Enhancing the Resiliency of Cyber-Physical Systems with Software-Defined Networks

Luis E. Salazar
UC Santa Cruz
luedsala@ucsc.edu

Alvaro A. Cardenas
UC Santa Cruz
alvaro.cardenas@ucsc.edu

ABSTRACT

Numerous research efforts have focused on intrusion detection in industrial control networks, however, few of them discuss what to do after an intrusion has been detected. Because the safety of most of these control systems is time-sensitive, we need new research on *automatic* incident response. In this paper, we extend our work on leveraging Software-Defined Networking (SDN) to automatically reconfigure an industrial control network to mitigate the impact of attacks, and to deceive adversaries.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; **Network security**; **Domain-specific security and privacy architectures**; • **Networks → Programmable networks**; • **Applied computing → Industry and manufacturing**.

KEYWORDS

Industrial Control Systems, Software-Defined Networks, Cyber-Physical Systems

ACM Reference Format:
Luis E. Salazar and Alvaro A. Cardenas. 2019. Enhancing the Resiliency of Cyber-Physical Systems with Software-Defined Networks. In *ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338499.3357356

1 INTRODUCTION

Industrial Control Systems (ICS) are responsible for operating various safety-critical infrastructures, such as power grids, water management, oil systems, and manufacturing. Recent events, like the blackout caused by the cyber-attack against the power-grid in Ukraine [9], show the dire need for improving the security of ICS.

Cybersecurity is a process that consists of (1) *protecting*, (2) *detecting*, and (3) *responding* to attacks [7]. Most of the literature on ICS security has focused on *preventing* and *detecting* attacks [3]; however, *responding* to attacks has received much less attention [1, 3]. In particular, most of the research papers focusing on intrusion detection for control systems do not discuss what to do after an attack has been detected [4].

CPS-SPC'19, November 11, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6831-5/19/11…$15.00
https://doi.org/10.1145/3338499.3357356

In this paper, we address this gap by continuing our work on intrusion response mechanisms to keep the control system operating safely while sustaining attacks. In particular, we extend our intrusion-response architecture that leverages Software-Defined Networking (SDN) to automatically reconfigure our attacked-network in real-time [12]. While our previous work focused on replacing compromised sensor and control messages to trustworthy sources, in this paper we focus on two new use-cases for network reconfiguration during an attack:

(1) We show how to create a honeypot of an industrial system so the attacker is rerouted to this honeypot.
(2) We show and work around the challenges of creating a honeypot when the attacker is already inside the industrial control network.

In addition to considering the new use-case of honeypots, this paper improves our previous open-source implementation of our ICS simulation, and the SDN response. In particular, we created several byproducts that can be of use to other researchers in Cyber-Physical Systems (CPS) security, and in general SDN security. Our new open-source software contributions in this work include,

(1) Our new design places the IDS in a secure network, implements a deep-packet inspection tool to extract the semantics of the ICS from the packets in the network, and improves the modularity of our design. The source code for all our SDN defenses is open source and available online: https://github.com/Cyphysecurity/ICS-SDN/tree/nids.
(2) We created a new Python API to interface with the ONOS SDN controller programmatically. Our tool is available online at https://github.com/Cyphysecurity/ICS-SDN/blob/nids/nids/pyonos.py.
(3) We also created a Web interface to manage our SDN network and our ICS emulation. Our new open-source tool is available at https://github.com/Cyphysecurity/ICS-SDN/blob/nids/nids/sdnidswebui.py.

The remainder of this paper is organized as follows: in Section 2 we discuss the background of our proposal and related work. As our solution is heavily dependent on SDN, we present an overview of this concept and the most relevant features that we use as part of our solution in Section 3. We then present the environment we developed for the ICS model to perform the tests in Section 4; in particular, we discuss the general setting, threat model, detection scheme, and design choices. We then propose the migration of an attacker from an existing network to an ICS honeypot in Section 5. Finally, we show the results of our solution in Section 6.

2 RELATED WORK

Our work is related to three lines of research: (1) honeypots for CPS, (2) SDN for managing security properties of a system, and (3) the visibility of industrial attacks.

The application of honeypots in CPS is a growing area of research. Irvene et al. present an example of a honeypot for a robotic vehicle [8], where they fool an attacker into believing that a given attack was successful by simulating unsafe actions within a honeypot environment of the protected robotic system. For ICS, Rubio et al. [13] present, among other intrusion detection solutions, a commercial use case of a honeypot solution developed for the acquisition and analysis of information related to a threat or attack against an ICS. Another relevant example developed under the Honeynet project is Conpot[1]: a low interactive server-side ICS honeypot which provides a range of common industrial control protocols. While Conpot presents examples of industrial systems to attackers, a sophisticated attacker can identify the honeypot by observing that the system does not satisfy the traditional physical properties of devices in a typical industrial setting. To further enhance the realism of the simulated system, Litchfield et al. present HoneyPhy, a honeypot that models the physics of the devices (e.g., delays) [10].

In addition to honeypots, our work is related to SDN applied to CPS. Skowyra et al. [14] showed how a verifiably-safe SDN can be implemented as part of a CPS. With the versatility of such networks, new ideas emerge involving this concept of programmable networks as part of a CPS and, more precisely, as part of an ICS. Antonioli et al. developed MiniCPS, a research tool that leverages the flexibility of SDN in order to present a framework with which any researcher can simulate an entire ICS network corresponding to a physical model of a known system [2]. In our previous efforts we extended MiniCPS by implementing a Physics-Based Anomaly Detection (PBAD) system [5] to identify attacks, and then implemented an incident-response system that removed the compromised sensor or controller from the network using SDN and then rerouted the network traffic to either a virtual sensor to replace the compromised sensor values, or a redundant controller to replace the compromised device [12].

In this paper we extend our previous work [12] in several ways: (1) first, we extend our IDS in a way that it receives network packets from a mirror port in order to acquire the data values (in our previous implementation, the IDS obtained the values of the system from a shared database used in the simulation of the process), (2) we implement our system on an industry-supported (and state of the art) SDN controller called ONOS[2] (Previously we implemented our system in an academic SDN controller called POX), (3) following security best practices for operating an IDS, we move the IDS from the same network being monitored to another segmented network that cannot send packets to the network being monitored, (4) we modularize our system to enable flexibility, and also higher fidelity with real-world systems (our previous work was implemented all within a single virtual machine, while in this paper we have three different virtual machines, one running the SDN network and physical process, another the SDN controller, and another the IDS), (5) we make all of our implementation and contributions open-source and available online (as stated in the introduction), and (6) in this paper we focus on the migration of an attacker from the real system, to a honeypot (Previously, we focused on the implementation of the IDS and the response to sensor attacks and controller attacks, but did not look at how to deceive the adversary and migrate them seamlessly to a honeypot). Our work emphasizes the importance of providing a seamless transition from the real system, to the honeypot environment, so the attacker does not detect that it has been transferred from the real target to a fake system.

Finally, inspired by the work of Giraldo et al. [6], we implement the network packet captures at the field-layer of industrial networks so our system can detect more attacks.

[1] http://conpot.org
[2] https://onosproject.org

2.1 Scope of the document

As Rubio et al. mentioned [13], the development of an effective IDS for ICS is still an open problem with different proposed solutions and innovations. Different techniques and detection mechanisms have been implemented throughout the years and new solutions are being developed as the industry itself evolves. Our implemented IDS is based on our previous work [12], and in this paper we consider details of the IDS performance out of scope, as our goal is to design solutions once an IDS has raised an alert.

