used to develop PLC programs.

Over the years, there have been extensive studies investigating control logic modification attacks, and formal verification-based defenses. To understand the current research progress in these areas, and to identify open problems for future research directions, we performed a systematization of current studies.

Scope of the paper. We considered studies presenting control logic modification attacks through modifying program payload (i.e. program code), or feeding special input data to trigger program design flaws. We also considered studies presenting formal verification techniques to protect the affected programs, including behavior modeling, state reduction, specification generation, and verification. Formal verification of network protocols is out of the scope of the paper. We selected the literature based on three criteria: (1) the study investigates control logic modification attacks or formal verification-based defenses, (2) the study is impactful considering its number of citations, or (3) the study discovers a new direction for future research.

Systematization methodology. Our systematization was based on the following aspects. We use "attack" to denote control logic modification, and "defense" to denote formal verification-based defense.

- Threat model: this refers to the requirements and

2021 IEEE European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1491-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/EUROSP51992.2021.00034

DOI 10.1109/EuroSP51992.2021.00034

assumptions to perform the attacks/defenses.
- Security goal: this refers to the security properties affected by attacks/defenses.
- Weakness: this refers to the flaw triggered to perform the attacks.
- Detection to evade: this refers to the detection that fails to capture the attacks.
- Challenge: this refers to the challenges in defending the attacks—the advance of attacks, and the insufficiency of defenses.
- Defense focus: this refers to the specific research topic in formal verification, e.g. behavior modeling, state reduction, specification generation, and formal verification.

We found that control logic modification attacks could happen under every threat model and considered various evasive techniques. The attacks have been fast evolving with the system design, through input channels from the sensors, the engineering stations, and other connected PLCs. The attacks could also evade dynamic state estimations and verification techniques through leveraging implicitly specified properties. Multiple attacks [54], [64], [83] even deceived engineering stations with fake behaviors.

We also found that applying formal verification has made great progress in improving code quality [97]. However, the majority of the studies investigated ad-hoc formal verification research targeting PLC programs. These studies face challenges in many aspects of formal verification, during program modeling, state reduction, specification generation, and verification. We found many studies manually define domain-specific safety properties, and verify them based on a few simple test cases. Despite the limitation of test cases, the implicitness of properties was not well explored, even though such properties have been used to conduct input manipulation attacks [68]–[70]. Besides implicit properties, specification generation has seen challenges in catching up with program modeling, to support semantics and rules from new attack surfaces. In addition, the real-time constraint limited runtime verification in supporting temporal features, event-driven features, and multitasks. The dependency on proprietary and vendor-specific techniques resulted in ad-hoc studies. The lack of open source tools impeded thorough evaluation across models, frameworks, and real programs in industry complexity.

As a call for solutions to address these challenges, we highlight the need of defending security issues besides safety issues, and we provide a set of recommendations for future research directions. We recommend future research to pay attention to plant modeling and to defend against input manipulation attacks. We recommend the collaboration between state reduction and stealthy attack detection. We highlight the need for automatic generation of domain-specific and incremental specifications. We also encourage more exploration in real-time verification, together with more support in open-source tools, and thorough performance and security evaluation.

Our study makes the following **contributions**:

- Systematization of control logic modification attacks and formal verification-based defenses in the last thirty years.

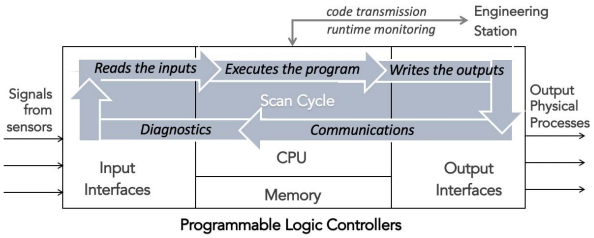

Figure 2: The architecture of a PLC.

- Identifying the challenges in defending control logic modification attacks, and barriers existed in current formal verification research.
- Pointing out future research directions.

The rest of the paper is organized as follows. Section 2 briefly describes the background knowledge of PLCs and formal verification. Section 3 describes the motivation of this work and the methodology of the systematization. Section 4 and Section 5 systematize existing studies on control logic modification attacks, and formal verification-based defenses—categorized on threat models and the approaches to perform the attack/defense. Section 6 provides recommendations for future research directions to counter existing challenges. Section 7 concludes the paper.

2. Background

2.1. PLC Program

2.1.1. Programming languages. IEC-61131 [87] defined five types of languages for PLC source code:
- Ladder diagram (LD),
- Structured text (ST),
- Function block diagram (FBD),
- Sequential function chart (SFC),
- Instruction list (IL).

Among them, LD, FBD, and SFC are graph-based languages. IL was deprecated in 2013. PLC programs are developed in *engineering stations*, which provide standard-compliant or vendor-specific Integrated Development Environments (IDEs) and compilers. Some high-end PLCs also support computer-compatible languages (*e.g.,* C, BASIC, and assembly), special high-level languages (*e.g.,* Siemens GRAPH5 [2]), and boolean logic languages [67].

2.1.2. Program bytecode/binary. An engineering station may compile source code to bytecode or binary depending on the type of a PLC. For example, Siemens S7 compiles source code to proprietary MC7 bytecode and uses PLC runtime to interpret the bytecode, while CODESYS compiles source code to binaries (i.e. native machine code) [55]. Unlike conventional software that follows well-documented formats, such as Executable and Linkable Format (ELF) for Linux and Portable Executable (PE) for Windows, the format of PLC binaries is often proprietary and unknown. Therefore, further exploration requires reverse engineering.

2.1.3. Scan cycle. Unlike conventional software, a PLC program executes by infinitely repeating a *scan cycle* that

consists of three steps (as Figure 2 shows). First, the *input scan* reads the inputs from the connected sensors and saves them to a data table. Then, the *logic execution* feeds the input data to the program and executes the logic. Finally, the *output scan* produces the output to the physical processes based on the execution result.

The scan cycle must comply with strict predefined timing constraints to enforce the real-time execution. The I/O operations are the critical part in meeting the cycle time.

2.1.4. Hardware support. PLCs adopt a hierarchical memory, with predefined addressing scheme associated with physical hardware locations. PLC vendors may choose different schemes for I/O addressing, memory organization, and instruction sets, making it hard for the same code to be compatible across vendors, or even models within the same vendor.

2.2. PLC program security

PLCs interact with a broad set of components, as Figure 3 shows. They are connected to sensors and actuators to interact with and control the physical world. They are connected to supervisory human interfaces (e.g. the engineering station) to update the program and receive operator commands. They may also be interconnected in a subnet. These interactions expose PLCs to various attacks. For example, communication between the engineering station and the PLC may be insecure, the sensors might be compromised, and the PLC firmware can be vulnerable.

2.2.1. Control logic modification. Our study considers control logic modification attacks, which we define as attacks that can change the behavior of PLC control logic. Control logic modification attacks can be achieved through *program payload/code modification* and/or *program input manipulation*. The payload modification can be applied to program source code, bytecode or binary (Section 2.1). The input manipulation can craft input data to exploit existed design flaws in the program to produce undesired states. The input may come from any interacting components showed in Figure 3.

Defending against these attacks is challenging. As we mentioned earlier, PLCs have to strictly maintain the scan cycle time to control the physical world in real-time. This requirement overweights security solutions requiring a large amount of dynamic analysis. Moreover, the security solution has to be accurate, since the controlled physical processes are critical in the industry, making false positives less tolerable.

2.2.2. Formal verification. Formal verification is a lightweight and accurate defense solution, which is often tailored for graphical languages. This makes it suitable to defend against control logic modification attacks.

Formal verification is a method that proves or disproves if a program/algorithm meets its specifications or desired properties based on a certain form of logic [32]. The specification may contain security requirements and safety requirements. Commonly used mathematical models to do formal verification include finite state machines, labeled transition systems, vector addition systems, Petri nets, timed automata, hybrid automata, process algebra, and formal semantics of programming languages, e.g. operational semantics, denotational semantics, axiomatic semantics, and Hoare logic. In general, there are two types of formal analysis: model checking and theorem proving [45]. Model checking uses temporal logic to describe specifications, and efficient search methods to check whether the specifications hold for a given system. Theorem proving describes the system with a series of logical formulae. It proves the formulae implying the property via deduction with inference rules provided by the logical system. It usually requires more background knowledge and nontrivial manual efforts. We will describe the commonly used frameworks and tools for formal verification in later sections.

An extended background in Appendix A provides an example of an ST program controlling the traffic lights in a road intersection, an example of an input manipulation attack, and the process of using formal verification to detect and prevent it.

3. Motivation and Methodology

In this section, we first explain our focus on control logic modification attacks and formal verification-based protection. Then, we use an example to introduce our systematization methodology.

3.1. Motivation

We focus on control logic modification due to its critical impact on the PLC industry. Control logic modification covers attacks from program payload (i.e. program code) modification to data input manipulation. These attacks result from frequently reported vulnerabilities, and also cause unsafe behaviors to the critical industrial infrastructure, as Figure 1 shows.

To mitigate control logic modification attacks, extensive studies have been performed using formal methods on PLC programs. Formal methods have demonstrated uniqueness and practicality to the PLC industry. For example, Beckhoff TwinCat 3 and Nuclear Development Environment 2.0 have integrated safety verification during PLC program implementation [56]. Formal methods have also been used in the PLC programs controlling Ontario Power Generation, and Darlington Nuclear Power Generating Station [76]. Nevertheless, we found existing research to be ad-hoc, and the area is still new to the security community. We believe our systematization can benefit the community with recommendations for future research directions.

Besides formal methods, there are additional defense techniques. At the design level, one can use encrypted network communication, private sensor inputs, and isolate different functionalities of the engineering station. These protections are orthogonal to formal methods and common for any type of software/architecture. In addition, one can leverage intrusion detection techniques with dynamic analysis. Such analysis often involves complex algorithms, such as machine learning or neural networks, which require extensive runtime memory, and may introduce false positives. However, PLCs have limited memory and are

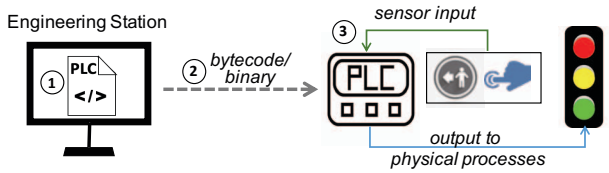

Figure 3: A PLC controlling traffic light signals.

less tolerant to false positives, given the controlled physical processes can be safety-critical. Thus, intrusion detection for PLC programs are less practical than for regular software. To improve PLC security, formal methods can cooperate with these techniques.

3.2. Methodology

3.2.1. Motivating Example. Figure 3 shows a motivating example with a PLC controlling traffic light signals at an intersection. In step ①, a PLC program developer programs the control logic code in one of the five languages described in Section 2.1.1, in an engineering station (e.g. located in the transportation department). The engineering station compiles the code into bytecode or binary based on the type of the PLC. Then in step ②, the compiled bytecode/binary will be transmitted to the PLC located at a road intersection through network communication. In step ③, the bytecode/binary will run in the PLC, by using the input from sensors (e.g. whether a pedestrian presses the button to cross the intersection), and producing output to control the physical processes (i.e. turning on/off a green light). The duration of lights will depend on whether a pedestrian presses the button to cross.

