2023 IEEE Symposium on Security and Privacy (SP)

SCAPHY: Detecting Modern ICS Attacks by Correlating Behaviors in SCADA and PHYsical

Moses Ike[†‡], Kandy Phan[‡], Keaton Sadoski[‡], Romuald Valme[‡], Wenke Lee[†]
[†]Georgia Institute of Technology, [‡]Sandia National Laboratories
Email: {mike, kphan, dksados, rvalme}@sandia.gov, mosesjike@gatech.edu, wenke@cc.gatech.edu

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179411

***Abstract*—Modern Industrial Control Systems (ICS) attacks evade existing tools by using knowledge of ICS processes to blend their activities with benign Supervisory Control and Data Acquisition (SCADA) operation, causing physical world damages. We present SCAPHY to detect ICS attacks in SCADA by leveraging the unique *execution phases* of SCADA to identify the limited set of legitimate behaviors to control the physical world in different phases, which differentiates from attacker's activities. For example, it is typical for SCADA to setup ICS device objects during *initialization*, but anomalous during *process-control*. To extract unique behaviors of SCADA execution phases, SCAPHY first leverages open ICS conventions to generate a novel physical process dependency and impact graph (PDIG) to identify disruptive physical states. SCAPHY then uses PDIG to inform a *physical process-aware* dynamic analysis, whereby code paths of SCADA *process-control* execution is induced to reveal API call behaviors unique to legitimate *process-control* phases. Using this established behavior, SCAPHY selectively monitors attacker's physical world-targeted activities that violates legitimate *process-control* behaviors. We evaluated SCAPHY at a U.S. national lab ICS testbed environment. Using diverse ICS deployment scenarios and attacks across 4 ICS industries, SCAPHY achieved 95% accuracy & 3.5% false positives (FP), compared to 47.5% accuracy and 25% FP of existing work. We analyze SCAPHY's resilience to futuristic attacks where attacker knows our approach.**

I. INTRODUCTION

Unlike Information Technology (IT) attacks, Industrial Control System (ICS) attacks cause physical damages to life-dependent physical processes such as power and water supply. ICS processes are controlled by Supervisory Control and Data Acquisition (SCADA) hosts, which run special programs to control the physical world [1–3]. Modern ICS attacks [4–7] are launched from SCADA, where attackers utilize legitimate ICS resources to blend their activities with benign SCADA operations and send malicious signal to disrupt processes.

To detect ICS attacks, statistical analysis of ICS traffic [3, 8–15] are effective against *noisy* behaviors (e.g., network scans and malformed protocols), but are evaded by modern attacks which use legitimate protocols and knowledge of ICS parameters to cause targeted (not noisy) disruptions [1, 2, 5]. In addition, physical models monitor sensors to know when *observed* physical states deviate from *expected*, by fitting historical sensor data into *linear* models [16–18]. However, [19, 20] show that in practice, such models may require experts to build, and a detailed process model may be unavailable. Further, physical models trigger false alarms when deployed in production due to noise and configuration changes, such that benign states appear outside the model [17, 19]. In general, existing ICS tools are evaded by modern attacks and prone to false alarms due to analyzing traffic/sensor data in isolation, and therefore they cannot tie their analysis to attack-execution context in SCADA. Detecting ICS attacks in SCADA is hard because attackers use the same API call behavior as benign SCADA programs. For example, Industroyer malware, which shutdown Ukraine power grid [1, 5], performed malicious actions that are part of normal SCADA activity such as accessing ICS device objects. Similarly, 2021 Florida water poisoning attack [4] used normal Human Machine Interface (HMI) commands to dump toxic chemicals into the water supply. Hence, existing host agents that looks for *"non-SCADA"* APIs will not detect them.

We found that while these attack behaviors are normal SCADA activities, they are anomalous when performed in atypical *execution phases* in SCADA. Therefore, in this work, instead of treating SCADA as one monolithic execution, we *specialize* its behaviors in unique execution phases, which are *initialization* and *process-control*. We observe that for Industroyer attack to work, the attacker had to execute API calls that are *atypical* of process-control but needed for the attack. For example, to hijack ICS device handles, Industroyer executed Registry Setup APIs in process-control, which is typical for *initialization*, hence anomalous. After infecting SCADA, attackers must "setup/connect" their tools to attack the physical world. These attack behaviors do not align with SCADA's phase-based behavior and leads to atypical APIs in wrong phases. Hence if we identify the limited set of legitimate process-control behaviors, we can selectively monitor and detect attacker's physical-targeted activities that violate them.

Further, because SCADA responds to physical world changes, we can induce SCADA to reveal process-control behaviors by stimulating relevant changes. However, this requires a physical model of ICS processes and their elements' states. Interestingly, we found that we can leverage ICS open platform communications (OPC) conventions [21–23] to characterize processes via ICS element configurations. We can then *toggle* each element state to induce SCADA to exhibit its process-control behaviors, enabling us to identify them. Further, we can derive the *impact* of these state changes to identify *disruptive* process states. This can allow us to detect disruptive physical world effects caused by (state-changing) control signals.

We present SCAPHY, a new hybrid technique to detect ICS attacks by correlating SCADA execution phase-specific behaviors with physical world impact. SCAPHY identifies the limited set of API calls unique to each SCADA execution phase, which differentiates from attacker's activities in these phases, allowing SCAPHY to detect them. To characterize the "required" internal host *channels* SCADA uses to control the physical world, we introduce a new reference model, *SCADA*

DOI 10.1109/SP46215.2023.00021

Software Stack (S^3), which SCAPHY leverages to selectively monitor steps along attacker's physical world-bound activities in each execution phases. Through S^3, SCAPHY can detect attacks that *circumvent* proper S^3 layers (e.g., SCADA rootkits) but sends disruptive control signals to the physical world.

SCAPHY uses a physical model to identify disruptive control signals sent to the physical world. SCAPHY generates this model by leveraging OPC conventions (*tags* and *alarms*) to extract and map ICS elements to their processes based on a novel process dependency and impact graph (PDIG) model. PDIG enables SCAPHY to assign each element state an *impact coefficient* ($I_C(s)$) based on how they impact (decrease or increase) process outcomes. Further, SCAPHY leverages PDIG to help induce and extract legitimate process-control behaviors. To do this, SCAPHY performs a *physical process-aware* dynamic analysis, whereby a SCADA engine [24] is induced to execute process-control code paths by iteratively switching ICS element states connected to the process. During this, SCAPHY records executed API calls to establish a set of PHYSical world Impact Call Specialization (PHYSICS) constraints to identify *legitimate* process-control behaviors.

SCAPHY detects modern ICS attacks that are missed by existing tools. By correlating behaviors in SCADA and physical, SCAPHY provides contextual alerts for ICS operators to respond to attacks at both SCADA and physical plant. Via PHYSICS constraints, SCAPHY limits the operations an attacker can execute to disrupt a physical process by detecting attacker's anomalous API behavior in atypical execution phases. SCAPHY's physical model detects when control signals cause a physical process to have *inconsistent state* or driven outside its *setpoint* ranges. SCAPHY's use of open OPC convention makes it device agnostic and not based on any device/controller, making it work for any OPC-supported ICS deployment. We evaluated SCAPHY at a U.S. national lab state-of-the-art ICS testbed. We launched 40 attacks on 24 diverse ICS scenarios across 4 industries, including an open-source Texas Pan Handle power grid [25]. SCAPHY detected 95% of all attacks with only 3.5% false positives, including real world ICS malware attacks. We make the following contributions:

1) We propose a hybrid technique to detect ICS attacks by correlating SCADA behaviors with physical world effects.
2) We present an ICS physical model via OPC conventions to both identify disruptive physical states and extract legitimate process-control phase behaviors in SCADA.
3) We introduce a new reference model in ICS, *SCADA Software Stack* (S^3), to characterize internal software & hardware layers of SCADA operation. Through S^3, host agents can selectively monitor S^3 layers to detect attacks.
4) Using diverse ICS scenarios & attacks, SCAPHY achieved 95% accuracy and 3.5% false positives (FP), compared to 47.5% accuracy and 25% FP of existing work [3, 16].
5) Due to limited resources and datasets for diverse ICS security research [26–29], we make available diverse ICS experiment scenarios[1] and datasets derived from them, both at the physical sensor layer and from the SCADA components. These experiments can be run in the FactoryIO ICS Engine [24] and Siemens Step7-based development suite, WinSPS [30].

[1]https://github.com/lordmoses/SCAPHY

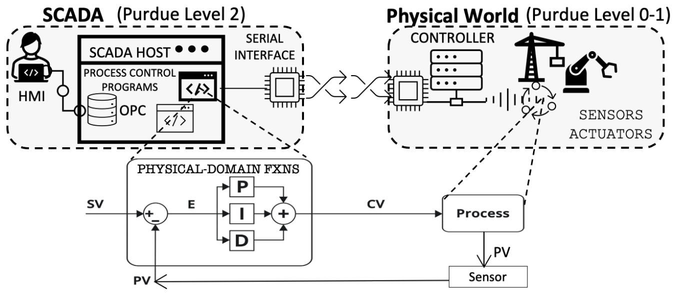


Fig. 1: Showing SCADA process-control operation: A physical-domain function compute a control variable *CV* to effect change on the physical world

II. BACKGROUND AND MOTIVATION

We present a background on ICS and use recent attacks as running examples to motivate our problem; 2021 Florida (FL) water poisoning attack [4], and 2016 Industroyer malware power grid attack [1]. In Section IV, we detail SCAPHY's approach in real world settings using the Industroyer example.

A. Real-World Motivating Examples

Florida Water Poisoning Attack. An attacker raised *dosing rate* of Sodium Hydroxide (NaOH) in FL water treatment plant to toxic levels, endangering citizens. NaOH is used to balance water PH but is toxic in high amounts. After gaining access to SCADA, the attacker started an HMI program to issue attack signals to disrupt the *level control* and *dosing* processes, increasing NaOH from normal 100 ppm to 11,100ppm [4, 31].
Industroyer Power Grid Attack. Industroyer shutdown a Ukranian power station by sending malicious signals from a SCADA host to a Siemens SPIROTEC device that runs circuit breaker Remote Terminal Units (RTUs). To perform the attack, Industroyer hijacked host serial COM ports, stole the breaker's *tag* from OPC to *address* its payload, and opened the breakers. This caused power imbalance that shutdown the station [1, 5].

B. ICS/SCADA Operations Background

Fig 1 describes ICS operation based on Purdue model [32]. SCADA hosts at Purdue Level 2 control physical processes, which run at Level 0 and 1 via ICS elements (comprising of actuators, sensors, and parameters in programmable logic controllers or PLC). SCADA constantly monitor running processes and when change is needed, they execute physical-domain logic such as *proportional integral derivative (PID)* to compute a control variable (*CV*) to modify element's states and effect the change (Fig 1). For example, the *level control* process in the FL water plant controls chemical level in a tank via a *PID* logic that control how much fluid enters and leaves the tank [29]. This is known as *process-control* and is the main function of SCADA. Unfortunately, ICS attacks are launched from SCADA due to direct access to the physical world as in Industroyer and FL attacks. Attacker gain access to SCADA via IT means such as phishing. Then he can leverage ICS resources such as OPC, HMIs, and SCADA channels to "address" and send malicious signals to ICS elements.

OPC is a functional part of SCADA and used for interoperability to exchange data about plant information in standardized convention [21–23]. OPC is used in widely deployed ICS platforms such as Siemens Tia Portal & Scheider OASYS. For

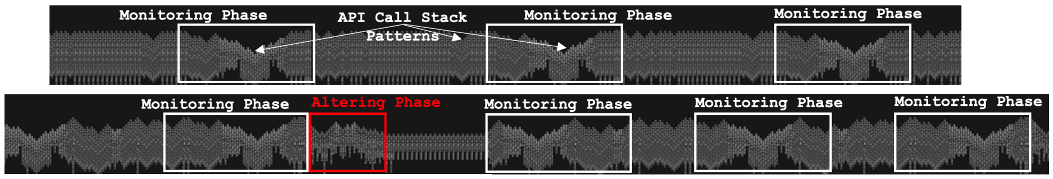


Fig. 2: Cycles of Process-Monitoring and Process-Altering phases based on the Step7-based WinSPS [30] control of Water Treatment Plant operation

example, Industroyer malware accessed the Ukrainian power grid's OPC server to extract the circuit breaker's OPC tag to address its attack payload. In our example, this OPC tag is *BRK.0.BOOL*, which specifies element *BRK* id=*0* with BOOL parameter — a heuristic about its possible states: 0 or 1 [33].

Limitations of Existing Work. Existing work to detect ICS attacks focus on analyzing traffic and/or sensor data in isolation, without SCADA execution context, which limits their ability to tie their analysis to SCADA for better accuracy and less false alarms. PLC techniques [34, 35] to detect malicious ladder logic on PLCs do not analyze SCADA control signals as in ladder logic but forwards signals to device. Lee [36] only monitors for host DLL injection, which may not happen in ICS attacks. Table I is a taxonomy of leading recent works in ICS attack detection, categorized by their technique and data point. We highlight their limitations such as not having ICS diversity in their evaluation and testing in-the-wild (adapted) attacks. We found that this is due to limited resources to emulate end-to-end and diverse ICS security research [26–29].

ICS Traffic approaches [8, 10, 11, 37] analyze abnormal traffic flows and protocols. However, modern attacks such as Industroyer and FL water plant attacks uses legitimate ICS protocols to emit traffic flows that blends with benign traffic. Traffic timing analysis [3, 9, 38] are effective for analyzing round trip time delays and inter-arrival times but are only effective against attack behaviors that are *chatty* [9] such as network scans, but not modern attacks which are targeted.

Physical Models detects deviation from expected physical behavior by training sensor data using linear and auto regressive models [16–18]. However, [19, 20] show that in practice, such models may require experts to build, and a detailed process model may be unavailable. Further they trigger false alarms in production due to noise and configuration changes such that benign states are outside the model. Reinforcement and Deep Learning [39–42] which uses game-theory to learn normal and attack behaviors, requires a high-interaction environment, known attacks, and expert reward function, which may limit its use in diverse ICS practice. Process-aware approaches [43–47] focus on specific physical functions, which reduces ambiguity in detection, but requires experts to specify process-specific violations (e.g., [43, 45] uses BRO rules for safety thresholds).

C. SCADA Host Attacks and Security Challenges

Due to physical ramifications of SCADA executions, existing host analysis techniques cannot be directly applied to secure SCADA hosts [5, 33]. Due to complex safety constraints in physical tasks, SCADA runs many proprietary and physical domain-specific programs such as ICS drivers and hardware tools. As such, unlike IT programs, using existing concolic analysis tools [48–50] will be intractable due to hardware-constrained code paths and environment need [48, 50]. Due to legacy and third-party components, code-signing cannot be strictly enforced to whitelist programs, enabling attackers to modify benign programs. For example, Industroyer executed custom APIs, and Stuxnet [6] injected into Siemens programs.