Before we discuss in detail our system, we need some background on software-defined networks.

3 SOFTWARE-DEFINED NETWORKS

In the traditional networking switch architecture, the assorted functions that a device must fulfill are segregated into three main categories represented as layers or planes: management, control, and data. Peer communications occur in the same plane, whereas cross-category communications occur between planes. The most common interactions between these categories are illustrated in figure 1.

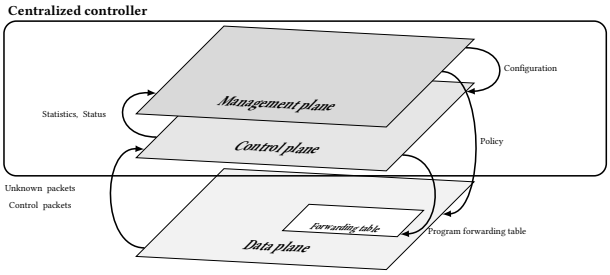


Figure 1: The main idea behind SDN is to have the control and management planes in a centralized controller.

The main idea behind SDN is to move all the software involving the decision-making process off network switches, and into a centralized system with the capability of optimizing decisions based on a complete overview of the managed network and not just a single device. Ultimately, SDN leaves the forwarding responsibilities to the device and centralizes the decision-making responsibilities within a controller.

In most scenarios, the vast majority of packets handled by an SDN switch involve the data plane, which takes care of packet buffering, scheduling, and forwarding. If the header information of any given packet is not recorded in the forwarding table, or the received packet is a control packet, these packets are forwarded to the control plane to obtain the corresponding table entry. The remaining features involve the interaction between the switch and

the administrator, which are handled by the management plane. In short, the raw data to be forwarded by the device is handled by the data plane, and the intelligence regarding the decision-making process is carried out by the control plane, based on the configuration made by the administrator in the management plane.

3.1 Southbound API

Since the decision-making process is being taken care of in a remote controller, there has to be a mechanism with which the network switches communicate with the controller. This communications channel is known as the southbound API, which handles the messages exchanged between the controller and the SDN devices to configure the network as a whole. The Open Networking Foundation developed OpenFlow as its standard southbound API and is supported by several manufacturers such as Cisco, Juniper, Huawei, Brocade, IBM, Dell, and HP, among others.

In addition to developing OpenFlow, the Open Networking Foundation published the OpenFlow switch specification [16], where they describe the actual requirements that a logical SDN switch must have to properly support OpenFlow as a southbound API. This includes the support of OpenFlow as the communications protocol between the switch and the controller, and the inclusion of the main components of an OpenFlow-compliant Switch. The main components of an OpenFlow logical switch can be summarized in a set of flow tables, a group table, and a set of OpenFlow channels with the controller.

By using OpenFlow as a southbound API, the controller can add, delete, or update flow entries within the tables in a logical device, both in a predefined manner or dynamically in response to a certain conditions in the packets.

At a high level, a flow table can be described as a set of flow entries. Each entry is comprised of match fields and a set of instructions or actions to carry out for each matching packet. This matching is carried out in priority order, starting from the first table within the switch and checking each subsequent table for a matching entry, taking into account the actual priority of every entry. If a matching entry is found, the instructions associated with that entry are followed. If there is no matching entry, the switch follows the instructions configured in the "table-miss" entry.

One important characteristic of a flow entry involves *actions* to be carried out for every matching packet (e.g. output port = x). However, the OpenFlow specification states that this action set can only contain one action of each type. This means that if the administrator wants to execute the same action type several times with different parameters for the same matching packet, one single entry is insufficient. This poses a problem since only the first matching entry will be executed, excluding any further actions contained in additional matching flow entries with lower priority.

To address this requirement, the OpenFlow specification defines the group table, which is comprised of group entries. The main idea behind group tables is depicted in figure 2. Every group entry contains an ordered list of action buckets, where each action bucket is essentially an action set of a regular flow entry. This allows an administrator to execute multiple action sets for a single matching packet. By stating a specific group as the action within the action set of a flow entry, the matching packet will be processed with the actions described in the action buckets of the corresponding group entry.

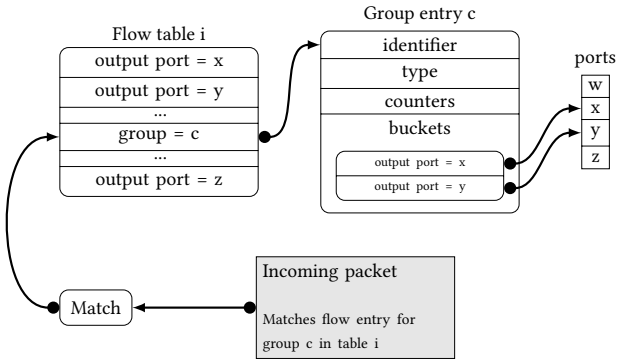


Figure 2: An example of a group table. An incoming packet that matches the entry on the flow table *i* that has an action set including the group *c* will instruct the switch to find the *c* entry within its group table and execute the actions defined in the buckets of the group entry. This allows the execution of multiple actions of the same type for a single matching packet.

3.2 Northbound API

In addition to having switches communicating with the controller, we need a way to let network applications reach the controller. The main idea behind the northbound API is to provide a mechanism with which an application can communicate with the controller to manage the network. The rationale behind this idea is that an application might handle a different set of information regarding the process implemented on top of a network that might not be directly related to network packets (e.g., sensor data, business analytics, market behavior, natural phenomena, etc.). With such information, different network decisions might be more effective towards the objective of the process, whatever it might be.

Due to the potential diversity of such applications, and the fact that the applications themselves might not be implemented within the controller, the latter must provide different mechanisms to fulfill the same requirements. That is where the northbound API comes into play.

Different controllers implement different types of northbound APIs. However, most controllers show a trend on how the APIs are implemented. For the most part, controllers implement at least two different APIs that achieve the same goals. The main API provides extensions written in the same programming language that the controller is built upon (i.e., Python API, Java API, etc.).

A secondary API is usually implemented as well, and it provides an external communications channel between an application and the controller via some specific protocol. For this communications channel, the northbound API is usually implemented by providing a REST API.

Regardless of the actual implementation of the northbound API, the goal is to provide a mechanism with which an application can make some method calls to the API, allowing it to reconfigure the SDN.

