AttkFinder: Discovering Attack Vectors in PLC Programs using Information Flow Analysis

John H. Castellanos
Singapore University of Technology and Design
Singapore
john_castellanos@mymail.sutd.edu.sg

Martín Ochoa
AppGate Inc.
Bogotá, Colombia
martin.ochoa@appgate.com

Alvaro A. Cárdenas
UC Santa Cruz
Santa Cruz, CA, USA
alacarde@ucsc.edu

Owen Arden
UC Santa Cruz
Santa Cruz, CA, USA
oarden@ucsc.edu

Jianying Zhou
Singapore University of Technology and Design
Singapore
jianying_zhou@sutd.edu.sg

ABSTRACT

To protect an Industrial Control System (ICS), defenders need to identify potential attacks on the system and then design mechanisms to prevent them. Unfortunately, identifying potential attack conditions is a time-consuming and error-prone process. In this work, we propose and evaluate a set of tools to symbolically analyse the software of Programmable Logic Controllers (PLCs) guided by an information flow analysis that takes into account PLC network communication (compositions). Our tools systematically analyse malicious network packets that may force the PLC to send specific control commands to actuators. We evaluate our approach in a real-world system controlling the dosing of chemicals for water treatment. Our tools are able to find 75 attack tactics (56 were novel attacks), and we confirm that 96% of these tactics cause the intended effect in our testbed.

CCS CONCEPTS

• **Security and privacy → Information flow control**; **Embedded systems security**; *Penetration testing*.

KEYWORDS

PLC program analysis, information flow, symbolic execution

ACM Reference Format:
John H. Castellanos, Martín Ochoa, Alvaro A. Cárdenas, Owen Arden, and Jianying Zhou. 2021. AttkFinder: Discovering Attack Vectors in PLC Programs using Information Flow Analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3471621.3471864

1 INTRODUCTION

Industrial Control Systems (ICSs) are a type of Cyber-Physical Systems (CPSs) in which sensors and actuators in the physical world are controlled by a distributed system of computing nodes called Programmable Logic Controllers (PLCs). These systems accomplish tasks such as water treatment and distribution, electricity generation, and manufacturing among others [4, 21]. Security and safety in such systems are thus critical to avoid physical harm to human beings operating or depending on such systems. Unfortunately, attacks are on the rise [12, 13].

The research community has proposed a variety of attack detection and attack-mitigation mechanisms for CPS [16], however, to evaluate their effectiveness, we need to test them with a robust set of possible attacks. For example a simple attack to overflow a tank can be to shut down the outflow pump, and turn on the motor valve at the water intake of the tank; however, a defence mechanism that only looks that this particular attack signature might miss other attacks that achieve the same objective (e.g., turning off the pump in a second stage of the system, which will create a cascade effect and overflow the first tank, or simply send false sensor signals reporting low water height will also overflow the tank). The question is, how confident are we that the attack benchmarks cover a wide range of possible attacks, and variations of these attacks to achieve the same objectives?

Unfortunately, identifying potential attack conditions is a time-consuming and error-prone process. Recently, Chen et al. [10, 11] proposed the idea of developing in software, a high-fidelity model of the physical system, and then use fuzz testing on the software simulation to automatically find several inputs (attacks) that cause the physical system to reach unsafe states. While the authors were able to find several attacks that were not identified in a manually-created benchmark, this approach has several drawbacks (1) we need to obtain a high-fidelity model of the physics of the system, (2) security researchers need to have expertise in identifying unsafe regions of the physical process, and (3) because physical systems have continuous variables, they must be represented by an infinite-dimensional state system, and finding inputs that drive this system to an unsafe state in a dynamic fashion (by executing the simulation multiple times) is computationally expensive.

In this work, we develop *AttkFinder:* a new automatic way to find attack conditions for industrial control system using a novel information flow-guided symbolic execution engine designed for the unique aspects of PLC programs and their interaction with industrial networks. Our approach (1) does not need a model of the physics of the system, (2) uses the safety conditions already coded on the PLC software to enumerate unsafe conditions, and (3) is efficient compared to alternative approaches because it only uses static analysis tools. Using a standard CPS security testbed, *AttkFinder* is able to find several attack vectors not previously discovered by previous proposals [10, 11] or other papers that developed attacks manually [2, 3, 22].

To achieve this in a variety of PLC programming languages, we also create a new intermediate representation of PLC code (which we call STIR). STIR can be applied to the most popular industrial programming languages such as Structured Text, Ladder Logic, and Function Block Diagrams. With this tool, a defender can find which variables can be used to drive the system to an unsafe state.

Surprisingly, we find several operational vulnerabilities that are not well documented in the plant's specification and are often unknown to operators. For instance, the operator of the system we tested believed that an event called *dead-head* (to be described later in the paper) was impossible because of the safety checks programmed in the PLC; however, with our tool, we found several ways to fool the safety checkers, as we explain in Table 3. Another example of previously unknown behaviour is a novel type of DoS attack we discovered against ICS components, which we can launch by abusing the use of indicators from actuators (Section 4.4). The vulnerabilities we found trigger safety conditions, such as tank overflows, pump dead-heads, component Denial-of-Service, or chemical contamination (Section 4.4). In summary, we discover 56 new attacks that go beyond traditional attacks against control-theoretical specifications.

In sum, our work makes the following contributions:

(1) We propose a new approach for automatically finding attacks that cause PLCs to send malicious control commands to the physical world based on information flow analysis and symbolic execution.
(2) We propose a new adversary model that considers different levels of access to a PLC (defined by the access control permissions). As far as we are aware, we are the first to identify and discuss how access control to variables works in PLCs. Previous work has not considered the access control granularity that adversaries may have; in particular, previous work has only assumed full control of the PLC logic [24, 25].
(3) We design new algorithms and tools to deal with PLC code intricacies for ladder logic, structured text, and function block diagrams.
(4) As far as we are aware, there is no open-source tool available to symbolically analyse PLC languages, let alone three different PLC languages. We are releasing our tools as open-source products[1].
(5) We evaluate our approach in a real-world system controlling the dosing of chemicals for water treatment. Our tools are able to find 75 ways to change the state of an actuator through the injection of malicious data or malicious commands. We compose our tactics to achieve well-known attacks like overflowing tanks [9, 35], as well as new and more sophisticated attacks like dead-head attacks (increase water pressure to either rupture a pipe or damage pumps), or dry-run attacks (damage a pump by turning it on while there is no liquid). Our automatic discovery of attacks shows a more comprehensive and diverse sets of attacks discovered than state-of-the-art approaches [10, 11].

[1]https://gitlab.com/jhcastel/attkfinder.

2 BACKGROUND

An ICS has four main components. (1) A ***plant*** represents the physical process under control, physical properties like pressure or temperature are called the state of the *plant*. (2) A set of ***sensors*** read the state of the *plant*, convert the physical properties into electrical signals, and deliver them to the controller. (3) One or more ***controllers*** monitor and control the state of the *plant*; they read the sensor signals, run the control strategy, and deliver the corresponding signals to the actuators. (4) A set of ***actuators*** receive electrical signals from the controller and modify the 'state' of the *plant* accordingly.

2.1 PLC Features

Industrial computers called Programmable Logic Controllers (PLC) are used in ICS. They are specialized computers that operate with high reliability. PLCs use multiple modules (physical add-ons to the main CPU) that provide new capabilities to the controller, including serial or Ethernet communications, discrete/analog input-output modules. PLCs are programmed using non-classical programming languages (described under Standard IEC-61131-3 [34]). The group of programs or routines are called the 'control logic'. As they control time-critical systems, PLCs must operate in a reliable and deterministic manner under industrial conditions (extreme temperatures, moisture, and electrically noisy environments).

2.2 PLC Access Control

PLC programming languages allow programmers to define access privileges per variable in each controller. In addition to a name and data type, programmers can assign access privileges to 'external' entities, allowing them to read/write variables remotely. There are three general options for remote access to internal PLC variables: (1) *Read/Write*: The variable can be accessed from any external device connected to the controller. (2) *Read Only*: External devices can read the variable but cannot modify its value. (3) *None*: The variable cannot be accessed from any external device.

All variables have *read/write* privileges by default [29, p. 63]. We use these features to enrich the threat model proposed in Section 2.6.

An external device may send a 'read' or 'write' request to the PLC with a particular variable as the payload. The PLC then looks up the requested variable, validates the read or write privileges, and either returns the value encapsulated in a 'read' response message or updates the value of the variable (if the external request was 'write').

Remark 1 (Remote access to PLCs' internal variables). *Any device in the industrial network such as the supervisor, Human-Machine Interfaces (HMIs), or other controllers can query PLCs. Queries include 'read' and 'write' messages. From a security perspective, a malicious actor could exploit such features and might compromise the confidentiality (reading PLCs' variables) and the integrity of the system (manipulating the variables that are used for control decisions).*

2.3 Scan Cycle

PLCs operate cyclically, repeating the same process over and over. This periodic process is called a ***scan cycle***.

Definition 1 (Scan cycle). *The scan cycle is a periodic process that handles the execution of the PLC program. In each scan cycle, the following process takes place. (1) The PLC reads sensors and stores their values in a local buffer. (2) Network messages are uploaded from the network module to local buffers and vice-versa. (3) The PLC executes the control logic. (4) The PLC updates output signals (to actuators) from values in the local buffer. (5) The PLC performs safety checks. (6) The cycle repeats.*

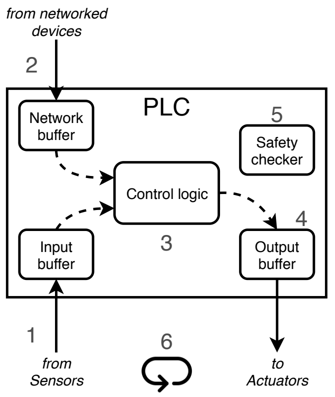


Figure 1: Scan cycle

PLCs enable real-time response guaranteeing the scan cycle does not exceed a pre-configured upper bound time in each execution period. If a routine exceeds the upper bound limit, the controller stops its execution and triggers an alarm.

2.4 Information Flow in PLCs

PLCs use internal 'buffers' to exchange data through modules. A module is a hardware addition to the PLC, usually to interconnect it to a supervisory network, or sensors and actuators. For instance, the input buffer stores the data obtained through the input module in a local memory space labelled with location, slot, and type [29, p. 16]. In Fig. 1, the input module (depicted as the arrow 'from Sensors') maps to the input buffer, and the data can be accessed by the control logic using the tag `RIO1:0:I.Data.0`. It is interpreted as the `RIO1` Module (Remote IO Module 1) attached to the PLC, Slot `0`, Type `I` (input), and the first bit of the data structure (`Data.0`). The network and the output modules have their local buffers, and they can be accessed from the control logic.

Network modules have separate *incoming*, *outgoing*, and *cached* buffers [30, Ch. 11]. Incoming buffers store receiving requests from other devices, outgoing buffers store information to be sent to other devices, and cached buffers keep outgoing connections open to be able to send multiple messages to the same recipient.

