A Specification-based State Replication Approach for Digital Twins

Matthias Eckhart
Christian Doppler Laboratory "SQI", Inst. of Information Systems Engineering, TU Wien
Vienna, Austria
matthias.eckhart@tuwien.ac.at

Andreas Ekelhart
SBA Research
Vienna, Austria
JRC TARGET
St. Pölten, Austria
andreas.ekelhart@sba-research.org

ABSTRACT

Digital twins play a key role in realizing the vision of a smart factory. While this concept is often associated with maintenance, optimization, and simulation, digital twins can also be leveraged to enhance the security and safety of cyber-physical systems (CPSs). In particular, digital twins can run in parallel to a CPS, allowing to perform a security and safety analysis during operation without the risk of disrupting live systems. However, replicating states of physical devices within a CPS in functionally equivalent virtual replicas, so that they precisely mirror the internal behavior of their counterparts, is an open research topic. In this paper, we propose a novel state replication approach that first identifies stimuli based on the system's specification and then replicates them in a virtual environment. We believe that replicating states of CPSs is a prerequisite for a multitude of security and safety enhancing features that can be implemented on the basis of digital twins. To demonstrate the feasibility of the specification-based state replication approach, we provide a prototypical implementation and evaluate it in an experimental CPS test bed. The results of this paper show that attacks against CPSs can be successfully detected by leveraging the proposed state replication approach.

CCS CONCEPTS

• **Security and privacy** → **Intrusion detection systems**; • **Computer systems organization** → *Embedded and cyber-physical systems*;

KEYWORDS

Cyber-physical systems; industrial control systems; digital twin; state replication; intrusion detection systems; AutomationML

1 INTRODUCTION

Cyber-physical systems (CPSs) are characterized by their ability to perform computations and to interact with the real world [2]. Industrial control systems (ICSs) can be considered as a subset of CPSs, as they integrate computational elements for the purpose of industrial process control. Due to the fact that ICSs have a direct impact on the environment they operate in, ensuring that these systems meet certain safety requirements is paramount. However, past incidents [21] have shown that a lack of adequate security measures may result in disrupted processes, potentially causing financial damage but also safety risks to personnel and the public [27]. As a consequence, CPSs and, in particular, ICSs gained much attention from security researchers in the past years [15, 20].

In contrast to IT systems, ICSs may consist of components which have a lifecycle of more than 15 years [19]. Consequently, they may rely on outdated technology that can often neither be replaced nor updated. Securing industrial systems is also a challenging endeavor, since a potential disruption through traditional IT security measures, e.g., network scanning [6], is typically not acceptable. Based on these characteristics of ICSs, it is evident that established IT security principles cannot be directly applied to the operational technology (OT) domain. Motivated by these challenges, it is a common requirement for ICS security measures to keep the impact on live systems as low as possible. This requirement concerns intrusion detection, intrusion prevention and, in particular, penetration testing. However, opting for less invasive or even purely passive methods and solutions may limit the desired level of security. For example, a passive monitoring approach does not allow a thorough inspection of the device, whereas active monitoring may cause a considerable overhead [23]. Although efforts have been made to combine active and passive security techniques to compensate the shortcomings of each other (e.g., [13]), there has been little discussion about how passive techniques can be further improved, eventually replacing active ones. For instance, little is known about how program states of devices can be passively replicated to obtain an accurate virtual representation of the CPS during its operation. In the context of this work, such a virtual representation consists of digital twins[1], i.e., virtual replicas of the network and the logic layer of physical devices, closely matching the physical devices' behavior on these layers. Due to the fact that these digital twins run in an isolated virtual environment and thus, are not exposed to external influences that originate from the real world, a state replication approach is necessary to keep the states of digital twins in sync with those of their physical counterparts.

In particular, we identify three use cases based on a state replication approach that would enhance the security and safety of a CPS.

First, the CPS can be indirectly analyzed without negatively affecting its operation. In other words, digital twins that maintain synced states with their physical counterparts, allow the use of active monitoring techniques on the digital twins, without causing any risk of interference with live systems, as they run in an isolated, virtual environment. In fact, the framework for controlling the digital twins may even provide an interface that allows

[1]While in the context of smart manufacturing, the term "digital twin" is more broadly applied to modeling, simulation and visualization aspects based on various artifacts and data collected during operation (e.g., [25]), we use it to describe emulated or simulated devices that may be connected to an emulated network.

a direct retrieval of states, making active monitoring via the network superfluous (cf. [7]). Moreover, such a framework could also automatically perform checks of security and safety rules (e.g., thresholds or consistency checks) that have been defined by users in advance [7].

Second, the input and output of physical devices can be compared to those of their digital twins for the purpose of intrusion detection. In essence, this can be considered as a variant of the behavior-specification-based intrusion detection technique [22, 28], provided that the CPS's correct behavior has been defined in a specification and is then used to generate the virtual replica. The beauty of this approach is that legitimate behavior of the CPS is already formulated during the engineering phase, reducing the configuration effort for such an intrusion detection system (IDS). During the operation of the CPS, the IDS would take the input and output of the physical devices and compare them to the input and output of the corresponding digital twins. The IDS would then raise an alarm, if differences were detected that indicate malicious activity. Although the findings by Mitchell and Chen [22] suggest that behavior-specification-based intrusion detection approaches generally yield a low false positive rate, the effectiveness of the IDS could be affected by inaccuracies in the specification of the CPS, inequalities in the implementation of the digital twins or by state mismatches.

Third, the data required for state replication could also be stored in a database for later use. In this way, historical states of digital twins can be recovered at a later point in time in order to repeat certain scenarios and analyze them in depth. For instance, this feature may provide valuable insights into how an infection occurred or what was the cause of a fault.

The paper at hand attempts to lay the foundation for the described use cases by proposing a technique to replicate program states of devices within a CPS to respective digital twins. To mirror the state of the physical environment, it is necessary to feed external stimuli to the digital twins. In consequence, they should exhibit the same behavior on the network and logic layer in the virtual environment. Our approach first identifies stimuli from specification that may trigger state transitions. Based on this knowledge, a variety of data sources (e.g., network traffic, system logs) can be used to identify concrete stimuli in order to replay them and, thereby, synchronize the program state between the digital twin and its physical counterpart.

To demonstrate the feasibility of our state replication approach, we extend the digital-twin framework *CPS Twinning* proposed in [7]. This framework generates digital twins based on the specification of CPSs, defined in AutomationML (AML) [5] artifacts. The physical connections and logical interactions between devices are also modeled to fully mirror the cyber-physical environment. Eckhart and Ekelhart [7] provide a proof of concept, including a prototype, that demonstrates how digital twins can be generated from specification and run independently from the physical environment. Yet, a replication mode in which they mirror the state of physical devices is missing. This paper aims to fill this gap.

We summarize the contributions of this work as follows:

- We propose a specification-based state replication approach for digital twins.
- We prove the feasibility of the proposed state replication approach by providing a proof-of-concept implementation named *CPS State Replication*.
- We evaluate the prototype in an ICS test bed by first measuring the state replication accuracy and then demonstrating how the concept of state replication can be leveraged to detect attacks against ICS.

The remainder of this paper has four sections. First, in Section 2, we cover related work in the context of state replication and intrusion detection. In Section 3, we formulate the proposed state replication approach. Section 4 presents the proof of concept and evaluates the implemented prototype. Finally, Section 5 concludes the paper and provides suggestions for future research directions.

2 RELATED WORK

Related work can be divided into the following categories: (i) state observers, (ii) state machine replication, (iii) state-based & process-aware intrusion detection, and (iv) physics-based state-aware intrusion detection.

State Observers. State observers can be used to estimate the state vector of a system based on observations of its inputs and outputs [18]. Although conceptually related to the field of control theory, our aim is not to *estimate* the systems' state vector, but rather to *replicate* the *program states* of systems to their virtual replicas. More precisely, we are not attempting to estimate the state of a system based on a mathematical model that describes its dynamics. Instead, our state replication approach feeds specific inputs to a digital twin that executes a functionally equivalent copy of the program running on the corresponding physical device. In further consequence, we can retrieve the digital twin's values of program variables with the expectation to receive an accurate representation of the variables that reside on the real, physical device.