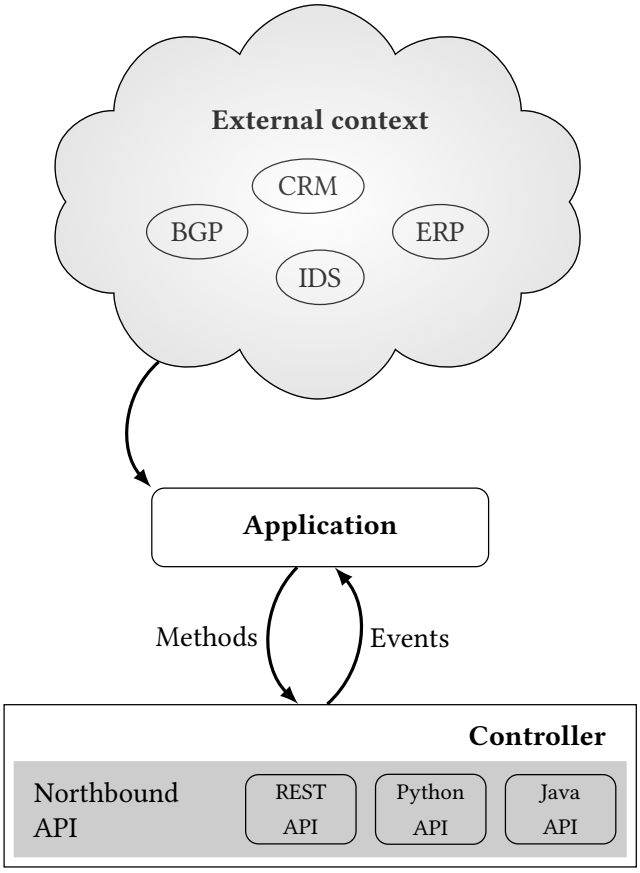


Figure 3: Generic northbound API communication scheme with an application.

3.3 SDN Controllers

There have been different types of SDN controllers for a variety of purposes. The first SDN controller that supported OpenFlow was NOX[3], an initial attempt by Nicira Networks (acquired by VMware) written in C++, which was later given to the community as an open-source project. Since then, several controllers supporting OpenFlow have spawned from this first attempt.

A direct successor from NOX is POX[4], an evolved Python version of NOX, intended to be used for prototyping and fast development of applications running within the controller. This was the SDN controller we used in our previous work [12]. The downside of this controller is its performance, a consequence of being developed in Python.

An early alternative to POX was Beacon[5], a Java-based SDN controller supporting OpenFlow and threaded operation. However, Beacon has not been in active development since its latest version, released on September 2013.

Originally forked from Beacon, Floodlight[6] formed the basis for commercial-grade SDN controllers. Its latest release was on March 2016, which indicates that it is not under active development.

Currently, the two most popular open-source SDN controllers are OpenDaylight[7] and ONOS[8]. The former is supported by the Open-Source Foundation, while the latter is supported by the Open Networking Foundation. While both controllers are Java-based and provide several desirable features, the fact that ONOS is being maintained by the same organization that defines the OpenFlow standard specifications guarantees seamless compatibility with the protocol. This difference can be appreciated within the performance variations between the two controllers. While both controllers are comparable, ONOS has been proven to out-perform OpenDaylight[15]. For these reasons we decide to implement our solution in ONOS.

3.4 Our use-case

We need to implement a way to give the IDS a copy of all packets in the field network of an ICS, and in addition we need to implement an incident-response application that can control (via the northbound API) the behavior of the ICS network.

Since the concept of grouping allows the packets to be sent through multiple ports of a switch, by placing an SDN device strictly to redirect all the traffic to a single passive interface in the monitor, group entries are then defined in order to acquire a copy of each packet sent through each port of each device within the SDN towards the IDS, effectively obtaining an entire copy of the network packets within the protected system.

Now that the IDS has an entire copy of the traffic, we can design the incident-response application. By using the northbound API, this incident-response application can alter the flow tables of the system, as needed, to protect it against a detected threat.

4 SYSTEM MODEL AND DESIGN CONSIDERATIONS

Our ICS is comprised of three main components: a physical process, a field network (a network between field devices and controllers), and a supervisory network (a network between controllers and SCADA systems) as seen in Figure 4. A secure architecture for ICS [11] suggests that these components should be placed in layers, so that ideally, an element within a certain layer can only communicate with elements of the adjacent layers when necessary. Other interactions between layers are possible but usually discouraged.

Since we are implementing a new SDN element within the ICS, this element must be placed accordingly. We conceive this element as the management component of all the network requirements of both the field network and the supervisory network. As such, it must interact, if needed, with SDN elements within these networks.

However, due to security concerns, and the actual role this element is going to be exerting over the scenario, direct communication between the components involved in the actual physical process and the SDN control network is not desired. Therefore,

[3]https://github.com/noxrepo/nox/
[4]https://github.com/noxrepo/pox
[5]https://openflow.stanford.edu/display/Beacon/Home
[6]http://www.projectfloodlight.org
[7]https://www.opendaylight.org/
[8]https://onosproject.org/

we take into consideration this constraint, and thus the SDN control network is placed in a new layer above both the field and the supervisory networks within the ICS architecture.

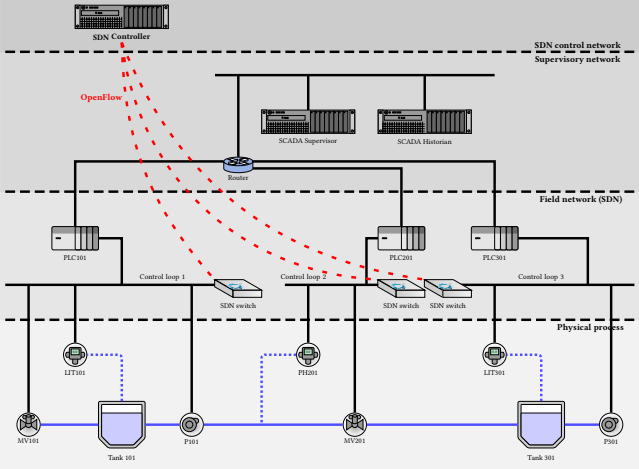


Figure 4: Our ICS has three PLCs, two tanks, two level sensors, one pressure sensor, and two flow actuators as well as two pump actuators.

Figure 4 depicts the scenario to be used as a general testbed for our IDS scheme, which extends our previous efforts on using SDN for incident response for ICS [12]. In this scenario, a physical process representing a water purification plant is simulated, and the process is then controlled by three different PLCs. The communications of all devices in the network is done with an SDN emulator.

4.1 Threat model

We assume the attacker has compromised a device in the field network. We assume that this compromise could have been achieved either externally or internally (Figure 5). The manner by which the attacker acquired access to the communications channel, and the privileges required to gain such access, are out of the scope of this document. It is assumed that s/he already executed such actions and gained the necessary privileges.

This attacker has the ability to communicate with the devices inside each control loop either directly or through some routing. Regardless of how it is established, the attacker has a communications channel between the attacker's location in the network and the target devices.

4.2 Proposed improvements over our previous work

This paper extends our previous work where we implemented an IDS on top of miniCPS [2] and then showed how to reconfigure the field network to survive sensor and controller attacks [12].