	ICS Traffic			PLC-Logic-			Physical Models										Has SCADA Context	
							Physical State			Reinforc-Learning			Process-Aware					
Technique	Yang [52]	Ihab [53]	Ponoma. [3]	Niang [54]	Mulder [34]	Formby [35]	Ghaeini [16]	Dina [18]	Aoudi [19]	Kurt [39]	Zhong [40]	Panfili [41]	Chromik [43]	Nivethan [45]	Remke [46]	Lin [44]	Lee [36]	SCAPHY
Traffic Analysis																		
Protocol fields	●		●															●
Time analysis	●	●	●															
PLC Logic																		
Logic execution								●	●									
Logic verification							●											
Physical State																		
State Deviation				●	●	●												
Physical Impact																		●
RL Models																		
Online POMDP										●								
Reward weights											●							
Multi-agent game											●	●						
Process-Aware																		
Predefined rules													●	●	●			
Power-flow specs															●	●		
Power prediction																●		
SCADA Context																		
DLL inject-based																	●	
PHYSICS constr.																		●
Evaluation																		
In-the-wild attacks							●											●
ICS diversity	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	4

TABLE I: Taxonomy of Recent Related works in ICS Attack detection

Our Insight. We found that the nature of physical tasks, which occur in cycles of repeated steps, requires SCADA to exhibit two distinct execution phases: *initialization* and *process-control*. Process-control comprise of *process-monitoring* and *process-altering* sub-phases. After infecting SCADA, attackers must "setup" and "connect" their tools to attack the physical world. These attack activities do not align with SCADA phase-specific operation, hence results in API calls executed in inappropriate SCADA phases. For example, to access target device tags, Industroyer made OPC calls in the process-monitoring phase, but OPC calls are used in *initialization*. To hijack SCADA physical channels (from benign program), Industroyer and Havex malware [51] created ICS device objects while in the process-altering phase, but that was a process-monitoring behavior, hence indicative of an attack. Therefore, SCAPHY deems API calls as "anomalous" when executed in atypical SCADA execution phase.

SCADA Execution Phases. SCADA starts with *initialization* which is performed once to setup environment such as loading ICS drivers. After *initialization*, process-monitoring starts. It involves updating process states in memory and operator HMIs. When physical change is needed, process-monitoring transitions to process-altering to perform the change and return until change is needed again. Process-altering invokes physical-domain logic on a *setpoint* variable *SV* to output a *CV* which is sent to the process (Fig 1). Fig 2 shows the unique

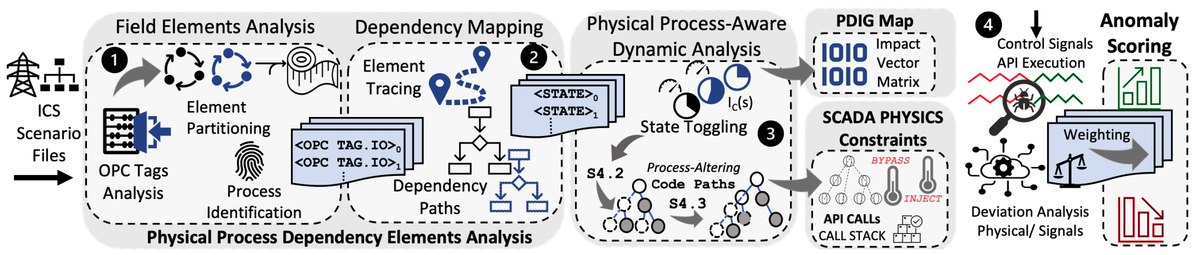


Fig. 3: SCAPHY Architecture: ICS scenario file is parsed. Extracted ICS elements are analyzed to identify processes and dependency elements. Element states are toggled to derive impact on process output over scan cycles, during which SCADA process-control PHYSICS constraints are identified for each process.

execution patterns of process monitoring and process-altering phases of a water treatment operation based on Siemens S7 WinSPS SCADA platform [30]. This pattern is based on API call stack behavior, which identifies these SCADA phases.

To identify behaviors unique to each process-control phase, SCAPHY first leverages domain-knowledge in OPC to build a physical model of ICS processes, which enables it to detect physical anomalies of process state changes such as when a process has *inconsistent state*. SCAPHY then uses the model to inform a *physical process-aware* dynamic analysis to induce and extract the limited set of legitimate API calls unique to each process-control phase behaviors. Through the API calls, SCAPHY establishes a PHYSical world Impact Call Specialization (PHYSICS) constraints, which attackers must violate (i.e, execute *atypical* APIs) to attack the physical world. Further, SCAPHY develops a new reference model, *SCADA Software Stack* (S^3) to characterize the "required" internal SCADA *channels* to access the physical based on the execution phases. For example, calling *ReadFile* indicates *process-monitoring* and *WriteFile* indicates *process-altering*. Both APIs access ICS device objects (e.g, COM Ports) in 3rd layer of S^3, to send signals to physical devices. Through S^3, SCAPHY can detect attacks that *circumvent or bypasses* required S^3 layers (e.g., SCADA rootkits) but sends disruptive signals to devices.

Resilience Against Evasion in SCADA. To attack the physical world, an attacker must communicate over SCADA's physical interface, through (only) which the physical can be accessed. SCAPHY monitors all network communication in the SCADA host via a hypervisor, which runs at a higher privilege than the SCADA system. SCAPHY enforces that all access to the physical world go through the expected physical world-bound API calls by monitoring accesses to the S^3 layers at runtime from the hypervisor using virtual machine introspection (VMI). If an attacker bypasses these expected APIs and sends signal to the physical world, SCAPHY will see that the emitted signal does not have the expected provenance (i.e., no matching API call occurred), and consequently detect the attacker's influence on the system.

III. THREAT MODEL AND ASSUMPTIONS

We assume a threat model similar to existing work for SCADA-originated ICS attacks [3, 9, 15, 16], whereby attacker has infected SCADA and can use available tools to send attack signals to devices. We developed SCAPHY for the Windows-based SCADA systems, which has unmatched dominance in ICS. We make the following practical assumptions: We do not consider attacks that do not originate from SCADA such as side-channel [55, 56]. Majority of *in-the-wild* ICS attacks are SCADA-originated [57]. Notional malware that *originates*/runs *only* on PLC such as in [58] are rarely seen in the wild due to attacker's cost of developing reusable malware for non-traditional CPUs [2, 59]. We note that PLC Man-In-The-Middle (MITM) has been addressed by existing work [60–62] and in practice via non-PLC diode gateways [63], and hence is outside the scope of this work. SCAPHY assumes the hypervisor as Trusted Computing Base (TCB) that cannot be compromised. In the hypervisor, SCAPHY sees all physical-bound API calls on SCADA virtual machine (VM). Further, SCAPHY relies on the Windows Kernel Patch Protection (KPP) to detect when rootkits inject into the ICS Device Stack (2nd layer of S^3) to MITM legitimate SCADA programs. KPP ensures that third-party kernel drivers (e.g., rootkits) cannot modify the kernel subsystem, which mediates hardware-access (i.e., rootkits cannot hide hardware-bound API calls). Relying on KPP is practical because it is widely used in Windows.

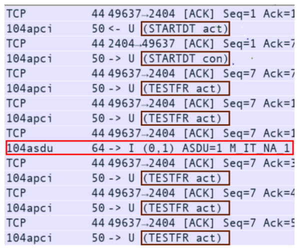
IV. SCAPHY APPROACH

Input and Output. SCAPHY's takes as input, an ICS scenario's OPC element data and function block diagram (FBD) and outputs (i) a physical model and (ii) PHYSICS constraints. The physical model assigns an *impact score* to each element state in a process, based on how the state impacts the process output. SCAPHY uses impact scores to detect signals that are disruptive to a process. PHYSICS constraints are a set of API calls of legitimate *process-control* phase behaviors, which differentiates from attacker's activities, such that he must *inject into* or *circumvent* them to attack processes, allowing SCAPHY to detect it as a violation of PHYSICS constraints.

Deployment. SCAPHY runs in hypervisors of SCADA VMs. Specifically, we deployed SCAPHY in Dom 0 of Xen hypervisor. This approach is practical because ICS plants are increasingly adopting virtualization to provide redundancy in SCADA such as in Enel power plants [64] and recent surveys [65, 66]. For example, Nozomi Networks, an ICS leader, provides several security solutions (e.g., Vantage and Guardian) that relies on virtualized end points [67–69]. In Xen, SCAPHY leverages LibVMI to trace API calls in SCADA VMs. At the network interface, SCAPHY monitor for control signals that changes element states. ICS protocols such as Modbus and DNP3 specifies function code for control signals (e.g., DNP3 $0x02$/Modbus $0x05$). Many traffic analyzers exist to identify function codes in ICS protocols [10, 11, 70].

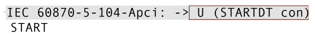
A. End-to-End Operation

SCAPHY works in four phases as shown in Fig 3. ❶ SCAPHY parses ICS scenario files to analyze and partition ICS elements



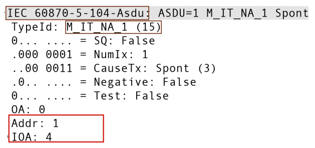
```
TCP        44 49637→2404 [ACK] Seq=1 Ack=1
104apci    50 <- U (STARTDT act)
TCP        44 2404→49637 [ACK] Seq=1 Ack=7
104apci    50 -> U (STARTDT con)
TCP        44 49637→2404 [ACK] Seq=7 Ack=7
104apci    50 -> U (TESTFR act)
TCP        44 49637→2404 [ACK] Seq=7 Ack=1
104apci    50 -> U (TESTFR act)
TCP        44 49637→2404 [ACK] Seq=7 Ack=1
104asdu    64 -> I (0,1) ASDU=1 M_IT_NA_1
TCP        44 49637→2404 [ACK] Seq=7 Ack=3
104apci    50 -> U (TESTFR act)
TCP        44 49637→2404 [ACK] Seq=7 Ack=4
104apci    50 -> U (TESTFR act)
TCP        44 49637→2404 [ACK] Seq=7 Ack=5
104apci    50 -> U (TESTFR act)
```

(a) Industroyer's IEC Attack Traffic



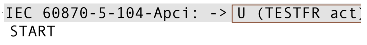
```
IEC 60870-5-104-Apci: -> U (STARTDT con)
 START
```

(b) IEC APCI Session START



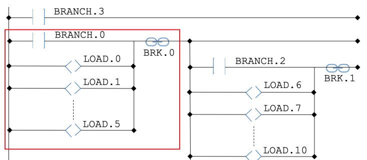
```
IEC 60870-5-104-Asdu: ASDU=1 M_IT_NA_1 Spont
 TypeId: M_IT_NA_1 (15)
 0... .... = SQ: False
 .000 0001 = NumIx: 1
 ..00 0011 = CauseTx: Spont (3)
 .0.. .... = Negative: False
 0... .... = Test: False
 OA: 0
 Addr: 1
 IOA: 4
```

(c) Industroyer Payload to Circuit Breaker IOA 4 OPC tag

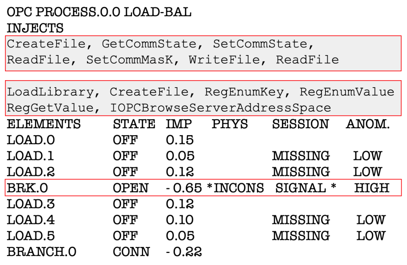


```
IEC 60870-5-104-Apci: -> U (TESTFR act)
 START
```

(d) IEC APCI TEST if device is alive



(e) Part FBD for Power Load Balancing



```
OPC PROCESS.0.0 LOAD-BAL
INJECTS
CreateFile, GetCommState, SetCommState,
ReadFile, SetCommMasK, WriteFile, ReadFile

LoadLibrary, CreateFile, RegEnumKey, RegEnumValue
RegGetValue, IOPCBrowseServerAddressSpace
ELEMENTS      STATE  IMP     PHYS      SESSION    ANOM.
LOAD.0        OFF    0.15
LOAD.1        OFF    0.05              MISSING    LOW
LOAD.2        OFF    0.12              MISSING    LOW
BRK.0         OPEN   - 0.65 *INCONS   SIGNAL *    HIGH
LOAD.3        OFF    0.12
LOAD.4        OFF    0.10              MISSING    LOW
LOAD.5        OFF    0.05              MISSING    LOW
BRANCH.0      CONN   - 0.22
```

(f) SCAPHY's Industroyer Detection Output

Fig. 4: Delineation of Industroyer's power grid attack signals based on the IEC 60870-5-104 ICS Protocol and SCAPHY's SCADA and physical anomaly output

into terminal and non-terminal sets based on heuristics from OPC. Terminal elements identifies processes. ❷ SCAPHY then traces each element's connections to map dependent element to their process. SCAPHY then loads the scenario in an ICS engine [24, 30] and performs a *physical process-aware* dynamic analysis ❸, whereby the engine is induced to execute code paths of *process-control* operations by iteratively switching element states. During this, SCAPHY records the API calls executed during process-altering and process-monitoring phases, to establish PHYSical world Impact Call Specialization (PHYSICS) constraints. Further, the change in process output caused by each state switching is averaged over several scan cycles to derive an impact score for each state relative to others. For states with oscillating impact (i.e., process output may increase or decrease), SCAPHY derives a *setpoint range*, which defines a minimum and maximum impact score for the process. ❹ SCAPHY raises alarms when executed APIs in each *process-control* phase are not in the PHYSICS constraints, when control signals causes *inconsistent state* or *outside setpoint* anomalies, and when *missing*, *extraneous*, and *out-of-order* signals are seen. SCAPHY then computes an anomaly score.

B. Detecting Industroyer Attack Behavior with SCAPHY

Industroyer shutdown Ukranian power grid by opening circuit breakers connected to load lines [1]. This attack disrupted load balancing of power demand & supply, a weakness in power systems, leading to outages [71]. To replicate the attack, we adapted an open-source Texas Pan Handle grid [25] setup in PowerWorld [72] at a U.S. National Lab. Our lab setup is detailed in Section VII. We ran Industroyer in its "intended" environment; a SCADA host with COM ports and OPC, connected to IEC 608070 device with simulated circuit breaker RTUs. SCAPHY raised detection alarm in under 9 seconds for a PHYSICS *injection* violation and an *inconsistent state* anomaly.

Industroyer PHYSICS Violation Industroyer made several *LoadLibrary* calls, although a normal SCADA API, was performed after the process-altering phase indicated by a prior *CreateFile(Write)* call. We found that *LoadLibray* is normal for process-monitoring and *initialization* but not process-altering, per the PHYSICS constraints established from the power scenario. We found that Industoryer used *LoadLibrary* to load *OPCClientDemo.dll*, to gain OPC capability. Thereafter, Industroyer transitioned to process-monitoring indicated by a *ReadFile* call. It then invoked *IOPCBrowseServerAddressSpace* OPC call to extract circuit breaker Information Object Address (IOA tag) to send its payload as shown in Fig 4c. OPC calls are typical of *initialization*, but not process-monitoring. In addition, Industroyer created new ICS device handles while already in the process-altering phase (to highjack COM Ports) which is malicious in process-altering.