Within each step, vulnerabilities can exist which allow attackers to affect the behavior of the control logic. The following describes the threat model assumptions for attackers to perform control logic modification attacks.

3.2.2. Threat Model Assumptions. T1: In this threat model, attackers assume accesses to the *program source code*, developed in one of the languages described in Section 2.1.1. Attackers generate attacks by directly modifying the source code. Such attacks happen in the engineering station as step ① in Figure 3. Attackers can be internal staffs who have accesses to the engineering station, or can leverage vulnerabilities of the engineering station [1], [50], [51] to access it.
T2: In this threat model, attackers have no access to program source code but can access *program bytecode or binary*. Attackers generate attacks by first reverse engineering the program bytecode/binary, then modifying the decompiled code, and finally recompiling it. Such attacks happen during the bytecode/binary transmission from the engineering station to the PLC (② in Figure 3). Attackers can intercept and modify the transmission leveraging vulnerabilities in the network communication [48], [49], [52] .
T3: In this threat model, attackers have no access to program source code nor bytecode/binary. Instead, attackers can guess/speculate the logic of the control program by accessing the *program runtime environment*, including the PLC firmware, hardware, or/and Input and Output traces. Attackers can modify the real-time sensor input to the program (③ in Figure 3). Such attacks are practical since within the same domain, the general settings of the infrastructure layout are similar, and infrastructures (e.g. traffic lights) can be publicly accessible [3], [43], [69].

3.2.3. Weaknesses. Attackers usually leverage existing program weaknesses for control logic modification. The following enumerates the weaknesses.
W1: Multiple assignments for output variables. Race condition can happen when an output variable depends on multiple timers or counters. Since one timer may run faster or slower than the other, at a certain moment, the output variable will produce a non-deterministic value. In the traffic light example, this may cause the green light to be on and off in a short time, or two lights to be on simultaneously.
W2: Uninitialized or unused variables. An uninitialized variable will be given the default value in a PLC program. If an input variable is uninitialized, attackers can provide illegal values for it during runtime. Similarly, attackers can leverage unused output variables to send private information.
W3: Hidden jumpers. Such jumpers will usually bypass a portion of the program, and are only triggered on a certain (attacker-controlled) condition. The attackers can embed malware in the bypassed portion of the program.
W4: Improper runtime input. Attackers can craft illegal input values based on the types of the input variables to cause unsafe behavior. For example, attackers can provide an input index that is out-of-the-bound of an array.
W5: Predefined memory layout of the PLC hardware. PLC addressing usually follows the format [6] of a storage class (e.g. I for input, Q for output), a data size (e.g. X for $BOOL$, W for $WORD$), and a hierarchical address indicating the location of the physical port. Attackers can leverage the format to speculate the variables during runtime.
W6: Real-time constraints. The scan cycle has to strictly follow a maximum cycle time to enforce the real-time execution. In non-preemptive multitask programs, one task has to wait for the completion of another task before starting the next scan cycle. To generate synchronization attacks, attackers can create loops or introduce a large number of I/O operations to extend the execution time.

Among the weaknesses, attackers need accurate program information to exploit **W1**, **W2**, and **W3**. Therefore, these attacks usually happen in **T1**. To disguise the modification to the source code, attackers in **T1** can include these weaknesses as bad coding practice, without affecting the major control logic. The other weaknesses are usually exploited in **T2** and **T3**.

3.2.4. Security Goals. The security goals of existing studies are related to the security properties of CIA: confidentiality, integrity, and availability.
GC: Confidentiality. The attacks violate confidentiality by stealthily monitoring the execution of PLC programs leveraging existing weaknesses (e.g. W2, W3). Formal verification approaches defend accordingly.
GI: Integrity. The attacks violate integrity by causing PLC programs to produce states that are unsafe for the physical process (e.g. plant), for example, overflowing a

water tank, or fluctuating power generation [11], [58], [85]. Formal verification approaches defend by verifying (i) *generic properties* that are process-independent, and (ii) *domain-specific properties* that consider the plant model. Due to the amount of studies targeting GI, we further split GI into generic payload modification (**GI1**) without program I/O nor plant settings, generic input manipulation (**GI2**) with program I/O, domain-specific payload modification (**GI3**) with plant settings, and domain-specific input manipulation (**GI4**) with program I/O and plant settings.

GA: Availability.: The attacks violate availability by exhausting PLC resources (memory or processing power) and causing a denial-of-service. Formal verification approaches defend accordingly.

4. Systematization of Attacks

This section systematizes PLC attack methodologies under the categorization of threat models. Within each category, we discuss the goals of the attacks and the underlying weaknesses. We also summarize the challenges of attack mitigation.

4.1. Attack Methodologies

Given the exposed threat models, the following section describes the attack methodologies of existing studies according to the security goals. Table 1 summarizes these studies.

4.1.1. T1: program source code. At the source code level, the code injection or modification has to be stealthy, in a way that no observable changes would be introduced to the major functionality of the program, or masked as novice programmer mistakes. In other words, the attacks could be disguised as unintentional bad coding practices.

Existing studies [84], [88] mainly discussed attacks on graphical languages, e.g. LD, because small changes on such programs could not be easily noticed.

Serhane *et.al* [84] focused on the weak points on LD programs that could be exploited by malicious attacks. Targeting *G1* to cause unsafe behaviors, attackers could generate uncertainly fluctuating output variables, for example, intentionally introducing two timers to control the same output variable, could lead to a race condition. This could damage devices, similar to Stuxnet [33], but unpredictably. Attackers could also bypass certain functions, manually force the values of certain operands, or apply empty branches or jumps.

Targeting *G2* to stay stealthy while spying the program, attackers could use array instructions or user-defined instructions, to log critical parameters and values. Targeting *G3* to generate DoS attacks, attackers could apply an infinite loop via jumps, and use nest timers and jumps to only trigger the attack at a certain time. This attack could slow down or crash the PLC in a severe matter.

Because PLC programmers often leave unused variables and operands, both the spying program and the DoS program could leverage unused programming resources.

These attacks leveraged weaknesses *W1*-*W4*, and focused on single ladder program. To extend the attacks to multi-ladder programs, Valentine *et.al* [88] further presented attacks that could install a jump to a subroutine command, and modify the interaction between two or more ladders in a program. This could be disguised as an erroneous use of scope and linkage by a novice programmer.

In addition to code injection and modification, McLaughlin *et.al* [69] presented input manipulation attacks to cause unsafe behaviors. This study analyzed the code to obtain the relationship between the input and output variables and deducted the desired range for output variables. Attackers can craft inputs that could lead to undesired outputs for the program. The crafted inputs have to evade the state estimation detection of the PLC. Since the input manipulation happens in *T3*, and more studies discussed input manipulation attacks without using source code, we will elaborate on these attacks in *T3*.

4.1.2. T2: program bytecode/binary. Studies at this threat model mainly investigated program reverse engineering, and program modification attacks. Instead of disguising as bad coding practices, like those in *T1*, the injection at the program binary aimed at evading behavior detectors.

To design an attack prototype, McLaughlin *et.al* [70] tested on a train interlocking program. The program was reverse engineered using a format library. With the decompiled program, they extracted the fieldbus ID that indicated the PLC vendor and model, and then obtained clues about the process structure and operations. To generate unsafe behaviors, such as causing conflict states for the train signals, they targeted timing-sensitive signals and switches. To evade safety property detection, they adopted an existed solution [34] to find the implicit properties of the behavior detectors. For example, variable r depends on p and q, so a property may define the relationship between p and q, as a method to protect r. However, attackers can directly change the value of r without affecting p and q, and the change will not alarm the detector. In this way, they automatically searched for all the Boolean equations, and could generate malicious payloads based on that.

Based on this prototype, SABOT [68] was implemented. SABOT required a high-level description of the physical process, for example, “the plant contains two ingredient valves and one drain valve”. Such information could be acquired from public channels, and are similar for processes in the same industrial sector. With this information, SABOT generated a behavioral specification for the physical processes and used incremental model checking to search for a mapping between a variable within the program, and a specified physical process. Using this map, SABOT compiled a dynamic payload customized for the physical process.

Both studies were limited to Siemens devices, without revealing many details on reverse engineering. To provide more information, and support CodeSys-based programs, Keliris *et.al* [55] implemented an open-source decompiler, ICSREF, which could automatically reverse engineer CodeSys-based programs, and generate malicious payloads. ICSREF targeted PID controller functions and manipulated the parameters such as the setpoints, proportional/integral/derivative gains, initial values, etc. ICSREF inferred the physical characteristics of the con-

TABLE 1: The studies investigating control logic modification attacks.

Threat Model	Paper	Weakness	Security Goal	Attack Type	Detection to Evade	Network Access	PLC Language/Type	Tools
T1 source code	Serhane'18 [84]	W1,2,3	GI1,GC,GA	both	Programmer	ES	LD, RSLogix	N/A
	Valentine'13 [88]	W1,2,3,6	GI1,GC	passive	Programmer	N/A	LD	PLC-SF, vul. assessment
	McLaughlin'11 [70]	W4	GI3	both	State verif.	ES	generic	N/A
T2 bytecode /binary	ICSREF [55]	W4	GI3	passive	NA	ES, PLC	Codesys-based	angr, ICSREF
	SABOT [68]	W4	GI3	passive	N/A	ES, PLC	IL	NuSMV
	McLaughlin'11 [70]	W4	GI3	both	State verif.	ES, PLC	generic	N/A
T3 runtime	PLCInject [58]	W5	GC	both	N/A	ES, PLC	IL, Siemens	PLCInject malware
	PLC-Blaster [85]	W5	GC,GA	active	N/A	ES, Sensor, PLC	Siemens	PLC-Blaster worm
	Senthivel'18 [83]	W4	GI1	active	ES	ES, PLC	LD, AB/RsLogix	PyShark, decompiler Laddis
	CLIK [54]	W4	GI1	both	ES	PLC	IL, Schneider	Eupheus decompilation
	Beresford'11 [11]	W4,5	GI2	both	N/A	ES, PLC	Siemens S7	Wireshark, Metasploit
	Lim'17 [64]	W4,5	GI4,GA	active	ES	ES, PLC	Tricon PLC	LabView, PXI Chassis, Scapy
	Xiao'16 [92]	W4	GI4	both	State verif.	Sensor, PLC	generic	N/A
	Abbasi'16 [3]	W4	GI2	both	Others	N/A	Codesys-based	Codesys platform
	Yoo'19 [94]	W5	GI1	both	Others	ES, PLC	Schneider/AB	DPI and detection tools
	LLB [43]	W4,6	GI1,GI2	both	Programmer	ES, PLC	LD, AB	Studio 5000, RSLinx, LLB
	CaFDI [69]	W4	GI4	both	State verif.	N/A	generic	CaFDI
	HARVEY [37]	W4,5	GI4,GC	both	ES	ES, PLC	AB	Hex, dis-assembler, EMS

*Engineering Station (**ES**), Allen-Bradley (**AB**). Tools: vulnerability (**vul.**). Detection to evade: verification (**verif.**).*

trolled process, so that modified binaries could deploy meaningful and impactful attacks.