Definition 2 (PLC variable classification by information flow). *We highlight three types of variables in PLC code from the data-flow perspective. (1) Remote variables ($\mathbb{R}$): The PLC gets these variables from remote sources, and they come as network messages or signals from sensors directly connected to the PLC. (2) Output variables ($\mathbb{O}$): These variables affect external entities. The PLC delivers signals to actuators directly connected or to other devices via network messages. (3) Local variables ($\mathbb{L}$): All the remaining variables used internally in the program.*

Vendors recommend using routines to read buffers at the beginning of the scan cycle to get deterministic behaviour during the program execution [29, p. 18]. Routines in the control logic can be programmed to access data from the input and network buffers.

If an attacker can reach the supervisory network of the PLC, they can potentially use the remote access privileges to local variables. The attacker can either read the data or even modify the value of the variable and cause the control logic to execute malicious commands.

2.5 PLC Programming Languages

There are several programming languages for PLCs under the standard IEC-61131-3 [34], the most popular are Structured Text (ST), Ladder Logic (LL), and Function Block Diagrams (FBD) [30, p. 31].

2.5.1 *Structured Text (ST).*
ST is an imperative programming language generally used to program simple routines expressed as state machines. Case structures are useful to code state machines as shown in Fig. 2a, where a two-state logic controls a simple motor. Routines written in the ST Language can be called from other languages like LL and FBD using the Jump-to-subroutine function (JSR).

2.5.2 *Ladder Logic (LL).*
LL evolved from the early age of automation, where electromechanical relays were connected to build logic gates to control industrial processes. LL is a graphical sequential programming language identified by two parallel lines called buses and logical circuits connecting the left bus to the right bus. Each circuit is called a rung. Contacts (input instructions), depicted as parallel bars, and coils (output instructions), represented as parentheses, compose rungs.

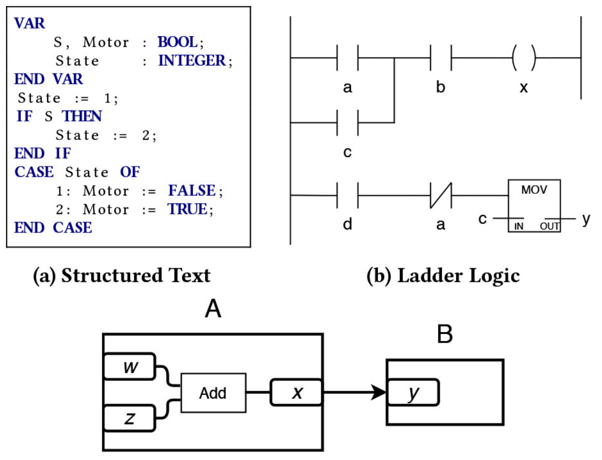
Fig. 2b shows an LL Program with two rungs, one composed of contacts *a*, *b*, and *c* and a coil *x*. The second rung has two contacts, *a* and *d* and the function `MOV`. Parallel connections correspond to a logical `OR` ($a \lor c$ in Rung 1), serial connections mean logical `AND`; logical `NEG` is drawn as a diagonal line ($d \land \neg a$ in Rung 2).

Special functions, like timers, counters, and Jump-to-Subroutine (`JSR`) can be configured in rungs.

2.5.3 *Function Block Diagram (FBD).*
FBD is a graphical programming language that offers an intuitive way to program a routine based on how different components communicate with each other. As the name suggests, the main components are blocks. Blocks have input parameters, output parameters, and internal routines.

Wires connect parameters and describe how data flows in the FBD Program.

Fig. 2c shows a small FBD program composed by two blocks, A and B. A has three parameters (variables), w and z as input parameters and x as output, they are connected through an ADD function which means $x = w + z$. A *wire* connects the parameter x to the input parameter y from the neighbour block B ($A.x \xrightarrow{\text{wire}} B.y$).



```
VAR
     S, Motor  : BOOL;
     State     : INTEGER;
END VAR
 State := 1;
 IF S THEN
     State := 2;
END IF
CASE State OF
     1: Motor := FALSE;
     2: Motor := TRUE;
END CASE
```

(a) Structured Text **(b) Ladder Logic**

(c) Function Block Diagram.

Figure 2: PLC programming languages.

2.6 Threat Model

We assume an attacker (1) is able to read any network message, (2) can attempt to read internal variables from the PLC, and (3) can attempt to write variables to the PLC (via network messages). However, the attacker cannot update the program running in the PLC, and cannot tamper with the PLC signals to the actuators.

Definition 3 (Attacker goal). *The goal of an attacker is to compose attack strategies that lead a system into a critical state via the indirect manipulation of actuator states.*

To change the plant's physical state, the attacker will try to change the output variables ($\mathbb{O}$) of the controller. Controller's output variables are the signals sent to the actuators, and therefore, they act as the final frontier between cyber and physical domains.

We assume that an attacker might, via espionage or social engineering, gain access to the controller's program and will be able to analyse it [6]. Moreover, we assume the attacker has a device with access to the PLC network, but they cannot modify the program running in the PLC. This assumption is realistic since PLCs implement hardware-based protections, and other researchers have previously proposed software attestation [25].

We define two attacker profiles for the threat model (Fig. 3), both have network access to the PLC, but have limitations based on the access control mechanisms implemented in PLCs:

A1: Full-access. The attacker can read and write arbitrarily local ($\mathbb{L}$) and remote ($\mathbb{R}$) variables.
A2: $\mathbb{R}$-access. The attacker has full read/write access to remote variables ($\mathbb{R}$) only.

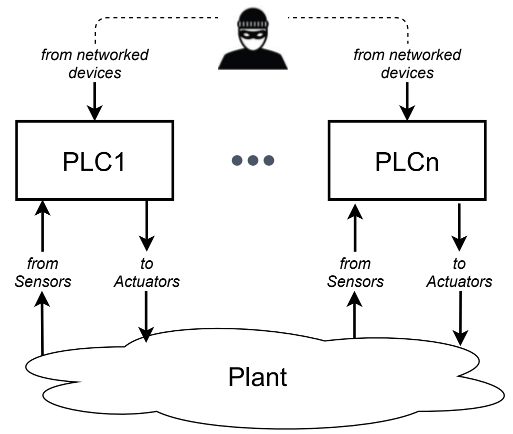


Figure 3: Threat Model

The key question we attempt to answer in this paper is ***how a controller can be manipulated without changing the software it runs by exploiting system compositions and information flows?***

Previous work focusing on symbolic execution for PLCs has explored threat models where the attacker can update the control logic of the PLC [24, 25]. However, changing the PLC program is not easy in practice: PLCs have a hardware-based protection mechanism that switches the PLC between 'run' and 'program' modes. If the PLC is in 'run' mode and the key of the PLC is not available, the PLC will not accept any software modifications. Unless the attacker is physically present in the plant and has access to the physical key to change the PLC operation, they will not be able to reprogram the PLC.

In this paper, we consider remote attackers that can break into the supervisory network of the PLC and then can send specific data to the PLC to alter its execution logic without physically changing the software on the PLC.

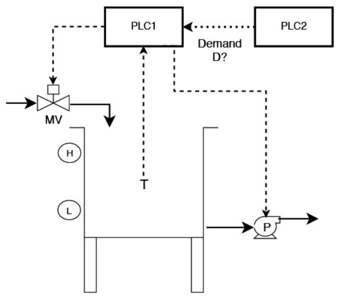
3 APPROACH

3.1 Simple Tank Filling Example

We use a simple tank filling system (Fig. 4a) as a running example to help us present our approach. The system's functional requirement is to provide water when needed. It has two actuators: a motor valve `MV` feeds the tank, and a pump `P` drains water from the tank. `MV` turns `ON` when the tank level (`T`) is below `L`, which guarantees that the tank always has sufficient liquid. `MV` turns `OFF` when the tank level exceeds `H`, which prevents the tank from overflowing. The system is considered to operate safely if the tank level remains between the physical limits $\mathtt{H}_{max}$ and $\mathtt{L}_{min}$. Once water is required (through the network message `D = ON`), and the tank level is above the `L` level, the pump (`P`) turns `ON`. The control logic described in Fig. **??** is the code running in PLC1 to guarantee the correct system operation.

3.2 Methodology to Compose Attack Vectors

The approach assumes the attacker has access to two primary information sources (See dashed boxes in Fig. 6): the source code of



(a) Diagram.

```
 1 'B1:' BEGIN
 2     T := ReadSensors();
 3     D := ReadNetMsg();
 4 'B2:' IF T > HIGH THEN
 5         MV := OFF;
 6         P := D;
 7 'B3:' ELSE IF T < LOW THEN
 8         MV := ON;
 9         P := OFF;
10 'B4:' ELSE
11         P := D;
12 'B5:' END
13     WriteActuator([MV,P]);
```

(b) PLC1 code.

Figure 4: Filling tank system.

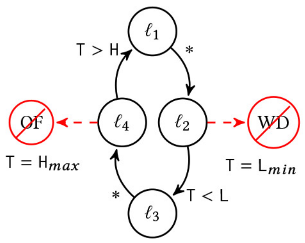
the controllers, and enough understanding of the system that is able to build the model of the system.

Based on the knowledge of the system, the attacker deduces the model shown in Fig 5. Four discrete states (locations) compose the model, each location has its corresponding actuator state, which controls how the tank behaves.

After understanding the system, the attacker can deduce the critical states, choose suitable targets, and design attack strategies. In the example, the attacker can identify two critical states, tank overflow and water deprivation, denoted as OF and WD in Fig. 5. The attacker also identifies actuator signals MV and P as their target.

The attacker only has access to the remote variables ($\mathbb{R} : \{D, T\}$). The goal is to 'control' the actuators MV and P only by modifying D and T. Once the attacker knows how to manipulate the actuators, he can compose attack vectors like tank overflow or water deprivation.

AttkFinder aims to discover multiple attack vectors through the analysis of PLC programs. AttkFinder is composed of three steps. (1) Statically analyse the PLC code to generate models and intermediate representations of the program. (2) Using symbolic execution, deduce attack tactics that can control variables in $\mathbb{R}$. (3) Compose attack vectors that affect the system (See the tool symbol in Fig. 6).



Locs.	MV	P	T
ℓ_1	OFF	D	↓
ℓ_2	OFF	D	↓
ℓ_3	ON	OFF	↑
ℓ_4	ON	D	↑

Figure 5: System model. Left: Finite state machine with critical states in red, states are called locations or modes in the literature [20, 31]. Right: Discrete and continuous variables in each location. ↑↓ shows if the continuous variable increases or decreases.

3.3 The PLC Parser

3.3.1 ***Modelling PLC Programs as Graphs.*** AttkFinder processes variable dependencies from each standard language and builds an information flow dependency graph (IFDG), as depicted in Fig. 7a. AttkFinder borrows the data-flow analysis from previous work [8] to build the IFDG. Nodes in IFDG represent variables, and edges refer to the relation among them. Edge labels show where the relationship takes place in the program. AttkFinder uses this information to search variable definitions more efficiently.