Industroyer's Physical Anomalies The Industroyer attack generated 8 IEC 608070 signals as shown in Fig 4a; two IEC 60870 Application Protocol Control Information (ACPI) *START* frames to begin communication, two ACPI *TEST* frames to check controller status, and one Application Service Data Unit (ASDU) payload sent to the circuit breaker RTU. Industroyer also issued 3 last *TEST* signals to verify controller's post attack status as shown in Fig 4d. Based on the physical model mapping of the element IOA in the payload, SCAPHY identified the target process as *load balancing* (LB). LB's dependency elements are load lines *LOADS.0-5*, breaker *BRK.0*, and *Branch.0* as shown in Fig 4d. SCAPHY output (Fig 4f) show that Industroyer issued a control signal to *BRK.0* (i.e., indicated by SIGNAL*), with none for other elements (i.e., missing). However, *BRK.0*'s new open state has an opposing impact vector to load lines's OFF state per SCAPHY physical model, allowing SCAPHY to detect an *inconsistent state* anomaly (detailed in Section V-C). Using *BRK.0*'s impact vector of 65%, SCAPHY derived a high anomaly score.

V. SYSTEM DESIGN

The goal of SCAPHY is to (*I*) identify the limited set of legitmate API calls of SCADA process-control execution phases that differentiates from attacker activities, and (*II*) build a physical model of ICS processes to identity disruptive physical impact of control signals. To achieve (*I*), we leverage the process dependency mappings in (*II*) to inform a *physical process-aware* dynamic analysis. To achieve (*II*), we leverage ICS OPC domain knowledge and impact scores derived during (*I*) to build a process dependency and impact graph (PDIG).

A. Automated Physical Process Comprehension

OPC Tag Analysis. SCAPHY parse OPC *tags* of ICS scenarios to discover ICS elements and their parameters. OPC naming conventions allows SCAPHY to extract element's possible states, which allows SCAPHY to automatically *switch* element's states during impact modelling. For example, the extracted OPC tag in Industroyer example *BRK.0.BOOL=1* specifies element *BRK*, ID of 0, and boolean parameter, which tells SCAPHY to switch its state 1↪0, as opposed to an integer.

ICS Element Partitioning and Process Identification. A process comprises of elements that work together to achieve a physical goal specified by the plant. This goal is determined by the state of a *terminal* element (a heuristic SCAPHY uses to identify unique processes). For example, the Tank in the Level Control (LC) process of the FL attack example is the terminal element because the LC goal depends on the Tank's level. This level is monitored by a level *Meter* sensor. When Tank's level reaches the plant's specified SV for LC, the process concludes. OPC captures this information via OPC *Alarms&Event*, a data structure that specifies *monitored* process parameters [73, 74]. To identify terminal elements, SCAPHY analyzes *Alarms&Event* and extracts process parameter tuples, which corresponds to a sensor and terminal element pair, e.g., the Meter and Tank. SCAPHY then partitions ICS elements into two sets; terminal and non-terminal sets (E_{Term}, E_{NTerm}). SCAPHY represents a process in the form of $P_j = (S_j,\ E_{Term_j})$; where S_j is the process sensor that monitors the terminal element E_{Term_j}, which corresponds to process P_j's output goal. For example, SCAPHY represents LC process as *LC=(Meter,Tank)*. Other elements in the plant such as Pumps and Valves are in E_{NTerm}. Based on identifying all processes, SCAPHY partitions the set of elements E such that:

$$(E = E_{Term} \cup E_{NTerm}) \tag{1}$$

$$\{\forall i, j \in (E_{Term}, E_{NTerm}); i \neq j; i \cap j = \emptyset\} \tag{2}$$

$$|E_{Term}| := |P| \tag{3}$$

$|E_{Term}|$ equals the number of processes, $|P|$

Process Dependency Mapping. After identifying processes, SCAPHY traces element's connections to identify dependency elements of a process. To do this, SCAPHY converts the ICS scenario's function block diagram (FBD) into Statement Lists (STL), a textual representation of FBD logic. STL are network-like statements which connects elements via logic arithmetic. An example STL is shown in (Section A). FBD are automatically generated by SCADA programs. From each E_{Term} node in STL, SCAPHY traces its connection *paths* until all connected elements are identified. Each process (identified by its terminal element) now has a list of paths, *PATHS*, which contain their dependency paths, *DepPath*. *DepPath* is a set of E_{NTerm} nodes arranged in sequential order from the E_{Term} node identifying process p, and given as follows:

$$DepPath(p) := \{E_{NTerm_0}, ..., E_{NTerm_n}\} \tag{4}$$

$$PATHS(p) := \{DepPath_i(p), ..., DepPath_n(p)\} \tag{5}$$

SCAPHY then aggregates all elements in all *DepPath* of a process into a dependency element set $DEP(p)$ such that:

$$\{\forall i \in PATHS(p) : DEP(p) := \textstyle\bigcup_{j=0}^{n} i(p)\} \tag{6}$$

$DEP(P)$ is the union of all elements in the dependency paths of P's pathlist. SCAPHY keeps tracks of the internal ordering among these elements as developed from Equation 4. SCAPHY uses this *ordering* constraints to detect out-of-order signals.

Functional Inter-Process Relationships SCAPHY identifies functional relationship between processes with common elements among them. If an element is in E_{Term} of P_1 and in E_{NTerm} in P_2, then P_1 depends on P_2. SCAPHY models such element as *inter-process transfer points (PTP)*, and P_1 and P_2 as *PTP sink* and *PTP source* respectively. PTP instances is common among Boolean elements such as valves and switches. SCAPHY leverages PTP to detect attacks spanning multiple processes, such as in the FL water attack (detailed in Section VII-B). PTP events occur when a control signal causes a PTP element to change state, which causes the PTP *sink* to assume the value of the PTP *source*'s E_{Term}. Through this, SCAPHY detects disruptive impact on the PTP *sink* process stemming from the PTP *source*.

B. Modelling Process Dependency and Impact

SCAPHY models how each dependency element state of a process impacts (decrease or increase) the process output using a novel *process dependency and impact graph* (PDIG) model.

Sketching PDIG Gragh. PDIG is a set of nodes which represents elements, and edges which connects element nodes based on their relationship. Example PDIG is shown in Section A for the LC process. Undirected *process edges* connect E_{Term} elements to each E_{NTerm} element in the *same* $DEP(P)$. Undirected *element edges* connect E_{NTerm} elements without any ordering constraint. Directed *element edges* connect two E_{NTerm} elements with ordering constraints among them in the direction of the ordering. Each E node is annotated with its possible states and impact score (derived later). In the PDIG, SCAPHY identifies and annotates PTP instances when there is an undirected element edge from a terminal element (of the PTP *source*) to a non-terminal element (i.e., the PTP element).

Deriving Impact to Model Physical Effects. SCAPHY assigns each element state s an impact score $I_C(s)$ based on how they impact a process relative to other states. When a control signal changes an element state, $I_C(s)$ is used to compute the anomaly score to quantify impact. SCAPHY also uses $I_C(s)$ to prune out non-impactful states, which reduces the number of elements to monitor. Note that SCAPHY aims to derive "relative" impact score of an element state, not actual score, or score for state combinations. As such, SCAPHY does not consider other elements' state when deriving $I_C(s)$, which avoids combinatoral state explosion [10, 75, 76]. To make the scoring robust/fair, all score derivation starts from the same initial process configuration and is driven until the process reaches steady state. To derive $I_C(s)$, SCAPHY leverages an ICS environment; Siemens S7 WinSPS [30] and FactoryIO [24] to load and drive the ICS scenarios' processes. SCAPHY then iteratively switches each state in the process and analyze the moving average of process outputs via reading the process sensor element. SCAPHY stops evaluating the change in output when successive changes become negligible (less than 1%). We normalize $I_C(s)$ with respect to scan cycles ran, which bounds its value between 0.0 and 1.0. This succinctly describes the impact of each state relative to other states.

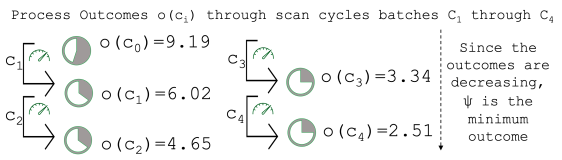
Formulation of $I_C(s)$. We define a process *outcome transition* set τ_n, which is a set of ordered outcomes o for a process from scan cycle c_j to c_n:

$$\tau_n := \{o(c_j), o(c_{j+1}), o(c_{j+2}), ..., o(c_n)\} \tag{7}$$

where c_j is the first scan cycle following SCAPHY switching of an elemetary state and c_n is the last or current scan cycle observed where $0 \leq j \leq n, n \in \mathbb{Z}$. SCAPHY keeps track of the highest or lowest recorded outcome ψ (i.e., the boundary outcome). For any scan cycle c_j, we define the $I_C(s)$ of the element state under analysis as follows:

$$I_C(s) = \frac{\sum_{i=j}^{n} \frac{o(c_i) - o(c_{i-1})}{ab(\psi - o(c_{i-1}))}}{|\tau_n|} \tag{8}$$

where $o(c_i) - o(c_{i-1})$ is change in process outcome, $ab(\psi -$



Process Outcomes o(c_i) through scan cycles batches C_1 through C_4
C_1
C_2
C_3
C_4
o(c_0)=9.19
o(c_1)=6.02
o(c_2)=4.65
o(c_3)=3.34
o(c_4)=2.51
Since the outcomes are decreasing, ψ is the minimum outcome

(a) Process operation scan cycles

Let the Calibrated range of P be 0 – 20.50cm, and P's outcome after cycle C_i be $O(c_i)$, Impact Coefficient $I_C(c_i)$ is as follows:

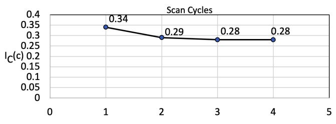
$$I_C(c_1)=\frac{\frac{(6.02-9.19)}{(9.19-0)}}{1} = -0.34$$

$$I_C(c_2)=\frac{\frac{(6.02-9.19)}{(9.19-0)}+\frac{(4.65-6.02)}{(6.02-0)}}{2} = -0.29$$

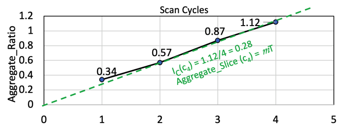
$$I_C(c_3)=\frac{\frac{(6.02-9.19)}{(9.19-0)}+\frac{(4.65-6.02)}{(6.02-0)}+\frac{(3.34-4.65)}{(4.65-0)}}{3} = -0.28$$

$$I_C(c_4)=\frac{\frac{(6.02-9.19)}{(9.19-0)}+\frac{(4.65-6.02)}{(6.02-0)}+\frac{(3.34-4.65)}{(4.65-0)}+\frac{(2.51-3.34)}{(3.34-0)}}{4} = -0.28$$

(b) Computation of Impact Coefficient



(c) A plot of $I_C(s)$ to scan cycles



(d) Plot of Aggregate Slices to scan cycles

Fig. 5: SCAPHY Impact Coefficient $I_C(s)$ Derivation for Each Element State

$o(c_i)$) is the absolute difference between the highest or lowest outcome and the preceding value at the scan cycle $i-1$. Further, $|\tau_n|$ is cardinality of the scan cycles from c_j to c_n.

We see that $\frac{o(c_i)-o(c_{i-1})}{ab(\psi-o(c_{i-1}))}$ is the ratio of the current process change (i.e., $o(c_i)-o(c_{i-1})$) to maximum change ($\psi-o(c_{i-1})$). If we aggregate this ratio for each scan cycle, we can compute $I_C(s)$ instantaneously at any scan cycles we chose without having to always compute $I_C(s)$ through all previous scan cycles. Using the aggregate ratio to compute the instantaneous derivation of $I_C(s)$ is given as follows:

$$I_C(s) = \frac{\delta}{\delta T} Aggregate\_Slice(o(c_n)) \tag{9}$$

where, for all scan cycles T, $Aggregate\_Slice(o(c_n))$ is sum of current change to Max change from c_j to c_n, defined as:

$$\{\forall c_i \in T : Aggregate\_Slice(o(c_n)) := \textstyle\sum_{i=j}^{n} \frac{o(c_i)-o(c_{i-1})}{\psi-o(c_{i-1})}\} \tag{10}$$

Fig 5b illustrates a derivation of $I_C(s)$ through scan cycles c_1 to c_4. At each scan cycle transition, the generated process outcomes, 9.19 through 2.15 were inputted into the $I_C(s)$ formula to compute the $I_C(s)$ scores. Notice that at scan cycle c_4, the difference in the subsequent $I_C(s)$ (i.e., from c_3) was negligible (i.e., 0). SCAPHY uses this observation as a heuristic to detect steady states. Otherwise, SCAPHY sets a maximum bound to stop the simulation. Fig 5d shows $I_C(s)$ for the end scan cycle c_n ($n=4$) using the iterative form and the instantaneous $I_C(s)$ form (green dotted line). The $Aggregate\_Slice(o(c_n))$ function is a straight line ($Aggregate\_Slice(c_n) = mT$) drawn from origin to the point $c_n \in T$, where m is the slope. Taking the derivative of $Aggregate\_Slice(c_n) = mT$ gives the instantaneous $I_C(s)$. Algorithm 1 shows SCAPHY algorithmic approach to derive to $I_C(s)$. $Aggregate\_Slice$ is the aggregate ratio of measured impact to maximum possible impact, and $Cycles$ is the total cycle batches in terms of scan cycles.

Algorithm 1 DeriveImpactCoefficient($I_C(s)$)

```
Input: ElementState $s$, Process $p$
Output: $I_C(s)$:                                         ▷ Initialization
  $Cycle, Cycle_{MAX} \leftarrow GetCycleBatchAndMax$
  $\psi \leftarrow GetOutcomeBoundCalib$
  $O_{PREV} \leftarrow GetProcessInitOutcome(p)$
  $SteadyState \leftarrow GetSteadyState(p)$
  $Aggregate\_Slice \leftarrow 0$
  while $Cycle_{MAX} > 0$ do
      $SDK\_RunSim(s, p)$
      $O_{c_i} \leftarrow GetProcessOutcome(p)$
      $O_{DIFF} \leftarrow O_{c_i} - O_{PREV}$
      $Aggregate\_Slice \leftarrow (Aggregate\_Slice + O_{DIFF})$
      $Aggregate\_Slice \leftarrow Aggregate\_Slice/ABS(O_{PREV}) - \psi)$
      $I_C(s) \leftarrow Aggregate\_Slice/Cycle$   ▷ check if steady state is reached, if so
return $I_C(s)$
      if $O_{DIFF} < SteadyState$ then
          Return $I_C(s)$
      else
          $Cycle++$
          $Cycle_{MAX}--$
          $\psi \leftarrow UpdateOutcomeBound(\psi, O_{c_i})$       ▷ update $\psi$ if neccesary
  Return $I_C(s)$
```

C. Characterizing Physical and Signal Anomalies

Inconsistent State. SCAPHY detects when a process has *inconsistent state* if two dependency element states have *opposing* impact *vectors*. To explain, recall that state's $I_C(s)$ in the PDIG drives process output in *one* direction towards its goal (i.e, not opposing goals). For example, Industroyer attacked *LB* process aims for a *LB* factor of 1.0 between power supply and demand. Although *LB* is affected by other factors, load lines and circuit breaker state play a role. If load decreases/disconnect (e.g., low demand), process-control responds by opening circuit breakers to bring back the balance. Hence, *load disconnecting* is *consistent* with *breaker opening* and drives the process towards its goal. However, *opening* circuit breakers and *connecting* load lines are inconsistent and disruptive to LB and should never occur in any legitimate setting. SCAPHY's $I_C(s)$ model captures this element state relationships and detects such *inconsistent state* physical anomalies. Through this, SCAPHY detected the Industroyer attack based on physical impact of the attack (Section IV).