4.1.3. T3: program runtime. At this level, existing studies investigated two types of attacks: the program modification attack, and the program input manipulation attack. The input of the program could either come from the communication between the PLC and the engineering station, or the sensor readings.

• ***Program modification attack.*** this requires reverse engineering and payload injection, similar to studies in *T2*. The difference is that, given the PLC memory layout available, and the features supported by the PLC, the design of payload becomes more targeted. Through injecting malicious payload to the code, PLCInject [58] and PLC-Blaster [85] presented the widespread impact of the malicious payload. PLCInject crafted a payload with a scanner and proxy. Due to the predefined memory layout of Siemens Simatic PLCs, PLCInject injected this payload at the first organization block (OB) to change the initialization of the system. This attack turned the PLC into a gateway of the network of PLCs. Using PLCInject, Spenneberg [85] implemented a worm, PLC-Blaster, that can spread among the PLCs. PLC-Blaster spread by replicating itself and modifying the target PLCs to execute it along with the already installed programs. PLC-Blaster adopted several anti-detection mechanisms, such as avoiding the anti-replay byte, storing at a less used block, and meeting the scan cycle limit. PLCInject and PLC-Blaster achieved *G3* and demonstrated the widespread impact of program injection attacks.

In addition to that, Senthivel *et.al* [83] introduced several malicious payloads that could deceive the engineering station. Since the engineering station periodically checks the running program from the PLC, the attackers could deceive it by providing an uninfected program while keep executing the infected program in the PLC. Senthivel achieved this through a self-developed decompiler (laddis) for LD programs. Senthivel also introduced three strategies to achieve this denial of engineering operation attacks.

In a similar setting, Kalle *et.al* [54] presented CLIK. After payload injection, CLIK implemented a virtual PLC, which simulated the network traffic of the uninfected PLC, and fed this traffic to the engineering station to deceive the operators. These two works employed a full chain of vulnerabilities at the network level, without accessing the engineering station nor the PLCs.

• ***Input manipulation through the network.*** several studies [11], [64] hijacked certain network packets between the engineering station and a PLC. Beresford *et.al* [11] exploited a packet (e.g. ISO-TSAP) between the PLC and the engineering station. These packets provided program information, such as variable names, data block names, and also the PLC vendor and model. Attackers could modify these variables to cause undesired behavior. With memory probing techniques, attackers could get a mapping between these names to the variables in the PLC. This would allow them to modify the program based on needs. This attack could cause damages to the physical processes. However, the chance for successful mapping of the variables through memory probing is small. In a nuclear power plant setting, Lim *et.al* [64] intercepted and modified the command-37 packets sent between the engineering station and the PLC. This packet provided input to an industrial-grade PLC consisted of redundant modules for recovery. The attack caused common-mode failures for all the modules.

These attacks made the entry point through the network traffic. However, they ignored the fact that security solutions could have enabled deep packet inspection (DPI) between the PLC and the engineering station. Modified packets with malicious code or input data could have been detected before reaching the PLC. To evade DPI, Yoo *et.al* [94], [95] presented stealthy malware transmission, by splitting the malware into small fragments and transmitting one byte per packet with a large size of noises. This is because DPI merges packets for detection and thus was not able to detect small size payload. On the PLC side, Yoo leveraged a vulnerability to control the received malicious payload, discard the padded noises, and configure the start address for execution. Although dependent on multiple vulnerabilities, this study provided insight for stealthy program modification and input manipulation at the network level.

• ***Input manipulation through sensor.*** existing studies [3], [43], [69], [92] explored different approaches to evade various behavior detection, and to achieve *G1*.

Govil *et.al* [43] presented Ladder logic bombs (LLB), which was a combination of program injection and input manipulation attacks. The malicious payload was injected into the existing LD program, as a subroutine, and could be triggered by a certain condition. Once triggered, this malware could replace legitimate sensor readings with manipulated values. LLB was designed to evade manual inspection, by giving instructions names similar to commonly used ones.

LLB did not consider behavior detection, such as state verification, or state estimation. To counter Büchi automaton based state estimation, CaFDI [69] introduced controller-aware false data injection attacks. CaFDI required high-level information of the physical processes, and monitored I/O traces of the program. It first constructed a Büchi automaton model of the program based on its I/O traces, and then searched for a set of inputs that may cause the model to produce the desired malicious behavior. CaFDI calculated the Cartesian product of the safe model and the unsafe model, and recursively searched for a path that could satisfy the unsafe model in the formalization. The resulting path of input would be used as the sensor readings for the program. To stay stealthy, CaFDI avoided noticeable inputs, such as an LED indicator. Xiao [92] fine tuned the undesired model to evade existing sequence-based fault detection [57]. An attacker could first construct a discrete event model from the collected fault-free I/O traces using non-deterministic autonomous automation with output (NDAAO), and then build a word set of NDAAO sequences, and finally, search for the undetectable false sequences from the word set to inject into the compromised sensors. Similarly, by combining the control flow of the program, Abbasi *et.al* [3] presented configuration manipulation attacks by exploiting certain pin control operations, leveraging the absence of hardware interrupts associated to the pins.

To evade the general engineering operations, Garcia [37] developed HARVEY, a PLC rootkit at the firmware level that can evade operators viewing the HMI. HARVEY faked sensor input to the control logic to generate adversarial commands, while simulated the legitimate control commands that an operator would expect to see. In this way, HARVEY could maximize the damage to the physical power equipment and cause large-scale failures, without operators noticing the attack. HARVEY assumed access to the PLC firmware, which was less monitored than the control logic program.

These studies make it practical to inject malicious payloads either through a compromised network or insecure sensor configurations. Because of the stealthiness, it remains challenging to design security solutions to counter. The following details the challenges.

4.2. Challenges

• ***Expanded attack input surfaces.*** The attack input surfaces for PLC programs are expanding. The aforementioned studies have shown input sources including (1) the communication from the engineering station, with certain packets intercepted and hijacked, (2) internet faced PLCs in the same subnet, and (3) compromised sensors and firmware. It becomes challenging for defense solutions to scope an appropriate threat model, since any component along the chain of control could be compromised.

• ***Predefined hierarchical memory layout.*** Multiple studies leveraged this weakness to perform the attacks. However, traditional defense solutions [22] have seen many challenges: (1) the address space layout randomization (ASLR) would be too heavy to meet the scan cycle requirements for the PLCs, and would still suffer from code-reuse attacks, (2) control flow integrity based solutions require a substantial amount of memory, and would be hard to detect in real-time, or to mitigate the attacks, and (3) the hierarchical memory layout is vendor-specific, and the attacks targeting them are product-driven, for example, Siemens Simatic S7 [11], [85]. It is challenging to design a lightweight and generalized defense solution.

• ***Confidentiality and integrity of the program I/O.*** The majority of the studies depended on the program I/O to perform attacks, either to extract information of the physical processes, and possible detection methods, or to manipulate input to produce unsafe behaviors. Protecting I/O is challenging in that (1) the input surfaces of the programs are expanding, (2) sensors and physical processes could be public infrastructure, and (3) the I/O has to be updated frequently to meet the scan cycle requirement.

• ***Stealthy attack detection.*** We have mentioned many stealthiness strategies based on different threat models, including (1) disguising malicious code as human errors, (2) code obfuscation with fragmentation and noise padding to evade DPI, (3) crafting input to evade state estimation and verification algorithms, (4) using specific memory block or configuration of the PLC, and (5) deceiving the engineering station with faked legit behaviors. It is challenging for a defense solution to capture these stealthy attacks.

• ***Implicit or incomplete specifications.*** Multiple studies have shown crafted attacks using the implicit properties [68]–[70]. The difficulties of defining precise and complete specifications lie in that (1) product requirements may change over time thus requiring update of semantics on inputs and outputs, (2) limited expressiveness can lead to incompleteness, while over expressiveness may lead to implicitness, and (3) domain-specific knowledge is usually needed. It is challenging to design specifications that overcome these difficulties.

5. Formal Verification based Defenses

A large body of research uses formal verification for PLC safety and security, as Table 3 shows. This study mainly focused on the following aspects:

- **Behavior Modeling**: Modeling the behavior of the program as a state-based, flow-oriented, or time-dependent representation.
- **State Reduction**: Reducing the state space to improve search efficiency.
- **Specification Generation**: Generating the specification with desired properties as a temporal logic formula.
- **Verification**: Applying model checking or theorem proving algorithms to verify the security or safety of the PLC program.

Based on these aspects, the following discusses defense methodologies. We use the same threat models, security goals, and weaknesses as mentioned in Section 3.2.

5.1. Behavior Modeling

The goal of behavior modeling is to obtain a formal representation of the PLC program behavior, so that given a specification, a formal verification framework can understand and verify it. The following discusses behavior modeling, based on different threat models.

5.1.1. T1: program source code. At the source code level, a line of studies [4], [30], [41], [76] have investigated the formal modeling of generic program behaviors. The majority of them translated programs to *automata* and *Petri nets*, since they were well supported by most formal verification frameworks [4]. These translations usually consider each program unit as an automaton, including the main program, functions, and function block instances. Each variable defined in the program unit was translated as a corresponding variable in the automaton. Input variables are assigned non-deterministically at the beginning of each PLC cycle. The whole program was modeled as a *network of automata*, where a *transition* represents the changes of variable values in different execution cycles, and a *synchronization* pair represents synchronized transitions of function calls. In a similar modeling method, Newell *et.al* [76] translated FBD programs to Prototype Verification System (PVS) models, since certain nuclear power generating station can only support such representation.

These studies could formally model most PLC behaviors, especially the internal logic within the PLC code. However, with only source code available, behavior modeling lacks the interaction with the PLC hardware, and the physical processes, which might cause unsafe or malicious behaviors to bypass later formal verification. The following discusses behavior modeling with more information available.

5.1.2. T2: program bytecode/binary. Fewer studies have investigated behavior modeling at the program binary level. The challenges lie in reverse engineering. As mentioned in existing works [71], [100], several PLC features are not supported in the normal instruction sets. PLCs are designed with hierarchical addressing using a dedicated memory area for the input and output buffers. The function blocks use a parameter list with fixed entry and exit points. PLCs also support timers that act differently between bit-logic instructions and arithmetic instructions.

Thanks to an open-source library [60], which can disassemble Siemens PLC binary programs into STL (IL equivalent for Siemense) programs, several works [21], [71], [93], [100] studied modeling Siemens binary programs. Based on the STL program, TSV [71] leveraged an intermediate language, ILIL, to allow more complete instruction modeling. With concolic execution, TSV obtained the information flow from the system registers and the memory. After executing multiple scan cycles, a temporal execution graph was constructed to represent the states of the controller code. After TSV, Zonouz *et.al* [100] adopted the same modeling. Chang *et.al* [21] and Xie *et.al* [93] constructed control flow graphs with similar executable paths. Chang deduced the output states of the timer based on the existing output state transition relationships, while Xie used constraints to model the program.