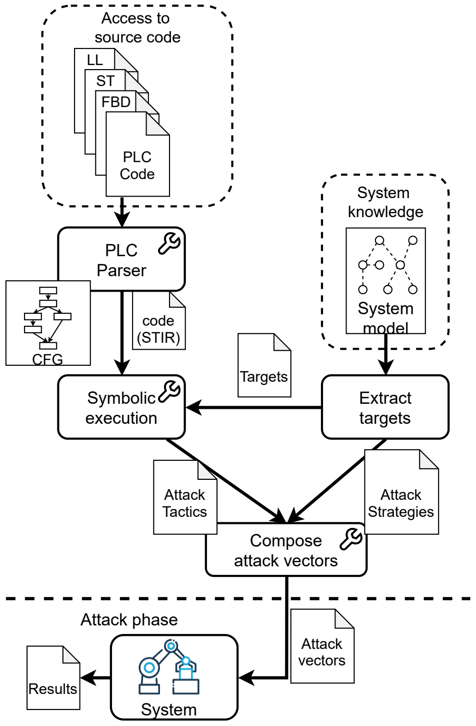
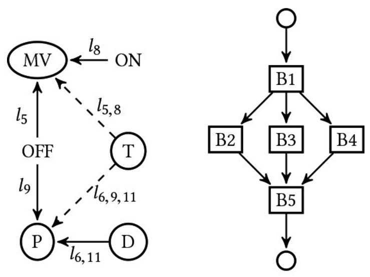


Figure 6: Diagram to compose attack vectors. The tool symbol shows the steps powered by *AttkFinder*.

A control-flow graph (CFG) models how a program is executed. AttkFinder builds a CFG as a chain of multiple *basic blocks* [27]. Edges represent the execution order of the basic blocks. A CFG has a single entry point and a single exit point. The first block that follows the entry point is called the 'root block'.

AttkFinder enriches produced CFGs adding attributes such as a *start line* and an *end line* for each block. It lets AttkFinder link CFGs and IFDGs. Fig. 7 shows the resulting IFDG and CFG for the tank filling example.



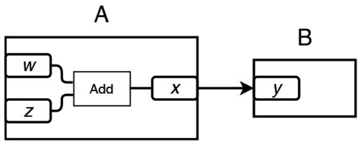
(a) IFDG (b) CFG

Figure 7: Graph models extracted from the static analysis of the PLC1 Code in Fig. 4a. The IFDG (left) describes *information flow dependencies*, solid edges denote direct flow, and dashed edges indirect. Control Flow Graph (right) describes different execution paths of the controller program.

3.3.2 *STIR (Structured Text Intermediate Representation).*

Instead of developing multiple tools to analyse each type of PLC programming language, we translate the standard programming languages [34] into an intermediate representation called **STIR** (Structured Text Intermediate Representation). STIR uses the programming guidelines to convert routines from FBD and LL to equivalent representations in ST. Vendors define these equivalences in the instructions reference manual [28]. We chose ST as the base for STIR because it allows smooth integration with an SMT engine.

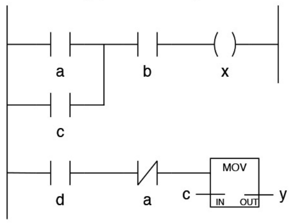
AttkFinder translates standard PLC into STIR as follows. In summary, FBD programs are processed in two steps. (1). We process the internal routines per block separately. (2). Assignments replace the wiring section, as follows: if we have two blocks A and B, A has an (output) variable x, and B has an (input) variable y, and they are wired $A.x \xrightarrow{\text{wire}} B.y$. We rewrite it as B.y := A.x in STIR notation (see Fig. 8a). In LL programs each rung is processed at the time, and routines written in ST language are copied directly to STIR without any modification.



(a) FBD Program

```
A.x := A.w + A.z
B.y := A.x;
```

(b) STIR



(c) LL Program

```
x := (a or c) and b;
if (d and not a)
    y := c
end if
```

(d) STIR

Figure 8: PLC programs to STIR

3.4 The Symbolic Execution Component of AttkFinder

The attacker feeds the algorithm with the set of targets deduced from Fig. 5 $t \in \mathbb{O}$, they are MV and P in the filling tank example. Then the algorithm searches for definitions of t in the IFDG. Then it gets the block ID for each definition. Blocks B2, B3, B4 change for example. The paths from the root block to the target block are symbolic traces ($\pi \in \Pi$). We include definition d into the trace π and evaluate it using symbolic execution to obtain a symbolic abstraction.

In the filling tank example, the attacker's goal is to learn how to manipulate actuators MV and P to compose attack vectors. AttkFinder returns the sets of symbolic abstractions that help the attacker to achieve their goal.

$\Pi_{(\mathtt{MV})}$ is the set of traces related to the operation of the motor valve MV, while $\Pi_{(\mathtt{P})}$ represents the set of traces that can change the behaviour of the pump P.

$$\Pi_{(\mathtt{MV})} = \begin{cases} \pi_2 : (\mathtt{MV} = \mathtt{OFF}) \land (\mathtt{T} > \mathtt{H}), \\ \pi_3 : (\mathtt{MV} = \mathtt{ON}) \land (\mathtt{T} < \mathtt{L}) \land \neg(\mathtt{T} > \mathtt{H}) \end{cases} \tag{1}$$

$$\Pi_{(\mathtt{P})} = \begin{cases} \pi_2 : (\mathtt{P} = \mathtt{D}) \land \mathtt{D} \in \{\mathtt{ON}, \mathtt{OFF}\} \land (\mathtt{T} > \mathtt{H}), \\ \pi_3 : (\mathtt{P} = \mathtt{OFF}) \land \neg(\mathtt{T} > \mathtt{H}) \land (\mathtt{T} < \mathtt{L}), \\ \pi_4 : (\mathtt{P} = \mathtt{D}) \land \mathtt{D} \in \{\mathtt{ON}, \mathtt{OFF}\} \land \\ \qquad \neg(\mathtt{T} > \mathtt{H}) \land \neg(\mathtt{T} < \mathtt{L}) \end{cases} \tag{2}$$

For example, Trace $\pi_3 \in \Pi_{(\mathtt{MV})}$ is deduced from Fig. 4b, π_3 corresponds to B3 from line 7–9 (l_{7-9}), where l_8 contains the MV definition (MV = ON). This definition depends on conditionals in l_4 and l_7. The conditional in l_7 has the predicate (T < L) and the ELSE condition suggests the predicate in l_4 must be negated (¬(T > H)).

Trace $\pi_4 \in \Pi_{(\mathtt{P})}$, B4 corresponds to lines l_{10-11} in the code. The definition in l_{11} depends on an input variable (D) which value cannot be determined at this point of the analysis $(\mathtt{P} = \mathtt{D}) \land \mathtt{D} \in \{\mathtt{ON}, \mathtt{OFF}\}$. This definition depends on conditionals in l_4, l_7 and l_{10}. The conditional in l_{10} suggests a negation of previous conditionals ¬(T < L) and ¬(T > H) for l_7 and l_4, respectively.

3.4.1 *Deducing Attack tactics.*

Attack tactics are concrete realisations of the abstract expressions described above. To get the set of attack tactics, AttkFinder uses an SMT solver engine in two steps (see Table 1: first checking satisfiability of the expression (SAT), and second generating a concrete case for it (SOLV).

For example, if an attacker wants to turn on a pump P, he may choose to evaluate π_2 from (2) with P = ON:

Target	Trace	Steps	
		SAT	SOLV
P = ON	π_2	✓	T = H + ϵ, D= ON
	π_3	✗	-
	π_4	✓	T = L + ϵ, D= ON

Table 1: Getting the attack tactics for target P = ON

A suitable attack for P = ON is thus [T = H + ϵ, D = ON], where ϵ is any small amount of T. Table 2 summarises all attack tactics in this example.

Additionally, we introduce the concept of effort indexing (the last column of Table 2).

Definition 4 (Effort index). *The effort index (ρ) is the number of variables an attacker needs to control to build an attack tactic.*

$\mathbb{O}$	#	Trace	$\mathbb{R}$		ρ
Target			T	D	
MV = ON	1	π_3	L − ϵ		1
MV = OFF	2	π_2	H + ϵ		1
P = ON	3	π_2	H + ϵ	ON	2
	4	π_4	L + ϵ	ON	2
P = OFF	5	π_2	H + ϵ	OFF	2
	6	π_3	L − ϵ		1
	7	π_4	L + ϵ	OFF	2

Table 2: Attack tactics for the filling tank example.

3.5 Composing Attack Vectors

This process is mainly manual. The attacker uses the system knowledge (Fig. 5) to deduce attack strategies, then use one or more attack tactics to compose attack vectors and achieve their goal. For example, if the attacker's goal is to cause a water deprivation via turning P OFF indefinitely, they can choose among attacks 5-7, where attack 6 ($\rho = 1$) is the easiest to perform because it only requires the attacker to control T. Additional conclusions from Table 2 are listed below.

- If an attacker controls input T, they can manipulate MV arbitrarily and P partially (can only be turned OFF).
- Input D does not cause changes to MV.
- If the attacker only controls D, they could conditionally manipulate P's state but only if T > L.

Initially, these conclusions might appear straightforward due to the example's simplicity, but our hypothesis is that this type of analysis will produce more valuable information in more complex systems, where the external variables are numerous, and the control strategy includes safety conditions and service routines.

4 EVALUATION

We test our analysis framework in two different types of PLCs (Allen Bradley and Wago) and from three different industrial systems (a chemical dosing process, a water filtering system, and a smart grid deployment).

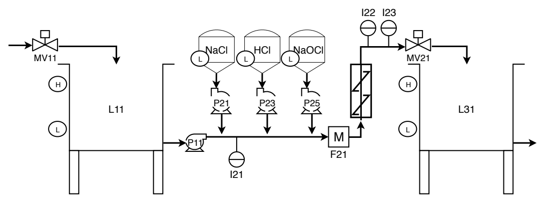
4.1 Chemical Dosing System

We start testing our approach in a chemical dosing system which is a two-tank cascade system. The system's goal is to have processed liquid available at L31 at all times (see Fig. 9). The system must guarantee that L31 and L11 operate under safe conditions (between H and L limits). L11 supplies liquid to L31, and pumps P21, P23, P25 add NaCl, HCl, and NaOCl chemicals, respectively. The magnetic sensor F21 measures the liquid flow in pipe P11-MV21, while sensors I21, I22, I23 measure the chemical properties of the liquid. The system has two *safety interlocks*, (1). P11 opens only if MV21 is already open, (2). Chemicals (P21, P23, P25) are applied only if liquid is flowing from L11 to L31, as measured by F21.