Outside Setpoint. SCAPHY detects when control signals drive a process to exceed what it is *operationally* caliberated for based on learning the highest and lowest bounds of the process output recovered during the $I_C(s)$ derivation.

Signal Anomalies. SCAPHY detects (i) *missing signals* when a process' control traffic has incomplete signals based on number of its dependency elements. This is useful for targeted attack signals (ii) *Extraneous signals* is when control traffic contains signals for elements not associated with the process. (iii) *Out-of-Order signals* occur signals sequences in control traffic are not in the expected ordered flow based on PDIG.

Scoring Anomalies. SCAPHY computes anomaly scores based on element's $I_C(s)$. Let m be number of affected elements, and s_i is the state transitioned by the control signal. Let n be number of process dependency elements and $I_{C_{MAX}}(j)$ be the Max $I_C(s)$ for an element. Anomaly score is given by:

$$\{\forall j \in DEP(p) : Anomaly\_Score = \frac{\sum_{i=0}^{m} I_C(s_i)}{\sum_{j=0}^{n} I_{C_{MAX}(j)}}\} \tag{11}$$

Using $I_{C_{MAX}}$ in the denominator normalizes and bounds the scores between 0,1 which succinctly captures the deviation relative to other elements. We then calibrated a detection threshold boundary for low, medium, high anomalies as 0.0-

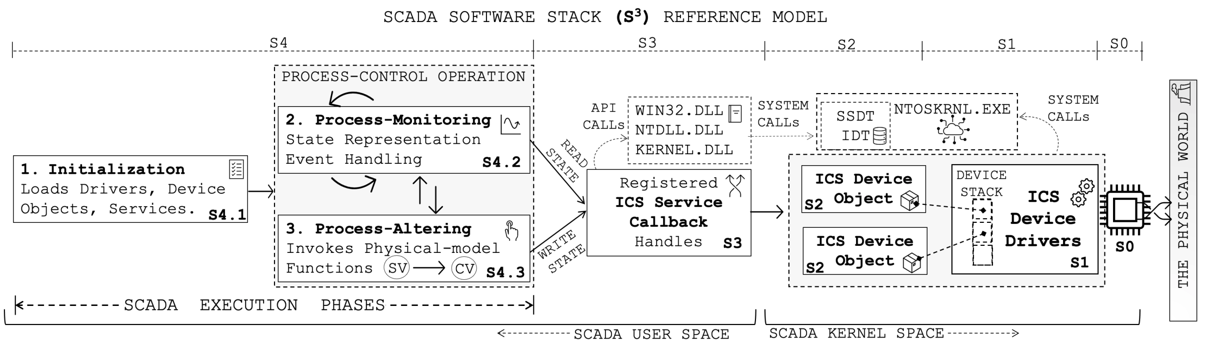


Fig. 6: SCADA Software Stack (S^3) and SCADA Execution Phases: Showing SCADA Host Interactions with S^3 Layers to access the Physical World

0.25, 0.26-0.60, and 0.61-1.0 respectfully. These gave the best accuracy for all scenarios tested. However, operators can fine tune these detection thresholds as needed.

D. Analyzing Physical World-Targeted Executions in SCADA

Given an ICS scenario, SCAPHY aims to generate the limited set of API calls unique to legitimate SCADA process-control operations, referred to as PHYSical world Impact Call Specialization (PHYSICS) constraints. To do this, SCAPHY leverages its physical model to inform a *physical process-aware* dynamic analysis, whereby a SCADA engine is induced to execute code paths of process-monitoring and altering behaviors. However, this requires first knowing each phase identifier and boundary.

Leveraging SCADA Software Stack (S^3) to Characterize Process-Control Behaviors. Through in-depth analysis of process-control in diverse ICS settings, we introduce a new reference model, *SCADA Software Stack* (S^3). S^3 does not replace Purdue Levels [32]. Rather, it breakdown Purdue Level 2 (i.e., control systems) into 5 layers to characterize the internal host layers involved in SCADA process-control, shown in Fig 6. We hope that via S^3, Antivirus companies can develop SCADA-specific host agents to monitor accesses to specific S^3 layers to detect attacks.

SCADA programs ($S4$) do not access ICS devices directly but do so using *device objects* ($S2$), which are software handles that enable the OS to mediate access to physical I/O ($S0$). To support diverse ICS devices, Windows provides a Driver Model (WDM) to allow device vendors to run ICS drivers ($S1$) in the kernel. In WDM, *driver objects* of an ICS driver represent instances of ICS devices the driver supports. For example, Windows supports 16550 UART devices via Windows Serial Driver, which allows SCADA programs to declare device objects, called COM ports, to communicate with devices.

ICS Callback Functions. To access devices, SCADA programs invoke ICS callback functions ($S3$) registered during ICS driver load. Callback functions invoke *CreateFile* which returns the device object handle as shown in Fig 6. Parameters *lpFileName* specifies device object name (e.g., COM1); *dwDesiredAccess* specifies *Read/Write* access mode; *dwShareMode* enables SCADA programs to deny other programs (e.g., malware) access to devices. Unfortunately, attackers (e.g., Havex, Industroyer, Oldrea) subvert this access control by killing SCADA processes to release their handles. For example, Industroyer killed *D2MultiCommService.exe* to hijack all COM ports to Siemens SPIROTEC device. Havex [51] scanned COM Ports to discover connected devices. After obtaining device handles, SCADA programs (and attackers) invoke *ReadFile* to read device states or *WriteFile* to send signals to them. Based on this behavior of S^3 layers, SCAPHY can monitor *CreateFile*, [*WriteFile* | *ReadFile*] on device objects as *identifiers* of process-altering and process-monitoring, respectively.

Identifying Process-Control Phase Windows. Existing work for web servers [77] rely on developer-supplied *boundaries* to identify phase transitions. This manual approach will not work if boundaries are not available such as in proprietary settings like ICS. However, SCAPHY leverages S^3 layers to analyze the *cyclical* nature of SCADA process-control to identify its transitions from process-monitoring to process-altering. Recall that accessing device objects ($S2$) using *CreateFile* in "Write" mode identifies process-altering, and in "Read" mode identifies process-monitoring, but we need to know when they begin and end (i.e., phase window) to specialize the extracted API calls. Based on their cyclical API call stack behavior such as shown in Fig 2, we found that process-altering *follows* process-monitoring. We also found that SCADA performs memory freeing operations thereafter to free up memory buffers used in physical-domain computations and then returns to monitoring.

We analyze process-monitoring to know its phase window. Process-Monitoring comprise of two sub-phases; *process state representation* (reads element states), and *event handling*, which checks if change is needed, otherwise it repeats as shown in Fig 2. SCAPHY analyzes the changing *EIP* register and call stack depth of the repeating loop to know the end of event handling the first time it returns to the *EIP* it started. When event handling ends but *EIP* returns to a different location and call stack depth, SCAPHY identifies this as the start of process-altering. Finally, SCAPHY performs a check to confirm the expected S^3 identifiers in each phase, which are [*CreateFile*, *ReadFile* || *WriteFile*], *CloseHandle* call sequences for process-monitoring & process-altering respectfully.

Physical Process-Aware Dynamic Analysis. SCAPHY leverages an ICS emulation engine (FactoryIO [24]) and a SCADA platform (Siemens S7 WinSPS [30]) to perform a *physical-process-aware* dynamic analysis of process-control behavior to generate PHYSICS constraints. FactoryIO (which also provides a SCADA tool) provides an environment to setup and drive physical processes. Its emulation engine supports hardware-in-the-loop PLCs using real protocols such as Modbus and allows loading of generic ICS scenario FBDs. On starting each process, we monitor the SCADA API executions to know when each process-control phase start based on the *phase windows*. For each process, we *induce* the SCADA execution to *re-*

		Physical World Dependency & Impact Model					PHYSICS Constraints					ICS program files		
ICS	Industry	physical		*OPC*	*PDIG*	I_C	behavior		verify		time	OPC points and tags		
scenarios	Domain	processes	ID	E_{Term}	nodes	avg/max	calls	stack	TP	FP	(min)	size	element	wires
power grid	Power Plant	load balancing	1.1	c-breaker	6	.71/.76	6	7	6	0	9.3	9K	19	35
		pwr distribution	1.2	load lines	4	.59/.66	4	6	4	0	10.4	9K	19	35
water treatment	Water Plant	level control	2.1	holding tank	4	.56/.86	10	12	10	0	8.2	11.5K	13	23
		dosing	2.2	dose valve	2	.66/.86	4	11	4	0	8.9	11.5K	4	23
auto warehouse	manufacture	pallet alignment	3.2	Axes X,Z	6	.47/.62	8	13	8	0	5.9	9K	10	11
		throughput	3.2	entry conveyor	2	.7/.84	4	11	3	1	5.5	9K	6	11
assembler	manufacture	product quality	4.1	clamp lid/base	2	.67/.77	4	7	4	0	7.5	9.5K	8	19
		load balancing	4.2	conveyor2	5	.8/.84	6	14	6	0	7.3	9.5K	11	19
palletizer	Shipping	load alignment	5.1	push clamp	3	.47/.71	7	13	7	0	5.9	7.8K	7	13
		prod protection	5.2	entry conveyor	6	.32/.69	4	17	4	0	5.7	7.8K	9	13
hvac system	manufacture	heat setpoint	6.1	room space	3	.46/.79	8	12	7	1	6.1	6K	8	14
		heat flow	6.2	vent	3	.6/.82	6	19	6	0	6	6K	7	14
converge station	Shipping	path throughput	7.1	load/unload	2	.47/.6	7	13	7	0	6.9	6.2K	9	15
		alt throughput	7.2	transfer	3	.67/.88	6	12	5	1	6.9	6.2K	9	15
production line	manufacture	alignment	8.1	control arm	2	.8/.8	6	15	5	1	8.6	8.5K	11	17
		throughput	8.2	conveyors	4	.6/.72	6	16	6	0	8.1	8.5K	11	17
sort station	Shipping	sort accuracy	9.1	unloader	2	.54/.85	4	13	4	0	5.7	9K	6	14
		throughput	9.2	conveyor	7	.5/.9	6	17	6	0	5	9K	16	14
separator	Shipping	accuracy	10.1	pusher1-2	6	.53/.72	5	12	4	1	5.9	4.9K	17	19
		throughput	10.2	conveyors	7	.69/.81	4	8	4	0	4.8	4.9K	15	19
elevator	manufacture	prod safety	11.1	conveyor1-3	5	.77/.83	6	13	6	0	10.2	11K	13	24
		throughput	11.2	entry conveyor	5	.33/.68	4	19	4	0	10	11K	12	24
queue processor	manufacture	spacing	12.1	buffer conveyor	6	.63/.71	6	13	6	0	6.7	9K	17	13
		throughput	12.2	entry conveyor	2	.67/.8	5	11	4	1	6.5	9K	14	13

TABLE II: ICS Scenarios: SCAPHY's physical world modelling results and generated SCADA PHYSICS constraints with Diverse ICS Industry Applications

compute the process control variable (*CV*) (i.e., to send to the physical) by iteratively switching each element state in the process. This drives execution down the process-monitoring and process-altering code paths to effect change on the process, enabling SCAPHY to record the API calls of each phase.

Process-Aware PHYSICS Constraints. SCAPHY's physical model makes generated PHYSICS constraints process-aware, i.e., captures process functions. Although many code paths exist in SCADA, our PHYSICS constraints cover *only* process-control paths by inducing the benign SCADA process-control logic to react to each element state change, ensuring that relevant "state-changing" logic paths are dynamically covered. Because element states are derived from the plant's deployed OPC, SCAPHY's PHYSICS constraints succinctly represent the limited legitimate APIs to control the physical.

PHYSICS Violations by Injection. SCAPHY detects anomalous APIs not in PHYSICS constraints as *injection* violations. As such, SCAPHY can detect attack code injected into SCADA programs (as done by Stuxnet [6]) and redirected API calls via Import Address Table *hooking*. Stuxnet-type attacks will evade existing tools that *whitelists* benign SCADA programs. However, because SCAPHY focuses on executed APIs, not the *executor*, injected APIs will be detected.

PHYSICS Violations by Bypass. Rootkits can bypass S^3 layers and directly send malicious signals to physical I/O using kernel calls such as *0x6b DeviceIOControl*. However, because SCAPHY sees all *WRITE* traffic on the physical interface ($S0$), it detects the attack as *Bypass* violation because no $S2$ activity was seen, allowing SCAPHY to know that a kernel-space entity bypassed proper process-altering S^3 channels to send signals.

VI. IMPLEMENTATION

We leveraged domain knowledge in OPC convention [21–23] to automatically extract and analyze ICS process information from input ICS scenarios files (OPC element data and FBD).

Accessing ICS Scenario Files in plants. We developed an OPC client [78] to perform $ReadRequest$ on OPC server (port 48031) using OPC UA protocol, which specifies a *nodesToRead* field [79] to return elements and parameter in JSON. Further, we read OPC Alarms&Event data from the $UaServer\_Event$ structure [73, 74] also in JSON. Lastly, we exported FBD's STL using SCADA software's provided API.

Process Identification and Element Tracing. We parse the OPC Alarm&Event data using python to extract the monitored (i.e., terminal) element and the sensor element. The terminal element is a fitting heuristic to identify processes because it represents a process goal. Next, to identify the OPC tags in the STL and trace element's connection, we first parsed the STL and matched its tags to tags from OPC. Then, we traced the STL statements from the terminal element tag to other (non-terminal) elements. Next, our python script "partitions" terminal and non-terminal elements in two sets.