Improving the IDS: Our previous IDS was implemented by extending the functionalities of the existing Programmable Logic Controller (PLC) elements within miniCPS. The PLC components store all the variables of the physical process it simulates within

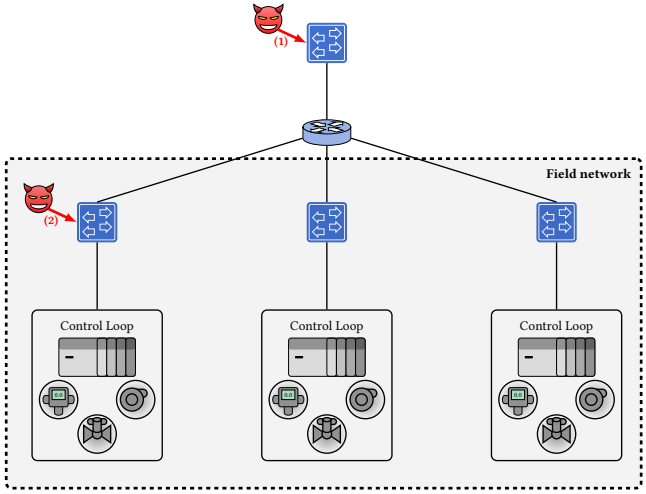
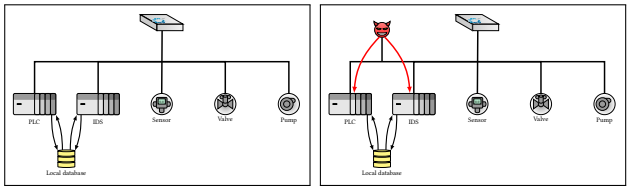


Figure 5: The threat model includes two different types of attacker. The first attacker contemplated by the model is an external attacker that gains access to the field network (1). The second one is an attacker located directly within the field network (2).

a local database. By extending this component, the IDS had direct access to the variables of each control loop stored within the local database. This previous implementation is depicted in figure 6a.



(a) Previous scenario. **(b) Compromised scenario.**

Figure 6: Previous IDS scheme within a control loop. The IDS is another component in the field network implemented as an extension of the PLC component. Both the PLC and the IDS can access the stored values in the local database.

However, if the field network itself is compromised as depicted in figure 6b, an attacker might have access to the IDS, and it may be compromised as well, defeating its purpose. To prevent such a scenario, we propose to remove the IDS from the field network. If the IDS is placed within a completely external, separate, machine outside the simulated environment, not only can this scheme prevent an attacker from gaining control of these security devices from within the field network, but also it will no longer be restricted to the simulated scenario, allowing it to be configured for a production environment. If the IDS is implemented externally, a single IDS can be used to oversee the entire process, not just a single control loop. Furthermore, if implemented correctly, an attacker within the field network should not even notice the existence of the IDS.

Figure 7 depicts our new design for an external IDS. The main idea behind this scheme is to obtain a copy of all the network traffic

flowing within the field network and sending that traffic to an external IDS. This system is implemented within the same SDN control network, as it must communicate with the controller to dynamically alter the SDN as a result of an alert.

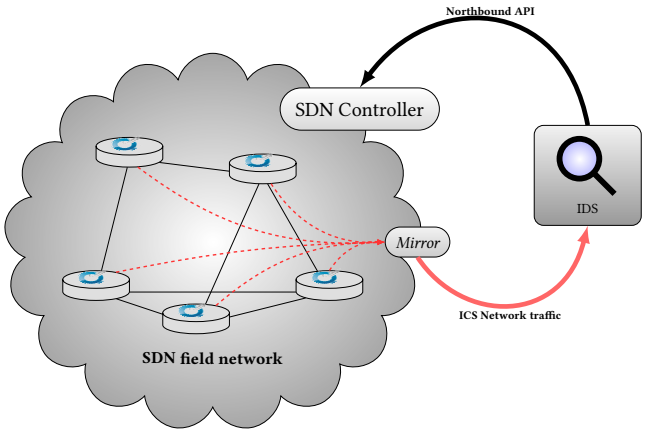


Figure 7: Intrusion detection scheme using mirrored traffic acquired from all the OpenFlow devices managed by the controller. Upon detection, the IDS reacts to a set of defined attacks by sending northbound API commands to the SDN controller in order to prevent further attacks.

By using an external IDS, it can not only detect anomalous behavior of the physical process itself, but it can also analyze the network traffic with traditional network IDS to identify known malicious techniques being launched from inside the field network. Moreover, any commands sent from the IDS to the SDN controller would not be accessible from within the field network, preventing the attacker from realizing any counter-measures taken by the IDS.

As for the data capture, some related work has already addressed some of the challenges that involve the usage of an IDS with SDN [17]. Despite the many benefits that the SDN provides, the segregation of the control and data planes makes it difficult to perform an inspection of the full packets circulating within an SDN. As a possible solution, Yoon et al. [17] present a conceptual design in which the application layer that handles the northbound API of a controller acquires the traffic from the SDN network through a separate NIC in an in-line fashion. However, they do mention the possible performance issues this might pose for the SDN as a whole. They also provide some insights regarding the implementation with passive-modes, in which additional network interfaces between the control and data planes are required to collect the full payload, which cannot be acquired via the control channel.

For our purposes, we consider that a passive-mode implementation is adequate for the intended solution. Not only does this implementation allows the inspection of the packets, but it also avoids affecting the overall performance of the SDN controller. Therefore, our scheme uses a passive-mode IDS in a separate machine, communicating with the SDN controller via the controller's northbound API.

Improving the SDN Controller: Regarding the SDN controller, our previous efforts used POX as the SDN controller, as it was already included with Mininet. Based upon the overall performance, compatibility, features, current development, and maintenance of the actual SDN controller, we consider that a reasonable candidate for the actual implementation of the SDN controller is ONOS, an open-source project that strives to provide a commercial-grade product ready to be deployed in production environments [15].

Improving the Modularity of Our Design: Based on the ideas discussed in this section, our scheme revolves around three main roles: the SDN network itself, the SDN controller and the IDS. Since the SDN network in a production environment is implemented with actual hardware that supports OpenFlow, the simulated environment must behave in the same way. Therefore, we consider that the obvious implementation of the SDN simulation must be in an independent machine from the actual SDN controller.

Our previous architecture [12] is illustrated in figure 8, and our design and implementation of the proposed improvements are shown in figure 9. The chosen SDN framework was Mininet[9], which simulated all the required devices within the SDN network, providing a reasonable simulation of a production environment.

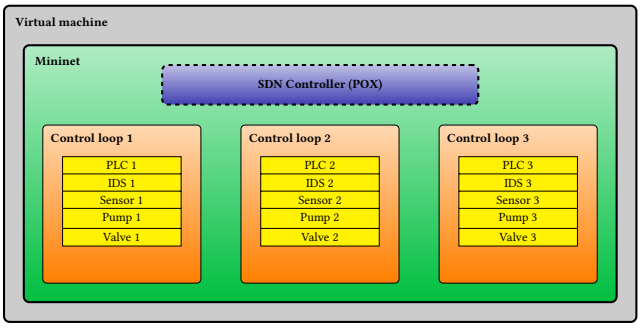


Figure 8: Previous lab environment. All the components were implemented in a single virtual machine running Mininet with POX as the SDN controller, and MiniCPS as the framework to simulate the ICS components.

The main idea of our defense mechanism is to improve the previous efforts by taking advantage of the SDN controller to execute the necessary actions to relocate an attacker detected within the field network into a honeypot network. This honeypot network must contain simulated replicas of the physical devices implemented in each control loop, creating the illusion of an actual working ICS with all its inner workings and communications between the components.

To implement the honeypot, the same miniCPS framework will be used, as it already provides a reasonable functionality to simulate an ICS. The focus of our work involves the actual usage of SDN technology rather than the simulation of the devices themselves. We point out that the previous efforts already tackled the simulation of the process, which includes the implementation of all the EtherNet/IP stack as the communications protocol for the simulated ICS.

[9]Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine: http://mininet.org/

4.3 SDN control network

As was discussed in section 4.2, our solution requires the segregation of the SDN framework, the SDN controller and the IDS. Due to this segregation of roles, there must be a control network established between these components as a communications channel. This channel allows the controller to communicate with both the simulated OpenFlow devices and the IDS.