State Machine Replication. Another related concept, yet associated with the research area of distributed computing and applied with a different objective, is state machine replication. In essence, state machine replication can be used to achieve fault tolerance in distributed systems, namely by first replicating services for the purpose of redundancy and then ensuring that the states of these services are kept in sync [26]. To reach an agreement among the services so that commands can be processed in a coordinated manner, consensus algorithms, such as Paxos [16], are used. Although state machine replication and the state replication approach presented in this paper share similar characteristics (e.g., requiring determinism), the key difference lies in the fact that both aim to solve different problems and, therefore, cannot be used interchangeably. In state machine replication, multiple replicas of one state machine must be coordinated, ensuring that every non-faulty one receives every request [26]. Instead, our proposed approach attempts to identify and replay only inputs that emanate from sources external to the virtual environment in order to to keep the state of digital twins in sync with their physical counterparts. Furthermore, due to the fact that only one digital twin is generated in the virtual environment for each device and physical traits are simulated, the state replication approach does not improve the CPS's fault tolerance. In

other words, if a device of the CPS fails, the respective digital twin cannot take over to prevent an interruption of processes.

State-based & Process-aware Intrusion Detection. As mentioned in Section 1, the rationale for replicating states to digital twins is to maintain a current virtual representation of physical devices that can then be used for security relevant use cases, such as intrusion detection. The notion of tracking the states of an ICS for the purpose of intrusion detection is not new. Several studies investigated how the sequence of system states can be virtually represented in order to detect malicious inputs, i.e., inputs that would cause the system to transition into a critical state [3, 10, 12].

In [10], the authors present an architecture of a state-based IDS that records the system state in "system virtual images" to identify malicious network requests. System virtual images are virtual representations of programmable logic controllers (PLCs) and remote terminal units (RTUs), including their memory registers, coils, inputs and outputs. To keep track of state transitions in a timely manner, an active monitoring approach has been implemented. Furthermore, they propose a language for specifying a rule set that is used by the IDS, allowing users to define malicious packets and critical states of systems. The approach proposed in [3] extends the work conducted by [10], as it examines historical states in order to determine a tendency towards a critical state. The prediction of critical states is achieved by leveraging distance metrics that indicate how far the monitored states and the defined critical states are apart from each other. Similar to [10], the IDS proposed by Carcano et al. [3] uses active monitoring to keep the states of the virtual representation in sync with the real system. Contrary to [3, 10], the IDS presented by Hadžiosmanović et al. [12] favors passive monitoring and is also able to interpret the semantics of PLC variables, enabling the derivation of behavioral models. Autoregressive modeling and control limits form the basis of the data modeling technique that is used in [12] to detect malicious activity.

Other, more recent works, such as [4, 24], propose process-aware intrusion detection techniques that focus specifically on states of the physical process, rather than mere system states. For instance, Nivethan and Papa [24] extend the work of Fovino et al. [10], as their IDS is able to interpret rules that have been formulated by process engineers. In this way, high-level constraints for process states can be specified by individuals who have expert knowledge about the physical process, which are then mapped by the IDS to corresponding system states.

In contrast to our approach, the IDSs presented in [3, 10, 12, 24] do not base their detection mechanisms on virtual replicas of the ICS, consisting of an emulated network layer and either emulated or simulated components (digital twins). Instead, they actively or passively retrieve the values of PLC variables in order to monitor the state of industrial processes. Moreover, the method for detecting intrusions also differs. While they rely on rules that define critical states [3, 4, 10, 24] or use statistical methods [12] to predict abnormal conditions of the process under control, we use certain features (e.g., inputs) to perform a comparison between the virtual replica of the ICS (generated from specification, thereby defining legitimate behavior) and its physical counterpart.

Physics-based State-aware Intrusion Detection. In recent years, a number of papers have been published which examine how physical models of a system can be leveraged to uncover intrusions [29]. For example, Urbina et al. [30] evaluate the effectiveness of a stateful detection method, which is based on the cumulative sum (CUSUM) algorithm to take historical changes of states into account. In [11], the authors present an improved version of the stateful intrusion detection approach [30] that enhances the CUSUM computation by adding a state-dependent parameter; thus, making it "state-aware".

Although the integration of a device's physical model into the corresponding digital twin for the purpose of detecting intrusions is worth researching, our work focuses on monitoring the program states in conjunction with network traffic.

3 STATE REPLICATION

Before discussing the proposed method, it is essential to clarify what constitutes a "state" of a device and how the term "replication" is used in this context. In this work, a deterministic program, P, denotes a finite-state machine (FSM), which is defined by a tuple $P \coloneqq (X, x_0, U, Y, \delta, \lambda)$ where $X \coloneqq \{x_0, x_1, \ldots, x_{n-1}\}$ is the finite set of *states*, $x_0 \in X$ is the initial state, $U \coloneqq \{u_0, u_1, \ldots, u_{k-1}\}$ is the finite set of inputs, $Y \coloneqq \{y_0, y_1, \ldots, y_{m-1}\}$ is the finite set of outputs, $\delta \colon X \times U \to X$ is the transition function and $\lambda \colon X \times U \to Y$ is the output function.

Furthermore, we expect that each digital twin runs a program $\hat{P}$ that is functionally identical to the one running on its physical counterpart P, such that $P = \hat{P}$. Thus, $\delta(x, u) = \hat{\delta}(\hat{x}, \hat{u}) \Leftrightarrow x' = \hat{x}'$, provided that $(x = \hat{x}) \wedge (u = \hat{u})$ (where x is the current state, and x' the next state of a physical device; likewise, $\hat{x}$ is the current state, and $\hat{x}'$ the next state of the device's digital twin).

Recall that an ICS consists of a variety of physical devices which may be represented as digital twins in the virtual environment. Each device can have an input $u \in U$ and an output $y \in Y$. Thus, $U \subset U^*$ and $Y \subset Y^*$, where U^* and Y^* is the set of inputs and outputs of *all* devices, respectively. Moreover, we define the set of stimuli, S, representing the roots of subsequent inputs, corresponding to one digital twin (i.e., stimuli emanation outside the modeled environment, such as sensor values or commands from an HMI), as follows:

$$S \coloneqq \{\, z \in \hat{U} \mid z \in U \wedge z \notin Y^* \,\}$$

where $\hat{U}$ is the set of inputs of a digital twin. In particular, $S \subset \hat{U}$ and, $\hat{U}$ may contain elements of $\hat{Y}^*$ as well as inputs that emanate from users interacting with the digital twin.

The term *replication*, on the other hand, describes the process of repeating the same stimulus, $s \in S$, from the physical environment on a digital twin, so that it leads to an identical program state ($x = \hat{x}$). If any members of S would not satisfy the right operand of the logical conjunction $z \in U \wedge z \notin Y^*$, then the digital twin may receive the same input twice. The reason for this result is that if such a stimulus is erroneously replicated, then the digital twin receives the stimulus in addition to an input that actually has been a prior output, since each digital twin should produce $\hat{y} \in \hat{Y}$ (where $\hat{Y}$ represents the set of a digital twin's outputs) by itself.

It is also worth mentioning that the period between two successive stimuli observed in the physical environment and the period between these two stimuli being fed to the respective digital twin

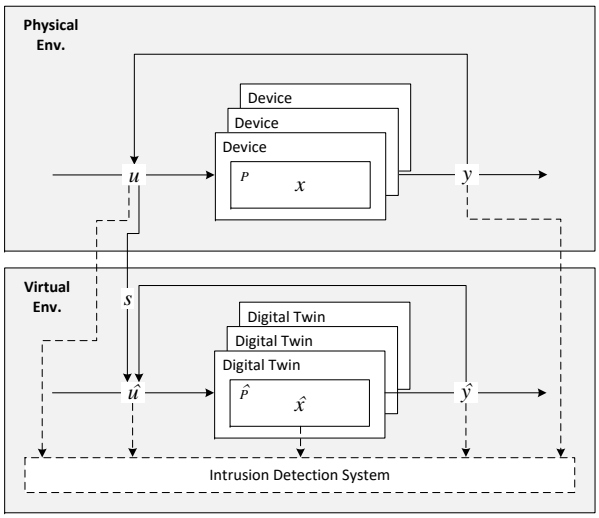


Figure 1: Structure of the state replication approach

in the virtual environment must be equal, to expect an identical state.