Tracing SCADA APIs with LibVMI. Our testbed SCADA runs in a Windows Dom U VM in Xen Hypervisor, installed in a bare-metal intel machine, with a Linux Dom 0 VM, where SCAPHY runs. In Dom 0, SCAPHY's monitoring tool is implemented in C++ (77 lines) which calls LibVMI's $altp2m$ module to initialize VM introspection on our SCADA VM via $create\_view$ method. To trace SCADA execution, our C++ code invokes LibVMI's *SETUP_INTERRUPT_EVENT* to "trap" and forward executed APIs to a SCAPHY's analysis tool (232 lines of python) via a Linux Pipe. We run SCAPHY as follows: *scaphy_monitor* | *scaphy_analysis.py*

VII. EVALUATION

We evaluate SCAPHY's ability to (i) detect a variety of ICS attacks across diverse ICS scenarios, and (ii) outperform existing tools in detection accuracy. We launched 40 ICS attacks on 24 diverse ICS scenarios across 4 ICS industries to show SCAPHY versatility, as shown in Table II. SCAPHY detected 95% of all attacks, including real ICS malware (Havex[2] and Industroyer[3]), with only 3.5% false positives. Due to lack of resources to support diverse ICS security research [26–29], we make available over 200GB of new ICS experiments and

[2]7f249736efc0c31c44e96fb72c1efcc028857ac7

[3]2cb8230281b86fa944d3043ae906016c8b5984d9

	MITRE ICS Attack ID	Attack Description	In-the-wild ICS Reference	MITRE ICS TTP	Physical Process IDs (in Table II))	PHYSICS bypass	PHYSICS inject	Physical Anom. incons	Physical Anom. setpoint	Signal Anom. miss	Signal Anom. extran	Signal Anom. ooo	Metrics TP	Metrics FP	detect time(s)
Process Altering	T872	wipe host/registers	Killdisk	Evasion	1.1, 1.2, 4.1	✓		✓			✓		3	0	9.9
	T836	modify parameter	Stuxnet	Impair proc contrl	4.1, 4.2, 8.1, 8.2		✓		✓		✓		3	0	9.0
	T831	contrl manipulation	Stuxnet	Impact control	2.1, 2.2, 6.2		✓			✓		✓	3	1	10.2
	T889	kernel driver attack	Blaster	Modify Program fxn	5.1, 5.2, 10.2	✓			✓				2	0	9.8
	T855	unauthorised cmd msg	Industroyer	Impair proc contrl	3.1, 3.2, 11.1		✓	✓			✓		3	1	7.8
	M1.2	corrupt registers	Triton	Impair proc ctl	7.1, 10.2, 11.2		✓	✓			✓	✓	4	0	8.8
Non-Process Altering	T874	library hooking	Triton	Execution	5.1, 5.2, 10.1, 10.2		✓						1	0	5.4
	T801	monitor proc state	Industroyer	Collection	6.1,6.2, 8.1		✓						1	0	4.1
	T861	points/tags identifica.	Backdoor.Oldrea	Collection (OPC)	6.1, 6.2, 8.2		✓						1	0	4.1
	T816	device shutdown	Industroyer	Inhibit Resp fxn	7.1, 9.1, 9.2, 7.2		✓						1	0	5.4
	T888	network Enumeration	Havex(as is)	Discovery fxn	4.1, 4.2, 8.1, 8.2		✓						1	0	4.8
	T805	block serial COM	Industroyer(as is)	Inhibit Resp fxn	1.1, 1.2, 9.1		✓						1	0	5.1

TABLE III: Deployed Attacks and Detection Metrics. We leverage the MITRE ICS Attack Framework [80] to categorize the attack TTPs

attacks in both SCADA and physical aspects, developed for FactoryIO Engine [24] and Siemens S7 platform [30].

Experimental Setup. We leveraged a U.S. national lab testbed, which supports fast deployment of ICS topologies, OPC, HMIs, and SCADA VMs, prepared with ICS tools to control physical processes such as UART interfaces and Windows Serial Driver. We used 3 SCADA platforms: S7 WinSPS, MyScada [81], and FactoryIO SDK. This makes our testbed suited to evaluate SCAPHY against ICS attacks in realistic settings. To support diverse processes, we leveraged Simulink [82], PowerWorld [72], and FactoryIO Engine to emulate physical processes in Remote Terminal Units (RTUs).

ICS Attacks Performed. We performed diverse modern ICS attacks from 4 categories: attacks that (**I**) maliciously alter element states in running processes, (**II**) blocks SCADA access to the physical, (**III**) collects attack-relevant data from SCADA, and (**IV**) exploit bugs in ICS devices. We leveraged Mitre ICS Attack Framework and ICSSploit [83] to develop realistic attacks, indicated in "In-the-wild" column of Table III and Table IV, each tailored against their pertinent ICS target.

Physical World Model and PHYSICS Constraints. Table II shows our ICS scenarios details and SCAPHY's derived physical world models and PHYSICS constraints. To verify accuracy of generated PHYSICS constraints (API calls), we produced forensic execution traces of our SCADA program using the Time-Travel debugging feature of Windows Debugger (WinDBG), which we manually stepped to see the APIs of each process-control path. Table II column 8-11 shows the number of unique *process-altering* APIs called on average per path. As shown, we found that only few unique APIs were seen (most times in loops) depicting that *process-altering* is very specialized per physical domain, but the high stack depth shows that *process-altering* is executed deep in SCADA logic. We found that the FPs were due to rare element states not parsed correctly from OPC. SCAPHY's physical model allowed it to prune non-impactful elements from the original pool extracted from OPC, which is efficient. E.g., many ICS elements such as repeaters do not have any impact on process output, hence were pruned off during $I_C(s)$ derivation. As such, we saw over 50% reduction (on avg) from extracted OPC elements to PDIG nodes. This outperforms naively analyzing all states regardless of impact as done in [16].

Further, the reduced element pool contributed to an $I_C(s)$ (impact) average above 70% of their process output, showing that SCAPHY modelled the relevant or *impactful* elements whose malicious state change are more disruptive to the process. SCAPHY physical-process-aware dynamic analysis takes about 8 mins, which is reasonable per deployed scenarios sizes (last 3 columns). Our ICS scenarios are adapted from real-world models developed. For example, our power grid scenario was adapted from an open-source Texas Pan Handle power grid [25], simulated in *PowerWorld* RTUs. We use the power grid example to explain Table II: SCAPHY accurately identified the process terminal element (E_{Term}) as shown. SCAPHY's impact analysis pruned the OPC element pool of 19 nodes to 10 PDIG nodes, which is efficient. The average impact of all PDG nodes was over 50% with Max at 0.76 and 0.66, meaning that attack involving them are most disruptive.

A. ICS Attack Detection

Table III shows attack categories **I**, **II**, & **III**, and SCAPHY's results. Attack category I are shown in the first row-group, "Process Altering". Category II & III are shown in the second-row group, Non-Process Altering. SCAPHY detected PHYSICS bypass and inject attacks for both categories. However, SCAPHY detected physical and signal anomalies for only category I, because category II & III do not send control signals but block SCADA access to the physical or collect data about devices. For example, in $T805$, Industroyer issued *CreateFile* on all *COM* ports to block our SCADA program from accessing the physical ("as is" mean we executed In-the-wild malware). Similarly, Havex in $T888$ enumerated all *COM* device objects stored in Registry keys *HKLMSYSTEMCurrentControlSetServices* (registry value *SERVICE_KERNEL_DRIVER* are device driver services) to identify connected devices by issuing loops of *OueryKey*, *OpenKey*, *QueryValueKey* API calls. These API calls were atypical of process-monitoring, allowing SCAPHY to detect their PHYSICS inject violations.

SCAPHY detected two PHYSICS bypass violations in category I, $T872$ & $T889$, which are kernel attacks that bypassed SCAPHY monitored S^3 layers. Specifically, $T889$ & $T872$ used *DeviceIOControl* direct driver call to send signals out the serial I/O. SCAPHY detected the attack because they sent out signals without accessing S^3 layers, allowing SCAPHY to know that a kernel entity bypassed proper *process-altering* S^3 channel to attack the physical. Further, $T872$ used the same call to send control code to the storage device driver to delete whole drives (/DosDevices/C:). The Last 3 columns show results. Ground truth was derived from open-source attack data [80, 84]. We found that FPs were due to missed APIs during PHYSICS analysis due to rare element states not parsed from OPC.

Evaluating SCADA Software Stack (S^3) Activity. For attack category **IV**, we demonstrate S^3's ability to *pinpoint* steps along SCADA access to the physical, which shows that S^3 layers are practical host *monitoring taps* for ICS attack detection. To do this, SCAPHY tracked access to each S^3 layer based on the API call identifiers for each layer. E.g., *ReadFile* accesses

ICSSPLOIT Attack ID	Attack Description	Real-world Device Targets	In-the-Wild CVEs/ICSAs	Exploit Type	*SCADA Software Stack* (S^3) Activity							*Metrics*		
					$S4.1$	$S4.2$	$S4.3$	$S3$	$S2$	$S1$	$S0$	TP	FP	FN
SPLOIT1.1	stop controller/CPU	Siemens Simatic-1200	ICSA-11-186-01	Unprotected Port	-	-	✓	✓	✓	✓	✓	5	0	2
SPLOIT1.2	remote code execution	QNX SDP 660	CVE-2006-062	Buffer Overflow	-	✓	✓	✓	✓	✓	✓	4	0	1
SPLOIT1.3	remote device halt	Schneider Quantum	ICSA-13-077-01	I/O corruption	-	✓	✓	✓	✓	✓	✓	5	0	1
SPLOIT1.4	crash RTOS service	QNX INETDd	CVE-2013-2687	Buffer Overflow	-	✓	✓	✓	✓	✓	✓	5	0	1
SPLOIT1.5	RPC device crash	WindRiver VXWorks	CVE-2015-7599	Integer Overflow	-	✓	✓	✓	✓	✓	✓	4	0	1
SPLOIT1.6	denial of service	Siemens S7-300/400	CVE-2016-9158	Input Validation	-	-	✓	✓	✓	✓	✓	4	0	2

TABLE IV: Deployed Attacks and Detection Metrics. We leverage Attack Modules ICSSPloit Attack Modules [83] to categorize the attack TTPs

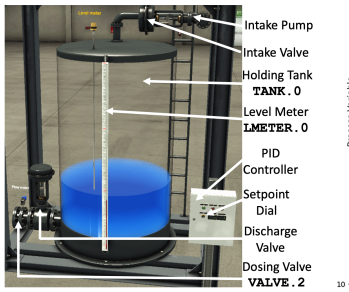
$S2$, the ICS device object. We leveraged ICSSPLOIT [83] to compile real-world exploits that were developed as proof-of-concepts exploit code against real bugs in ICS devices. We launched these attacks against their simulated ICS targets and then check which S^3 component was accessed in the SCADA host. Table IV details the results. SCAPHY detected all five layers of S^3 in all attacks as shown. However, in layer $S4$, SCAPHY did not detect *initialization* behavior ($S4.1$). This is because the exploits where self-contained and did not issue any API to setup environment. However, SCAPHY detected their *process-altering* behavior ($S4.3$) when the *WriteFile* API was called to send signal to the physical. We hope that via S^3, Antivirus companies can develop SCADA-specific host agents to correlate accesses to specific S^3 layers to detect attacks.

B. Case Study: 2021 Florida Water Plant Poisoning Attack

We replicated the FL incident with realistic water treatment scenario using FactoryIO and open-source data [26, 27]. The attacker targeted the chemical dosing operation by manipulating a *proportional* P parameter to raise toxic levels of NaOH in the water outside the setpoint (*SV*) [4]. Chemical dosing involves two processes: *Level control* (LC) and *Dosing*. The HMI is shown in Fig 7a. LC aims to fill a holding tank, *TANK.O* with chemical based on *SV*, after which Dosing will open a valve, *VALVE.2* to let chemical into the water supply [26, 27]. LC is controlled by a physical domain logic, *Proportional Integral Derivative* (*PID*). Because *SV* cannot be reached in one shot, PID operation involves several "intake" and "discharge" cycles, (shown in Fig 7d) whereby an intake pump fills chemical into *TANK.O*, and a discharge valve remove accesses. P controls how aggressive the intake and discharge cycles are driven. E.g, a high P pumps an *initial* excessive volume into *TANK.O*. Fig 7c shows how different P values affect how *SV* is reached. *SV* is set via a hardware dial on the PID controller, so attacker cannot modify it via cyber.

Attack and Detection. The attacker issued 2 control signals to raise P (dumps excess chemical into *TANK.O*) and open *VALVE.2*. We launched the attack using a Modbus payload, but made the attack code self-contained, without triggering any PHYSICS violation. At the SCADA side, the attack triggered all S^3 layers to send out signal, which was not malicious by itself. We now focus on the physical aspects. SCAPHY's detected a high *outside set point* physical anomaly and a low *Extraneous* signal anomaly. Fig 7b shows SCAPHY's outputs.

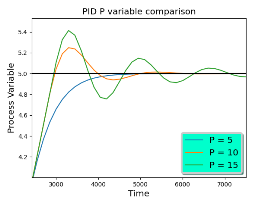
Explanation SCAPHY detection is based on the functional process relationship captured in the physical model between LC and Dosing. Recall Section V-A, *VALVE.2* is a PTP element between LC (PTP source) and Dosing (PTP sink). When *VALVE.2*'s state changes from *CLOSE* to *OPEN*, the Dosing process output *assumes TANK.O*'s value (i.e., the E_{Term} of the PTP Source) which is measured by the meter sensor *LMETER.0*. This Dosing process outcome (*LMETER.0*'s value)



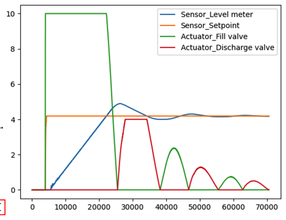
(a) Chemical Dosing Operation HMI

```
OPC PROCESS.0.0 LEVEL CTR
OPC PROCESS.0.1 DOSING

ELEMENTS   STATE     IMP   PHY  SESSION    ANOMA.
PUMP.0     ON        0.15       MISSING    LOW
VALVE.0    OPEN      0.15       MISSING    LOW
VALVE.1    CLOSE     - 0.2      MISSING    LOW
PVAR.0     VOLT 10   0.63       SIGNAL *   LOW
LMETER.0   VOLT 31   -          -          -
VALVE.2    OPEN     0.87  *OUT SETP  EXTRAN* HIGH
```

(b) SCAPHY Output



(c) Effects of different P's



(d) Intake/discharge cycles

was outside the setpoint for Dosing derived during $I_C(s)$ derivation, which allow SCAPHY to detect it. Finally, a high anomaly score is calculated based on *VALVE.2*'s $I_C(s)$ of 0.87, allowing SCAPHY to detect an *outside setpoint* anomaly. Further, SCAPHY detects an *extraneous* signal anomaly because the signal to open *VALVE.2* was "extraneous" in LC.

C. Case Study: Reliability of SCAPHY against Modern Rootkits that knows SCAPHY Approach

To test SCAPHY's reliability against rootkits, we selected two modern rootkit techniques based on recent works [85–87]: (i) direct kernel object manipulation (DKOM), which tampers kernel objects, and (ii) Hypervisor-level DKOM. We show that SCAPHY can be reliable against DKOM rootkits, but not hypervisor-level DKOM. We leveraged Metasploit [84] to develop these rootkits. We note that SCAPHY only applies to rootkits that *attack the physical world from SCADA* (i.e., SCADA rootkits). That is, SCAPHY is not aimed to detect traditional IT-only rootkits (e.g., exploit attacks or backdoors). SCAPHY's ability to detect *evasion* in SCADA is because any SCADA entity must ultimately send out traffic via the physical interface to access the physical. To do that legitimately, they must traverse proper SCADA's S^3 channels (monitored by SCAPHY). If a rootkit naively *circumvents* these channels, but sends signals via the physical interface, SCAPHY detects the evasion because no S^3 API calls was seen, which means a kernel entity (e.g., rootkits) bypassed proper S^3 channels. This was the case in Section VII-A, where rootkits $T872$ & $T889$ used kernel call *DeviceIOControl* to send signals, without S^3 activities. Hence the goal of a SCADA rootkit in evading SCAPHY is to deceptively "present" SCAPHY with proper S^3 API behavior, while sending commands to the physical.