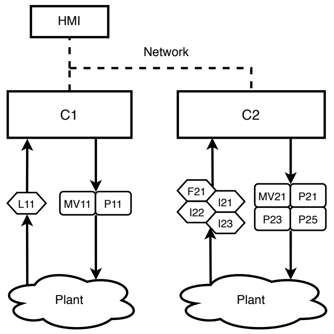
The system operates in two modes, automatic and manual operation. Two PLCs (C1, C2) control the system. C1 and C2 monitor that the system runs under safe conditions, such as (1). Tanks (L11, L31) must maintain a level between the H and L limits. (2). Chemical levels in the water must not exceed safe limits. (3). Safety interlocks must be guaranteed.

C1 controls the first half of the system; it reads sensor L11 and controls MV11, P11. Fig. 10 shows the system model governed by C1. OF and UF stands for Overflow and Underflow of tank L11. Hmax and Lmin are the bounders where the system reaches unsafe states. Similarly, C2 controls the second half of the Testbed; C2 reads I21, I22, I23, F21, sensors and controls MV21, P21, P23, P25. The reader can refer to the appendix the see C2's model, Fig. 14 shows how the system evolves and highlights Chemical contamination (CC) besides OF and UF unsafe states.

C1 and C2 are Allen Bradley PLCs (1756-L71) and share their states through network messages based on the Ethernet/IP and CIP industrial protocols as shown in Fig. 9b.

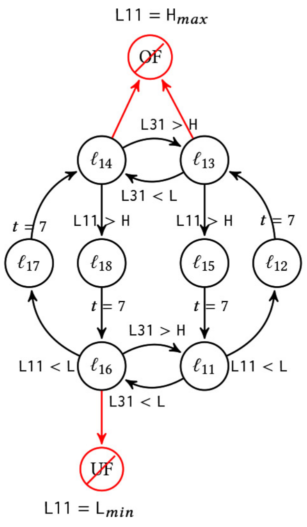


(a) System



(b) Connection diagram

Figure 9: Chemical Dosing System



Locs.	MV11	P11	L11
ℓ_{11}	OFF	OFF	-
ℓ_{12}	OFF⟶ON	OFF	↑
ℓ_{13}	ON	OFF	↑
ℓ_{14}	ON	ON	↑
ℓ_{15}	ON⟶OFF	OFF	↑
ℓ_{16}	OFF	ON	↓
ℓ_{17}	OFF⟶ON	ON	↑
ℓ_{18}	ON⟶OFF	ON	↑

Figure 10: System model managed by C1. Left: Finite-state machine of discrete states (locations), critical states in red. Right: Location list and discrete and continuous variables. Arrow (⟶) represents a transition state in the actuators. ↑ means the level increases, ↓ means the level decreases.

4.2 Analysis of C1 and C2 Programs

C1 has 11 routines: 8 ST, 2 LL, and 1 FBD, and C2 has 13 routines: 9 ST, 2 LL, and 2 FBD. AttkFinder produced a 1000+ LOC code in STIR format (see Section 3.3.2), a 227-block CFG for C1, a 2700+ LOC STIR file, and a 573-block CFG for C2.

From the system models, we define as target the set of actuators $A_{C1} \in \{\text{MV11}, \text{P11}\}$, and $A_{C2} \in \{\text{MV21}, \text{P21}, \text{P23}, \text{P25}\}$ with values $V \in \{\texttt{OFF}, \texttt{ON}\}$. We feed AttkFinder with these targets.

4.3 Attack tactics

After processing C1 and C2 programs, AttkFinder automatically identifies the targets in the corresponding CFGs. Each target is processed by AttkFinder and produces a set of traces Π, with its corresponding symbolic expression.

When the symbolic expression is a logical disjunction, attack tactics are evaluated in different predicates that satisfy the symbolic expression. For example, the trace

$$\pi_{105} : \texttt{MV11.C} = 2 \land \texttt{AUTO.OFF} \land (\texttt{MV11.FO} \lor \texttt{MV11.FC}) \quad (3)$$

can be interpreted that during manual operation ($\texttt{AUTO.OFF} = 1$) and after MV11 turns ON ($\texttt{MV11.C} = 2$), C1 checks if there was a failure opening MV11 ($\texttt{MV11.FO} = 1$) *OR* closing MV11 ($\texttt{MV11.FC} = 1$). We split the original symbolic expression (π_{105}) into two; one checking for the opening failure MV11.FO ($\pi_{105.1}$), and the other for the closing failure MV11.FC ($\pi_{105.2}$).

Concrete executions of such expressions produce the attack tactics. In total, *AttkFinder* discovers 75 attack tactics.

Remark 2 (limitations on symbolic execution). *In contrast to classical computer programs, PLC programs execute the code continuously governed by the scan-cycle. It means PLC program variables that seem not to have any dependency in classical programs might have dependencies from previous executions of the program. To overcome this limitation in symbolic execution, we perform a complementary n-cycle analysis (η) to discover an $\mathbb{L} \xrightarrow{\eta} \mathbb{R}$ mapping.*

The intuition behind the n-scan-cycle analysis is that variables from $\mathbb{L}$ used in attack expressions might depend on values of variables in $\mathbb{R}$ computed in previous scan cycles. The reader can refer to the appendix for a more detailed explanation on $n-$cycle analysis.

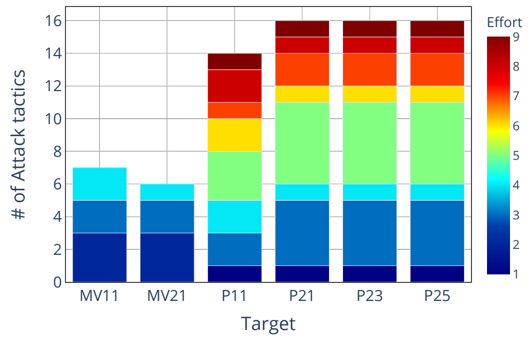


Figure 11: Number of attack tactics by target. Colour scale denotes the effort index (ρ) ranged from 1 to 9.

Fig. 11 shows how the attack tactics are distributed among the targets. They are ranked by effort index (ρ). We define the effort index as the number of variables that an attacker needs to control to launch an attack tactic (see Section 3.4.1). The larger this index is the harder the attack will be to launch.

Among the targets, motor valves (MV in Fig. 11) have the attack tactics with considerable low ρ. It suggests these targets are easier to manipulate by attackers than pumps.

Remark 3. *Effort index (ρ) can be a valuable metric to rank attack tactics, as higher the index as harder the attack, because it is proportional to the number of variables involved. Risk assessment is a domain where ρ can ignite novel indicators.*

4.4 Composing Attack Vectors

So far, we have shown that attackers can send inputs to a PLC so that it changes the state of some actuators. But changing the state of an actuator alone might not be an attack. For example, turning ON MV11 when L11 is far below H does not cause problems, whereas if an attacker can deliver a similar attack to L11=H, this might cause a tank overflow. We now discuss how we discover safety threats and how to create attack vectors that override these conditions.

These safety conditions should be validated before changing the state of actuators. Controllers check the safety conditions based on sensors or via network messages when they are monitored by other controllers. Thus the *safety conditions are a subset of* $\mathbb{R}$.

Remark 4. *The necessary conditions for all attack tactics linked to a state is an approximation of the safety conditions associated with the particular actuator.*

For instance, based on Table 9, for P11=ON, the invariant (necessary conditions) $\pi_{145} \land \pi_{158}$ is

$$\{\texttt{C2} : \texttt{MV21.ST} = 2, \texttt{LL} > \texttt{AI0} > \texttt{T}_{max}, \texttt{DI2} = 1, \texttt{DI4} = 0\}$$

We identify six possible attack scenarios where safety conditions are compromised, or the operational goal is disrupted. These conditions are *tank overflow*, *tank underflow*, *dead-headed pump*, *dry-running pump*, *component Denial-of-Service (cDoS)*, *chemical contamination*, and *membrane fouling*.

Tank overflow. If an attacker wants to overflow tank L11, he needs to choose the appropriate context (when L11≈ H), ℓ_{13} or ℓ_{14} in Fig. 10. Then he needs to increase the L11 level. This is possible via tactics 6 or 7. tactic 6 exploits the manual operation of MV loading $\texttt{AUTO.OFF} = 1$ and $\texttt{MV11.C} = 2$ to C1. tactic 7 changes the tank level readings from AI0 making C1 believe the tank is at a low level. The attacker also needs to keep the AI0 value under 1000 to bypass overflow checkers (tactic 5).

Tank underflow. Similarly, an attacker might choose to underflow L11 via spoofing the tank level sensor AI0 to show a level above the 200mm limit.

Dry-running pumps. Pumps are designed to operate with a pumped liquid that opposes its action. A lack of liquid causes component overheat and may cause damage. In our scenario, this strategy is linked to the tank underflow case presented before. It is achieved using the same attack tactics in the context of L11 empty ($\texttt{AI0} \approx 0$).

Dead-headed pump operation. Dead-head occurs when a pump runs and, the discharge line[2] is closed. As the liquid cannot flow, the pressure in the pipe increases causing the pump to overheat.

This strategy requires two phases: (1) keep MV21 closed and (2) open P11 to increase the pressure at the pipe P11-MV21. To close MV21, an attacker might combine tactics 22, 24, and 25. He can then use tactics 20 or 21 to open P11.

Interestingly, this was an attack strategy unknown by the operator as he believed C1 would manage the undesired behaviour.

DoS of components (cDoS). In this strategy, the attacker aims to disable a particular component by falsely portraying unsafe conditions. When a controller detects unsafe conditions, it turns `OFF` the particular component causing a Denial-of-Service.

For instance, in C2, `DI10` indicates if P23 is in a faulty state, an attacker can induce P23 to be turned `OFF` by a safety mechanism forcing `DI10` = 1 as shown in tactics 46 and 51. All cases of cDoS attacks are shown in Table 3.

Chemical contamination. Pumps P21, P23, and P25 control the dosing process for the chemicals NaCl, HCl, and NaOCl chemicals, respectively. Each chemical controls a particular property of the liquid. For instance, HCl controls the pH value of the water, and C2 should keep it between 6.5 and 7.5 to be acceptable for human consumption[3].

If the pH level in the water drops to acidic levels (< 6.5), the liquid is considered contaminated and corroding metal pipes thus becoming unsafe for human consumption. In our scenario, an attacker can achieve this goal via spoofing sensor `AI2` > 7.05 in C2, which will make C2 think pH level is always high, shutting down the HCl source (P23) as shown in the attack tactic 48.

Membrane fouling. This is the loss of filtering properties of membranes in water treatment systems. Variations outside the operational limits of the chemicals NaCl and NaOCl chemicals can cause membrane fouling [15].