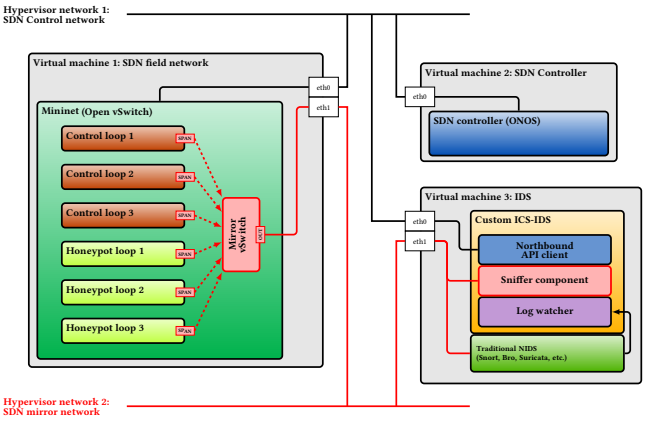


Figure 9: The lab environment is comprised of three different roles. One virtual machine handles the simulation of the physical processes, the honeypot and the actual SDN hardware that would be implemented in the field network (Section 4.4). Another machine implements the chosen SDN controller –ONOS– (Section 4.5). The last machine implements the custom IDS (Section 4.6).

As ONOS publishes both the OpenFlow and northbound API services in the same network interface, this is the actual control network that will be used to establish the communications channel between the three roles. Furthermore, in every OpenFlow device, this control network is effectively isolated from the actual SDN and is merely used as a communications channel between the device itself and the SDN controller. If the IDS is placed within this same network, not only will it still be unreachable from within the SDN, but also it will have the ability to freely communicate with the controller as well.

To implement our improved scheme, a set of three virtual machines are interconnected as depicted in figure 9, one for each required role. The scenario makes use of a hypervisor to run three separate virtual machines, each one fulfilling a specific role in the scenario. We take advantage of the hypervisor's networking capabilities to establish the communications channels between the different roles in the improved scheme. Since we need to have a control network and also sniff the SDN network traffic, we follow the conceptual design of Yoon et al. [17] and add another network interface for this purpose between the SDN field network and the IDS.

4.4 ICS simulation VM

The first virtual machine runs both Mininet and MiniCPS. This machine is in charge of simulating the field networks needed to represent the actual physical process to be defended. This includes both the industrial control devices as well as the SDN OpenFlow devices. In addition to the devices simulated in the previous scheme, our solution replicates the entire scenario to provide the honeypot, a replica of the physical process. In a production environment, this machine would only simulate the honeypot, and the physical process would be included in the SDN scheme with actual hardware devices that support OpenFlow.

To be able to extract the traffic from the SDN devices, an additional SDN device is placed within Mininet. This device has a link to every other OpenFlow device in the network. By manipulating flow and group tables, the network traffic from each control and honeypot loop is mirrored into this new device. The entire traffic received by this device is then redirected towards a particular port in the device that is linked to a network interface in the virtual machine, effectively sending all the traffic captured from the SDN network into the virtual machine's network interface. This network traffic will be then sniffed by the IDS.

4.5 SDN Controller VM

The next virtual machine handles the SDN controller role. In this machine, ONOS has been installed and configured in such a way that both Mininet and the IDS can communicate with it. The former to establish the communication between the simulated OpenFlow devices and the controller. The latter to send instructions to the controller via the northbound API. These communications are received via the SDN control network.

4.6 IDS VM

The last virtual machine implements the IDS role as a whole. As the scope of our current efforts is not in the attack itself, but rather in how to leverage SDN features once an alert is raised, we assume that the detection of an attacker has already been implemented and we merely use this detection mechanism as part of the new implementation. Following the same implementation practices of our previous efforts, our custom-made IDS role is also implemented using Python.

As the intrusion detection schemes from our previous efforts depend on the actual values of the process, these values are now acquired directly from the sniffed network packets rather than from the local database. This is because the new IDS is an external entity, independent from the actual network that is being defended. Therefore, any data acquisition must be done directly from the sniffed payloads. This is taken care of by a sniffer component within the custom IDS, based on Scapy[10].

Any decisions regarding the SDN itself as a result of the detection mechanisms of the sniffer component are handled by a northbound API client component included in the custom IDS. To the best of our knowledge, there is no official Python module to directly interact with ONOS, which presented a challenge for including ONOS in our solution. However, ONOS does provide a northbound REST API as a generic northbound API. Because of this, we developed 'pyonos', a Python module that can communicate with the northbound REST

[10]Scapy is a Python program that enables the user to send, sniff, dissect and forge network packets. This capability allows the construction of tools that can probe, scan or attack networks. https://scapy.readthedocs.io/en/latest/

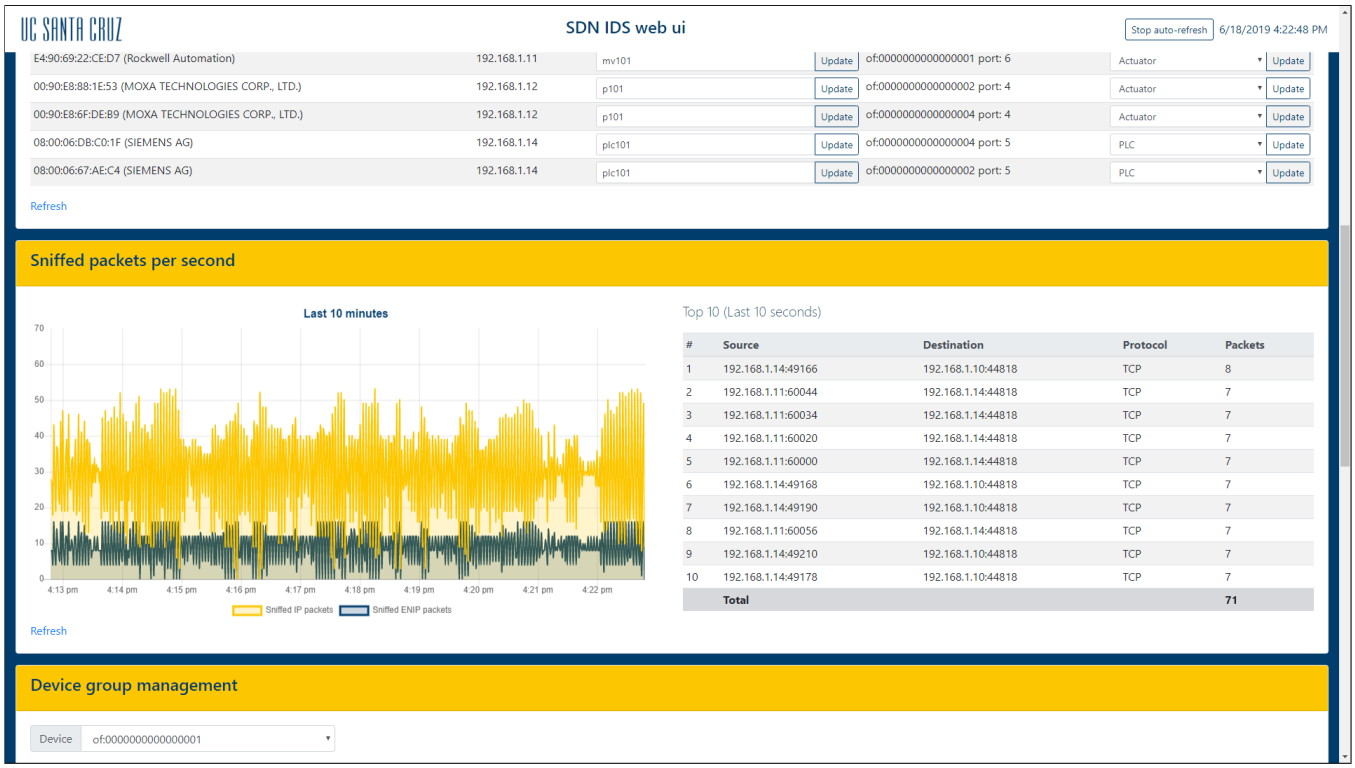


Figure 10: The IDS web UI. It allows rudimentary management of the custom IDS and the relevant SDN devices via the controller's northbound API.