DKOM Rootkit. This rootkit installs as a kernel driver and uses DKOM to obtain (and execute on) the userspace

ICS device object ($S2$ layer). Unlike naive rootkits $T872$ & $T889$, this rootkit exploits *ZwCreateFile* native API which handles parameters differently in kernel than the userspace *CreateFile* [88]. Calling *ZwCreateFile* with the driver name returns the driver base object address (a kernel object) which is traversed to locate *ICS device objects* in driver data structure. With this object, the rootkit (i) inserts itself on the ICS Device Stack of the physical interface, from which it can send signals, and (ii) invokes *ZwWriteFile* to complete fake S^3 behavior.

Detection. Executing the rootkit emitted $S2$ *process-altering*

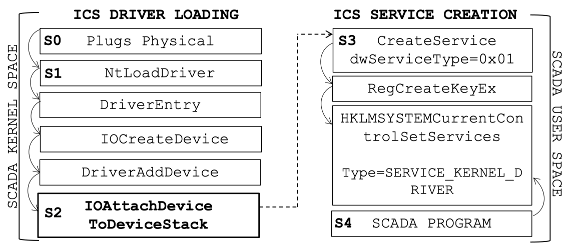


Fig. 8: API sequence for ICS Driver Loading and Registration.

APIs with no PHYSICS violations based on *ZwCreateFile* & *ZwWriteFile* calls. This evades SCAPHY. However, SCAPHY can detect this rootkit by simply monitoring for ICS driver attachment to the Device Stack, a protected kernel object.

Explanation For the attack to work, the rootkit had to *attach* itself on the *ICS Device Stack* of the physical interface. This is done using *IoAttachDeviceToDeviceStack* during driver loading as shown in Fig 8, which shows the steps ICS drivers (and rootkit) must follow to attach themselves to a Device Stack. Device Stack is a kernel structure that tells the kernel which drivers can access I/O of a physical device. Hence SCAPHY can track when new drivers are added to a Device Stack, allowing it to catch when *additional or unknown* drivers are introduced.

We note that rootkits cannot modify the kernel subsystem (*NTOSKRNL.EXE*) per *Windows Kernel Patch Protection (KPP)* [89], which prevents third-party code (e.g., drivers) from "patching" the kernel subsystem, which includes Device Stack. KPP prevents unauthorized access of device I/O. Hence ICS drivers (and rootkits) must follow the API steps in Fig 8, which SCAPHY can monitor with high-fidelity. To test this, we added this API behavior to SCAPHY model and ran the rootkit. On observing a repeat of *IoAttachDeviceToDeviceStack* (i.e., *after* the benign ICS driver has already been attached), SCAPHY detects this malicious Device Stack tampering. Making this addition to SCAPHY was seemless because the API sequence in Fig 8 *follows* SCAPHY's proposed S^3 layers, which shows the ubiquitousness of the S^3 reference model. For example, the argument *dwServiceType*=$0x01$ in *CreateService* registers an ICS driver service, which returns a handle used by ICS callback functions ($S3$) to access device objects ($S2$).

Hypervisor-Level DKOM. This second attack requires the rootkit to *escape* the VM and launch DKOM in Xen hypervisor (Dom 0 Linux) where SCAPHY runs. Although this is outside our threat model, as the hypervisor is our TCB (Section III), we wanted to show a concrete way to evade SCAPHY. Recall in Section VI that SCAPHY's LibVMI-based monitoring tool uses a Linux Pipe to feed API traces to SCAPHY analysis tool. This DKOM works by redirecting this Pipe's memory buffer in kernel to another pipe the rootkit controls. Specifically, our

Techniques	Approach	Attacks/	**Attack Detection Metrics**								
			SCADA			Physical			CTRL Signals		
		Normal	TP	FP	FN	TP	FP	FN	TP	FP	FN
Sensor [16] Analysis	Linear Regressive M.	40/146	-	-	-	19	37	21	-	-	-
Traffic [3] Analysis	Decision Tree Classifier	40/146	-	-	-	-	-	-	11	18	29
Hybrid SCAPHY	SCADA Corr. with Physical	40/146	36	5	4	21	14	19	18	21	22

TABLE V: Comparison with Existing Techniques

rootkit modified the Page Pointer, *PAGE* *, the first argument of *pipe_buffer*, a struct member of *pipe_inode_info* kernel object, which manages Pipe operations in Linux kernel. Since Linux Pipes are mounted in file-system "*pipe:/*" it was easy to find SCAPHY's pipe via *ls*. Then, the rootkit modified SCAPHY's Pipe's *PAGE* * in kernel to a malicious Pipe buffer address created with *dcom_write* | *dcom_read*, which allowed attacker to control what the SCAPHY analysis sees, without SCAPHY knowing. In our attack, we killed the LibVMI side of SCAPHY after the pipe manipulation for the attack to work.

D. *Comparing* SCAPHY *with Existing Techniques*

We compared SCAPHY to existing ICS techniques that use physical models [16] and traffic classifier [3]. [16] is based on [17] and analyzes sensor data's cumulative sum of residuals, and raises alarm if the difference between sensor and expected behavior is higher than a threshold. [3] analyzes spatial-temporal properties of ICS signals such as packet arrival times and size, using a REBTree classifier. [3] raises alarm when traffic features are *outside* a running average. To do this comparison, we use sensor and traffic data from the experiments in Table II, which we parsed into *@.ARFF* format (sample shown in Section A), and make available via this work We leveraged open-source tools to setup these techniques. For [16], we leveraged Scikit-Learn to generate a linear regressive model to fit the sensors values of the physical elements in the normal running mode. For [3], we leveraged WEKA [90] to generates a REPTree classifier that builds a decision tree using information gain and variance in the extracted traffic fields.

To launch attacks, we follow the format in Table III, which produced 40 attacks and 146 normal instances. Table V shows the results. Existing tools did not detect any SCADA attacks due to no SCADA context. However, this is where SCAPHY detected most attacks (90%, and 95% overall). We note that diverse ICS experiments (such as ours) may affect the performance of tools designed for specific ICS domains. [16] detected 19 attacks (47.5%), with a high FP of 37 (25%). Its FP is due to flagging high sensor deviations that are part of benign behaviors. One instance is the FL water attack where a high P variable causes a high but *temporary* sensor deviation. Although it deviates greatly from the *setpoint*, it is benign in Level Control (LC), but anomalous in the Dosing process if dosing valve is open. Unlike SCAPHY's physical model, existing linear models [16–19] do not capture inter-process operations (e.g., between Dosing and LC), hence are prone to false alarms due to high but temporary benign deviations. SCAPHY's physical model lower TP is due to approximating analog states such as switching 0-10v with only 3 levels 0v,5v,10v, which saves space but is less precise. [3] detected 11 attacks (27.5%), but with low FP (12.3%) because most modern attack is similar to benign. SCAPHY's signal-

SCAPHY Pipeline Requirement	Where Used	Manual Step	Avg. time	Times repeated
Exporting FBD to STL JSON	Element Tracing	Several Clicks in SCADA GUI	under 1 minutes	1 per scenario
Matching STL to OPC tags	Element Tracing	Seeding a Regex script with match parameter	under 10 minutes	1 per scenario
Setting up SCADA Modelling ENV	Impact Derivation	Configuring testbed networking & devices	under 1 hour	1 per scenario

TABLE VI: Quantifying amount of manual work to apply SCAPHY in practice

based detection had more FPs because it flagged many benign missing signals, showing that missing signals are not effective, which we can mitigate by raising its detection threshold.

E. *Quantifying Manual Work to use SCAPHY in Practice*

We quantify the amount of manual work that may be required to apply SCAPHY's pipeline in practice. For example, although our SCADA platform provides interfaces to automatically export FBDs to STL, other platforms may require manual export via the GUI. In addition, automatically matching and tracing OPC tags in the STL require SCADA and OPC to use the same namespace (e.g., "*BRK*" in OPC is also "*BRK*" in STL). This was the case in our testbed and in integrated platforms such as Siemen's TIA. However, in plants using third-party tools such that STL and OPC have different namespace (e.g., "BRK" v. "BR"), an STL-to-OPC element matching can be achieved semi-automatically by manually seeding a regular expression script to perform the matching. Table VI summaries the time to perform each step and repetitions needed.

VIII. DISCUSSIONS: SCAPHY'S PRACTICALITY

Runtime Overhead. SCAPHY leverages LibVMI state-of-the-art VM analysis tool in Xen Baremetal hypervisor, which enables detection of malicious APIs in SCADA VMs around 9 seconds, as shown in Section VII-A. LibVMI achieves near-native speed access of guest VM memory pages when run on Baremetal Hypervisors, and now used in production [91–96]. This makes SCAPHY a practical detection technique. In contrast, existing work based on sensor data [16, 17], must observe several sensor deviations before making a *sound* decision, which pushes their *time-to-detect* to the minute range [16, 17].

Real vs. Emulated State Switching. As switching real processes can be dangerous, we used emulated processes. However, the switched states were informed by deployed OPC server in the plant. Real-world gaps exist if poor emulation is used. To reduce this practicality gap, SCAPHY leveraged state-of-the-art Simulink and PowerWorld ICS emulation.

ICS Attack Dev. Difficulty. Unlike IT, developing ICS attacks is hard due to finding their SCADA and physical target. As such, we spent months developing many modern attack scenarios, and tested more attacks than existing work.

Robustness of PHYSICS Constraints. PHYSICS constraints are generated per ICS scenario and SCADA. This is practical because plants rarely update physical domain logic, since they are based on immutable laws of physics.

Limitations and Future Work. SCAPHY cannot detect attacks that originate outside of SCADA such as side channels and device hardware. We will investigate similar execution phase-based analysis these ICS threat models. Further, although SCAPHY can rely on Window's KPP to detect rootkits evasion, with time new rootkits may bypass KPP. We will investigate SCADA-specific defenses for rootkit such as integrating Call Stack analysis. This shows promise given that SCADA *process-control* have well-defined Call Stack behavior (Fig 2) which can be robust (but expensive) than API calls.

IX. RELATED WORKS

What differentiates ICS from IT is that physical tasks follow immutable laws of physics [16, 17], which can be learned to build prediction models [97]. Existing work build sensor behavior models [16–19] to predict when behaviors deviate from expected. However, in practice, such models are not always available [19, 20], and raise false alarms due to noise and config changes [19]. Offensive ways such as Harvey [58] can MITM sensors and present "false" data. [60–62] proposed ways to address MITM. [63] uses non-PLC diode gateways to avoid MITM. Reinforcement and Deep Learning [39–42] uses game-theory to learn normal and attack behaviors, but requires a high-interaction system, known attacks, and expert reward function, which may limit its use in diverse ICS practice.

Statistical analysis of ICS traffic [3, 8–15] are effective for *noisy* and abnormal traffic such as illegal protocols and scans. However, modern attacks evade them using benign protocols and knowledge of parameters to cause specific (not noisy) attacks [1, 2, 5]. Flow-based approaches [8, 10, 11, 37] analyze abnormal function codes/channels, such as shown in [98, 99], but are evaded by attacks such as Industroyer, which uses legitimate HMIs. Timing analysis [3, 9, 38] analyze anomalous round trip time delays and inter-arrival times. However, they are effective for signals that are *chatty* [9] such as attacker scans, not modern attacks that are targeted. Lee [36] only monitors for host DLL injection, which may not happen in ICS attacks. In contrast, SCAPHY deems API calls as anomalous when executed in the wrong SCADA execution phase. Side channel defenses such as EMF and power [55, 56] may require close proximity (motherboard-level) to controllers.

Pattern-based analysis [10, 76, 100] detects anomalous signals such as isolated signals but suffers from model's high sensitivity, which causes false alarms due to slight config changes (e.g., addition of new devices) [10, 76]. State-based tools [76] detects *critical states* in ICS but requires manual rules, which do not scale. Further, because they analyze all state transitions, can suffer from state explosion when parameters increase. SCAPHY is stateless and only analyze current and *impactful* element states. Process-aware tools [18, 19, 43–47] analyze sensor data unique to specific process functions, which may reduce ambiguity in detection, but requires experts to specify safety threshold violations per ICS-domain. This results in a tradeoff of being manual or domain specific. For example, [43, 45] uses manual BRO rules (e.g., heat level must not exceed 20). [44] uses expert-software to predict power flows. [46] uses pre-defined power flow rules to analyze deviations in measurements. In contrast, SCAPHY's process-aware physical model automatically detects physical anomalies by analyzing the disruptive "impact" of control signals. Further, unlike the above techniques, SCAPHY correlates physical "impact" with behaviors in SCADA (i.e., anomalous API calls in atypical execution phase) for contextual "end-to-end" attack detection.

X. Conclusion

We present SCAPHY to detect ICS attacks by leveraging unique *execution phases* of SCADA to identify the limited set of benign behaviors to control the physical world in different phases, which differentiates from attacker's activities. To do this, SCAPHY first leverages OPC to build a physical model, which enables it to detect physical anomalies. SCAPHY then uses this model to inform a *physical process-aware* dynamic analysis, whereby SCADA is induced to reveal API calls unique to *process-control*. Through this, SCAPHY detects attack behaviors that violates *process-control* phase. SCAPHY achieved high accuracy and outperformed existing work.

XI. Acknowledgement

We thank the anonymous reviewers for their helpful and informative feedback. We also thank Marshall Daniels for his guidance. This material was supported in part by Sandia National Laboratories, the Office of Naval Research (ONR) under grants N00014-19-1-2179, N00014-17-1-2895, and N00014-18-1-2662, the Defense Advanced Research Projects Agency (DARPA) under contracts HR00112090031 and HR00112190087. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy National Nuclear Security Administration under contract DE-NA0003525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] *WIN32/INDUSTROYER: A new threat for industrial control systems*. URL: https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf.

[2] Ruimin Sun et al. "SoK: Attacks on Industrial Control Logic and Formal Verification-Based Defenses". In: *arXiv preprint arXiv:2006.04806* (2020).

[3] Stanislav Ponomarev. *Intrusion Detection System of industrial control networks using network telemetry*. Louisiana Tech University, 2015.

[4] *Oldsmar Water Treatment Facility Cyber Attack*. URL: https://www.dragos.com/blog/industry-news/recommendations-following-the-oldsmar-water-treatment-facility-cyber-attack/.

[5] Defense Use Case. "Analysis of the cyber attack on the Ukrainian power grid". In: *Electricity Information Sharing and Analysis Center (E-ISAC)* (2016).

[6] *W32.Stuxnet Dossier*. URL: https://www.wired.com/images_blogs/threatlevel/2011/02/Symantec-Stuxnet-Update-Feb-2011.pdf.

[7] *Ransomware Attack leads to shutdown of Major U.S. Pipeline System*. URL: https://www.washingtonpost.com/business/2021/05/08/cyber-attack-colonial-pipeline/.

[8] Jeong-Han Yun et al. "Burst-based anomaly detection on the DNP3 protocol". In: *International Journal of Control and Automation* 6.2 (2013), pp. 313–324.

[9] Celine Irvene et al. "If I Knew Then What I Know Now: On Reevaluating DNP3 Security using Power Substation Traffic". In: *Proceedings of the Fifth Annual Industrial Control System Security (ICSS) Workshop*. 2019, pp. 48–59.

[10] Niv Goldenberg and Avishai Wool. "Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems". In: *international journal of critical infrastructure protection* 6.2 (2013), pp. 63–75.

[11] James Halvorsen and Julian L Rrushi. "Target Discovery Differentials for 0-Knowledge Detection of ICS Malware". In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing*. IEEE. 2017, pp. 542–549.