3.1 Drawbacks of Active Monitoring

At first sight, the obvious solution to capture x would be an active monitoring approach, i.e., continuously polling for changes of the state. However, this way of state replication has several disadvantages. First of all, actively monitoring each device in the physical environment via the network would incur a significant increase in network traffic and potentially disrupt real-time communication within industrial networks. Second, active monitoring may not only affect the performance on a network level, but also on devices themselves. This downside is particularly critical for legacy systems, as they are still prevalent in industrial environments and typically limited in computational resources. Third, an attacker could tamper the response of a poll request in order to hide the real program state, x. In this way, a security and safety analysis based on the states of digital twins could be circumvented and, as a consequence, malicious activity would remain undetected.

Due to the aforementioned negative effects of using an active monitoring technique for the purpose of replicating states to digital twins, a passive approach will be proposed.

3.2 A Passive State Replication Approach

In a purely passive setting, the trigger of a state transition must be tracked down in order to understand which u constitutes a stimulus so that it can be fed to the transition function of the digital twin, $\hat{\delta}(\hat{x}, s)$. Since the proposed state replication approach should not add any overhead to the network, nor to devices themselves, the identification of stimuli must exclusively rely on sources that are already available due to the architecture of the CPS, enabling a purely passive operation. These sources can be diverse, ranging from the passive monitoring of network traffic to log files.

As Figure 1 illustrates, the devices which are located within the physical environment (i.e., CPS), receive an input u that may trigger a state transition, such that $\delta(x, u) = x'$. For instance, a PLC could receive an input via its analog-, digital- or network interface that may cause a transition into the next state x', as defined by the control logic. Since u must be considered as a determining factor that drives x', u could be directly fed to the PLC's digital twin ($u = \hat{u}$), leading to $\hat{\delta}(\hat{x}, \hat{u}) = \hat{x}'$, provided that $u = s$. This can be exemplified by considering a PLC that is used to control the liquid level in a tank. The PLC is connected to a liquid level sensor (input), as well as a valve (output), for the purpose of maintaining a specific liquid level inside the tank. In addition, it is important to note that only the PLC exists in the virtual environment as a digital twin. In this setup, $S = U$ of the PLC, due to the fact that the sensor measurements represent the root of inputs and the liquid level sensor is not part of the virtual environment. Although u is governed by y, as the valve influences the liquid level, the data that is coming from sensor readings constitute external influences. As a result, the input, u, received by the physical PLC can be directly replicated within the virtual environment to reach $x' = \hat{x}'$.

However, replicating states of CPSs that are more complex may lead to two issues. First, detecting inputs without proper sensors in place, such as user actions, may be limited. Second, if s cannot be correctly distinguished from u, a state mismatch ($x \neq \hat{x}$) may occur. For instance, an input u that is a consequence of a prior state transition should have been produced by the respective digital twin on its own. Hence, replicating u would only duplicate the input and potentially lead to a state mismatch. However, differentiating between s and u may be challenging, since there can be long chains of state transitions that increase complexity, especially when multiple devices are involved.

To overcome these challenges, we propose to utilize the specification of the CPS in order to identify stimuli. First, the specification of the CPS will be parsed to define the partial function f, i.e., the mapping of stimuli indices, I, to stimuli of *all* digital twins, S^*. Let $f : U^* \cup Y^* \nrightarrow S^*$ be a partial function, then I is defined as follows:

$$I := \{\, j \in U^* \cup Y^* \mid f(j) \in S^* \,\}.$$

Next, $j \in U^* \cup Y^*$ will be observed and checked whether $j \in I$. Since $j \in I \Leftrightarrow f(j)\downarrow$, meaning that j is a member of I if and only if $f(j)$ is defined, $s \in S^*$, the value of f of j, is fed to the respective digital twin, provided that j is indeed in the set of I. Hence, $\hat{\delta}(\hat{x}, s) = \hat{x}'$.

Example. A packaging line is managed by a human machine interface (HMI) that communicates with a PLC via Modbus TCP/IP in order to control a conveyor belt. The characteristics of these devices, including their programs, and the physical as well as logical network have been specified in an AML-based engineering artifact. As discussed in [7], the virtual environment along with digital twins can be automatically generated based on the data extracted from this specification. Furthermore, external influences can be inferred by examining the associated role of components. Based on the role definition of the HMI, i.e., `AutomationMLExtendedRoleClassLib/HMI`, it is evident that this component receives inputs from users. Due to the fact that user actions represent the roots of subsequent state transitions, this external influence qualifies as a stimulus, $s \in S$,

for the digital twin of the HMI. However, an input of the HMI, $u \in U$, cannot be directly observed without effort, as no method that would record human input (e.g., camera, data logging) has been set up beforehand. Nevertheless, a user input generates an output in the form of a Modbus request, making x indirectly observable by passively monitoring the network traffic in order to capture outputs of the HMI. Next, the partial mapping (f) must be created based on the specified logical network (cf. Listing 1) in two steps. First, `InternalElement` elements (Listing 1, starting at line 2) within the logical network are extracted for the purpose of classifying inputs and outputs, as they declare how devices communicate, including details concerning the used network protocol (e.g., Modbus function code). Second, `InternalLink` elements (Listing 1, line 11) are resolved to identify source and destination of packets, and which program variable(s) they may affect. Finally, if an output of the HMI, y, has been classified as an outcome of a stimulus ($y \in I$), $f(y)$ is defined, meaning that s can be inferred from y by evaluating function f. In further consequence, s is replicated in the virtual environment.

```
1  <InternalElement Name="LogicalNetwork" ID="c51...">
2    <InternalElement Name="ModbusRequests" ID="ce1...">
3      <InternalElement Name="StartConveyorBeltModbusReadRequest"
       ↪  ID="0e5...">
4        <Attribute Name="functionCode"
         ↪  AttributeDataType="xs:integer">
5          <Value>3</Value>
6        </Attribute>
7        <Attribute Name="startingAddress"
         ↪  AttributeDataType="xs:integer">
8          <Value>0</Value>
9        </Attribute>
10       ...
11       <InternalLink Name="HMI1 StartConveyorBelt - PLC1 Modbus
         ↪  400001" RefPartnerSideA="{068...}:StartConveyorBelt"
         ↪  RefPartnerSideB="{29b...}:1" />
12       <RoleRequirements
         ↪  RefBaseRoleClassPath="/ModbusReadHoldingRegisters"
         ↪  />
13     </InternalElement>
14     ...
```

Listing 1: Excerpt of the logical network specification

It is also worth mentioning that u may not constitute an input emanating from sources that are external to the device (e.g., incoming network packet), but rather from the control logic itself (e.g., triggered by a timer instruction). However, in this case we still expect $x = \hat{x}$, provided that $P = \hat{P}$, i.e., the digital twin and its physical counterpart execute a functionally identical version of the program and both have the same initial state. However, achieving $s_0 = \hat{s}_0$ represents a key challenge in implementing the proposed approach in a real-world setting, since program states in live systems cannot be arbitrarily altered or reset.

4 PROOF OF CONCEPT

In this section, we present a proof of concept of the proposed state replication approach. First, we introduce the test bed and evaluation scenario. After that, we describe the architecture, followed by the presentation of the proof-of-concept implementation and the evaluation results.

4.1 Scenario

To evaluate the implemented state replication approach, we built an experimental ICS test bed that represents a fraction of a candy production line. In particular, our ICS aims to improve efficiency in quality control activities, as selected candies can be removed from the conveyor in an automated manner, for the purpose of manual inspection.

Figure 2 depicts the architecture of the ICS test bed. In general, the architecture can be divided into three layers, viz., (i) the physical process (level 0), (ii) basic control and connectivity (level 1), and (iii) supervisory control and IT/OT services (level 2). Since we wanted to examine whether state replication is also feasible when more recent technology is involved, we designed the ICS in line with key principles of a smart factory, emanating from the vision of Industry 4.0 [14].