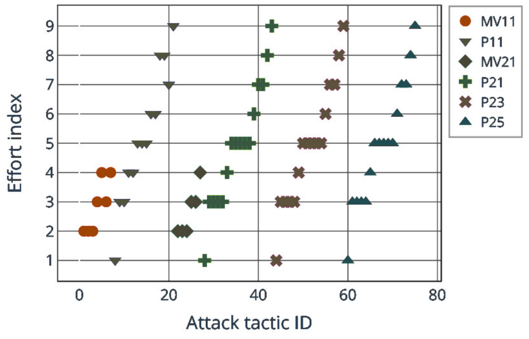
In our scenario, an attacker can spoof sensors `AI1` and `AI3` to change states of pumps P21 and P25, respectively. The attack tactics associated with this strategy are 32 and 43 for P21, and 64 and 75 for P25.

Table 3 summarises the attack vectors.

The effort index (ρ) shows how difficult it is to launch the attack, i.e. in L11 overflow (row 1 in Table 3), three attack tactics can compose the strategy, 5 ($\rho = 4$), 6 ($\rho = 3$) and 7 ($\rho = 4$). If an attacker uses tactics 5 + 6, the final effort is $\rho = 7$. If he uses tactic 7, the effort is $\rho = 4$. According to the effort index, cDoS and chemical contamination strategies are the cheapest to compose because they use one tactic only.

4.5 Evaluating Performance on Different Systems

Finally, to test the performance of our algorithms, we select a diverse set of PLC programs from different real-world systems: the SmartGrid, chemical dosing, and water filtering. The SmartGrid



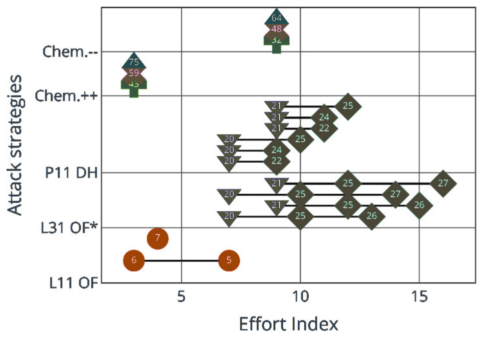


Figure 12: Top: All 75 attack tactics with its effort index, colours and markers identify affected actuator. Bottom: Attack vectors composed by attack tactics, solid lines connect them to form an attack vector, numbers refer to tactic ID . OF: Overflow, UF: Underflow, DH: Dead-head, DR: Dry–run, Chem.++: Increment of chemical level, Chem.—: Reduction of chemical level. * L31 OF/L11 UF/P11 DR are achieved with the same attack vector group.

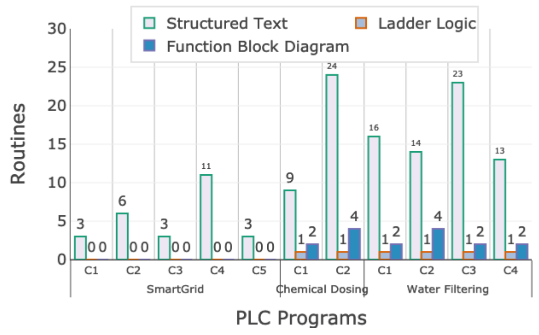
has 5 WAGO PLCs to control generation, transmission, and consumption processes. Their programs were written in Structured Text language. The chemical dosing and water filtering systems have Allen Bradley's PLCs to control their sub-processes. The PLC programs use a combination of Structured Text, Ladder Logic, and Function Block Diagram routines to handle the processes. PLC Programs vary in size, from a 3-routine ST-only program to a 29-routine multi-language program (C1-SmartGrid and C2-chemical dosing, respectively) as shown in Fig. 13a.

The PLC programs are coded in diverse programming languages; some use a combination of LL, FBD, and ST while others use pure ST. We test the *PLC-Parser* in all eleven PLC programs to evaluate its performance and the consistency of the graphs and the STIR representation. Fig. 13b shows the average time that the *PLC-Parser* takes to process PLC programs of different sizes and complexity. The plot also depicts the size of the outcomes, graphs (in nodes), and the STIR representation code (in Lines-of-Code). The produced CFG graphs vary in size from 37 blocks to 575 blocks; the size of the CFG depends on the complexity of the control strategy in the PLC.

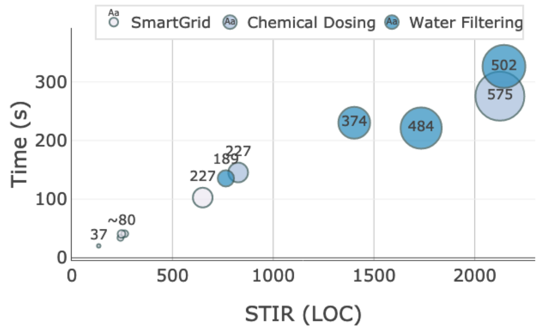
[2]Discharge line refers to a pipe or conduit connected to the pump output
[3]The US Environmental Protection Agency suggests keeping the pH level of drinking water between 6.5 and 8.5 https://www.epa.gov/sdwa/drinking-water-regulations-and-contaminants

Goal	Attack tactics	Attack vectors	ρ	Capability	
				A1	A2
L11 overflow	7 or (6 and ¬5)	2	[4, 7]	✓	✓
L31 overflow	(20, 21) and ¬(26, 27) and ¬(25)	4	[13, 16]	✓	✗
L11 underflow	(20, 21) and ¬(26, 27) and ¬(25)	4	[13, 16]	✓	✗
P11 dead-head	(20, 21) and (22, 24, 25)	6	[9, 12]	✓	✗
P11 dry-run	(20, 21) and ¬(26, 27) and ¬(25)	4	[13, 16]	✓	✗
MV11 DoS	3	1	2	✓	✓
P11 DoS	8, 10, 13, 14, 17	5	[1, 6]	✓	✓
MV21 DoS	24	1	2	✓	✓
P21 DoS	28, 30, 33, 34, 37	5	[1, 5]	✓	✓
P23 DoS	44, 46, 51, 52, 55	5	[1, 5]	✓	✓
P25 DoS	60, 62, 67, 68, 71	5	[1, 5]	✓	✓
Chem. Decr.	32, 48, 64	3	3	✓	✓
Chem. Incr.	43, 59, 75	3	9	✓	✗

Table 3: Attacker capabilities of composing attack strategies



(a) Program sizes by number of routines and languages



(b) *PLC-Parser* Performance and outcome of the PLC-Parser tool

Figure 13: Performance of the *PLC-Parser* evaluated in PLC programs of different sizes from 3 routines to up to 29 routines. Circle diameters represent the size of the resultant CFG graphs.

5 DISCUSSION

5.1 Taxonomy of Discovered Attacks

We classify the attack tactics based on the variables of interest they target in the PLC code. We identify the following four groups: physics-inspired, internal state, component signalling, and inter-controller communication.

5.1.1 ***Physics-inspired.*** They are the most intuitive set of attacks because they are directly linked to the control strategy. They have been explored in the literature as *false data injection attacks* [37] or *deception attacks* [5, 7]. Values in these attacks are associated with setpoints where actuator states change, i.e. C2 adds NaCl to the liquid to increase the conductivity until it reaches the upper setpoint at 265μS/cm. Then C2 changes the strategy to cut the NaCl dosing. The attacker aims to lead the controller to a wrong control strategy spoofing sensor readings like tank level, flow sensors, etc.

For instance, in our analysis, the attacker can change MV11 (`AUTO.ON` = 1) by modifying values in sensor `AI0` to make C1 think that the tank is below the lower limit of 500cm, so C1 will open MV11. Alternatively, the attacker can spoof the sensor with the value 1000cm to fool C1 to close MV11 because the tank seems to have reached the upper limit. 16 out of 75 attack tactics correspond to physics-inspired attacks (21.33%).

5.1.2 ***Internal state.*** Internal state attacks target only internal variables in the code. They aim to force the execution of particular traces in the PLC Program to achieve a goal. An example of an internal state attack is tactic 1, where the attacker forces `AUTO.OFF` = 1 and `MV11.C` = 1. Here the attack aims to activate the manual routine of MV11 that turns it `OFF`. A static program analysis can deduce this attack tactic. Still, it can be challenging to deduce it from a control-theoretic perspective since it is not related to the physical behaviour of the system. 25 out of 75 attack tactics correspond to internal state attacks (33.33%).

5.1.3 ***Component signalling.*** Actuators report their status back to the controller using a set of specific indicators. For instance, an actuator indicates if its status is 'running' or 'stop' or, in case of a hardware malfunction, a 'failure' signal. All these signals are connected to their respective controller. In our scenario, `DI3` is the P11's 'running' state indicator, and `DI4` is the P11's 'failure' indicator. Both are connected to C1, and C1 takes control decisions based on their states.

An attacker can hijack such signals and report a wrong status to the controller. To the best of our knowledge, we are the first to study this type of attacks.

For instance, if the attacker intercepts a signal from the `DI4` sensor to C2 (it corresponds to a 'faulty' signal of P21), he can overwrite `DI4` = 1, and C2 will think P21 suffers a hardware failure. Then C2 will deliver an `OFF` signal to P21 as a safety measure. 27 out of 75 attack tactics correspond to component signalling attacks (36.00%).

5.1.4 ***Inter-Controller communication.*** Controllers share their states using network messages; if an attacker compromises this communication, it can affect the system severely via state mismatching among controllers. This state mismatching can lead the system to operate under unsafe conditions.

Examples of this type of attacks are 16, 21, and 27. Attack 16 can cause starvation (running out of water) at tank L31 closing P11 due to the spoofed message `C2 : MV21.ST = 1`. There is a safety mechanism in place that does not allow P11 to turn `ON` if MV21 is closed, so the spoofed message will make C1 think MV21 is `OFF`, activating the safety mechanism. Attacks 21 and 27 can cause L31 overflow via spoofing of message `C3 : L31.L = 1` causing C1 and C2 to believe tank L31 is always below the lower limit.

A more elaborate set of attacks can cause severe damage in pipe P11-MV21 if the attacker writes a network message to `C2 : MV21.ST = 2` while MV21 is closed. We present the composition of attack vectors in Section 4.4. 7 out of 75 attack tactics correspond to inter-controller communication attacks (9.33%).

Remark 5. *Physics-inspired attacks have been explored extensively in the literature [5, 7, 37], usually proposed by the intuition of the researchers. On the other hand, our approach systematically discovers attack tactics from the PLC Code (including physics-inspired) ≈ 79% novel attacks. This method expands the diversity of data-oriented attacks and can ignite the study of novel offensive/defensive techniques in CPS security.*

5.1.5 ***Time-constrained features (TC).*** As the name suggests, TC attacks have a time-domain component associated with them. The attack vector has two phases. First, a *timing condition* must be held, then the attack *condition* is validated. The timing component is described by a two-tuple (x, t), where x is the predicate holding for t time.

For example, the attack tactic 3 is a TC attack with the following symbolic expression:

$\pi_{120.3} : \texttt{AUTO.ON} \land ((\texttt{MV11.Open} \times 6) \land \neg \texttt{DI8})$

Can be read as: *'if* `DI8 = 0`*, during 6 seconds after* `MV11.Open = 1` *then* `MV11 = OFF`*'*. In other words, the result of attack 3 will take effect 6 seconds after launching the attack.