[12] Basem Al-Madani, Ahmad Shawahna, and Mohammad Qureshi. "Anomaly detection for industrial control networks using machine learning with the help from the inter-arrival curves". In: *arXiv preprint arXiv:1911.05692* (2019).

[13] Leandros A Maglaras and Jianmin Jiang. "Intrusion detection in SCADA systems using machine learning techniques". In: *2014 Science and Information Conference*. IEEE. 2014, pp. 626–631.

[14] Barnaby Stewart et al. "A novel intrusion detection mechanism for scada systems which automatically adapts to network topology changes". In: *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems* 4.10 (2017).

[15] Leandros A Maglaras, Jianmin Jiang, and Tiago Cruz. "Integrated OCSVM mechanism for intrusion detection in SCADA systems". In: *Electronics Letters* 50.25 (2014), pp. 1935–1936.

[16] Hamid Reza Ghaeini et al. "State-aware anomaly detection for industrial control systems". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1620–1628.

[17] David I Urbina et al. "Limiting the impact of stealthy attacks on industrial control systems". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 1092–1105.

[18] Dina Hadziosmanovic et al. "Through the eye of the PLC: towards semantic security monitoring for industrial control systems". In: *Proc. ACSAC*. Vol. 14. Citeseer. 2014, p. 22.

[19] Wissam Aoudi, Mikel Iturbe, and Magnus Almgren. "Truth will out: Departure-based process-level detection of stealthy attacks on control systems". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 817–831.

[20] Istvan Kiss, Bela Genge, and Piroska Haller. "A clustering-based approach to detect cyber attacks in process control systems". In: *2015 IEEE 13th international conference on industrial informatics (INDIN)*. IEEE. 2015, pp. 142–148.

[21] *OPC Foundation. OPC Unified Architecture*. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[22] Mai Son and Myeong-Jae Yi. "A study on OPC specifications: Perspective and challenges". In: *International Forum on Strategic Technology 2010*. IEEE. 2010, pp. 193–197.

[23] Wolfgang Mahnke. "2.4 OPC Unified Architecture". In: *Collaborative Process Automation Systems* (2010), p. 86.

[24] *Next Gen PLC Training*. URL: https://factoryio.com.

[25] *ACTIVSg2000: 2000-bus synthetic grid on footprint of Texas*. URL: https://electricgrids.engr.tamu.edu/electric-grid-test-cases/activsg2000/.

[26] Jonathan Goh et al. "A dataset to support research in the design of secure water treatment systems". In: *International conference on critical information infrastructures security*. Springer. 2016, pp. 88–99.

[27] *Secure Water Treatment (SWaT)*. URL: https://itrust.sutd.edu.sg/itrust-labs-home/itrust-labs_swat/.

[28] Thomas Morris and Wei Gao. "Industrial control system traffic data sets for intrusion detection research". In: *International Conference on Critical Infrastructure Protection*. Springer. 2014, pp. 65–78.

[29] Thomas Morris et al. "A control system testbed to validate critical infrastructure protection concepts". In: *International*

Journal of Critical Infrastructure Protection 4.2 (2011), pp. 88–103.

[30] *WinSPS-S7 Programming AND simulation tool for Siemens S7-300-PLCs.* URL: https://www.mhj-tools.com/?page=winsps-s7.

[31] *Dangerous Stuff: Hackers Tried to Poison Water Supply of Florida Town.* URL: https://www.nytimes.com/2021/02/08/us/oldsmar-florida-water-supply-hack.html.

[32] Rockwell Automation. *Converged Plantwide Ethernet (CPwE) Design and Implementation Guide.* 2011.

[33] Benjamin Green, Marina Krotofil, and Ali Abbasi. "On the significance of process comprehension for conducting targeted ICS attacks". In: *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy*. 2017, pp. 57–67.

[34] John Mulder et al. "WeaselBoard: zero-day exploit detection for programmable logic controllers". In: *Sandia report SAND2013-8274, Sandia national laboratories* (2013).

[35] David Formby and Raheem Beyah. "Temporal execution behavior for host anomaly detection in programmable logic controllers". In: *IEEE Transactions on Information Forensics and Security* 15 (2019), pp. 1455–1469.

[36] Jae-Myeong Lee and Sugwon Hong. "Keeping host sanity for security of the SCADA systems". In: *IEEE Access* 8 (2020), pp. 62954–62968.

[37] Chetna Singh, Ashwin Nivangune, and Mrinal Patwardhan. "Function code based vulnerability analysis of DNP3". In: *2016 IEEE International Conference on Advanced Networks and Telecommunications Systems*. IEEE. 2016, pp. 1–6.

[38] Ihab Darwish and Tarek Saadawi. "Attack Detection and Mitigation Techniques in Industrial Control System-Smart Grid DNP3". In: *2018 1st International Conference on Data Intelligence and Security (ICDIS)*. IEEE. 2018, pp. 131–134.

[39] Mehmet Necip Kurt et al. "Online cyber-attack detection in smart grid: A reinforcement learning approach". In: *IEEE Transactions on Smart Grid* 10.5 (2018), pp. 5174–5185.

[40] Kai Zhong et al. "An Efficient Parallel Reinforcement Learning Approach to Cross-Layer Defense Mechanism in Industrial Control Systems". In: *IEEE Transactions on Parallel and Distributed Systems* (2021).

[41] Martina Panfili et al. "A game-theoretical approach to cyber-security of critical infrastructures based on multi-agent reinforcement learning". In: *2018 26th Mediterranean Conference on Control and Automation (MED)*. IEEE. 2018, pp. 460–465.

[42] David Wilson et al. "Deep learning-aided cyber-attack detection in power transmission systems". In: *2018 IEEE Power & Energy Society General Meeting (PESGM)*. IEEE. 2018, pp. 1–5.

[43] Justyna J Chromik, Anne Remke, and Boudewijn R Haverkort. "Bro in SCADA: dynamic intrusion detection policies based on a system model". In: *5th International Symposium for ICS & SCADA Cyber Security Research 2018 5*. 2018, pp. 112–121.

[44] Hui Lin et al. "Runtime semantic security analysis to detect and mitigate control-related attacks in power grids". In: *IEEE Transactions on Smart Grid* 9.1 (2016), pp. 163–178.

[45] Jeyasingam Nivethan and Mauricio Papa. "A SCADA intrusion detection framework that incorporates process semantics". In: *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. 2016, pp. 1–5.

[46] Justyna J Chromik, Anne Remke, and Boudewijn R Haverkort. "What's under the hood? Improving SCADA security with process awareness". In: *2016 Joint Workshop on Cyber-Physical Security and Resilience in Smart Grids (CPSR-SG)*. IEEE. 2016, pp. 1–6.

[47] Helge Janicke et al. "Runtime-monitoring for industrial control systems". In: *Electronics* 4.4 (2015), pp. 995–1017.

[48] Omar Alrawi et al. "Forecasting Malware Capabilities From Cyber Attack Memory Images". In: *30th USENIX Security Symposium*. 2021.

[49] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *Acm Sigplan Notices* 46.3 (2011), pp. 265–278.

[50] Insu Yun et al. "QSYM: A practical concolic execution engine tailored for hybrid fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 745–761.

[51] Julian Rrushi et al. "A quantitative evaluation of the target selection of havex ics malware plugin". In: *Industrial Control System Security (ICSS) Workshop*. 2015.

[52] Huan Yang, Liang Cheng, and Mooi Choo Chuah. "Deep-learning-based network intrusion detection for SCADA systems". In: *2019 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2019, pp. 1–7.

[53] Ihab Darwish and Tarek Saadawi. "Attack Detection and Mitigation Techniques in Industrial Control System -Smart Grid DNP3". In: *2018 1st International Conference on Data Intelligence and Security (ICDIS)*. 2018, pp. 131–134. DOI: 10.1109/ICDIS.2018.00028.

[54] Mohamed Niang et al. "Formal Verification for Validation of PSEEL's PLC Program." In: *ICINCO (1)*. 2017, pp. 567–574.

[55] Yi Han et al. "Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 1095–1108.

[56] Yannan Liu et al. "On code execution tracking via power side-channel". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 1019–1031.

[57] Thomas Miller et al. "Looking back to look forward: Lessons learnt from cyber-attacks on Industrial Control Systems". In: *International Journal of Critical Infrastructure Protection* 35 (2021), p. 100464.

[58] Luis Garcia et al. "Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit." In: *NDSS*. 2017.

[59] *GuardLogix Controller Systems.* URL: https://literature.rockwellautomation.com/idc/groups/literature/documents/rm/1756-rm093_-en-p.pdf.

[60] Anhtuan Le, Utz Roedig, and Awais Rashid. "LASARUS: Lightweight Attack Surface Reduction for Legacy Industrial Control Systems". In: June 2017, pp. 36–52. ISBN: 978-3-319-62104-3. DOI: 10.1007/978-3-319-62105-0_3.

[61] Mohsen Salehi and Siavash Bayat-Sarmadi. *PLCDefender: Improving Remote Attestation Techniques for PLCs Using Physical Model.* 2021. DOI: 10.1109/JIOT.2020.3040237.

[62] Zeyu Yang et al. "PLC-Sleuth: Detecting and Localizing PLC Intrusions Using Control Invariants". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 2020, pp. 333–348.

[63] *Prevent intrusion and maintain network integrity with Data Diodes.* URL: https://advenica.com/en/cds/data-diodes.

[64] Giuseppe Campagna et al. "New flexibility services and technologies: ENEL experiences in Italy". In: *2020 AEIT International Annual Conference (AEIT)*. IEEE. 2020, pp. 1–6.

[65] Qais Qassim et al. "A survey of SCADA testbed implementation approaches". In: *Indian Journal of Science and Technology* 10.26 (2017), pp. 1–8.

[66] Daniel T Sullivan and Edward J Colbert. *Demonstration of SCADA Virtualization Capability in the US Army Research Laboratory (ARL)/Sustaining Base Network Assurance Branch (SBNAB) SCADA Hardware Testbed*. Tech. rep. RAYTHEON TECHNICAL SERVICES CO LLC DULLES VA, 2015.

[67] *NOZOMI Networks - Vantage*. URL: https://www.nozominetworks.com/downloads/US/Nozomi-Networks-Vantage-Data-Sheet.pdf.

[68] *NOZOMI Networks - Guardian*. URL: https://www.nozominetworks.com/downloads/US/Nozomi-Networks-Guardian-Data-Sheet.pdf.

[69] *NOZOMI Networks - Case Studies*. URL: https://www.nozominetworks.com/resources/case-studies/.

[70] *VTSCADA*. URL: https://www.vtscada.com/help/Content/Welcome.htm?tocpath=Welcome.

[71] Saleh Soltan, Prateek Mittal, and H Vincent Poor. "BlackIoT: IoT botnet of high wattage devices can disrupt the power grid". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 15–32.

[72] *The Visual Approach to Electric Power System*. URL: https://www.powerworld.com.

[73] *Alarm function blocks in IEC 61131-3 programming*. URL: https://www.plcnext.help/te/Service_Components/Alarms/Alarm_Function_Blocks_IEC61131.htm.

[74] *Unified Automation OPC*. URL: https://documentation.unified-automation.com/uasdkc/1.3.2/html/group__UaServerEvents.html#gaa87b4e02150191876901697eb2eeb86c.

[75] Andrea Carcano et al. "A multidimensional critical state analysis for detecting intrusions in SCADA systems". In: *IEEE Transactions on Industrial Informatics* 7.2 (2011).

[76] Alfonso Valdes and Steven Cheung. "Communication pattern anomaly detection in process control systems". In: *2009 IEEE Conference on Technologies for Homeland Security*. IEEE. 2009, pp. 22–29.

[77] Seyedhamed Ghavamnia et al. "Temporal system call specialization for attack surface reduction". In: *29th Security Symposium (USENIX Security 20)*. 2020, pp. 1749–1766.

[78] *OPC Foundation. OPC Client*. URL: https://opcfoundation.org/products/view/opc-ua-client-free-product.

[79] *Unified Automation OPC UA*. URL: https://documentation.unified-automation.com/uasdknet/2.5.7/html/classUnifiedAutomation_1_1UaBase_1_1ReadRequest.html.

[80] *ATTACK for Industrial Control Systems*. URL: https://collaborate.mitre.org/attackics/index.php/Main_Page.

[81] *MYSCADA SCADA Automation and HMI Solutions*. URL: https://www.myscada.org/en/.

[82] *Simulation and Model Based Design*. URL: https://www.mathworks.com/products/simulink.html.

[83] *ICSSPLOIT*. URL: https://github.com/dark-lbp/isf/tree/master/icssploit.

[84] *Metasploit Modules for SCADA-related Vulnerabilities*. URL: https://scadahacker.com/resources/msf-scada.html.

[85] Joseph Connelly et al. "CloudSkulk: A Nested Virtual Machine Based Rootkit and Its Detection". In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2021, pp. 350–362.

[86] Mohammad Nadim, Wonjun Lee, and David Akopian. "Characteristic features of the kernel-level rootkit for learning-based detection model training". In: *Electronic Imaging* 2021.3 (2021), pp. 34–1.

[87] Aniket Sahu et al. "A Survey on Rootkit Techniques". In: ().

[88] *Using Nt and Zw Versions of the Native System Services Routines*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines.

[89] Mark Ermolov and Artem Shishkin. *Microsoft Windows 8.1 Kernel Patch Protection Analysis*.

[90] Mark Hall et al. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[91] Alfred Melvin, G Jaspher Kathrine, and J Immanuel Johnraja. "The Practicality of using Virtual Machine Introspection Technique with Machine Learning Algorithms for the Detection of Intrusions in Cloud". In: (2021).

[92] Hyun wook Baek, Abhinav Srivastava, and Jacobus Van der Merwe. "Cloudvmi: Virtual machine introspection as a cloud service". In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 153–158.

[93] Siqi Zhao et al. "Seeing through the same lens: introspecting guest address space at native speed". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 799–813.

[94] Zhihong Tian et al. "A real-time correlation of host-level events in cyber range service for smart campus". In: *IEEE Access* 6 (2018), pp. 35355–35364.

[95] Bhavesh Borisaniya and Dhiren Patel. "Towards virtual machine introspection based security framework for cloud". In: *Sadhana* 44.2 (2019), p. 34.

[96] Tamas K Lengyel et al. "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014, pp. 386–395.

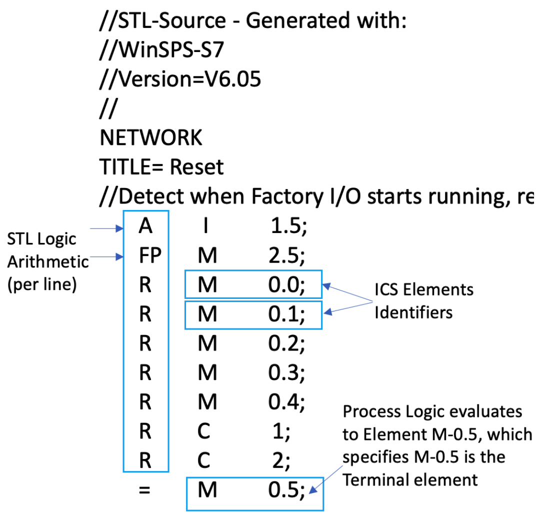
[97] Lennart Ljung. "System identification". In: *Signal analysis and prediction*. Springer, 1998, pp. 163–173.