In the scenario process at hand, the goods are transported on `Conveyor Belt 1`. `PLC 1`, a Siemens SIMATIC S7-1200 PLC, controls `Motor 1`, a stepper motor that drives the conveyor belt. `HMI 1` communicates with the PLC via Modbus TCP/IP, allowing users to set the state (start/stop) and the velocity of the conveyor belt. Furthermore, each candy carries an RFID tag, enabling the identification of the candy's type (e.g., mint flavor). If `RFID Reader 1`, an ESP8266, reads a tag, the type of the candy is transmitted wirelessly via `Access Point 1` to `MQTT Broker 1` in order to publish it on the MQTT `candy` topic. As `IIoT Gateway 1` is subscribed to this topic, the type of the detected candy is received by the gateway via MQTT and then sent via Modbus TCP/IP to `PLC 1`. If the PLC receives such a Modbus request, the type of the detected candy is saved in memory. Besides allowing users to control the conveyor belt by interacting with `HMI 1`, they can also select a type of candy that should be removed from the conveyor belt. If a candy type has been selected, a Modbus TCP/IP request is sent to `PLC 1`, which then saves the user's selection in memory. If the detected and selected candy — both stored in memory of `PLC 1` — match, the candy ejection takes place in two steps. First, the PLC controls `Motor 1` in a way that it moves the candy to an absolute position, ensuring that it is within reach of the pusher. Then, the PLC activates the pusher by starting `Motor 2` in order to eject the candy. After the candy has been successfully removed from the conveyor belt, `PLC 1` resets the variable that is used to store the user's selection.

As mentioned in Section 3.2, the input u and output y of a physical device is passively monitored to identify stimuli indices, I. In this scenario, we use the log output of `RFID Reader 1` and the network traffic between `HMI 1` and `PLC 1` as sources for the stimuli indices identification.

4.2 Architecture

As shown in Figure 3, the overall architecture can be divided into the digital-twin framework *CPS Twinning* [7] and its replication

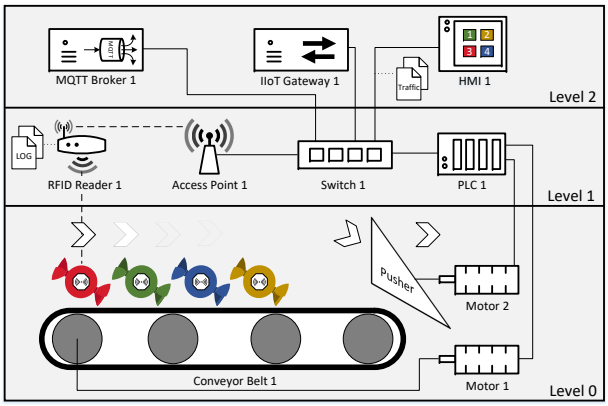


Figure 2: Test bed used for evaluation

mode, named *CPS State Replication*. In the following subsections, each component of CPS State Replication will be explained in detail.

4.2.1 Data Sources. As the inputs, U, and outputs, Y, of a physical device must be retrieved in a passive manner, pre-existing data sources such as (i) system logs, (ii) network traffic, and (iii) sensor measurements can be used. Depending on the nature of the physical device, relying only on one data source for obtaining stimuli may be insufficient. In this case, combining multiple data sources is required. However, if multiple data sources are used in conjunction and the sets of recorded inputs or outputs overlap, duplicates must be filtered out before the data is passed onto the ingestion component.

4.2.2 Ingestion. The ingestion component is in charge of taking unstructured data that originated from the physical or virtual environment, transforming it into a consistent format and then transmitting the output of the transformation process to the collection component. The ingestion of data (i.e., members of U^*, Y^*, $\hat{U}^*$ and $\hat{Y}^*$, where $\hat{U}^*$ and $\hat{Y}^*$ represent the inputs and outputs of *all* digital twins, respectively) is supported in two ways: offline and online. In offline mode, the ingestion is performed in a static manner, signifying that inputs and outputs are processed file-based. In contrast, online ingestion refers to the continuous processing of streamed data. Thus, online ingestion is the preferred mode for consistently maintaining synced states with the physical environment. On the other hand, the offline mode may facilitate the execution of tests performed in the virtual environment, as the same states can be replicated to digital twins any number of times.

It is also worth highlighting that the ingestion is performed in a decentralized manner, allowing to balance the load of incoming data.

4.2.3 Collection. This component consists of a distributed messaging system that transports data between the ingestion, stimuli-processing and intrusion detection component. In addition, the collection component transfers the identified stimuli to CPS Twinning in order to replay them in the virtual environment.

Upon receiving data from the ingestion component, the transmitted messages are categorized based on the type of data source (e.g., network traffic) and the environment they originated from (e.g., physical environment). Besides incoming messages from the ingestion component, stimuli and intrusion detection results are also collected accordingly, ensuring that they are distributed to respective consumers.

4.2.4 Specification & Parser. The specification of the CPS represents a key component in CPS Twinning and is reused for the identification of stimuli in CPS State Replication. Ideally, the specification has been created in the course of the engineering phase and is then maintained throughout the CPS's lifecycle. In the proposed specification-based state replication approach, the stimuli identification is performed based on topological data, meaning that engineering artifacts must precisely describe the hierarchy and properties of the plant's components. Since the engineering data format AutomationML (AML) [5] is specifically designed to model the plant's structure, it is the data format of choice for state replication.

The parser transforms the specification of the CPS to S^*, I, f and passes them onto the stimuli-processing component. As discussed in Section 3.2, building the sets S^* and I is achieved by interpreting the characteristics of physical devices and by analyzing the specified data flows, defined in the logical network. One way to infer the characteristics of a device (e.g., receives user input, transmits sensor readings) is by interpreting its role definition. For instance, in AML, instances (e.g., a concrete HMI) can be associated with roles by using the `RoleRequirements` element. On the other hand, the data flows can be defined by using the notion of a logical network (cf. Listing 1), i.e., a model of the network that considers OSI layers 3–7 [1]. Each element of the logical network (e.g., request) is then analyzed based on the device's characteristics in order to determine whether it represents an indication of a stimulus, $i \in I$. If the element of the logical network can indeed be represented as a member of I, the corresponding stimulus is constructed. The construction of the stimulus is determined by the set membership of i. In particular, if $i \in U^*$ is true, then i can be replicated in the virtual environment as it is ($i = s$, where $s \in S^*$). However, if $i \in Y^*$ is true, then the stimulus must be created from scratch, for example by making appropriate calls to the digital-twin framework's API. As a result, the sets S^* and I can be built based on the inferred device characteristics and the data flows. The partial function f is computed in consequence of the set building, as it maps stimuli indices to stimuli.

4.2.5 Stimuli Processing. The stimuli-processing component receives inputs and outputs of devices from the collection component and checks for each incoming message, $j \in U^* \cup Y^*$, whether $j \in I$. If j is indeed a member of I, then $f(j)\downarrow$ and the partial function f of j evaluates to $s \in S^*$. The output of $f(j)$, i.e., the stimulus, is then transmitted to CPS Twinning through the collection component in order to replicate it in the virtual environment.

4.2.6 Intrusion Detection. Assuming that the specification of the CPS precisely describes the correct behavior and each digital twin follows the state of its physical counterpart, a divergence between the physical and virtual environment should be detectable. Based on this assumption, comparing the inputs and outputs of the physical environment with those of the virtual one may reveal either device faults or malicious activity. Supplementary, the states of digital