Remark 6 (Reducing the opportunity window in TC). *The 'attack window' in attacks with a time-constrained feature is bound by the timer duration (τ) and the moment the timer is reset (*`T.RST`*) as changing τ might cause changes in the operation of the CPS. The only option to reduce the window is to move* `T.RST` *as close as possible to τ. This strategy can be deployed in the PLC Code enforcing a* ***reset-after-use*** *policy of all timer variables.*

From the whole set of attack tactics, 17 out of 75 have TC features. The reader can refer to Table 7 in the appendix to review all variables with TC features.

In summary, our attacks cover a more diverse set of criteria and conditions than any other previous work on automatic attack discovery or manual attack benchmarks, as illustrated in Table 4. *AttkFinder* finds novel attacks not previously discussed, such as the Dead-headed pump attack, the dry-running pump attack, chemical contamination attacks, and even DoS attacks. Our work shows the limitations of previous fuzzers of models of the physical system [10, 11]; not only are they more expensive to deploy and use (they require high-fidelity models of the physics of the system), but their reliance on the physics of the system ignores other safety conditions such as the possibility of running a pump in dry conditions (these device conditions are usually not captured in the physical models of a process). We believe our tool offers a complementary and robust alternative to semi-automatically finding attacks on industrial systems.

5.2 Effectiveness of Attack tactics During the Attack Phase

The attack tactics are evaluated under two different attack profiles described in Section 2.6. The attacker sits on the industrial network and can intercept messages that reach the PLCs, and can query the PLC via network messages. **A1** is a powerful attacker that can modify variables arbitrarily in both spaces ($\mathbb{R}$ and $\mathbb{L}$). **A2** has total access to $\mathbb{R}$, but they are totally blind about what is happening in $\mathbb{L}$.

We test all attack tactics in a chemical dosing system described in Section 4.1. The variables in $\mathbb{R}$ are intercepted before being processed by the control logic, and values in $\mathbb{L}$ are forced via the engineering workstation. Libraries such as *pycomm*[4] allow engineers to program python scripts to communicate with PLCs. An attacker or malware in the industrial network can use similar tools to interact with a target PLC. An attacker can launch a Man-in-the-Middle attack (e.g., by compromising an industrial switch [36]) to spoof network messages carrying messages from other controllers or remote sensors. At the same time, it can read and write internal variables of the PLC using *Pycomm*.

We classify the results of the tested attack tactics as *Successful*, *Conditional*, *Failed*, or *Unable*. In successful attacks, the attacker achieves to change the target state. *Conditionally Successful Attack* means the attacker can launch[5] the attack, but the success depends on variables that the attacker does not control (in $\mathbb{L}$ space). Then they have to deduce or read them. *Failed* means the attacker launches the attack, but the target does not change its state. And finally, *Unable* means the attacker faces limitations to launch the attack, i.e., they cannot change all needed variables ($\mathbb{L}$ and $\mathbb{R}$).

Table 5 summarises all successful (**S**) and conditional successful (**C**) attack tactics order by target and attacker profiles. As **A1** is the strongest attacker, he can launch a higher number of successful attacks (72 in total) while **A2** achieves 54 attacks. The reader can refer to the appendix (Tables 9 and 10) to see the detailed results of the 75 attack tactics.

From the results, we conclude **A2** cannot turn `ON` any pump (P11, P21, P23, and P25) in the system, see rows with value 0 in Table 5. The reason is that the set of attack tactics PXX=`ON` (20, 21, . . . , 75) strongly depends on variables in $\mathbb{L}$, and as **A2** cannot read $\mathbb{L}$ states, they need to guess these values. It makes it difficult to catch the right moment to launch the attack. To identify this precise moment becomes a probabilistic problem *predicting with high confidence when conditions in $\mathbb{L}$ will satisfy.*

Experiments reveal a non-explicit behaviour of the system. C1 synchronises System's global states, it means C1 updates `AUTO.OFF` and `AUTO.ON` variables in C2 via network messages. These variables belong to $\mathbb{L}$ in C1, but to $\mathbb{R}$ in C2. it means attacker **A2** can influence C2 more than C1, in other words, C2 is more vulnerable than C1. In C1, **A2** can launch only one **S** attack, while in C2, the number of **S** attacks increases to 13.

[4]https://github.com/ruscito/pycomm

[5]launching an attack means the attacker changes required variables to desired values.

Work	Threat model			Attack crafting	Attack strategy					
	S	A	N		OF	UF	DH	DR	cDoS	Chem.
SSMP [2]	✗	✗		M	✓	✓				
Noise matters [3]	✗		✗	M	✓	✓				
Smart fuzzing [10]		✗		A	✓	✓				
Active fuzzing [11]			✗	A	✓	✓				
Tabor [22]	✗	✗		M	✓	✓				✓
AttkFinder	✗		✗	S	✓	✓	✓	✓	✓	✓

Table 4: Comparing attack diversity with previous works. Threat model refers to main attack vector, (S)ensor, (A)ctuator or (N)etwork . Attack crafting: (M)anual, (A)utomatic or (S)emi-automatic. Attack strategy: Tank overflow (OF), tank underflow (UF), Dead–headed pump (DH), Dry–running pump (DR), DoS of components (cDoS) and chemical contamination (Chem.)

		Attacker	
Target		A1	A2
C1	MV11 = OFF	3	2
	MV11 = ON	2	1
	P11 = OFF	12	11
	P11 = ON	2	0
$\sum$		19S	1S+13C
C2	MV21 = OFF	3	1
	MV21 = ON	2	0
	P21 = OFF	14	13
	P21 = ON	2	0
	P23 = OFF	14	13
	P23 = ON	2	0
	P25 = OFF	14	13
	P25 = ON	2	0
$\sum$		53S	13S+27C
Total		**72** (72S)	**54** (14S+40C)

Table 5: Summary of successful (S) and conditionally successful (C) attack tactics by attacker profile.

6 RELATED WORK

Applications of the symbolic execution (SE) vary from automatic test generation to malware analysis [33]. In the context of CPS security, symbolic execution has been explored to identify if a PLC code contains attacks. SABOT [24] aims to find a mapping from a semantically meaningful set to a memory set; the authors coined the term *variable to device mapping* (VTDM). SABOT requires the attacker to gain detailed knowledge of the system. An attacker writes a specification that describes how the control logic is supposed to execute. Then the tool matches this specification against the program in the PLC. Once the attacker understands how the program is coded, he can write malicious code and update the PLC to attack the system.

In a similar fashion, McLaughlin et al. [25] studied how to address attacks where the actor can upload malicious code to the PLC. TSV uses a combination of symbolic execution and model checking to verify PLC programs satisfy the safety properties of the CPS. Engineers define safety properties as a Linear Temporal Logic proposition. TSV is deployed as a bump-in-the-wire device. It captures a program before being uploaded to the PLC. TSV converts the PLC's assembly code into an Intermediate Language representation called ILIL. Then it builds a symbolic scan cycle that models all possible executions of the PLC Program. The Authors propose a Temporal Execution Graph to model the relation through multiple scan cycles. Using model checking, TSV verifies the safety properties and produces a counterexample in case of violation. Although TSV and our approach use similar techniques, the reasoning of the threat model is different. While TSV's threat model involves an attacker that changes the PLC Program, ours explores a more restricted actor, an attacker that wants to exploit the operational vulnerabilities remotely.

In automatic attack generation, Sarkar et al. [32] present "I came, I saw, I hacked". In their approach, the authors propose an automatic discovery tool to build attacks for ICS: (1) Their method collects screenshots from HMI devices and binaries from controllers. (2) Machine learning algorithms classify the type of system the CPS is controlling. (3) Through control-theoretic techniques, the tool creates modified versions of PLC programs to manipulate the system's behaviour. A significant difference is that AttkFinder aims to affect the system via the injection of network packets. In contrast, their attack vector is a subtle change on the controller program.

Other researchers have previously proposed information-flow analysis in Cyber-Physical systems, but they aimed to solve different research Challenges. Morris et al. [26] propose modelling a CPS as a state machine and apply information flow analysis to quantify the effects of an attacker controlling components of the system. Castellanos et al. [8] propose the application of information flow to perform a risk analysis on an ICS; they aim to classify which components are the most vulnerable in the system. Our approach has similarities in the use of information flow analysis, but we complement it with symbolic execution to get concrete cases of attack tactics in the system. Additionally, we test our approach with multiple PLC vendors in a variety of realistic systems.

From a software testing perspective, Guo et al. [18] propose SymPLC by translating Structured Text PLC programs into C. Then SymPLC uses a symbolic execution engine to generate test cases. While SymPLC considers particular data types such as timers, it evaluates only the output of this data type as a boolean. On the other hand, in our analysis, we include the timing parameter as a key component for the time-constrained attacks (see Section 5.1.5). Another difference is that our approach also processes LL and FBD languages and can be adapted to other vendors.

All these previous efforts consider that the attacker can modify the program running in the PLC. In contrast, our contributions focus on identifying inputs that the attacker can send to the PLC to force it to change the actuators in a way desired by the attacker without the need to modify the software running on the PLC. It is a realistic scenario as PLCs are meant to run for several years without any updates, and engineers usually protect the code of a PLC by moving a physical key from "programming" to "running". Without the key, the attacker cannot load new software in the PLC. We also point out that our attacks can bypass control-flow integrity checkers for PLCs [1] as our attacks pass through approved execution paths of the software in the PLC.

Furthermore, these previous efforts only considered one PLC programming language. However, our work is applicable to three different and widely-used programming languages in PLCs. In addition, we translate programs that use different routines in different languages converted into an intermediate representation STIR file. Our approach also evaluates special features from PLCs like timing components and scan-cycle implications (which are ignored by previous work [18]). Our tools will be released as open-source products to help engineers analyse PLC programs.

Several other papers considering the security of control systems [9, 14, 17, 19, 23, 35]. Most of these previous papers consider how to attack a control system by driving the system to an unsafe state, focusing only on the main variables under control, for example, overflowing or under-flowing a tank, or adding chemicals to the water to drive the pH to unsafe levels. Our analysis, however, discovers vulnerabilities in hard-coded safety features that are not part of the main control logic, such as preventing that a pump is turned on when no water is flowing through or preventing dead-headed pump operations.

7 CONCLUSION

We introduced the concept of *attack tactics* as system features that can be exploited out-of-context by an attacker and can cause undesired consequences in an Industrial Control System. We developed an information-flow-guided symbolic execution engine that processes standard industrial programming languages [34] and semi-automatically discovers attack tactics.

We tested our approach in a two-controller chemical dosing system in the presence of three attacker profiles. We found 72 successful attack tactics, where 56 (removing physics-inspired attacks) are novel attacks so far unknown by operators.