Combined with studies at *T1*, these studies could handle more temporal features, such as varied scan cycle lengths, and enabled input dependent behavior modeling. With control flow based representation, nested logic and pointers could also be supported. However, without concrete execution of the programs, the drawbacks are obvious: (1) the input vectors were either random or have to be manually chosen, (2) the number of symbolic states limited the program sizes, (3) the temporal information further increased resource consumption. Next, we discuss behavior modeling with runtime information.

5.1.3. T3: program runtime. With runtime information, existing research [19], [53], [65], [98], [99] modeled programs considering its interactions with the physical processes, the supervisor, and the operator tasks. This allowed more realistic modeling for timing sensitive instructions, and domain-specific behavior modeling.

Automated frameworks [91], [99] were presented to model PLC behaviors with interrupt scheduling, function calls, and IO traces. Zhou *et.al* [99] adopted an environment module for the inputs and outputs, an interruption module for time-based instructions, and a coordinator module to schedule these two modules with the main program. Wang *et.al* [91] automated a BIP (Behavior, Interaction, Priority) framework to formalize the scanning mode, the interrupt scheduler, and the function calls. Mesli *et.al* [72] presented a component-based modeling for the whole control-command chain, with each component described as timed automata.

To automate modeling of domain-specific event behavior, VetPLC [98] generated timed event causality graphs (TECG) from the program, and the runtime data traces. The TECG maintained temporal dependencies constrained by machine operations.

These studies removed the barrier from modeling event-driven and domain-specific behaviors. They could mitigate attacks violating security and safety requirements via special sequences of valid logic.

5.1.4. Challenges.

• ***Lack of plant modeling.*** Galvao *et.al* [36] have pointed out the importance of plant models in formal verification. However, existing studies focused on the formalization of PLC programs, rather than the I/O of the programs that directly reflect the behavior of the physical processes (e.g. plant). Under *T3*, a few studies considered program I/O during behavior modeling. However, they either consider I/O as a generic module working together with the other modules [91], [99], or informally use data mining on program I/O to extract program event sequences [98]. It remains challenging to formalize plant models in improving PLC program security.

• ***Lack of modeling evaluation.*** The majority of the studies only adopted one modeling method to obtain a program representation. We understood the representation is compatible with the formal verification framework.

However, there were no scientific comparisons between models from different studies, except some high-level descriptions. Within one model, only a few studies [20], [69], [98] evaluated the number of states in their representations. It is even more difficult to understand the performance of the model from the security perspective.

• ***State explosion.*** The aforementioned studies have already adopted an efficient representation that transforms a program unit as a state automaton, and formalizes the state transition between the current cycle and the next cycle. A less efficient model representation transforms each variable of a program as a state, and formalize the transition between the states. Even though such representation can benefit PLC programs in any language, it produces large size models containing too many states to be verified, even for small and medium-sized programs. Therefore, in practice, most of the programs are modeled in the former efficient representation. For large size programs, however, both representations will produce large amounts of state combinations, causing the state explosion problem. The following describes research works in state reduction.

5.2. State Reduction

The goal of state reduction is to improve the scalability and complexity of PLC program formalization. There are two common steps involved. First, we have to determine the meaningful states related to safety and security properties. Then, we trim the less meaningful states.

5.2.1. T1: program source code. At the source code level, a line of studies [25], [29], [42], [79] performed state reduction. Gourcuff *et.al* [42] considered the meaningful states as those related to the input and output variables, since they directly control the behavior of the physical processes. To obtain the dependency relations of the input and output variables, Gourcuff conducted static code analysis to get variable dependencies in a ST program, and found a large portion of the unrelated states. Even though this method significantly reduced the state search space, it also skipped much of the original code for the following verification.

To improve the code coverage of the formalization, Pavlovic *et.al* [79] presented a general solution for FBD programs. They first transformed the graphical program to textual statements in *textFBD*, and further substituted the circuit variables to *tFBD*. This approach removed the unnecessary assignments connecting continuous statements and merged them into one. On top of this approach, Darvas et.al fine tuned the reduction heuristics with a more complete representation [25], [29]. Besides unnecessary variable or logic elimination, these heuristics adopted the Cone of influence (COI)-based reduction, and the rule-based reduction. The COI-based reduction first removed unconditional states that all possible executions go through. It then removed variables that do not influence the evaluation of the specification. The rule-based reduction could be specified based on the safety requirements of the application domain. Additionally, math models were also used to abstract different components. Newell *et.al* [76] defined additional structure, attribute maps, graphs, and block groups to reduce the state space of their PVS code.

These studies successfully reduced the size of program states. They were limited, however, to basic Boolean representation reduction. For programs with complex time-related variables, function blocks, or multitasks, these studies were insufficient. It was also unclear whether the reduction could undermine program security. The following discusses other reduction techniques when such information is present.

5.2.2. T2: program bytecode/binary. Studies at the binary level mostly adopted symbolic execution combined with flow-based representation. This demonstrated that meaningful states lead to different symbolic output vectors. TSV [71] merged the input states that could all lead to the same output values. It also abstracted the temporal execution graphs, by removing the symbolic variables based on their product with the valuations of the LTL properties.

To further reduce the unrelated states, Chang *et.al* [21] reduced the overlapping output states of the same scan cycle, and removed the output states that had been analyzed in previous cycles. To reduce the overhead of timer modeling, they employed a deduction method for the output states of timers, through the analysis of the existing output state transition relationships These reductions did not undermine the goal of detecting malicious behaviors spanning multiple cycles.

Compared with *T1*, these studies were more interested in preserving temporal features, and targeted the reduction from random inputs in symbolic execution. However, without undermining the temporal feature modeling, the reduction of input states was inefficient given the lack of real inputs. The following discusses the reduction techniques when runtime inputs are available.

5.2.3. T3: program runtime. With runtime information, we could gain a better understanding of the real meaningful states. These include the knowledge from event scheduling for subroutine and interrupts, and the real inputs and outputs from the domain-specific processes.

Existing studies [53], [65], [98], [99] presented state reduction in different approaches. To reduce the scale of the model, Zhou *et.al* [99] modeled timers inline with the main program instead of a separate automata, since their model had considered the real environment traces, the interruptions, and the scheduling between them. Similarly, Wang *et.al* [91] compressed segments without jump and call instructions into one transition.

Besides merging unnecessary states, the real inputs and domain-specific knowledge could narrow down the range for modeling numerical and float variables. In Zhang's study [98], continuous timing behavior was discretized to multiple time slices with a constant interval. Since the application-specific IO traces are available, the time interval was narrowed to a range balancing between efficiency and precision.

Compared with studies at *T1* and *T2*, state reduction at *T3* was more powerful, not only with more realistic temporal and event-driven features supported, but also helped to extract more meaningful states with domain-specific runtime information.

5.2.4. Challenges.

• ***Lack of "ground truth" for continuous behavior.*** We discussed that runtime traces helped to determine a "realistic" granularity for continuous behaviors. However, choosing the granularity was still experience-based and ad-hoc. In fact, a too coarse granularity would fail to detect actual attacks, while a too fine granularity expected infeasible attack scenarios [36]. Abstracting a "ground truth" model for continuous behavior remains challenging.

• ***Implicitness and stealthy attacks from reduction.*** Although existing studies have considered property preservation, the reduced "unrelated" states may undermine PLC security. We mentioned in Section 4 that implicit specification had led to attacks. The reduced states may cause the implicit mapping between the variables in the program and its specification, or they may contain stealthy behaviors that were simply omitted by the specification. The following discusses research on specification generation.

5.3. Specification Generation

The goal of these studies is to generate safety and security specifications with formal semantics. Specifying precise and complete desired properties is difficult. Existing studies focused on two aspects: (1) process-independent properties that describe the overall requirements for a control system, and (2) domain-specific properties that require domain expertise.

5.3.1. T1: program source code. At the source code level, a line of studies [13], [27], [28], [41], [47] investigated specification generation with process-independent properties. These properties include avoiding variable locks, avoiding unreachable operating modes, operating modes that are mutually exclusive, and avoiding irrelevant logic [81].

Existing studies [4], [10], [16], [74], [81] usually adopted CTL or LTL-based formulas to express these properties. LTL describes the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc. CTL describes *invariance* and *reachability*, e.g., the system never leaves the set of states, the system can reach the set of states, respectively. Other variants included ACTL, adopted by Rawlings [81], and ptLTL, adopted by Biallas [12].

Besides CTL and LTL-based formulas, a *proof assistant* was also investigated to assist the development of formal proofs. To formally define the syntax and semantics, Biha *et.al* [13] used a type theory based proof assistant, Coq, to define the safety properties for IL programs. The semantics concentrated on the formalization of on-delay timers, using discrete time with a fixed interval. Besides Coq, K framework [47] was also adopted to provide a formal operational semantics for ST programs. K is a rewriting-based semantic framework that has been applied in defining the semantics for C and Java. Compared with Coq, K is less formal but lighter and easier to read and understand. The trade-off is that manual effort is required to ensure the formality of the definition.

These studies limited specification generation to certain program models. To enable formal semantics for state-based, data-flow-oriented, and time-dependent program models, Darvas *et.al* [27] presented PLCspecif to support various models.

These studies provided opportunities for engineers lacking formalism expertise to generate formal and precise requirements. The proof assistant frameworks even allowed generating directly executable programs, e.g. C program. Nevertheless, only process-independent properties could be automated, the following discusses specification generation with more information available.

5.3.2. T2: program bytecode/binary. As mentioned earlier, symbolic execution allowed these studies to support program modeling with numeric and float variables. These variables provided more room for property definitions in the specification. TSV [71] defined properties bounding the numerical device parameters, such as the maximum drive velocity and acceleration. Others *et.al* [21], [93], [100] defined properties to detect malicious code injection, parameter tampering attacks. Xie *et.al* [93] expanded the properties to detect stealthy attacks, and denial of service attacks.

Similar to studies at *T1*, these studies all adopted LTL-based formalism, and could automate process-independent property generation. To accommodate certain attack strategies, the specification generation was manually defined.

5.3.3. T3: program runtime. With runtime information available, specification generation concentrated more on domain-specific properties. In a waste water treatment plant setting, Luccarini *et.al* [65] applied artificial neural networks to extract qualitative patterns from the continuous signals of the water, such as the pH and the dissolved oxygen. These qualitative patterns were then mapped to the control events in the physical processes. The mapping was logged using XML and translated into formal rules for the specification. This approach considered the collected input and output traces as ground truth for security and safety properties, and removed the dependencies on domain expertise.

In reality, the runtime traces might be polluted, or contain incomplete properties for verification. To ensure the correctness and completeness of domain-specific rules, existing studies [36], [98] also considered semi-automated approaches, which combined automated data mining and manual domain expertise. VetPLC [98] formally defined the safety properties, through automatic data mining and event extraction, aided with domain expertise in crafting safety specifications. VetPLC adopted timed propositional temporal logic (TPTL), which was more suitable to quantitatively express safety specifications.