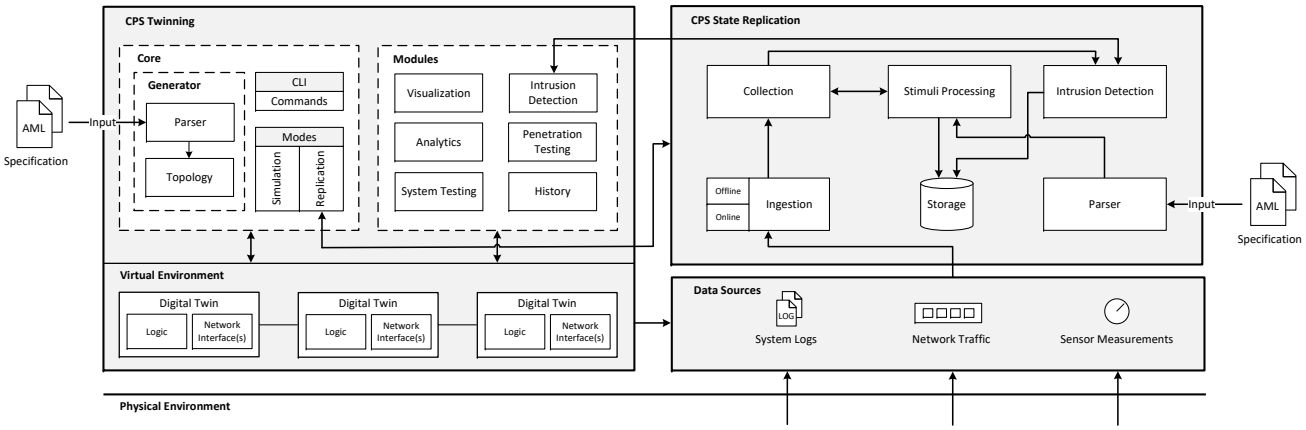


Figure 3: Architecture of CPS Twinning [7] and CPS State Replication

twins can also be factored into the comparison in order to consider physical process states. Thus, the intrusion detection component processes elements of the sets U^*, Y^*, $\hat{U}^*$, $\hat{Y}^*$ and $\hat{X}^*$, where $\hat{X}^*$ is the set of states of *all* digital twins.

The comparison between $p \in U^* \cup Y^*$ and $v \in \hat{U}^* \cup \hat{Y}^*$ relies on predefined features. To give an example, Modbus TCP/IP network traffic from the physical and virtual environment could be compared based on (i) source & destination MAC addresses, (ii) source & destination IP addresses, and (iii) Modbus TCP/IP ADUs. If the comparison based on the selected features yields a mismatch, the IDS raises an alarm to alert users of potential intrusions. It is also worth mentioning that the selection of features significantly impacts the intrusion detection performance (e.g., false positive rate), as some characteristics of p and v may vary. This could be caused, for example, by differences in the implementation (e.g., network stack) of physical devices and digital twins [7].

The beauty of this intrusion detection approach is that it enables automatic in-depth checks of the CPS during operation that reveal whether the behavior of the CPS conforms to the defined legitimate behavior, without causing any risks of interference with live systems.

However, we also identified two drawbacks that are inherent to our approach. First, it must be ensured that the specification indeed describes the correct behavior of the CPS. In particular, if engineering artifacts have been modified with malicious intent before they are passed onto the generation step done by the digital-twin framework, then the comparison would be to no avail. As a result, the mitigation of insider threats and the protection of the engineering process is of utmost importance. Second, if the digital twins receive malicious inputs to which they are similarly vulnerable as their physical counterparts, the comparison used for intrusion detection would not yield any differences. However, this risk may be mitigated by continuously performing safety and security checks on digital twins based on additional rules defined in the specification [7] or by employing dynamic malware analysis techniques [8]. Since the execution of digital twins can be thoroughly analyzed without negatively affecting the operation of live systems, both mitigation strategies are applicable.

The main limitation of this approach is that we rely on the assumption that the digital twins model the correct behavior and that we can passively obtain the genuine inputs of the real devices.

4.2.7 Storage. The storage component consists of one or multiple databases that are used to store the input (i.e., elements of the sets U^*, Y^*, $\hat{U}^*$, $\hat{Y}^*$) and output (i.e., elements of the set S^*) of the stimuli-processing component as well as intrusion detection logs. Persisting stimuli would enable users to recover historical states of digital twins and resume the digital twins' execution in simulation mode. In this way, the behavior of digital twins can be thoroughly analyzed over time. Such a "history" feature would also allow users to test security measures in an isolated environment, yet with states that the real CPS had in the past. Additionally, the stored data could be used for other non-security related applications (e.g., analytics).

4.3 Implementation

To evaluate how the proposed state replication approach can be realized in practice, we implemented all components of CPS State Replication (cf. Section 4.2), except the storage component, as we leave the "history" use case for future work.

The source code of CPS State Replication[2] and CPS Twinning[3] is available on GitHub.

4.3.1 CPS Twinning. As already mentioned, we used CPS Twinning [7] as a basis, since it already supports the generation and execution of digital twins in a virtual environment. CPS Twinning uses Mininet [17] to emulate the network layer [7]. However, Mininet does not yet support the virtualization of wireless networks, meaning that we cannot use it to generate the digital twins for wireless devices, viz., `RFID Reader 1` and `Access Point 1` (cf. Section 4.1). Since Mininet does not fit our needs, we adopted Mininet-WiFi [9], a fork of Mininet that allows the virtualization of stations and access points.

[2]https://github.com/sbaresearch/cps-state-replication
[3]https://github.com/sbaresearch/cps-twinning

Furthermore, we seamlessly integrated our replication mode into the framework in order to ensure that individuals can use it via the command-line interface (CLI) as any other module. Yet, we aimed to design the replication mode in CPS Twinning as lightweight as possible. The reason for this design choice is to reduce the stimuli-processing load on the digital-twin framework by running CPS State Replication as a self-contained application, ideally in a distributed manner. As a result, we implemented only the stimuli retrieval from the collection component and the algorithm to replicate them in the virtual environment in CPS Twinning. The replication mode in CPS Twinning was implemented in Python.

When users activate the replication mode by issuing the command `start_replication` via the CLI in CPS Twinning, two threads are started, one for enqueueing retrieved stimuli and one for replicating the dequeued stimuli in the virtual environment. The actual algorithm implemented in CPS Twinning that is used for stimuli replication can be seen in Algorithm 1. The stimuli replication algorithm has two inputs: (i) a Boolean value to define the exit condition for a state mismatch, and (ii) a minimum priority queue to retrieve stimuli for replication. Since we use the timestamp of occurrence of each stimulus as a priority number, the oldest stimuli are retrieved first. If no stimulus has been replicated yet, the conditional expression in line 5 of Algorithm 1 is true, leading to the instantaneous replication of the first incoming stimulus. However, before the actual method for replicating the stimulus is called, the timestamp of the stimulus is stored in a variable (cf. Algorithm 1, line 7). For any subsequent stimuli, the statements in the `else` branch, starting in line 11 of Algorithm 1, are executed.

Recall that the timespan between the replication of two successive stimuli must equal to the timespan between these two stimuli observed in the physical environment (cf. Section 3). We attempt to fulfill this requirement by calculating the timespan for which the execution of the thread must be suspended in order to replicate the stimulus in due time. It is worth highlighting that for the calculation of the sleep time (cf. Algorithm 1, line 16), we always use the timestamp when the initial stimulus was replicated, instead of the timestamp when the most recent one was replicated. The reason for this lies in the fact that the point in time at which the stimulus is replicated should always match the point in time at which it had been observed in the physical environment, whilst taking the inevitable delay into account that is caused by CPS State Replication when replicating the first stimulus.

Due to the distributed nature of CPS State Replication, we cannot rely on a strict ordering of streamed stimuli. For example, the thread, which is used for receiving stimuli from the collection component, may enqueue a stimulus that has an older timestamp than the one that currently awaits replication. In this case, the current stimulus replication must be prematurely interrupted by resuming the execution of the algorithm, enqueuing the stimulus again and skipping the actual replication (cf. Algorithm 1, line 19–21).

4.3.2 CPS State Replication. The actual implementation of the ingestion, collection and stimuli-processing component heavily relies on open-source software. In fact, the ingestion and collection component were realized by leveraging the ready-made tools Apache Flume and Apache Kafka, respectively. Consequently, there was more work involved in configuring both platforms, than actual programming effort. Yet, the parser, stimuli processing and intrusion detection components were implemented based on Apache Spark in Scala.