Using the attack tactics, we showed how to compose attack vectors like tank overflow/underflow, dead-headed pumps, dry-running pumps, component Denial-of-Service, and chemical contamination that can have a significant impact on the system operation. Finally, we showed that this approach can be adapted to other industries and can be adjusted to process PLC programs from multiple vendors. Our implementation is available as open-source software.

ACKNOWLEDGMENTS

This research is partially supported by the National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure" (Award Number: NSoE_DeST-SCI2019-0010). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

This material is also supported by Air Force Research Laboratory under agreement FA8750-19-2-0010, by Army Research Office under agreement W911NF-20-1-0253, and by NSF CNS-1931573.

REFERENCES

[1] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. 2017. ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Annual Computer Security Applications Conference (ACSAC).*
[2] Sridhar Adepu and Aditya Mathur. 2016. Distributed detection of single-stage multipoint cyber attacks in a water treatment plant. In *Asia Conference on Computer and Communications Security (AsiaCCS).*
[3] Chuadhry Mujeeb Ahmed, Jianying Zhou, and Aditya P Mathur. 2018. Noise matters: Using sensor and process noise fingerprint to detect stealthy cyber attacks and authenticate sensors in cps. In *Annual Computer Security Applications Conference (ACSAC).*
[4] Rajeev Alur. 2015. *Principles of cyber-physical systems.* MIT Press.
[5] Saurabh Amin, Xavier Litrico, Shankar Sastry, and Alexandre M Bayen. 2012. Cyber security of water SCADA systems—Part I: Analysis and experimentation of stealthy deception attacks. *IEEE Transactions on Control Systems Technology* (2012).
[6] Michael J Assante and Robert M Lee. 2015. The industrial control system cyber kill chain. *SANS Institute InfoSec Reading Room* (2015).
[7] Alvaro A Cárdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. 2011. Attacks against process control systems: risk assessment, detection, and response. In *ACM symposium on information, computer and communications security (AsiaCCS).*
[8] John H Castellanos, Martín Ochoa, and Jianying Zhou. 2018. Finding dependencies between cyber-physical domains for security testing of industrial control systems. In *Annual Computer Security Applications Conference (ACSAC).*
[9] Yuqi Chen, Christopher M Poskitt, and Jun Sun. 2018. Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system. In *IEEE Symposium on Security and Privacy (SP).*
[10] Yuqi Chen, Christopher M Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-guided network fuzzing for testing cyber-physical system defences. In *Automated Software Engineering (ASE).*
[11] Yuqi Chen, Bohan Xuan, Christopher M Poskitt, Jun Sun, and Fan Zhang. 2020. Active fuzzing for testing and securing cyber-physical systems. In *International Symposium on Software Testing and Analysis (ISSTA).*
[12] Anton Cherepanov. 2017. *WIN32/INDUSTROYER: A new threat for industrial control systems.* Technical Report.
[13] AC Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. 2018. TRITON: The first ICS cyber attack on safety instrument systems. Black Hat USA.
[14] Cheng Feng, Venkata Reddy Palleti, Aditya Mathur, and Deeph Chana. 2019. A Systematic Framework to Generate Invariants for Anomaly Detection in Industrial Control Systems.. In *Network and Distributed System Security Symposium (NDSS).*
[15] Wei Gao, Heng Liang, Jun Ma, Mei Han, Zhong-lin Chen, Zheng-shuang Han, and Gui-bai Li. 2011. Membrane fouling control in ultrafiltration technology for drinking water production: A review. *Desalination* (2011).
[16] Jairo Giraldo, Esha Sarkar, Alvaro A Cardenas, Michail Maniatakos, and Murat Kantarcioglu. 2017. Security and privacy in cyber-physical systems: A survey of surveys. *IEEE Design & Test* 34, 4 (2017), 7–17.
[17] Benjamin Green, Marina Krotofil, and Ali Abbasi. 2017. On the significance of process comprehension for conducting targeted ICS attacks. In *Workshop on Cyber-Physical Systems Security and PrivaCy (CPS-SPC).*
[18] Shengjian Guo, Meng Wu, and Chao Wang. 2017. Symbolic execution of programmable logic controller code. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE).*
[19] Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H Hartel. 2014. Through the eye of the PLC: semantic security monitoring for industrial processes. In *Annual Computer Security Applications Conference (ACSAC).*
[20] Thomas A Henzinger. 2000. The theory of hybrid automata. In *Verification of digital and hybrid systems.*
[21] Edward Ashford Lee and Sanjit A Seshia. 2016. *Introduction to embedded systems: A cyber-physical systems approach.* Mit Press.
[22] Qin Lin, Sridha Adepu, Sicco Verwer, and Aditya Mathur. 2018. TABOR: A graphical model-based approach for anomaly detection in industrial control systems. In *Asia Conference on Computer and Communications Security (AsiaCCS).*
[23] Stephen McLaughlin. 2013. CPS: Stateful policy enforcement for control system device usage. In *Annual Computer Security Applications Conference (ACSAC).*
[24] Stephen McLaughlin and Patrick McDaniel. 2012. SABOT: specification-based payload generation for programmable logic controllers. In *ACM conference on Computer and communications security (CCS).*
[25] Stephen E McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. 2014. A Trusted Safety Verifier for Process Controller Code.. In *Network and Distributed System Security Symposium (NDSS).*
[26] Eric Rothstein Morris, Carlos G Murguia, and Martín Ochoa. 2017. Design-time quantification of integrity in cyber-physical systems. In *Workshop on Programming Languages and Analysis for Security (PLAS).*
[27] Steven Muchnick et al. 1997. *Advanced compiler design implementation.* Morgan kaufmann.

[28] Rockwell Automation Publication. 2014. *Logix5000 Controllers General Instructions Reference Manual.* Technical Report.
[29] Rockwell Automation Publication. 2018. *Logix 5000 Controllers I/O and Tag Data.* Technical Report.
[30] Rockwell Automation Publication. 2019. *Logix 5000 Controllers Design Considerations.* Technical Report.
[31] Ricardo G Sanfelice. 2015. Analysis and design of cyber-physical systems: a hybrid control systems approach. *Cyber-Physical Systems* (2015).
[32] Esha Sarkar, Hadjer Benkraouda, and Michail Maniatakos. 2020. I came, I saw, I hacked: Automated Generation of Process-independent Attacks for Industrial Control Systems. In *Asia Conference on Computer and Communications Security (AsiaCCS).*
[33] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE symposium on Security and privacy (SP).*
[34] Michael Tiegelkamp and Karl-Heinz John. 2010. *IEC 61131-3: Programming industrial automation systems.* Springer.
[35] David I Urbina, Jairo A Giraldo, Alvaro A Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. 2016. Limiting the impact of stealthy attacks on industrial control systems. In *ACM SIGSAC Conference on Computer and Communications Security (CCS).*
[36] Eduard Kovacs Security week. 2019. Critical Vulnerabilities Found in WAGO Industrial Switches. https://www.securityweek.com/critical-vulnerabilities-found-wago-industrial-switches Available on February 2020.
[37] Qingyu Yang, Jie Yang, Wei Yu, Dou An, Nan Zhang, and Wei Zhao. 2013. On false data-injection attacks against power system state estimation: Modeling and countermeasures. *IEEE Transactions on Parallel and Distributed Systems* (2013).

APPENDIX

Execution order

The PLC program is divided into multiple routines, each routine can be programmed with a different programming language [30, p. 31]. The *entry point* of the program is the routine that should be executed first by the PLC; this entry point is called the 'main routine', and it is added by the engineers in the PLC's configuration file. Routines can do *inter-procedural calls* using the *JSR* function (Jump to SubRoutine), this is how engineers control the execution order of the program.

Different features of the program are implemented using separate routines. Routines can include updates from external signals, reading/writing of remote variables, safety checkers, etc.

To build a consistent CFG requires we follow the execution order, that is the reason our approach implements an algorithm that creates the CFG following the same order, instead of processing the program sequentially as it is in classical programming analysis [27].

Model of subsystem controlled by C2

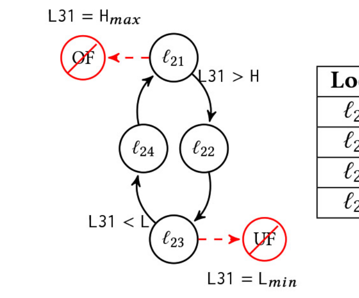
Symbolic Expressions

n-cycle analysis

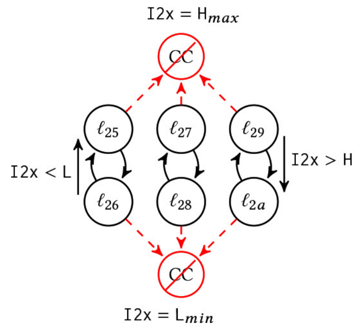
Effects of scan cycle on symbolic execution

The scan cycle, as described in the section 2.1, can be understood as a global infinite loop that covers the controller's program. Then we can have a 1-cycle analysis that is similar to classical symbolic execution mapping traces from the root block at the CFG, or we can extend this analysis to a *n*-cycle method (η). The latter aims to discover dependencies that require multiple cycles, like validation of variables in previous cycles.

The way how we extend the symbolic execution to a *n*-cycle method is described in Alg. 1. By taking the results of the 1-cycle method as new targets and repeat the process again, this method will produce symbolic expressions of previous scan cycles. We repeat this process until the results contain variables in $\mathbb{R}$ or variables converge to a fixed point.



Locs.	MV21	L31
ℓ_{21}	OFF	↑
ℓ_{22}	OFF	↑
ℓ_{23}	ON	↓
ℓ_{24}	ON	↓



Locs.	P2x	I2x
ℓ_{25}	ON	↑
ℓ_{26}	OFF	↓
ℓ_{27}	ON	↑
ℓ_{28}	OFF	↓
ℓ_{29}	ON	↑
ℓ_{2a}	OFF	↓

Figure 14: System model managed by C2. Left: Finite state machine with critical states in red. Right: Discrete and continuous variables in each location. ↑ means the level increases, ↓ means the level decreases.