Besides (semi)-automated specification generation, Mesli *et.al* [72] manually defined a set of rules for the interaction between each component along the chain of control. The requirements are also written in CTL temporal logic. To assist domain experts in developing formal rules, Wang *et.al* [91] formalized the semantics for a BIP model for all types of PLC programs. It automated process-independent rules for interrupts, such as, following the first come first serve principle.

These studies enabled specification generation with domain-specific knowledge. They thus expanded security research with more concentration on safety requirements.

5.3.4. Challenges.

• ***Lack of specification-refined programming.*** Since these studies already assumed the existence of the PLC programs (source code or binary), the generated specification could help refine the programming and program modeling. We have mentioned earlier that state reduction considered property preserving, and removed "irrelevant logic" from program modeling. However, generated properties did not provide direct feedback to the program source code. In fact, program refinement in a similar approach of state reduction is promising in eliminating irrelevant stealthy code from the source.

• ***Ad-hoc and unverified specification translations.*** Despite the availability of formal semantics and proof assistants, such as Coq, PVS, HOL, existing requirements are informally defined in high-level languages, and vary across industrial domains. Existing studies translating these requirements encountered many challenges: (1) tradeoff between an automated but unverified approach, or a formal but manual rewriting, (2) the dependencies on program language (many studies were based on IL [47], deprecated in IEC 61131-3 since 2013), (3) the rules were based on sample programs without the complexity of the real industry.

• ***Barrier for automated domain-specific property generation.*** Although Luccarini [65] presented a promising approach, it was based on two unrealistic assumptions: (1) the trace collected from the physical processes was complete and could be trusted, and (2) the learning algorithm extracted the rules completely and accurately. Without further proofs (manual domain expertise) to lift these two assumptions, the extracted properties would be an incomplete "white list" which may also contain implicitness, leading to false positives or true negatives in the verification or detection.

• ***Specification generation with evolved system design.*** Increasing requirements were laid on PLC programs, considering the interactions from new components. In the behavior modeling, we have observed studies formalizing the behaviors of new interactions, on top of existed models, for example, adding a scheduler module combing an existed program with a new component. Compared with that, we saw fewer studies investigating incremental specification generation, based on existing properties. It was still challenging to define the properties to synchronize PLC programs with various components, especially in a timing-sensitive fashion.

5.4. Verification

We already discussed the modeling of program behavior, and specification generation. With these, a line of studies [9], [10], [16], [17], [20], [74], [75], [77], [80], [81], [96] applied model checking and theorem proving to verify the safety and security of the programs.

These studies applied several formal verification frameworks, summarized in Table 2. The majority of them used Uppaal and Cadence SMV. Uppaal was used for real-time verification representing a network of timed automata extended with integer variables, structured data types, and channel synchronization. Cadence SMV was used for untimed verification.

5.4.1. T1: program source code. At the source code level, formal verification studies aimed at verifying weaknesses *W1-W4*, to defend against general safety problems. They had been applied by programs from different industries.

To defend *G1*, Bender *et.al* [10] adopted model checking for LD programs modeled as timed Petri nets. They applied model checkers in the Tina toolkit to verify LTL properties. Bauer *et.al* [9] adopted Cadence SMV and Uppaal, to verify untimed modeling and timed modeling of the SFC programs, respectively. They identified errors from three reactors. Similarly, Niang *et.al* [77] verified a circuit breaker program in SFC using Uppaal, based on a recipe book specification. To defend *G2*, Hailesellasie *et.al* [44] applied Uppaal and compared two formally generated attributed graphs, the *Golden Model* with the properties, and a random model formalized from a PLC program. The verification is based on the comparison of nodes and edges of the graphs. They detected stealthy code injections.

Instead of adopting existing tools, several studies developed their own frameworks for verification. Arcade.PLC [12] supported model checking with CTL and LTL-based properties for all types of PLC programs. PLCverif [28] supported programs from all five Siemens PLC languages. NuDE 2.0 [56] provided formal-method-based software development, verification and safety analysis for nuclear industries. Rawlings *et.al* [81] applied symbolic model checking tools st2smv and SynthSMV to verify and falsify a ST program controlling batch reactor systems. They automatically verified process-independent properties, rooted in *W1-W4*.

Besides model checking, existing studies [76] also adopted PVS theorem proving to verify the safety properties described in tabular expressions in a railway interlocking system.

These studies are limited to general safety requirements verification. To defend *G2* and *G3*, more information will be needed, as discussed in the following.

5.4.2. T2: program bytecode/binary. This line of studies [21], [71], [91], [93], [100] allowed us to detect binary tampering attacks.

TSV [71] combined symbolic execution and model checking. It fed the model checker with an abstracted temporal execution graph, with its manually crafted LTL-based safety property. Due to its support for random timer values within one cycle, TSV was limited by checking the code with timer operations, and still suffered from state explosion problems. Xie *et.al* [93] mitigated this problem with the use of constraints in verifying random input signals. Xie used nuXmv model checker. Chang *et.al* [21] applied a less formal verification based on the number of states.

These studies successfully detected malicious parameter tempering attacks, based on sample programs controlling traffic lights, elevator, water tank, stirrer, and sewage injector.

5.4.3. T3: program runtime. With runtime information, existing studies could verify domain-specific safety and

security issues, namely all the weaknesses and security goals discussed in Section 4.

To defend *G1* by considering the interactions to the program, Carlsson *et.al* [18] applied NuSMV to verify the interaction between the Open Platform Communications (OPC) interface and the program, using properties defined as server/client states. They detected synchronization problems, such as jitter, delay, race condition, and slow sampling caused by the OPC interface. Mesli [72] applied Uppaal to multi-layer timed automata, based on a set of safety and usability properties written in CTL. They detected synchronization errors between the control programs and the supervision interfaces.

To fully leverage the knowledge from the physical processes, VetPLC [98] combined runtime traces, and applied BUILDTSEQS to verify security properties defined in timed propositional temporal logic. HyPLC [38] applied theorem prover KeYmaera X to verify the properties defined in differential dynamic logic. Different from VetPLC, HyPLC aimed at a bi-directional verification between the physical processes, and the PLC program, to detect safety violations.

These studies either assumed an offline verification, or vaguely mentioned using a supervisory component for online verification. To provide an online verification framework, Garcia *et.al* [40] presented an on-device runtime solution to detect control logic corruption. They leveraged an embedded hypervisor within the PLC, with more computational power and integration of direct library function calls. The hypervisor overcame the difficulties of strict timing requirements and limited resources, and allowed verification to be enforced within each scan cycle.

5.4.4. Challenges.

• ***Lack of benchmarks for formal verification.*** Similar to the challenges in behavior modeling, an ideal evaluation should be multi-dimensional: across modeling methods, across verification methods, and based on a set of benchmark programs. Existing evaluations, if performed, were limited to one dimension and based on at most a few sample programs. These programs were often vendor-specific, test-case driven, and failed to reflect the real industry complexity. Without a representative benchmark and concrete evaluation, the security solution design would still be ad-hoc.

• ***Open-source automated verification frameworks.*** Existing studies have presented several open-source frameworks taking a PLC program as input, and automatically generating the formal verification result over generic properties. These frameworks (e.g. Arcade.PLC, st2smv and the SynSMV) lowered the bar for security analysis using formal verification. However, over the years, such frameworks were no longer supported. No comparable replacement emerged, except PLCverif [26] targeting Siemens programs.

• ***High demand for runtime verification.*** The challenges include (1) expanded attack landscapes due to increasingly complex networking, (2) tradeoff between limited available resources on the PLC and real-time constraints, (3) runtime injected stealthy attacks due to insecure communication, and (4) runtime denial of service attacks omitted by existing studies.

6. Recommendations

We have described and discussed the security challenges in defending against PLC program attacks using formal verification and analysis. Next, we offer recommendations to overcome these challenges. Our recommendations highlight promising research paths based on a thorough analysis of the state-of-the-art and the current challenges. We consider these recommendations equally relevant regardless of any particular factor—neither mentioned nor considered in this section—that may change this perception.

6.1. Program Modeling

6.1.1. Plant Modeling. We discussed the lack of formalized plant modeling in Section 5.1.4. We recommend more research in plant modeling to formalize more accurate and complete program behaviors. Future research should consider refinement techniques to define the granularity and level of abstraction for the plant models and the properties to verify. The refinement techniques should consider the avoidance of state explosion, by extracting feasible conditions of the plant that can trigger property violations in the program.

6.1.2. Input manipulation verification. Plant modeling is also promising in mitigating program input manipulation attacks. As mentioned in Section 4, input manipulation is widely adopted by the attackers. Future research should consider the Orpheus [23] prototype in a PLC setting. Orpheus performs event consistency checking between the program model and the plant model to detect input manipulation attacks. To perform event consistency checking in a PLC, future research may consider instrumentation on the input and output variables of the programs, and compare the values with these from the plant models.

6.2. State Reduction

In Section 4.1.1, we discussed code level attacks that could disguise themselves as bad coding practice, and are hard to be noticed. During the state reduction, based on an existed specification, "unrelated" states are trimmed to avoid state explosion problems. However, as mentioned in Section 4.2, existing studies failed to investigate the relationship between the "unrelated" states and the original program. It could be hidden jumps with a stealthy logger to leak program critical information. The specification might only consider the noticeable unsafe behaviors, which can disturb the physical processes, while let the states from the stealthy code be recognized as "unrelated". We, therefore, recommend future research to investigate the security validation of unrelated code, and consider automatic program cleaning for the stealthy code.

6.3. Specification Generation

6.3.1. Domain-specific property definition. As mentioned in Section 5.3.4, there are barriers in automatic generation of domain-specific properties, and manually

TABLE 2: Common frameworks for formal verification

Frameworks	Modeling Languages	Property Languages/Prover	Supported Verification Techniques
NuSMV/nuXMV	SMV input language (BDD)	CTL, LTL	SMT, Model checking, fairness requirements
Uppaal	Timed automata with clock and data variables	TCTL subset	Time-related, and probability related properties
Cadence SMV	SMV input language (BDD)	CTL, LTL	Temporal logic properties of finite state systems
SPIN	Promela	LTL	Model checking
UMC	UML	UCTL	Functional properties of service-oriented systems
Coq	Gallina (Calculus of Inductive Constructions)	Vernacular	Theorem proof assistant
PVS	PVS language (typed higher-order logic)	Primitive inference	Formal specification, verification and theorem prover
Z3	SMT-LIB2	SMT-LIB2 Theories	Theorem prover

TABLE 3: Existing studies using formal verification to detect control logic attacks