Algorithm 1: Stimuli replication algorithm in CPS Twinning

Input: A Boolean *bailOut*, a minimum priority queue *pq*

```
1  s_0 ← NULL                    // first replicated stimulus
2  t_0 ← 0    // timestamp of first replicated stimulus
3  s_{i-1} ← NULL                // last replicated stimulus
4  while state replication mode is active do
5      if s_0 = NULL then
6          s_0 ← Dequeue(pq)
               // may block until stimulus is available
7          t_0 ← GetTimestamp()
8          ReplicateStimulus(s_0)
9          s_{i-1} ← s_0
10     else
11         s_i ← Dequeue(pq)
               // may block until stimulus is available
12         Δ_{s_{i-1}} ← GetTimestamp(s_i) − GetTimestamp(s_{i-1})
13         if Δ_{s_{i-1}} < 0 and bailOut = true then
14             break                    // state mismatch
15         Δ_{s_0} ← GetTimestamp(s_i) − GetTimestamp(s_0)
16         t_sleep ← (t_0 + Δ_{s_0}) − GetTimestamp()
17         if t_sleep > 0 then
18             f ← Sleep(t_sleep)
                       // true, if wake up was forced
19             if f = true then
20                 Enqueue(pq, s_i)
21                 continue
22         else if t_sleep < 0 then
               // state replication delay of t_sleep * −1
23         ReplicateStimulus(s_i)
24         s_{i-1} ← s_i
```

Data Sources, Ingestion & Collection. As already discussed in Section 4.1, we use the log output of the `RFID Reader 1` as well as the network traffic between `PLC 1` and `HMI 1` as data sources for state replication. When the RFID reader detects a candy, the candy's type (e.g., cherry) is printed to the serial port. To ingest these log messages, we implemented a Python program that reads the messages via the RFID reader's serial port and publishes them to a Kafka topic. On the other hand, the network traffic is first captured by use of tshark and then ingested through a Kafka sink in Flume. To implement the offline and online mode of this ingestion instance, we configured two sources in Flume. For the offline mode, we set up a spooling directory that ingests tshark captures in the form of JSON files. In contrast, the online mode was realized by configuring a source of type `exec`, which allows to specify a Unix command (in our case tshark) that is executed on start-up of Flume. It is also worth mentioning that we implemented a Flume interceptor that

adds the device's or digital twin's identifier to the header of Kafka messages in order to perform semantic partitioning of network traffic data. In this way, multiple instances of the stimuli-processing component can consume Kafka messages in parallel.

Besides the log messages of the RFID reader and the network traffic between the PLC and the HMI, we also use the output of a sensor, which detects ejected candies, and the network traffic between the RFID reader and the MQTT broker as data sources. Although both data sources serve no purpose for state replication, they significantly enhance our intrusion detection approach (cf. Section 4.4.2). While the candy sensor provides valuable details about the candy ejection process (i.e., time of ejection, candy's type), the network traffic between `RFID Reader 1` and `MQTT Broker 1` may reveal attacks that targeted the detection of candies.

The candy sensor relies on a video stream to detect candies based on the color of their wrapping. The program used to process the video stream was implemented in Python with OpenCV.

Parser, Stimuli Processing & Intrusion Detection. The parser traverses the instance hierarchy defined in the AML file and extracts the data required to build f, i.e., a data structure that maps stimuli indices (consisting of specific inputs and outputs) to stimuli. Similar to [7], the AML parser has been specifically designed for the scenario at hand (cf. Section 4.1). Consequently, any modifications to the scenario and, in further consequence, to the specification, may require adaptations to the AML parser.

After parsing the specification, a streaming client is instantiated that subscribes to the Kafka topics used for stimuli processing and intrusion detection. When the Spark application receives new messages from Kafka, the incoming dataset is split into subsets by filtering based on the messages' topic. After the streamed data has been partitioned, only the two datasets (network traffic and system logs) that contain messages, which originated from the physical environment, are used for stimuli processing. By contrast, the datasets, which contain network traffic, system logs and sensor measurements for both the phyiscal and virtual environment, are passed onto the intrusion detection system. Owing to the concept of resilient distributed datasets (RDDs) [31] that is implemented in Spark, the stimuli processing and the intrusion detection is executed in parallel.

The stimuli processing was implemented as follows: In the first step, the messages of the two datasets are unmarshalled into instances of Scala classes, which have been modeled according to the types of inputs and outputs (e.g., Modbus TCP/IP packet). Next, each input and output object is used to determine whether the partial map (i.e., f) transforms it to a stimulus. Finally, the retrieved stimuli are marshalled and then published to the corresponding Kafka topic.

As already mentioned, the intrusion detection system processes messages from all subscribed Kafka topics. Yet, we limited the comparison between the physical and virtual environment to MQTT publish requests and candy sensor measurements. Although the proposed intrusion detection technique would enable a more comprehensive comparison, we wanted to measure the intrusion detection performance with a minimal amount of features used. In particular, we selected (i) the source/destination MAC address, (ii) the source/destination IP address, (iii) the MQTT header flags, (iv) the MQTT payload, and (v) the detected candy type (coming from the candy sensor) as features for the comparison. In case of detected differences between the physical and virtual environment, the compared data, which was captured from the physical environment, is logged to the console in order to warn users.

4.3.3 Known Limitations. The presented prototype has several limitations that result from differences between the implemented logic layer of digital twins and those of physical devices. Since we use CPS Twinning for the generation of digital twins, the extended version of the digital-twin framework suffers from the same drawbacks as those reported by Eckhart and Ekelhart [7]. For instance, we had to port parts of the program running on `PLC 1` to structured text (ST). In particular, vendor-specific blocks (e.g., motor control) were mocked in ST. In addition, we had to set the scan time of `PLC 1` to 50 ms, due to performance issues.

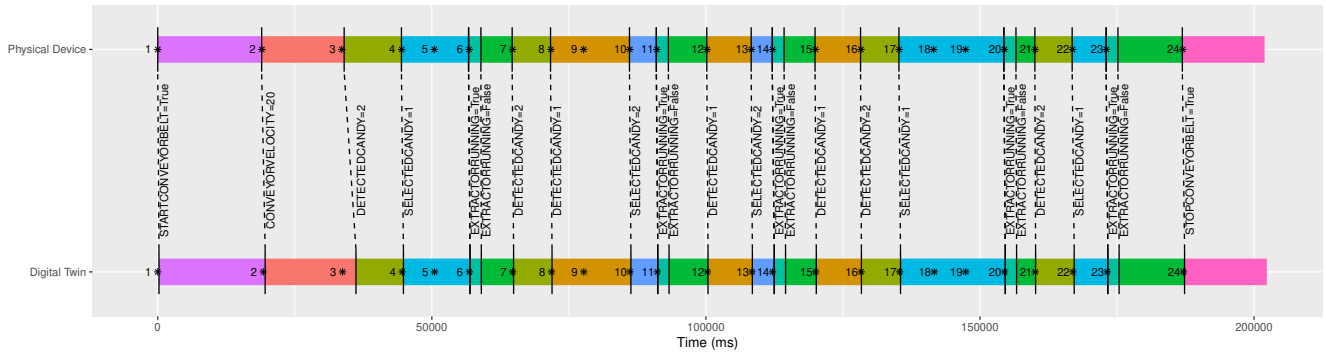
4.4 Evaluation

We aim to evaluate the prototype and the proposed state replication approach by performing two types of experiments. In the first type of experiments, we try to determine the accuracy of the state replication approach, which is measured by the time differences between state transitions of the physical and virtual environment. The second type of experiments aim to demonstrate the effectiveness of the intrusion detection system.

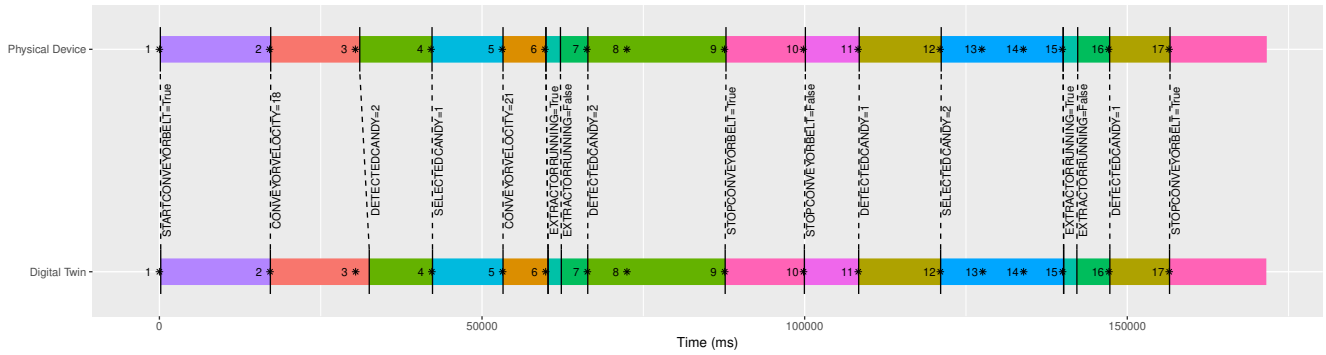
4.4.1 State Replication Accuracy. To determine the time differences between the state transitions of a physical device and those of its digital twin, we configured a logger on the physical device and implemented a "state logging" module in CPS Twinning. In particular, we added a logger function block to the program running on `PLC 1` that logs all program variables every 100 ms to a CSV file. On the other hand, when activating the "state logging" module of CPS Twinning, the state transitions of all digital twins are captured. In this way, we were able to track the sequence of states and the timestamp of state transitions for both the digital twin and its physical counterpart.