Algorithm 1: *n*-scan-cycle search (η)

```
Data: A subset of internal variables J ∈ 𝕃, the set of external
      variables ℝ
Input: J
Result: Provide a set of conditional expressions for each
        element in J.
R:= {} ;                                        // Result set
X:= {} ;   // Queue to process elements in multiple scan-cycles
foreach j ∈ J do
    X:= {j};
    S:= {} ;                          // Secondary result set
    while X ≠ ∅ do
        x:= X.pop();
        Y:= GetSymbExpressions(x);
        foreach y ∈ Y do
            if y ∈ ℝ then
                S.push(y);
            else if y ∉ X ∧ y ≠ x then
                X.push(y);
    R.push(S);
return R
```

Target		Trace(π)	$\mathbb{R}$		$\mathbb{L}$	ρ
			Sensor	Network		
P11	ON	π_{145}	DI2 ∧ ¬DI4	C2 : MV21.ST = 2	RST.ON ∧ L11.HTY ∧ ¬L11.LL ∧ P11.C = 2 ∧ AUTO.OFF	8
		π_{158}	DI2 ∧ ¬DI4	C2 : MV21.ST = 2 ∧ C3 : L31.L	RST.ON ∧ L11.HTY ∧ ¬L11.LL ∧ AUTO.ON ∧ P.Sel = 1 ∧ P1.ST = 2	10
	OFF	π_{141}	DI2		P11.C = 1 ∧ AUTO.OFF	3
		$\pi_{146.1}$	DI2	C2 : MV21.ST ≠ 2	P11.C = 2 ∧ P11.ST = 2 ∧ AUTO.OFF ∧ P11.Start	6
		$\pi_{146.2}$	DI2		P11.C = 2 ∧ L11.HT ∧ L11.LL ∧ AUTO.OFF ∧ P11.Start	6
		$\pi_{146.3}$	DI2 ∧ DI4		P11.C = 2 ∧ AUTO.OFF ∧ P11.Start	5
		π_{149}*	DI2 ∧ ¬DI3*		P11.C = 2 ∧ AUTO.OFF ∧ P11.Start	4
		$\pi_{159.1}$	DI2 ∧ ¬DI4	C2 : MV21.ST ≠ 2	RST.ON ∧ AUTO.ON ∧ P.Sel = 1 ∧ P1.ST = 2 ∧ P11.Start	8
		$\pi_{159.2}$	DI2 ∧ ¬DI4		RST.ON ∧ L11.HTY ∧ L11.LL ∧ P1.ST = 2∧ P.Sel = 1 ∧ AUTO.ON ∧ P11.Start	9
		π_{161}*	DI2 ∧ ¬DI3*		P1.ST = 2 ∧ AUTO.ON ∧ P11.Start ∧ P.Sel = 1	6
		$\pi_{163.1}$	DI2 ∧ DI4		AUTO.ON	3
		$\pi_{163.2}$	DI2	C2 : MV21.ST ≠ 2	P11.ST = 2 ∧ AUTO.ON	4
		$\pi_{163.3}$*	DI2	C2 : F21.LL*	P11.ST = 2 ∧ AUTO.ON	4
		π_{168}	¬DI2			1
MV11	ON	π_{107}			RST.ON ∧ MV11.C = 2 ∧ AUTO.OFF	3
		π_{117}			RST.ON ∧ AUTO.ON ∧ P1.ST = 2 ∧ L11.L	4
	OFF	π_{101}			MV11.C = 1 ∧ AUTO.OFF	2
		$\pi_{105.1}$			MV11.C = 2 ∧ AUTO.OFF ∧ MV11.FO	3
		$\pi_{105.2}$			MV11.C = 2 ∧ AUTO.OFF ∧ MV11.FC	3
		$\pi_{120.1}$			AUTO.ON ∧ PTS = 121 ∧ L11.HH	3
		$\pi_{120.2}$*			AUTO.ON ∧ MV11.FC*	2
		$\pi_{120.3}$*			AUTO.ON ∧ MV11.FO*	2

Table 6: Symbolic attack expressions from C1's PLC code analysis. The effort index (ρ) shows the number of variables an attacker needs to control to launch a particular attack. * These attacks expressions have a time-constrained component associated.

	Variable	Timer condition		Attack condition		Attack
		Enable condition	τ (secs)	$\mathbb{R}$	$\mathbb{L}$	Complexity
C1	MV11.FC	MV11.Close	7	¬DI9	MV11.C = 1 ∧ AUTO.OFF	[1, 0, 2]
	MV11.FO	MV11.Open	6	¬DI8	MV11.C = 2 ∧ AUTO.OFF	[1, 0, 2]
	P11=OFF	P11.Start	1	DI2 ∧ ¬DI3	P11.C = 2 ∧ AUTO.OFF	[2, 0, 2]
		P11.Start	1	DI2 ∧ ¬DI3	P1.ST = 2 ∧ AUTO.ON ∧ P11.Start ∧ P.Sel = 1	[2, 0, 4]
		P11.ST = 2 ∧ F21.LL	10	DI2	P11.C = 2 ∧ AUTO.OFF ∧ P11.Start	[1, 1, 4]
		P11.ST = 2 ∧ F21.LL	10	DI2	AUTO.ON	[1, 1, 1]
C2	MV21.FC	MV21.Close	10	¬DI9	MV21.C = 1 ∧ AUTO.OFF	[1, 0, 2]
	MV21.FO	MV21.Open	9	¬DI26	MV21.C = 2 ∧ AUTO.OFF	[1, 0, 2]
	P21=OFF	P21.Start	0	DI2 ∧ ¬DI3	P21.C = 2 ∧ AUTO.OFF	[2, 0, 2]
		P21.Start	0	DI2 ∧ ¬DI3	P2.ST = 2 ∧ AUTO.ON ∧ P21.Start ∧ P.Sel = 1	[2, 0, 4]
		AI0 = 0.0 ∧ P21.ST = 2∧ MV21.ST = 2	3	DI2	AUTO.ON	[1, 1, 4]
	P23=OFF	P23.Start	4	DI8 ∧ ¬DI9	P23.C = 2 ∧ AUTO.OFF	[2, 0, 2]
		P23.Start	4	DI8 ∧ ¬DI9	P2.ST = 2 ∧ AUTO.ON ∧ P23.Start ∧ P.Sel = 1	[2, 0, 4]
		AI0 = 0.0 ∧ P23.ST = 2∧ MV21.ST = 2	3	DI8	AUTO.ON	[1, 1, 4]
	P25=OFF	P25.Start	8	DI14 ∧ ¬DI15	P25.C = 2 ∧ AUTO.OFF	[2, 0, 2]
		P25.Start	8	DI14 ∧ ¬DI15	P2.ST = 2 ∧ AUTO.ON ∧ P25.Start ∧ P.Sel = 1	[2, 0, 4]
		AI0 = 0.0 ∧ P25.ST = 2∧ MV21.ST = 2	3	DI14	AUTO.ON	[1, 1, 4]

Table 7: Time-constrained attacks

	$\mathbb{L}\,(i)$	$\mathbb{R}\,(e : \eta(i))$
C1	L11.L	AI0 < L
	¬L11.LL	AI0 > LL
	L11.LL	AI0 < LL
	L11.HH	AI0 > HH
	L11.HT	0 < AI0 < T_{max}
	MV11.C = 2	DI8
	MV11.FC*	¬DI9
	MV11.FO*	¬DI8
	P11.C = 2	DI3
C2	MV21.FC*	¬DI27
	MV21.FO*	¬DI26
	MV21.ST = 2	DI26

Table 8: n-cycle analysis. * These variables have a time-constrained component associated.

Detailed Results

Target	#	π	External variables ($\mathbb{R}$)									Internal variables ($\mathbb{L}$)												ρ	Results	
			C2:F21.LL	C2:MV21.ST	C3:L31.L	AI0	DI2	DI3	DI4	DI8	DI9	AUTO.OFF	AUTO.ON	L11.HT	L11.LL	MV11.C	P.Sel	P1.ST	P11.C	P11.ST	P11.Start	PTS	RST		A1	A2
MV11 = OFF	1	101										1				1								2	S	⊥
	2	120.2^{*+}									0		1											2	F	F
	3	120.3^{*+}								0*			1											2	S	C
	4	105.2^{*+}								1	0	1				+								3	F	F
	5	120.1^{+}				1000					0		1									121		4	S	C
MV11 = ON	6	107										1				2							1	3	S	⊥
	7	117^{+}				500							1					2					1	4	S	C
P11 = OFF	8	168					0																	1	S	S
	9	141					1					1							1					3	S	⊥
	10	163.1					1		1				1											3	S	C
	11	163.2		1			1					1								2				4	S	C
	12	163.3^{*}	1*				1						1							2*				4	S	C
	13	146.3^{+}					1	1	1			1							+		1			5	S	C
	14	149^{*}					1	0*				1							2		1			5	S	C
	15	146.2^{+}				200	1	1				1		+	+				+		1			5	S	C
	16	146.1^{+}		1			1	1				1							+	2	1			6	S	C
	17	161^{*}					1	0*					1				1	2			1			6	S	C
	18	159.1		1			1		0				1				1	2			1		1	8	S	C
	19	159.2^{+}				200	1		0				1	+	+		1	2			1		1	8	S	C
P11 = ON	20	145^{+}		2		700	1		0			1		+	+				2				1	7	S	⊥
	21	158^{+}		2	1	700	1		0				1	+	+		1	2					1	9	S	⊥

Table 9: Attack primitives for C1's targets *MV11*, and *P11*. They are realisations of the symbolic expressions in the table 6. * These variables have a time-constrained component associated. + Extended cases using $\mathbb{L} \xrightarrow{\eta} \mathbb{R}$ mapping from table 8. Shaded cells represent *sufficient conditions* in each successful (S) attack. Shaded $\mathbb{L}$ variables denote the variables that cannot be overwritten due to updates in each scan cycle.

Target	#	π	External variables ($\mathbb{R}$)														Internal variables ($\mathbb{L}$)									ρ	Results	
			C3:L31.H	C3:L31.L	AI0	AI1	AI2	AI3	DI2	DI3	DI4	DI26	DI27	DI28	AUTO.OFF	AUTO.ON	MV21.C	MV21.FC	MV21.FO	MV21.ST	P2.ST	P21.C	P21.ST	P21.Start	RST		A1	A2
MV21 = OFF	22	77													1		1									2	S	⊥
	23	96.2^{+*}											0			1		+								2	F	F
	24	96.3^{+*}										0*				1			+							2	S	S
	25	96.1	1													1					2					3	S	⊥
MV21 = ON	26	83													1		2								1	3	S	⊥
	27	93		1												1					2				1	4	S	⊥
P21 = OFF	28	208							0																	1	S	S
	29	181							1						1							1				3	S	⊥
	30	203.1							1		1					1										3	S	S
	31	203.4							1					1		1										3	S	S
	32	203.5				265			1							1										3	S	S
	33	203.2^{+}							1			0				1				+			2			4	S	C
	34	203.3^{*}			0.0				1							1				2*			2*			5	S	C
	35	186.3							1		1				1							2		1		5	S	C
	36	189^{*}							1	0*					1							2		1		5	S	C
	37	186.2							1					1	1							2		1		5	S	C
	38	201^{*}							1	0*						1					2			1		5	S	C
	39	186.1^{+}							1			0			1					+		2	2	1		6	S	C
	40	199.1^{+}							1		0	0				1				+	2			1	1	7	S	C
	41	199.2							1		0			1		1					2			1	1	7	S	C
P21 = ON	42	185			2.1				1		0			0	1					2		2			1	8	S	⊥
	43	198			2.1	245			1		0			0		1				2	2				1	9	S	⊥

Table 10: Attack vectors for controller C2 for actuators *MV21* and *P21*.