API provided by ONOS to manage the SDN devices. Using this new module the custom IDS can manage the SDN network, allowing the dynamic manipulation of the flow tables of the field network devices.

A final component of the custom IDS is the log watcher. This feature was conceived in the design of the custom IDS but has not yet been implemented. The idea behind the log watcher is to have a traditional IDS such as Snort inspect the sniffed traffic for traditional IT malicious traffic. The log watcher component of the custom IDS should be able to read the alerts generated by the traditional IDS, and new decisions can be based on these alerts. The inclusion of this traditional IDS is left for future work.

Throughout the implementation of our solution, we noticed an increasing complexity regarding the management of the SDN infrastructure. Therefore, we started the development of a simple web interface to interact with the IDS. The web interface allows the management of the SDN devices, SDN group manipulation, basic flow table management, simple role assignment of the detected endpoints and the display of some statistics regarding the sniffed packets within the last minutes (figure 10).

5 RELOCATING THE ATTACKER TO THE HONEYPOT

We are now ready to implement our honeypot relocation use-case. There are four components to the defense scheme as a whole: the SDN controller (ONOS), the field and honeypot networks (Mininet and MiniCPS), and the IDS. ONOS will control the networking itself, managing the interactions between the field network and the honeypot, as well as the mirroring scheme used to deliver the network traffic to the IDS. The latter, in turn, will receive the raw network packets from both the field and honeypot networks to make the appropriate security decisions, which are then translated to REST API commands that are sent to the controller as needed (figure 11).

The primary idea behind this defense scheme is that neither the field nor the honeypot networks are aware that there is an IDS monitoring the network traffic. In other words, there is no communication between the devices inside either network and the IDS. Nonetheless, the IDS does inspect all the traffic flowing throughout both networks in search of the anomalous behavior implemented in previous solutions.

In addition to these external components, we need to communicate both the field and honeypot networks whenever needed. However, this communication has to be restricted to specific cases in which the field devices might be in jeopardy. In other words, there has to be a bridge between both networks that should only allow traffic to flow upon specific requests by the IDS. Under regular circumstances, this bridge should effectively insulate both networks.

While probably the most intuitive manner of implementing this bridge would be to establish a link between each pair of SDN switches in each control loop between the field network and the

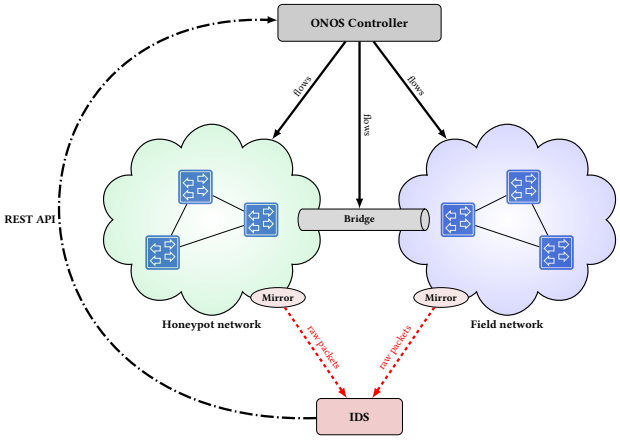


Figure 11: Fundamental interactions between the components of the defense scheme.

honeypot, this approach could depend on the router to forward traffic between different control loops within the honeypot. This would mean that the router itself must have knowledge of the honeypot when the whole idea behind the honeypot is for it to be invisible from the actual network. Therefore, having a centralized bridge between the field network and the honeypot facilitates the communication between any control loop in the field network and the honeypot as needed, without depending on the router. Every routing decision can be defined via flow table entries as needed, while ordinary communications between these networks can be restricted by default, making the networks invisible to each other.

Whenever an attacker is identified and isolated inside the field network, the IDS takes actions to effectively relocate the attacker within the honeypot. We send the appropriate instructions to ONOS from the IDS via the REST API. These instructions include the flow table entries that allow the SDN switches to reroute all the traffic incoming from the attacker through the bridge and into the honeypot, as well as the corresponding replies from the fake devices within the honeypot back to the attacker (figure 12).

6 PERFORMANCE TESTS

Since this scheme is meant to be a seamless transition between the real network and the honeypot network, it is paramount that attackers do not realize that they have been moved to the honeypot network. This means that not only the real scenario must be replicated, but the behavior of the network must be maintained.

To accomplish a seamless transition, the traffic sent by the attacker must be redirected to the honeypot in such a way that the activities performed by the attacker appear to be continuous.

We performed two main tests in the scenario depicted in figure 13. The first one involves a custom UDP service. We chose a UDP service because we need to measure the response time of the traffic itself while transitioning between the two networks. Moreover, a connection-oriented service could change the outcome of the test due to possible re-connection times. Therefore, a service which does not require the establishment of a connection will indicate the actual response times of the network as a whole. This means that

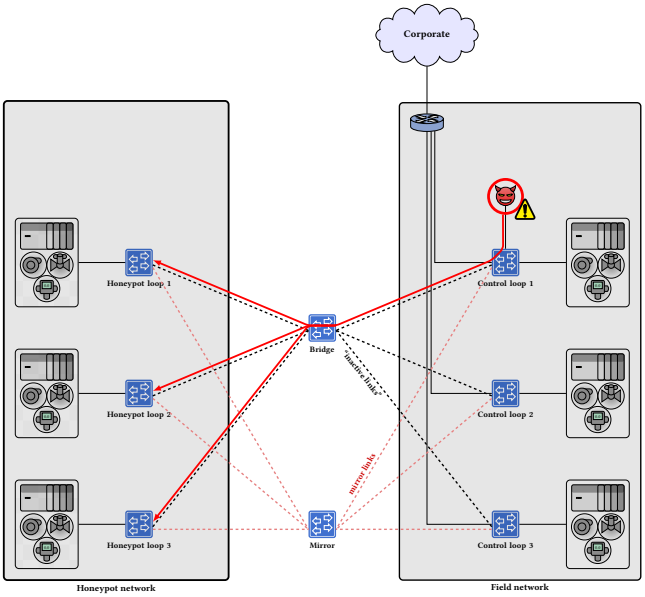


Figure 12: Under normal circumstances, flow tables in the bridge switch prevent any traffic from flowing between the field network and the honeypot, hence the name "inactive" links. Upon detection of an attacker, flow tables are modified as necessary to reroute all the traffic involving the attacker into the honeypot.

any delays in the communication will be the result of the different updates in the SDN devices along the paths.