Two experiments based on the scenario discussed in Section 4.1 were conducted. Although both experiments were performed with the same ICS test bed, they are completely independent from each other, meaning that the sequence of stimuli (i.e., inputs of the HMI and RFID reader) varies. Figure 4a and 4b each depict two timelines, showing how the values of program variables of the physical PLC and its digital twin change over the course of time. In both figures, the stimuli are marked with asterisks. It is also worth mentioning that the candy types are internally mapped to integers — namely, the value 1 indicates a candy with the flavor cherry and the value 2 a mint-flavored candy. Furthermore, the colored segments only show the elapsed time until *any* variable change occurred. As a result, there are stimuli marked on both timelines of Figure 4a and 4b that do not change any variables' value. For example, in Figure 4a, the 18^{th} and 19^{th} stimulus represents an input that originated from the RFID reader, detecting a candy with the flavor mint. Since a candy with the same flavor has been detected before (16^{th} stimulus), the variable's value does not change.

In the first experiment (cf. Figure 4a), 24 stimuli were replicated in total. The timespan between the first observed/replicated stimulus



(a) Experiment 1



(b) Experiment 2

Figure 4: Sequences of changes of PLC program variables from the first (a) and second (b) experiment[4]

and the first variable change is 3 ms in the physical environment and 220 ms in the virtual. The state replication delay, i.e., the timespan between the first observed stimulus in the physical environment and the first replicated stimulus in the virtual environment, was 11.701 seconds. Furthermore, we measured the differences between the intervals of a program variable change on the physical device and its digital twin. The first quartile is 22 ms, the median is 98 ms, and the third quartile is 149 ms. The minimum and maximum interval deviations are 2 ms and 1789 ms, respectively.

As can be seen in Figure 4b, 17 stimuli were replicated in the second experiment. In this experiment, the timespan between the first stimulus and the first variable change amounts to 114 ms in the physical and 207 ms in the virtual environment. Furthermore, the state replication delay measured in the second experiment is 10.8 seconds. In the second experiment, we measured slightly higher deviations between the intervals of a program variable change on the physical device and the corresponding digital twin than in the first experiment. In this run, the first quartile is 63 ms, the median is 114 ms, and the third quartile is 198 ms. Moreover, the minimum and maximum interval deviations are 4 ms and 1521 ms, respectively.

It is worth pointing out that certain intervals between changes to the digital twin's program variables were shorter than the respective intervals measured on the physical device. Due to a lack of a physical model of the pusher, we mocked the block that controls the motor of the pusher with a timer on delay (TON) instruction and hard coded a value of 1961 ms for the timer (cf. Section 4.3.3). Since this value represents the average timespan between `EXTRACTORRUNNING=True` and `EXTRACTORRUNNING=False` (i.e., the pusher is active) of only 5 runs, this part of the digital twin's program inherently introduces inaccuracies. Interestingly, when replicating the third stimulus in the first and second experiment (cf. Figure 4a and 4b), we measured a considerable difference of 1517 ms and 1521 ms, respectively, between the program variable change (`DETECTEDCANDY=2`) observed on the physical device and its digital twin. Although it is not entirely clear from the data that we obtained what caused the delay, we assume that the initial connection to `MQTT Broker 1` or `IIoT Gateway 1` in the virtual environment consumes more time than any connections that were subsequently established.

In both experiments, CPS Twinning and CPS State Replication were executed on a single machine. Accordingly, we we would expect a decrease in the state replication delay, if CPS State Replication is deployed to a multi-node cluster.

4.4.2 Detection of Attacks. We evaluated the effectiveness of the intrusion detection system by testing it against two types attacks: (i) a man-in-the-middle (MITM) attack, and (ii) an insider attack. Both attacks targeted the candy ejection process, i.e., ejecting a candy with a different flavor than selected by the user.

[4]Those `DETECTEDCANDY` labels that precede `EXTRACTORRUNNING=True` have been omitted for the sake of clarity.

```
14:04:55.178 - Count [pCandy=1,vCandy=1].
+------+
| candy|
+------+
|Cherry|
+------+
14:06:06.392 - Count [pMQTT=8,vMQTT=1].
+---------+---------+------------+------------+------------+------------+--------+-----------+--------+----------+--------+
|  eth.src|  eth.dst|      ip.src|      ip.dst|mqtt.msgtype|mqtt.dupflag|mqtt.qos|mqtt.retain|mqtt.len|mqtt.topic|mqtt.msg|
+---------+- -------+------------+------------+------------+------------+--------+-----------+--------+----------+--------+
|08:00:...|f8:1e:...|192.168.0.61|192.168.0.32|           3|           0|       0|          0|      11|     candy|    Mint|
...
|08:00:...|f8:1e:...|192.168.0.61|192.168.0.32|           3|           0|       0|          0|      11|     candy|    Mint|
+---------+---------+------------+------------+------------+------------+--------+-----------+--------+----------+--------+
```

Listing 2: Output of the IDS for attack 1

MITM Attack. For the MITM attack, our aim was to position the attacker between `RFID Reader 1` and `MQTT Broker 1` by launching an ARP spoofing attack. When the RFID reader detects a candy, the attacker intercepts the MQTT publish packet, changes the detected candy type in the packet's payload and sends it to the MQTT broker. In further consequence, the forged candy type is transmitted to the PLC, leading to the ejection of a wrong candy.

In this attack scenario, the user selects the flavor mint via the HMI in order to remove such a candy from the conveyor belt. Then, the RFID reader detects a candy of type cherry and attempts to transmit this data via MQTT to the MQTT broker. Yet, the attacker — as the man in the middle — intercepts the communication and changes the type of the detected candy to mint. As the variable values in the PLC program for the detected and selected candy match, the wrong candy with the flavor cherry is ejected.

Due to the fact that the IDS relies on the network traffic between the RFID reader and the MQTT broker as well as on the candy sensor log output, the attack has left its traces in both data sources. Besides the varying detected candy type, the amount of MQTT publish packets observed in the physical environment differs from that observed in the virtual environment. As can be seen from the output of the IDS depicted in Listing 2, eight MQTT publish packets were observed in the physical environment (cf. line 7, packet count for the physical and virtual environment is denoted by `pMQTT` and `vMQTT`, respectively), due to retransmissions caused by the ARP spoofing attack. In contrast, only one MQTT publish packet was captured in the virtual environment.

Insider Attack. The second attack emanates from an insider. An engineer modifies the ejection logic in the PLC program with malicious intents and transfers it to the device via a memory card. Thus, the attacker leaves no traces in the network traffic.

This time, the user selects the flavor cherry on the HMI. After the selection, the RFID reader detects a candy of type cherry. However, the candy ejection process has not been initiated by the PLC, due to malicious changes in the PLC logic. Instead, upon detecting a mint-flavored candy, the ejection is executed. Since the attacker was not able to adapt the specification of the PLC accordingly, the PLC digital twin still reflects the correct behavior. Thus, the correct candy (i.e., a candy of type cherry) is ejected virtually, leading to the discovery of the intrusion. Listing 3 shows that both the physical and virtual candy sensor detected one candy (denoted by `pCandy` and `vCandy`, respectively), yet the output of the real, physical candy sensor indicates that a mint-flavored candy was ejected, instead one with a cherry flavor.

```
15:07:21.065 - Count [pCandy=1,vCandy=1].
+-----+
|candy|
+-----+
| Mint|
+-----+
```

Listing 3: Output of the IDS for attack 2

5 CONCLUSIONS

In this paper, we have presented an approach to replicate program states from physical devices to their digital twins. State transitions of physical devices can be detected by passively monitoring their inputs and outputs. Based on these passive data sources and the system's specification, we deduce stimuli and replicate them in a virtual environment. Since the digital twins execute equivalent versions of the programs running on their physical counterparts, we expect them to exhibit an identical behavior. The digital twins enable a detailed inspection of otherwise hidden states and, furthermore, highlight deviations by comparing them to the behavior of their real, physical counterparts.