Threat Model	Paper	Security Goal	Defense Focus	Verification Techniques	Property	PLC Language	Tools
T1 source code	Adiego'15 [4]	GI1	BM, SG	MC	CTL, LTL	ST,SFC	nuXmv, PLCVerif, Xtext, UNICOS
	Bauer'04 [9]	GI1,GI3	FV	MC	CTL	SFC	Cadence SMV, Uppaal
	Bender'08 [10]	GI3	SG, FV	MC	seLTL	LD	Tina Toolkit
	Biallas'12 [12]	GI1,GI3	SG, FV	MC	∀CTL, ptLTL	generic	PLCopen, ***Arcade.PLC****, CEGAR
	Biha'11 [13]	GI1	SG	TP	N/A	IL	SSReflect in Coq, CompCert
	Brinksma'00 [16]	GI3	SG	MC	N/A	SFC	SPIN/Promela, Uppaal
	Darvas'14 [25]	GI1	SR	MC	CTL, LTL	ST	**COI reduction**, NuSMV
	Darvas'15 [27]	GI1,GI3	SG	EC	N/A	ST	**PLCspecif**
	Darvas'16-1 [28]	GI1	SG, FV	N/A	temporal logic	ST	***PLCverif***, nuXmv, Uppaal
	Darvas'16-2 [29]	GI1	SR	MC, EC	temporal logic	LD,FBD	***PLCverif***, NuSMV, nuXmv, etc.
	Darvas'17 [30]	GI1	BM	N/A	temporal logic	IL	***PLCverif***, Xtext parser
	Giese'06 [41]	GI1	BM, SG	EC	N/A	ST	GROOVE, ISABELLE, FUJABA
	Gourcuff'06 [42]	GI1,GI3	SR	MC	N/A	ST,LD,IL	NuSMV
	Hailesellasie'18 [44]	GI1,GC	FV	MC	N/A	SFC,ST,IL	BIP, nuXmv, Uppaal,**UBIS model**
	Huang'19 [47]	GI1	SG	N/A	N/A	ST	K framework, ***KST model***
	Kim'17 [56]	GI1,GI3	FV	MC, EC	CTL	FBD,LD	CASE tools (Nude 2.0), NuSCR
	Moon'94 [74]	GI1	SG	MC	CTL	LD	N/A
	Newell'18 [76]	GI1,GI3	BM, SR	TP	N/A	FBD	PVS Theorem prover
	Niang'17 [77]	GI3	FV	MC	N/A	generic	Uppaal, program translators
	Pavlovic'10 [79]	GI1,GI3	SR	MC	CTL	FBD	NuSMV
	Rawlings'18 [81]	GI1	SG, FV	MC	CTL, ACTL	ST	***st2smv, SynthSMV****
	Mader'00 [66]	GI1	BM	N/A	N/A	generic	N/A
	Ovatman'16 [78]	GI1,GI3	BM, FV	MC	N/A	generic	N/A
	Moon'92 [75]	GI1,GI3	ALL	MC	CTL	LD	**a CTL model checker**
	Bohlender'18 [14]	GI1,GI3	SR	MC	N/A	ST	Z3, PLCOpen, Arcade.PLC
	Kuzmin'13 [61]	GI1	BM	N/A	LTL	ST	Cadence SMV
	Bonfe'03 [15]	GI3	BM	N/A	CTL	generic	SMV, CASE tools
	Chadwick'18 [20]	GI3	BM, SG	TP	FOL	LD	Swansea
	Frey'00 [35]	GI1,GI3	BM	N/A	N/A	N/A	N/A
	Yoo'09 [96]	GI3	ALL	MC, EC	CTL	FBD	NuSCR, Cadence SMV, VIS, CASE
	Lamperiere'99 [62]	GI1	BM	N/A	N/A	generic	N/A
	Kottler'17 [59]	GI3	ALL	N/A	CTL	LD	NuSMV
	Younis'03 [97]	GI1,GI3	BM	N/A	N/A	generic	N/A
	Rossi'00 [82]	GI1	BM	MC	CTL, LTL	LD	Cadence SMV
	Vyatkin'99 [89]	GI1	BM	MC	CTL	FBD	SESA model-analyser
	Canet'00 [17]	GI1,GI3	ALL	MC	LTL	IL	Cadence SMV
T2 bytecode /binary	Chang'18 [21]	GI1	ALL	MC	LTL, CTL	IL	DotNetSiemensPLCToolBoxLibrary
	McLaughlin'14 [71]	GI1,GI3	ALL	MC	LTL	IL	**TSV**, Z3, NuSMV
	Xie'20 [93]	GI1,GC,GA	BM, SG, FV	MC	LTL	IL	SMT, NuXMV
	Zonouz'14 [100]	GI1,GI3	BM, SG, FV	MC	LTL	IL	Z3, NuSMV
T3 runtime	Carlsson'12 [18]	GI	FV	MC	CTL, LTL	N/A	NuSMV
	Cengic'06 [19]	GI2	BM	MC	CTL	FBD	Supremica
	Galvao'18 [36]	GI3,GI4	SG	MC	CTL	FBD	ViVe/SESA
	Garcia'16 [40]	GI3	FV	MC	DFA	LD,ST	N/A
	Janicke'15 [53]	GI1,GI2	BM, SR	MC	ITL	LD	Tempura
	Luccarini'10 [65]	GI3,GI4	BM, SR, SG	TP	CLIMB	N/A	SCIFF checker
	Mesli'16 [72]	GI	BM, SG, FV	MC	TCTL	LD,FBD	Uppaal
	Wang'13 [91]	GI1,GI2	BM, SR, SG	MC	LTL, MTL	IL	BIP
	Zhang'19 [98]	GI,GC	ALL	MC	TPTL	ST	**BUILDTSEQS** algorithm
	Zhou'09 [99]	GI	BM, SR	MC	TCTL	IL	Uppaal
	Wan'09 [90]	GI1,GI2	BM, FV	TP	Gallina	LD	Coq, Vernacular
	Garcia'19 [38]	GI	BM	TP	differential dL	ST	KeYmaera X
	Mokadem'10 [73]	GI3	BM	MC	TCTL	LD	Uppaal
	Cheng'17 [23]	GI2,GC	BM	N/A	eFSA	N/A	LLVM DG
	Ait'98 [5]	GI2	SG	TP	FOL	N/A	Atelier B

*Defense Focus: Behavior modeling (**BM**), State Reduction (**SR**), Specification Generation (**SG**), and Formal Verification (**FV**). Verification tehcniques: model checking (**MC**), equivalence checking (**EC**), and theorem proving (**TP**). In tools: items in bold are **self-developed**, bold italics are **open-source** and * represent tools no longer mantained.*

defined properties can cause implicitness. We recommend future research to consider domain-specific properties as a *hybrid program* consisted of continuous plant models as well as discrete control algorithms. These properties can be formalized using differential dynamic logic and verified with a sound proof calculus. Existing research [38] has

formalized the dynamic logic model of a water treatment testbed controlled by a ST program. The formalization aims to understand safety implications, and can only support one task with boolean operations. Future research should explore the formalization of dynamic logic with the goal of security verification, and support arithmetic operations, multitask programs, and applications in other domains.

6.3.2. Incremental specification generation. We discussed attacks using expanded input surfaces or a full chain of vulnerabilities in Section 4.2. We also discussed the challenges given the fast-evolving system design in Section 5.3.4. This leads us to think about incremental specification generation, with a full chain of behaviors, and update in a dynamic spectrum. Incremental specification generation [5] has been designed for interactive systems. In the PLC chain of control, interactions should consider both the physical process changes, and the inclusion of the engineering station. Modeled behaviors from these new interactions should be compatible with existing properties. To update in a dynamic spectrum, the behavior changes from each interactive component should support automatic generation and comparison. This requires automatic translations between the behavior models of each component. The closest study is HyPLC [38], which supported automatic translation between the PLC program, and the physical plant model. However, incremental specification generation was not considered. We encourage future research to investigate this direction, and seek interactive mutual refinement.

6.4. Verification

6.4.1. Real-time attack detection. As shown in Sections 5.4.3 and 5.4.4, there is a high demand for runtime verification beyond a high-level prototype. To perform runtime verification, existing studies depend on engineering stations. However, Section 4.1.3 has demonstrated runtime attacks aiming at evading or deceiving the engineering station from runtime detection. The engineering stations have been exposed to various vulnerabilities [1], [50], [51], due to the rich features supported outside the scope of security. Therefore, we recommend future research to consider a dedicated security component, such as the bump-in-the-wire solution provided by TSV [71]. This component is promising in eliminating the resource constraints within a PLC, and allows the program to meet the strict cycle time. In addition to the real-time requirement, future research should also learn from existing attack studies [37], [54], and consider exploring the verification between the PLC and the other interacting components, including the engineering station.

6.4.2. Open-source tools and benchmarks. We discussed in Section 5.4.4 that the lack of open-source tools and benchmarks have led to adhoc studies without evaluations on models and verification techniques. It is promising to see PLCverif [26] become open-source and support integration of various model checking tools. We recommend future studies to continue the development of open-source tools, to cover program modeling, state reduction, specification generation, and formal verification. To adapt to broad use cases, we suggest the tools to be IEC-61131 compliant, compatible with existing open-source PLC tools [7], and consider long time maintenance. We also recommend future studies to develop PLC security benchmarks, including a collection of open-source programs that are vendor-independent and can represent industrial complexities, and a set of security metrics that can support concrete evaluations.

6.4.3. Multitasks Verification. In Section 4.1.3, we have discussed attacks that can use PLC multitasks to perform a denial-of-service attacks, and spread stealthy worms. To defend against multitask attacks, existing studies [39], [73] only considered checking the reaction time between tasks to detect failures of meeting the cycle time requirement. We recommend future research to consider more attack scenarios involved in multitask programs, for example, using one task to spy or spread malicious code to the other co-located tasks, as did in PLCInject [58] and PLC-Blaster [85], or manipulating shared resources (e.g. global variables) between tasks to produce non-deterministic output to disturb the physical processes. Future research should explore the verification of these attack scenarios, with the consideration of task intervals and priorities at various granularities.

7. Conclusion

This paper provided a systematization of knowledge based on control logic modification and formal verification-based defense. We categorized existing studies based on threat models, security goals, and underlying weaknesses. We discussed the techniques and approaches applied by these studies. Our systematization showed that control logic modification attacks have been evolved with the system design. Advanced attacks could compromise the whole chain of control, and in the meantime evade various security detection methods. We found that formal verification based defense studies focus more on integrity than confidentiality and availability. We also found that the majority of the research investigate ad-hoc formal verification techniques, and the barriers exist in every aspect of formal verification.

To overcome these barriers, we suggest a full chain of protection and we encourage future research to investigate the following: (1) formalize plant behaviors to defend input manipulation attacks, (2) explore stealthy attack detection with state reduction techniques, (3) automate domain-specific specification generation and incremental specification generation, and (4) explore real-time verification with more support in open-source tools and thorough evaluation.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This project was supported by the National Science Foundation (Grant#: CNS-1748334) and the Army Research Office (Grant#: W911NF-18-1-0093). Any opinions, findings, and conclusions or recommendations expressed in this paper are

those of the authors and do not necessarily reflect the views of the funding agencies or sponsors.

References

[1] Siemens SIMATIC PCS7, WinCC, TIA Portal (Update D). https://www.us-cert.gov/ics/advisories/ICSA-19-134-08.

[2] Simatic S5 PLC. https://en.wikipedia.org/wiki/Simatic_S5_PLC.

[3] Ali Abbasi and Majid Hashemi. Ghost in the PLC designing an undetectable programmable logic controller rootkit via pin control attack. *Black Hat Europe*, 2016:1–35, 2016.

[4] Borja Fernandez Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor Manuel González Suárez. Applying model checking to industrial-sized PLC programs. 11(6):1400–1410, 2015.