The custom UDP service is a simple service that replies 8 bytes of data to any arriving packet, the 8 bytes are just a timestamp coded in a long integer. Whenever the client sends a packet, it records its local timestamp and calculates the difference between the timestamp it sends and the timestamp of the client by the time it receives the response. Note that the difference is only calculated with the local timestamp. The remote timestamp is only used as a payload for the packet and is only meant to add a standard-sized payload to the traffic between the client and service. This difference in time will be the round-trip time (RTT) of the service, which is the time it takes to send a packet, be "processed" by the service, and receive a response.

While the test is being performed, we configure a set of flow table rules in the SDN controller, forcing the traffic that is being sent by the client (the "attacker") to be redirected to the honeypot network. Since all the packets will have the wrong Ethernet frames, the attacker will be forced to request the "new" MAC address of the service to continue using the service. Furthermore, the SDN devices that have just updated their flow tables will completely add the rules whenever corresponding traffic is matched against every new rule. In the meantime, the attacker experiences a timeout in the communication, as the packets do not reach the intended destination (figure 13).

The result of this redirection, as seen by the SDN controller, is depicted in figure 14. All the traffic generated by the host with IP address 192.168.1.100 located in the field network on the right

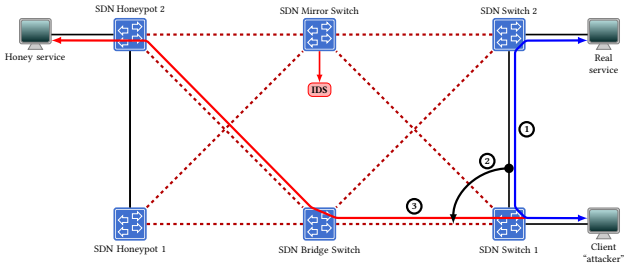


Figure 13: Performance test scenario. Two machines will be running the same service both in the field and honeypot networks. An "attacker" located in the field network will initially exchange data with the real service (1). Upon the execution of the relocation mechanism, the traffic will be redirected to the honeypot service (2). The attacker then exchanges data with the honeypot service (3). Throughout the whole process, the attacker logs the RTT of each packet sent to the service, unaware of the service's location.

is being sent to the honeypot network on the left. In this particular case, the host is contacting the service published by the IP address 192.168.1.10, which is exposed in both hosts that share that IP address in the field and honeypot networks, making a seamless transition possible. The remaining hosts in both networks are communicating with each other within the boundaries of each network.

The collected results of a sample execution are shown in figure 15. For each packet sent, the attacker logs the RTT of that particular packet with the current timestamp. This data can depict the behavior of the SDN network while transporting the data between two hosts. The gap highlighted in grey represents the downtime experienced due to the relocation of the attacker from the field network to the honeypot via flow tables. As with any other regular traffic in an SDN, the first packet of a new connection has a larger RTT than the rest of the traffic associated with a particular communications channel between two hosts.

To estimate the behavior of the SDN to such relocation, this test was performed 100 times. Each time, the attacker logged the RTT of every sent packet and either the time it took to get a response or a timeout of 1 second. Each execution was performed in the following manner: the attacker sent data for 15 seconds, then the relocation took place, and finally, an additional 15 seconds of data transmission took place. The idea is to have a standard 30-second uptime for the payloads to reach the "intended" destination. In the field network, the intended destination would be the real service; in the honeypot, the fake service.

Since the main challenge of the relocation is to avoid the detection of the honeypot by the attacker, the relocation timeout (i.e., the time it takes to receive a response after the relocation is issued in the controller) is a variable that must be considered as part of the performance of the protection scheme, this delay in the communication must be small enough to avoid any concerns by the attacker. Figure 16 presents the measured downtime on every execution of the test. The delay experienced by the client service appears to have a random component to it. The changes in the delay revolve around 25 seconds, which means that this is the amount of time the attacker is expected to have a service outage.

To fix this issue, we look at the origin of this delay, which is due to the way the flow entries are generated. At this point, the flow entries match the attacker's address and as an instruction, the entries have an OUTPUT action towards the port connected to the bridge between the field network and the honeypot network. While this effectively redirects all the traffic involving the attacker towards the honeypot, there is an issue with the MAC addresses. All the traffic that the attacker is sending is intended for the original device and, while the honeypot device has the same IP address, the MAC address of the emulated device is different from the original one.

A naive solution would be to simulate the honeypot devices with the same MAC addresses. However, as the internal indexing of hosts within the SDN controller is based on MAC addresses, this would be counter-productive as legitimate traffic would be sent to the honeypot and innocuous traffic from the honeypot would be sent to the field network, ultimately affecting the physical process.

To solve this issue, we added new flow entries in the honeypot switches. The idea behind these new flow entries is that they allow us to modify the packets in order to change the destination MAC addresses to the corresponding addresses of the honeypot device whenever a packet is meant for each particular device; the same thing must be done for the opposite direction, every packet originating from the simulated device must be altered in order to change the source MAC address to that of the original device. By doing so, the requests meant for the service will be redirected to the honeypot service and the responses will be correctly routed to the attacker. After implementing these changes, figure 17 shows the RTT of a sample execution under these conditions. By changing the MAC addresses, the response time greatly improved to the point that, at most, a single packet is lost.

7 CONCLUSIONS

In this paper, we present a way to create a honeypot network of an ICS in which an identified attacker can be dynamically relocated by leveraging the inherent properties of SDN. By stealthily acquiring a copy of all the traffic flowing throughout the field network, an IDS that implements a northbound API client can reconfigure the SDN as needed to isolate and fool the attacker, while avoiding any direct contact with the affected network.

We show that if the transition is handled carefully and takes into account all the relevant network layers, the relocation of an attacker can not only be seamless but also very difficult to detect, as the behavior of the network from the perspective of the attacker is the same both in the field and the honeypot networks, and the packet loss during the relocation is avoided when the appropriate rules are used.

Finally, we show that an effective defense can be implemented covertly by handling all the defense traffic within a separate SDN control network, isolated from the field and corporate networks, while still receiving the traffic from the field network over specific interfaces instantiated within the SDN devices.