While the results of this work are promising, there are still some challenges to be addressed to improve the state replication approach. For example, depending on the nature of the device or caused by inaccuracies of the specification, inferring the device's characteristics based on the role definition may be infeasible. In this case, users are required to either add details about stimuli or explicitly label inputs as stimuli. Another possible option, yet more complex to implement, may be to develop a code analyzer that automatically inspects all programs, which are referenced in a CPS's specification, for the purpose of stimuli identification. Furthermore, as our approach focuses only on states of software programs, support for replicating stimuli that manifest themselves in the form of analog signals is lacking. Thus, analog values that constitute stimuli must

be converted to a binary representation, before they can be replicated in the virtual environment. However, considering that CPSs typically consist of analog-intensive components, further research in this area is worth pursuing.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital, Business and Enterprise and the National Foundation for Research, Technology and Development, and COMET K1, FFG - Austrian Research Promotion Agency is gratefully acknowledged. Furthermore, this work was supported by the Austrian Science Fund (FWF) and netidee SCIENCE under grant P30437-N31.

REFERENCES

[1] AutomationML. 2014. *Whitepaper: Communication.* Technical Report V_1.0.0. AutomationML consortium.

[2] Radhakisan Baheti and Helen Gill. 2011. Cyber-physical systems. *The impact of control technology* 12 (2011), 161–166.

[3] A. Carcano, A. Coletta, M. Guglielmi, M. Masera, I. Nai Fovino, and A. Trombetta. 2011. A Multidimensional Critical State Analysis for Detecting Intrusions in SCADA Systems. *IEEE Transactions on Industrial Informatics* 7, 2 (May 2011), 179–186. https://doi.org/10.1109/TII.2010.2099234

[4] Justyna Joanna Chromik, Anne Katharina Ingrid Remke, and Boudewijn R.H.M. Haverkort. 2016. *What's under the hood? Improving SCADA security with process awareness.* IEEE. https://doi.org/10.1109/CPSRSG.2016.7684100

[5] R. Drath, A. Luder, J. Peschke, and L. Hundt. 2008. AutomationML - the glue for seamless automation engineering. In *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. 616–623. https://doi.org/10.1109/ETFA.2008.4638461

[6] David Duggan, Michael Berg, John Dillinger, and Jason Stamp. 2005. Penetration testing of industrial control systems. *Sandia National Laboratories* (2005).

[7] Matthias Eckhart and Andreas Ekelhart. 2018. Towards Security-Aware Virtual Environments for Digital Twins. In *Proceedings of the 4th ACM Workshop on Cyber-Physical System Security (CPSS '18)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/3198458.3198464

[8] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.* 44, 2, Article 6 (March 2008), 42 pages. https://doi.org/10.1145/2089125.2089126

[9] R. R. Fontes, S. Afzal, S. H. B. Brito, M. A. S. Santos, and C. E. Rothenberg. 2015. Mininet-WiFi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*. 384–389. https://doi.org/10.1109/CNSM.2015.7367387

[10] I. N. Fovino, A. Carcano, T. D. L. Murel, A. Trombetta, and M. Masera. 2010. Modbus/DNP3 State-Based Intrusion Detection System. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 729–736. https://doi.org/10.1109/AINA.2010.86

[11] Hamid Reza Ghaeini, Daniele Antonioli, Ferdinand Brasser, Ahmad-Reza Sadeghi, and Nils Ole Tippenhauer. 2018. State-Aware Anomaly Detection for Industrial Control Systems. In *The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC)*. https://doi.org/10.1145/3167132.3167305

[12] Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H. Hartel. 2014. Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 126–135. https://doi.org/10.1145/2664243.2664277

[13] William Jardine, Sylvain Frey, Benjamin Green, and Awais Rashid. 2016. SENAMI: Selective Non-Invasive Active Monitoring for ICS Intrusion Detection. In *Proceedings of the 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy (CPS-SPC '16)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/2994487.2994496

[14] Henning Kagermann, Wolfgang Wahlster, and Johannes Helbig. 2013. *Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0 – Securing the Future of German Manufacturing Industry.* Final Report of the Industrie 4.0 Working Group. acatech – National Academy of Science and Engineering, München.

[15] M. Krotofil and D. Gollmann. 2013. Industrial control systems security: What is happening?. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*. 670–675. https://doi.org/10.1109/INDIN.2013.6622964

[16] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[17] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. ACM, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466

[18] D. G. Luenberger. 1964. Observing the State of a Linear System. *IEEE Transactions on Military Electronics* 8, 2 (April 1964), 74–80. https://doi.org/10.1109/TME.1964.4323124

[19] T. Macaulay and B.L. Singer. 2016. *Cybersecurity for Industrial Control Systems: SCADA, DCS, PLC, HMI, and SIS.* CRC Press.

[20] S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A. R. Sadeghi, M. Maniatakos, and R. Karri. 2016. The Cybersecurity Landscape in Industrial Control Systems. *Proc. IEEE* 104, 5 (May 2016), 1039–1057. https://doi.org/10.1109/JPROC.2015.2512235

[21] Bill Miller and Dale Rowe. 2012. A Survey of SCADA and Critical Infrastructure Incidents. In *Proceedings of the 1st Annual Conference on Research in Information Technology (RIIT '12)*. ACM, New York, NY, USA, 51–56. https://doi.org/10.1145/2380790.2380805

[22] Robert Mitchell and Ing-Ray Chen. 2014. A Survey of Intrusion Detection Techniques for Cyber-physical Systems. *ACM Comput. Surv.* 46, 4, Article 55 (March 2014), 29 pages. https://doi.org/10.1145/2542049

[23] Andrew Nicholson, Helge Janicke, and Antonio Cau. 2014. Position Paper: Safety and Security Monitoring in ICS/SCADA Systems. In *Proceedings of the 2nd International Symposium on ICS & SCADA Cyber Security Research 2014 (ICS-CSR 2014)*. BCS, UK, 61–66. https://doi.org/10.14236/ewic/ics-csr2014.9

[24] Jeyasingam Nivethan and Mauricio Papa. 2016. A SCADA Intrusion Detection Framework That Incorporates Process Semantics. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference (CISRC '16)*. ACM, New York, NY, USA, Article 6, 5 pages. https://doi.org/10.1145/2897795.2897814

[25] Roland Rosen, Georg von Wichert, George Lo, and Kurt D. Bettenhausen. 2015. About The Importance of Autonomy and Digital Twins for the Future of Manufacturing. *IFAC-PapersOnLine* 48, 3 (2015), 567 – 572. https://doi.org/10.1016/j.ifacol.2015.06.141 15th IFAC Symposium onInformation Control Problems inManufacturing.

[26] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[27] Jill Slay and Michael Miller. 2008. Lessons Learned from the Maroochy Water Breach. In *Critical Infrastructure Protection*, Eric Goetz and Sujeet Shenoi (Eds.). Springer US, Boston, MA, 73–82.

[28] Prem Uppuluri and R. Sekar. 2001. *Experiences with Specification-Based Intrusion Detection.* Springer Berlin Heidelberg, Berlin, Heidelberg, 172–189. https://doi.org/10.1007/3-540-45474-8_11

[29] David I. Urbina, Jairo Giraldo, Alvaro A Cardenas, Junia Valente, Mustafa Faisal, Nils Ole Tippenhauer, Justin Ruths, Richard Candell, and Henrik Sandberg. 2016. *Survey and new directions for physics-based attack detection in control systems.* Technical Report. NIST. https://doi.org/10.6028/nist.gcr.16-010

[30] David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. 2016. Limiting the Impact of Stealthy Attacks on Industrial Control Systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1092–1105. https://doi.org/10.1145/2976749.2978388

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2.