[5] Yamine Aït-Ameur, Patrick Girard, and Francis Jambon. Using the b formal approach for incremental specification design of interactive systems. In *IFIP International Conference on Engineering for Human-Computer Interaction*, pages 91–109. Springer, 1998.

[6] Thiago Alves. PLC addressing. https://www.openplcproject.com/reference/plc-addressing/.

[7] Thiago Alves and Thomas Morris. Openplc: An iec 61,131–3 compliant open source industrial controller for cyber security research. *Computers & Security*, 78:364–379, 2018.

[8] Michael J Assante. Confirmation of a coordinated attack on the ukrainian power grid. *SANS Industrial Control Systems Security Blog*, 207, 2016.

[9] Nanette Bauer, Sebastian Engell, Ralf Huuck, Sven Lohmann, Ben Lukoschus, Manuel Remelhe, and Olaf Stursberg. Verification of PLC programs given as sequential function charts. In *Integration of software specification techniques for applications in Engineering*, pages 517–540. Springer, 2004.

[10] Darlam Fabio Bender, Benoît Combemale, Xavier Crégut, Jean Marie Farines, Bernard Berthomieu, and François Vernadat. Ladder metamodeling and PLC program validation through time Petri nets. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 121–136. Springer, 2008.

[11] Dillon Beresford. Exploiting siemens simatic s7 plcs. *Black Hat USA*, 16(2):723–733, 2011.

[12] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Arcade. PLC: A verification platform for programmable logic controllers. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 338–341. IEEE, 2012.

[13] Sidi Ould Biha. A formal semantics of PLC programs in Coq. In *2011 IEEE 35th Annual Computer Software and Applications Conference*, pages 118–127. IEEE, 2011.

[14] Dimitri Bohlender and Stefan Kowalewski. Compositional verification of PLC software using horn clauses and mode abstraction. *IFAC-PapersOnLine*, 51(7):428–433, 2018.

[15] Marcello Bonfe and Cesare Fantuzzi. Design and verification of mechatronic object-oriented models for industrial control systems. In *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 03TH8696)*, volume 2, pages 253–260. IEEE, 2003.

[16] Ed Brinksma and Angelika Mader. Verification and optimization of a PLC control schedule. In *International SPIN Workshop on Model Checking of Software*, pages 73–92. Springer, 2000.

[17] Géraud Canet, Sandrine Couffin, J-J Lesage, Antoine Petit, and Philippe Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. volume 4, pages 2449–2454. IEEE, 2000.

[18] Henrik Carlsson, Bo Svensson, Fredrik Danielsson, and Bengt Lennartson. Methods for reliable simulation-based PLC code verification. *IEEE Transactions on Industrial Informatics*, 8(2):267–278, 2012.

[19] Goran Cengic, Oscar Ljungkrantz, and Knut Akesson. Formal modeling of function block applications running in IEC 61499 execution runtime. In *2006 IEEE Conference on Emerging Technologies and Factory Automation*, pages 1269–1276. IEEE, 2006.

[20] Simon Chadwick, Phillip James, Markus Roggenbach, and Tom Wetner. Formal Methods for Industrial Interlocking Verification. In *2018 International Conference on Intelligent Rail Transportation (ICIRT)*, pages 1–5. IEEE, 2018.

[21] Tianyou Chang, Qiang Wei, Wenwen Liu, and Yangyang Geng. Detecting plc program malicious behaviors based on state verification. volume 11067 of *Lecture Notes in Computer Science*, pages 241–255, Cham, 2018. Springer International Publishing.

[22] Eyasu Getahun Chekole, Sudipta Chattopadhyay, Martín Ochoa, Huaqun Guo, and Unnikrishnan Cheramangalath. Cima: Compiler-enforced resilience against memory safety attacks in cyber-physical systems. *Computers & Security*, page 101832, 2020.

[23] Long Cheng, Ke Tian, and Danfeng Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 315–326, 2017.

[24] Stephen Chong, Joshua Guttman, Anupam Datta, Andrew Myers, Benjamin Pierce, Patrick Schaumont, Tim Sherwood, and Nickolai Zeldovich. Report on the NSF workshop on formal methods for security. *arXiv preprint arXiv:1608.00678*, 2016.

[25] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Vinuela, and Víctor M González Suárez. Formal verification of complex properties on PLC programs. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 284–299. Springer, 2014.

[26] Dániel Darvas, Enrique Blanco, and Switzerland V Molnár. PLCverif Re-engineered: An Open Platform for the Formal Analysis of PLC Programs. ICALEPCS.

[27] Dániel Darvas, Enrique Blanco Vinuela, and István Majzik. A formal specification method for PLC-based applications. 2015.

[28] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Generic representation of PLC programming languages for formal verification. In *Proc. of the 23rd PhD Mini-Symposium*, pages 6–9.

[29] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety PLC based control software. In *International Conference on Integrated Formal Methods*, pages 508–522. Springer, 2016.

[30] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. PLC program translation for verification purposes. *Periodica Polytechnica Electrical Engineering and Computer Science*, 61(2):151–165, 2017.

[31] Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. Triton: The first ics cyber attack on safety instrument systems. In *Proc. Black Hat USA*, pages 1–26, 2018.

[32] Rolf Drechsler et al. *Advanced formal verification*, volume 122. Springer, 2004.

[33] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[34] Alessio Ferrari, Gianluca Magnani, Daniele Grasso, and Alessandro Fantechi. Model checking interlocking control tables. In *FORMS/FORMAT 2010*, pages 107–115. Springer, 2011.

[35] Georg Frey and Lothar Litz. Formal methods in PLC programming. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, volume 4, pages 2431–2436. IEEE, 2000.

[36] Joel Galvão, Cedrico Oliveira, Helena Lopes, and Laura Tiainen. Formal verification: Focused on the verification using a plant model. In *International Conference on Innovation, Engineering and Entrepreneurship*, pages 124–131. Springer, 2018.

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

[38] Luis Garcia, Stefan Mitsch, and André Platzer. HyPLC: Hybrid programmable logic controller program translation for verification. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 47–56, 2019.

[39] Luis Garcia, Stefan Mitsch, and André Platzer. Toward multi-task support and security analyses in plc program translation for verification. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 348–349, 2019.

[40] Luis Garcia, Saman Zonouz, Dong Wei, and Leandro Pfleger De Aguiar. Detecting PLC control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*, pages 67–72. IEEE, 2016.

[41] Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, and Robert Wagner. Towards verified model transformations. In *Proc. of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV 2a), Genova, Italy*, pages 78–93. Citeseer, 2006.

[42] Vincent Gourcuff, Olivier De Smet, and J-M Faure. Efficient representation for formal verification of PLC programs. In *2006 8th International Workshop on Discrete Event Systems*, pages 182–187. IEEE, 2006.

[43] Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. On ladder logic bombs in industrial control systems. In *Computer Security*, pages 110–126. Springer, 2017.

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

[45] Joseph Y Halpern and Moshe Y Vardi. Model checking vs. theorem proving: a manifesto. *Artificial intelligence and mathematical theory of computation*, 212:151–176, 1991.

[46] Daavid Hentunen and Antti Tikkanen. Havex hunts for ics/scada systems. In *F-Secure*. 2014.

[47] Yanhong Huang, Xiangxing Bu, Gang Zhu, Xin Ye, Xiaoran Zhu, and Jianqi Shi. KST: Executable Formal Semantics of IEC 61131-3 Structured Text for Verification. *IEEE Access*, 7:14593–14602, 2019.

[48] ICS-CERT. CVE-2017-12088. https://nvd.nist.gov/vuln/detail/CVE-2017-12088.

[49] ICS-CERT. CVE-2017-12739. https://nvd.nist.gov/vuln/detail/CVE-2017-12739.

[50] ICS-CERT. CVE-2017-13997. https://nvd.nist.gov/vuln/detail/CVE-2017-13997.

[51] ICS-CERT. CVE-2018-10619. https://nvd.nist.gov/vuln/detail/CVE-2018-10619.

[52] ICS-CERT. CVE-2019-10922. https://nvd.nist.gov/vuln/detail/CVE-2019-10922.

[53] Helge Janicke, Andrew Nicholson, Stuart Webber, and Antonio Cau. Runtime-monitoring for industrial control systems. *Electronics*, 4(4):995–1017, 2015.

[54] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. CLIK on PLCs! Attacking control logic with decompilation and virtual PLC. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*, 2019.

[55] Anastasis Keliris and Michail Maniatakos. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society, 2019.

[56] Eui-Sub Kim, Dong-Ah Lee, Sejin Jung, Junbeom Yoo, Jong-Gyun Choi, and Jang-Soo Lee. NuDE 2.0: A Formal Method-based Software Development, Verification and Safety Analysis Environment for Digital I&Cs in NPPs. *Journal of Computing Science and Engineering*, 11(1):9–23, 2017.

[57] Stéphane Klein, Lothar Litz, and Jean-Jacques Lesage. Fault detection of discrete event systems using an identification approach. *IFAC Proceedings Volumes*, 38(1):92–97, 2005.

[58] Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, and Volker Roth. Internet-facing PLCs-a new back orifice. *Blackhat USA*, pages 22–26, 2015.

[59] Sam Kottler, Mehdy Khayamy, Syed Rafay Hasan, and Omar Elkeelany. Formal verification of ladder logic programs using NuSMV. In *SoutheastCon 2017*, pages 1–5. IEEE, 2017.

[60] Jochen Kühner. Dotnetsiemensplctoolboxlibrary. *https://github.com/jogibear9988/DotNetSiemensPLCToolBoxLibrary*.

[61] Egor Vladimirovich Kuzmin, AA Shipov, and Dmitrii Aleksandrovich Ryabukhin. Construction and verification of PLC programs by LTL specification. In *2013 Tools & Methods of Program Analysis*, pages 15–22. IEEE, 2013.

[62] Sandrine Lampérière-Couffin, Olivier Rossi, J-M Roussel, and J-J Lesage. Formal validation of PLC programs: a survey. In *1999 European Control Conference (ECC)*, pages 2170–2175. IEEE, 1999.

[63] Robert M Lee, Michael J Assante, and Tim Conway. German steel mill cyber attack. *Industrial Control Systems*, 30:62, 2014.

[64] Bernard Lim, Daniel Chen, Yongkyu An, Zbigniew Kalbarczyk, and Ravishankar Iyer. Attack induced common-mode failures on plc-based safety system in a nuclear power plant: Practical experience report. In *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 205–210. IEEE, 2017.

[65] Luca Luccarini, Gianni Luigi Bragadin, Gabriele Colombini, Maurizio Mancini, Paola Mello, Marco Montali, and Davide Sottara. Formal verification of wastewater treatment processes using events detected from continuous signals by means of artificial neural networks. Case study: SBR plant. *Environmental Modelling & Software*, 25(5):648–660, 2010.

[66] Angelika Mader. A classification of PLC models and applications. In *Discrete Event Systems*, pages 239–246. Springer, 2000.

[67] PLC Manual. Basic Guide to PLCs: PLC Programming. https://www.plcmanual.com/plc-programming.