[98] Samuel East et al. "A Taxonomy of Attacks on the DNP3 Protocol". In: *International Conference on Critical Infrastructure Protection*. Springer. 2009, pp. 67–81.

[99] Nicholas Rodofile, Kenneth Radke, and Ernest Foo. "Real-time and interactive attacks on DNP3 critical infrastructure using Scapy". In: *Proceedings of the 13th Australasian Information Security Conference (AISC 2015)[Conferences in Research and Practice in Information Technology (CRPIT), Volume161]*. Australian Computer Society Inc. 2015, pp. 67–70.

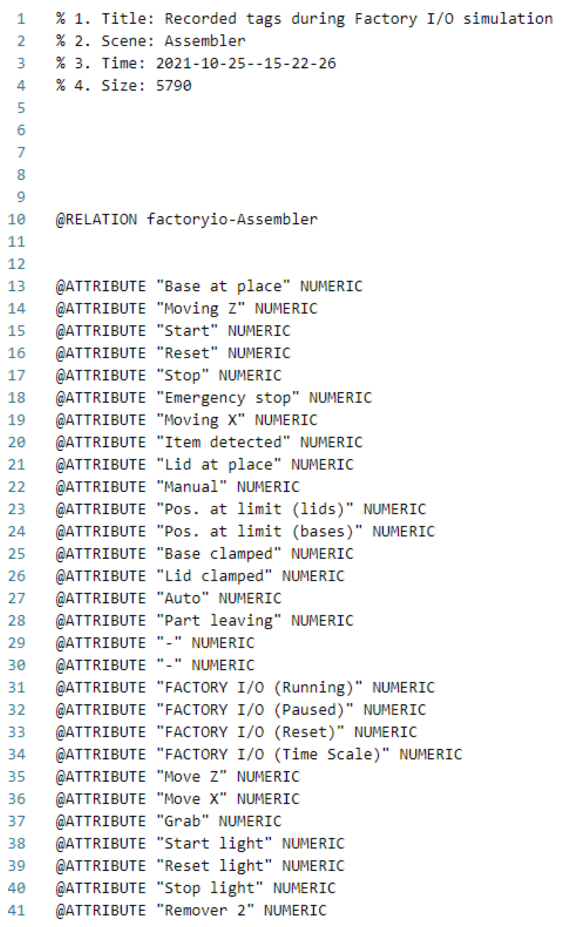
[100] Igor Nai Fovino et al. "Modbus/DNP3 state-based intrusion detection system". In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE. 2010, pp. 729–736.

APPENDIX



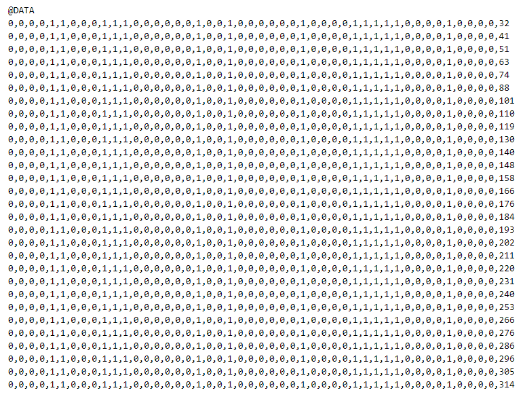
```
//STL-Source - Generated with:
//WinSPS-S7
//Version=V6.05
//
NETWORK
TITLE= Reset
//Detect when Factory I/O starts running, re
A    I    1.5;
FP   M    2.5;
R    M    0.0;
R    M    0.1;
R    M    0.2;
R    M    0.3;
R    M    0.4;
R    C    1;
R    C    2;
=    M    0.5;
```

Fig. 9: Our Example Statement List (STL) TEXT FILE Exported from WinSPS-S7 Function Block Diagram. Annotated to show the different parts



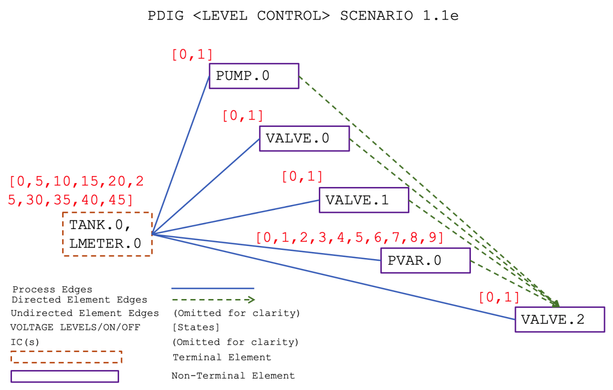
```
% 1. Title: Recorded tags during Factory I/O simulation
% 2. Scene: Assembler
% 3. Time: 2021-10-25--15-22-26
% 4. Size: 5790





@RELATION factoryio-Assembler


@ATTRIBUTE "Base at place" NUMERIC
@ATTRIBUTE "Moving Z" NUMERIC
@ATTRIBUTE "Start" NUMERIC
@ATTRIBUTE "Reset" NUMERIC
@ATTRIBUTE "Stop" NUMERIC
@ATTRIBUTE "Emergency stop" NUMERIC
@ATTRIBUTE "Moving X" NUMERIC
@ATTRIBUTE "Item detected" NUMERIC
@ATTRIBUTE "Lid at place" NUMERIC
@ATTRIBUTE "Manual" NUMERIC
@ATTRIBUTE "Pos. at limit (lids)" NUMERIC
@ATTRIBUTE "Pos. at limit (bases)" NUMERIC
@ATTRIBUTE "Base clamped" NUMERIC
@ATTRIBUTE "Lid clamped" NUMERIC
@ATTRIBUTE "Auto" NUMERIC
@ATTRIBUTE "Part leaving" NUMERIC
@ATTRIBUTE "-" NUMERIC
@ATTRIBUTE "-" NUMERIC
@ATTRIBUTE "FACTORY I/O (Running)" NUMERIC
@ATTRIBUTE "FACTORY I/O (Paused)" NUMERIC
@ATTRIBUTE "FACTORY I/O (Reset)" NUMERIC
@ATTRIBUTE "FACTORY I/O (Time Scale)" NUMERIC
@ATTRIBUTE "Move Z" NUMERIC
@ATTRIBUTE "Move X" NUMERIC
@ATTRIBUTE "Grab" NUMERIC
@ATTRIBUTE "Start light" NUMERIC
@ATTRIBUTE "Reset light" NUMERIC
@ATTRIBUTE "Stop light" NUMERIC
@ATTRIBUTE "Remover 2" NUMERIC
```

Fig. 10: Our Example ARFF Header showing fields (Element states) we used for specific element in the Sorting Station Scenario



```
@DATA
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,32
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,41
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,51
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,63
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,74
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,88
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,101
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,110
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,119
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,130
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,140
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,148
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,158
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,166
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,176
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,184
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,193
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,202
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,211
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,220
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,231
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,240
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,253
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,266
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,276
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,286
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,296
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,305
0,0,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,314
```

Fig. 11: Sample ARFF-formatted Element State reporting every 100 scan cycles



(a) Example PDIG Model for the Level Control Process of the FL Attack Incident

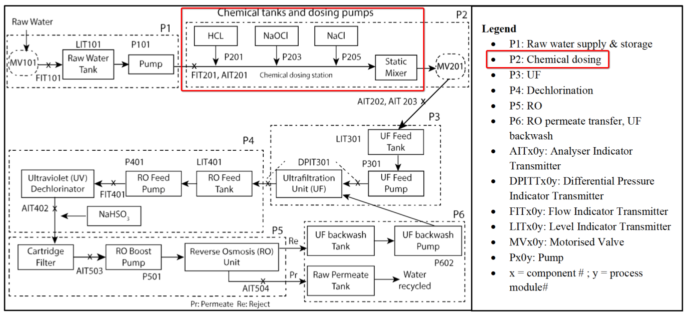
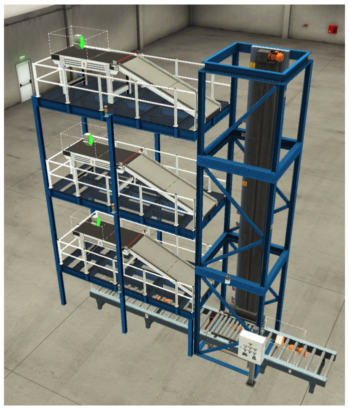
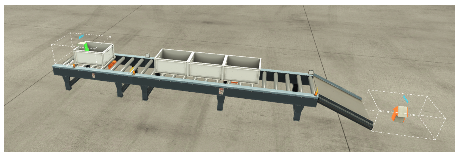
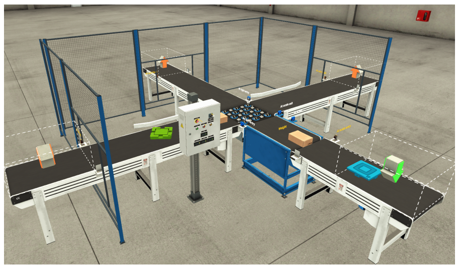


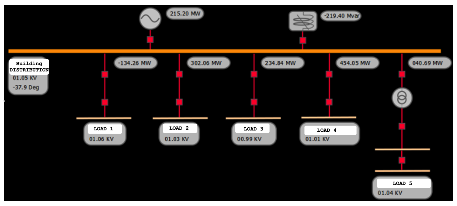
Fig. 13: Complete water treatment plant based on [26, 27]: Showing the chemical dosing operation in reference to the FL water poisoning attack


(a) HMI for Elevator (Advanced) Scene

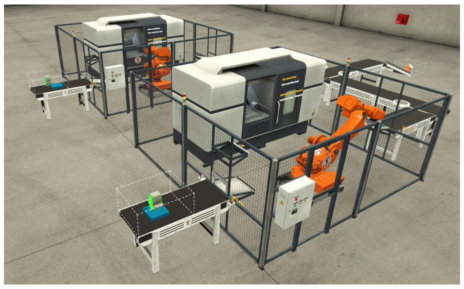

(b) HMI for Queue Processor Scene

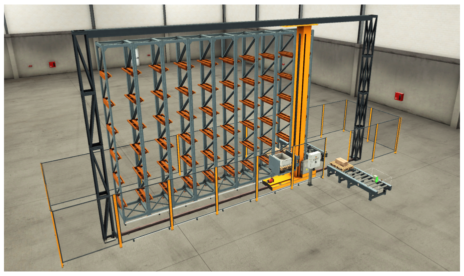
(c) HMI for Converge Station Scene

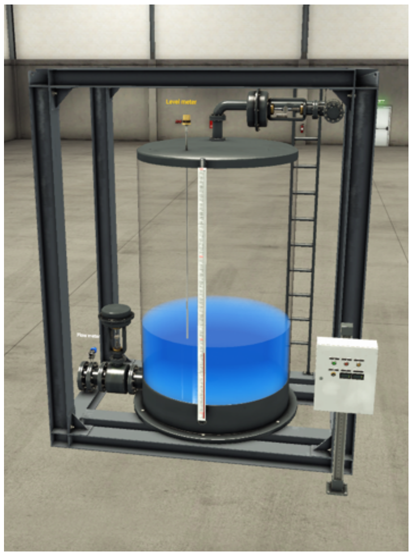

(d) HMI for Sorting Station Scene

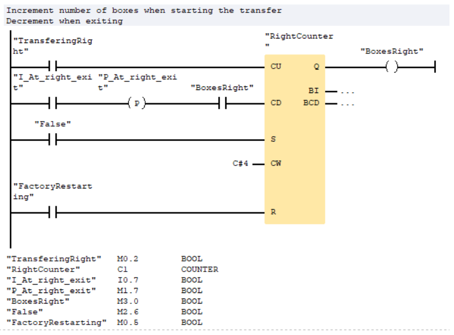


(e) HMI of Power Scenario

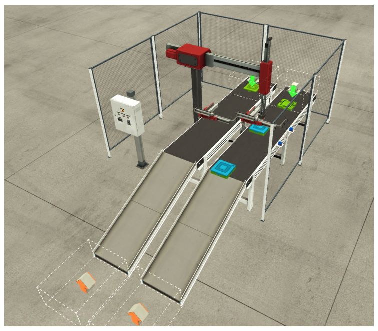

(f) HMI for Production Line Scene

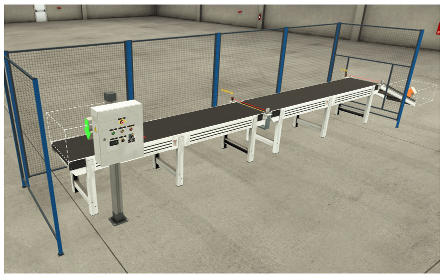

(g) HMI for Automated Warehouse Scene


(h) HMI for Chemical Dosing Scene

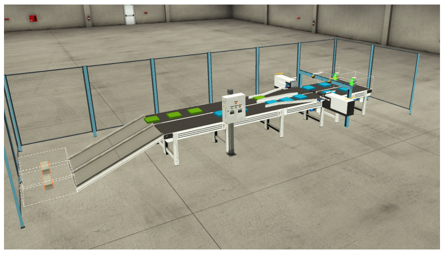

```
Increment number of boxes when starting the transfer
Decrement when exiting
```

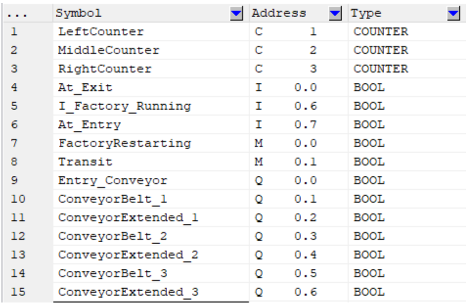
```
"TransferingRight"   M0.2       BOOL
"RightCounter"       C1         COUNTER
"I_At_right_exit"    I0.7       BOOL
"P_At_right_exit"    M1.7       BOOL
"BoxesRight"         M3.0       BOOL
"False"              M2.6       BOOL
"FactoryRestarting"  M0.5       BOOL
```
(i) Example 2-wire or FBD for Sorting Station Scenario


(j) HMI for Assembler Scene


(k) HMI for Buffer Station Scene

(l) HMI for Palletizer Scene


(m) HMI for Separating Station Scene



...	Symbol	Address		Type
1	LeftCounter	C	1	COUNTER
2	MiddleCounter	C	2	COUNTER
3	RightCounter	C	3	COUNTER
4	At_Exit	I	0.0	BOOL
5	I_Factory_Running	I	0.6	BOOL
6	At_Entry	I	0.7	BOOL
7	FactoryRestarting	M	0.0	BOOL
8	Transit	M	0.1	BOOL
9	Entry_Conveyor	Q	0.0	BOOL
10	ConveyorBelt_1	Q	0.1	BOOL
11	ConveyorExtended_1	Q	0.2	BOOL
12	ConveyorBelt_2	Q	0.3	BOOL
13	ConveyorExtended_2	Q	0.4	BOOL
14	ConveyorBelt_3	Q	0.5	BOOL
15	ConveyorExtended_3	Q	0.6	BOOL

(n) Example Element Parameter Tags for Sorting Station Scenario