[68] Stephen McLaughlin and Patrick McDaniel. SABOT: specification-based payload generation for programmable logic controllers. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 439–449, 2012.

[69] Stephen McLaughlin and Saman Zonouz. Controller-aware false data injection against programmable logic controllers. In *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 848–853. IEEE, 2014.

[70] Stephen E McLaughlin. On Dynamic Malware Payloads Aimed at Programmable Logic Controllers. In *HotSec*, 2011.

[71] Stephen E McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. A Trusted Safety Verifier for Process Controller Code. In *NDSS*, volume 14, 2014.

[72] S Mesli-Kesraoui, A Toguyeni, A Bignon, F Oquendo, D Kesraoui, and P Berruet. Formal and joint verification of control programs and supervision interfaces for socio-technical systems components. *IFAC-PapersOnLine*, 49(19):426–431, 2016.

[73] Houda Bel Mokadem, Béatrice Berard, Vincent Gourcuff, Olivier De Smet, and Jean-Marc Roussel. Verification of a timed multitask system with UPPAAL. *IEEE Transactions on Automation Science and Engineering*, 7(4):921–932, 2010.

[74] Il Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[75] Il Moon, Gary J Powers, Jerry R Burch, and Edmund M Clarke. Automatic verification of sequential control systems using temporal logic. *AIChE Journal*, 38(1):67–75, 1992.

[76] Josh Newell, Linna Pang, David Tremaine, Alan Wassyng, and Mark Lawford. Translation of IEC 61131-3 function block diagrams to PVS for formal verification with real-time nuclear application. *Journal of Automated Reasoning*, 60(1):63–84, 2018.

[77] Mohamed Niang, Alexandre Philippot, François Gellot, Raphaël Coupat, Bernard Riera, and Sébastien Lefebvre. Formal Verification for Validation of PSEEL's PLC Program. In *ICINCO (1)*, pages 567–574, 2017.

[78] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An overview of model checking practices on verification of PLC software. *Software & Systems Modeling*, 15(4):937–960, 2016.

[79] Olivera Pavlovic and Hans-Dieter Ehrich. Model checking PLC software written in function block diagram. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 439–448. IEEE, 2010.

[80] Mathias Rausch and Bruce H Krogh. Formal verification of PLC programs. In *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*, volume 1, pages 234–238. IEEE, 1998.

[81] Blake C Rawlings, John M Wassick, and B Erik Ydstie. Application of formal verification and falsification to large-scale chemical plant automation systems. *Computers & Chemical Engineering*, 114:211–220, 2018.

[82] Olivier Rossi and Philippe Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In *Proceedings of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM 2000)*, pages 177–182. Citeseer, 2000.

[83] Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, and Vassil Roussev. Denial of engineering operations attacks in industrial control systems. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 319–329, 2018.

[84] Abraham Serhane, Mohamad Raad, Raad Raad, and Willy Susilo. PLC code-level vulnerabilities. In *2018 International Conference on Computer and Applications (ICCA)*, pages 348–352. IEEE, 2018.

[85] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. Plc-Blaster: A worm living solely in the plc. *Black Hat Asia, Marina Bay Sands, Singapore*, 2016.

[86] Ruimin Sun. PLC-control-logic-CVE. https://github.com/gracesrm/PLC-control-logic-CVE/blob/master/README.md.

[87] Michael Tiegelkamp and Karl-Heinz John. *IEC 61131-3: Programming industrial automation systems*. Springer, 1995.

[88] Sidney E Valentine Jr. Plc code vulnerabilities through scada systems. 2013.

[89] Valeriy Vyatkin and H-M Hanisch. A modeling approach for verification of IEC1499 function blocks using net condition/event systems. In *1999 7th IEEE International Conference on Emerging Technologies and Factory Automation. Proceedings ETFA'99 (Cat. No. 99TH8467)*, volume 1, pages 261–270. IEEE, 1999.

[90] Hai Wan, Gang Chen, Xiaoyu Song, and Ming Gu. Formalization and verification of PLC timers in Coq. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 315–323. IEEE, 2009.

[91] Rui Wang, Yong Guan, Liming Luo, Xiaoyu Song, and Jie Zhang. Formal modelling of PLC systems by BIP components. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 512–518. IEEE, 2013.

[92] Min Xiao, Jing Wu, Chengnian Long, and Shaoyuan Li. Construction of false sequence attack against PLC based power control system. In *2016 35th Chinese Control Conference (CCC)*, pages 10090–10095. IEEE, 2016.

[93] Yaobin Xie, Rui Chang, and Liehui Jiang. A malware detection method using satisfiability modulo theory model checking for the programmable logic controller system. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e5724.

[94] Hyunguk Yoo and Irfan Ahmed. Control logic injection attacks on industrial control systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 33–48. Springer, 2019.

[95] Hyunguk Yoo, Sushma Kalle, Jared Smith, and Irfan Ahmed. Overshadow plc to detect remote control-logic injection attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 109–132. Springer, 2019.

[96] Junbeom Yoo, Eunkyoung Jee, and Sungdeok Cha. Formal modeling and verification of safety-critical software. *IEEE software*, 26(3):42–49, 2009.

[97] M Bani Younis, Georg Frey, et al. Formalization of existing PLC programs: A survey. In *Proceedings of CESA*, pages 0234–0239, 2003.

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

[99] Min Zhou, Fei He, Ming Gu, and Xiaoyu Song. Translation-based model checking for PLC programs. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 553–562. IEEE, 2009.

[100] Saman Zonouz, Julian Rrushi, and Stephen McLaughlin. Detecting industrial control malware using automated PLC code analytics. *IEEE Security & Privacy*, 12(6):40–47, 2014.

Appendix

1. Extended Background

This section offers an example of an ST program controlling the traffic lights in a road intersection. We demonstrate an input manipulation attack and the process of using formal verification to detect and prevent it.

1.1. An ST code Example. Code 1 shows a simplified traffic light program written in ST. The program controls the light status (e.g. green, yellow, red) at an intersection between two roads in the north-south (NS) direction and the east-west (EW) direction. The program takes input from sensors telling if emergency vehicles are approaching (line 4), and whether pedestrians press the button to request crossing the intersection (line 5). In Figure A.1, lines 8 to 11 define the output variables representing the status of lights at the NS and the EW directions. By default, the light status in the NS direction is green, and the light status in the EW direction is red. Then, lines 13 to 23 define the logic of changing light status based on the values of the input variables.

```
1  TYPE Light : (Green, Yellow, Red); END_TYPE;
2  PROGRAM TrafficLight
3    VAR_INPUT
4      SensorNS : BOOL; SensorEW : BOOL;
5      ButtonNS : BOOL; ButtonEW : BOOL;
6    END_VAR
7
8    VAR_OUTPUT
9      LightNS : Light := Green;
10     LightEW : Light := Red;
11   END_VAR
12
13   IF LightNS = RED AND LightEW = RED AND NOT(ButtonNS)
         AND NOT(SensorEW) THEN
14     (* turn green when light is red, button is reset,
           and no emergency detected *)
15     LightNS := Green;
16   ELSIF LightNS = GREEN AND LightEW = RED AND SensorEW
         THEN
17     (* light must change when emergency approaches in
           EW direction *)
18     LightNS := Yellow;
19   ELSIF LightNS = GREEN THEN
20     LightNS := Green;
```

```
21   ELSE THEN
22     LightNS := Red;
23   END_IF;
24
25   (* The EW light status changes in a similar way *)
26   (* Omitted *)
27 END_PROGRAM
```

Code 1: A traffic light program in ST.

1.2. An attack Example. Normally, when the NS light is red, and an emergency vehicle is sensed in the NS direction, the sensor will be on until the NS light is switched to green. However, an attacker can manipulate the emergency sensor by switching it to on (e.g. $SensorNS := TRUE$) when the NS light is red and the EW light is green, and switching it to off (e.g. $SensorNS := FALSE$) when the NS light is red and the EW light is yellow. This can cause the green lights of both the NS direction and the EW direction to be on simultaneously.

1.3. Formal Verification. Next, we show how formal verification can catch the above-mentioned input manipulation attack.

We first model the ST program using the SMV language. This can be manually written or automatically generated through open-source tools, such as *st2smv*. As Code 2 shows, input variables are defined as $IVAR$ in lines 2 to 6. Other variables are defined as VAR in lines 7 to 9, and initialized in $ASSIGN$ using the $init$ function in lines 10 to 12. Lines 14 to 25 define the transition of light status, representing the program logic in Figure A.1 from lines 13 to 26.

We then specify the property that the green lights of the NS direction and the EW direction will never be on simultaneously. This is achieved in line 28 in which A denotes "always" and G denotes "global".

```
1  MODULE main
2     IVAR
3        button_NS: boolean;
4        button_EW: boolean;
5        sensor_NS: boolean;
6        sensor_EW: boolean;
7     VAR
8        light_NS: {RED, YELLOW, GREEN};
9        light_EW: {RED, YELLOW, GREEN};
10    ASSIGN
11       init(light_NS) := GREEN;
12       init(light_EW) := RED;
13
14       next(light_NS) := case
15          light_NS = RED & light_EW = RED & button_NS =
                FALSE & sensor_EW = FALSE: GREEN;
16          light_NS = GREEN & light_EW = RED & sensor_EW
                = TRUE: YELLOW;
17          light_NS = GREEN: GREEN;
18          TRUE: {RED};
19       esac;
20
21       next(light_EW) := case
22          light_EW = RED & light_NS = RED & button_EW =
                FALSE & sensor_NS= FALSE: GREEN;
23          light_EW = GREEN & light_NS = RED & sensor_NS
                = TRUE: YELLOW;
24          light_EW = GREEN: GREEN;
25          TRUE: {RED};
26       esac;
27
28    SPEC AG ! (light_NS = GREEN & light_EW = GREEN)
```

Code 2: SMV for the traffic light program.

Last, we use NuSMV to verify the property and obtain the following counterexample.

```
-> State: 1.1 <-
  light_NS = GREEN
  light_EW = RED
-> Input: 1.2 <-
  button_NS = FALSE
  button_EW = FALSE
  sensor_NS = FALSE
  sensor_EW = TRUE
-> State: 1.2 <-
  light_NS = YELLOW
-> Input: 1.3 <-
  sensor_EW = FALSE
-> State: 1.3 <-
  light_NS = RED
-> Input: 1.4 <-
-> State: 1.4 <-
  light_NS = GREEN
  light_EW = GREEN
```

Listing 1: A counterexample from the formal verification.

Listing 1 shows that the initial state (State 1.1) has NS light of green and EW light of red. Then, in State 1.2, the program receives an input of True $SensorEW$, so the NS light switches to yellow. Next, in State 1.3, the input of $SensorEW$ changes to False, but the NS light still has to change from yellow to red. Finally, in State 1.4, the EW light switches to green due to an earlier emergency request ($True\ SensorEW$) in State 1.2, while the NS light also switches to green since the emergency request has been cleared ($False\ SensorEW$) in State 1.3.

From the above counterexample, the input manipulation attack in Section A.2 is revealed. To prevent this attack, one can either forbid the input pattern of the counterexample, or redevelop the ST program accordingly.