Our contributions are the direct result of the extension of our previous work [12], resulting in a refined IDS scheme that leverages

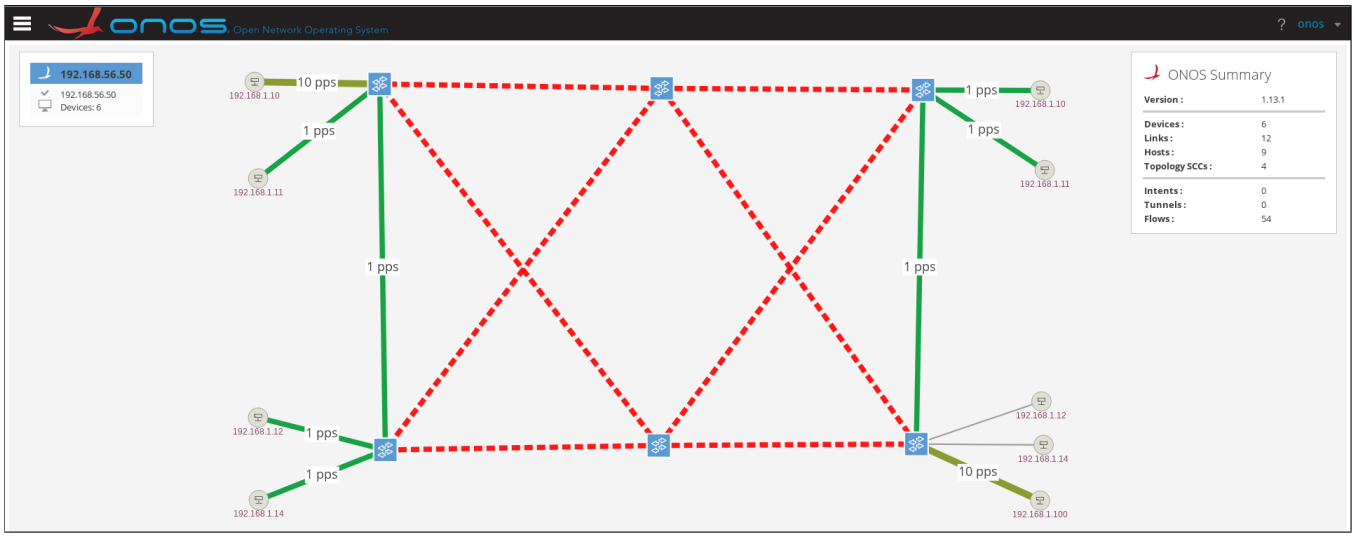


Figure 14: An execution of the test as seen by the SDN controller. Two isolated networks are linked exclusively by switches that only relay traffic as directed by the IDS. The traffic between the attacker (192.168.1.100) and a victim device (192.168.1.10) is redirected from the real network (right side) to the honeypot (left side).

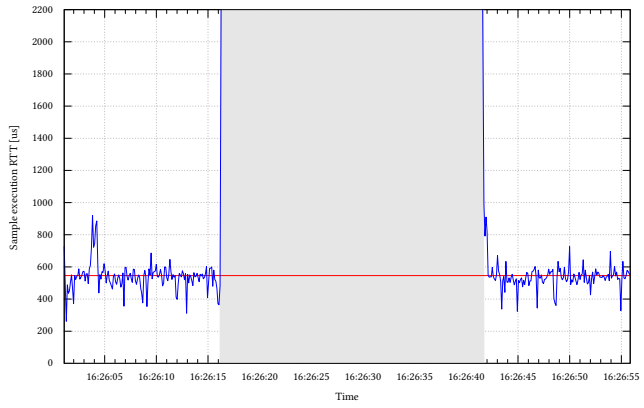


Figure 15: An execution of the test. The collected data represents the round trip time between the "attacker" and the "victim" host, in microseconds. The gap represents the downtime generated by the relocation of the "attacker" into the honeypot.

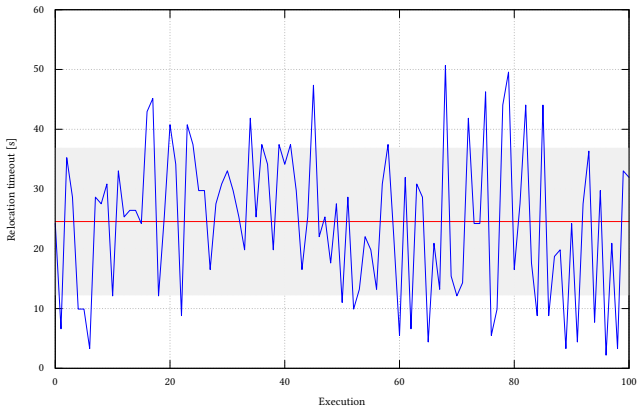


Figure 16: Downtime measured on each one of the 100 executions of the test. Every sample represents the time between the relocation of the attacker from the field network to the honeypot, and the reception of the first response from the honeypot, in seconds.

SDN with a commercial-grade controller and an ICS simulation framework as a honeypot that can isolate a detected attacker within a secure environment, protecting the field network. The byproducts of these results include the management web UI and the Python module to communicate with the SDN controller ONOS via a REST API.

7.1 Future work

As the current implementation was solely developed for a specific ICS scenario, future work includes the improvement of the modular qualities of the IDS scheme to be extensible for different scenarios. Furthermore, the current implementation of the Python interface with the northbound API is strictly limited to the particular features we required for the current scenario, in the future, additional features can be implemented to provide a complete northbound client.

ACKNOWLEDGMENTS

This work is based on research sponsored by the U.S. Department of Commerce, National Institute of Standards and Technology (NIST) award 70NANB17H282N, the National Science Foundation through CMMI-1925524, CNS-1929406, CNS-1929410, and CNS-1931573, and the Air Force Research Laboratory under agreement number FA8750-19-2-0010. The U.S. Government is authorized to reproduce

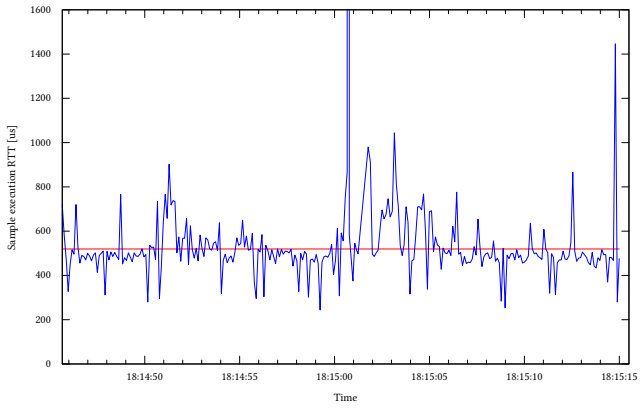


Figure 17: Another execution of the test after the MAC address issue was resolved. The collected data represents the round trip time between the "attacker" and the "victim" host, in microseconds. The downtime was reduced to a single lost packet between the client and the fake service at 18:15:00.6, a great improvement when compared to the initial behaviour depicted in figure 15.

and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] Cristina Alcaraz and Sherali Zeadally. 2015. Critical infrastructure protection: requirements and challenges for the 21st century. *International journal of critical infrastructure protection* 8 (2015), 53–66.

[2] Daniele Antonioli and Nils Ole Tippenhauer. 2015. MiniCPS. In *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy - CPS-SPC '15*. ACM Press, New York, New York, USA, 91–100. https://doi.org/10.1145/2808705.2808715

[3] Jairo Giraldo, Esha Sarkar, Alvaro A. Cardenas, Michail Maniatakos, and Murat Kantarcioglu. 2017. Security and Privacy in Cyber-Physical Systems: A Survey of Surveys. *IEEE Design & Test* 34, 4 (aug 2017), 7–17. https://doi.org/10.1109/MDAT.2017.2709310

[4] Jairo Giraldo, David Urbina, Alvaro Cardenas, Junia Valente, Mustafa Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candell. 2018. A Survey of Physics-Based Attack Detection in Cyber-Physical Systems. *Comput. Surveys* 51, 4 (jul 2018), 1–36. https://doi.org/10.1145/3203245

[5] Jairo Giraldo, David Urbina, Alvaro Cardenas, Junia Valente, Mustafa Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candell. 2018. A survey of physics-based attack detection in cyber-physical systems. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 76.

[6] Jairo Giraldo, David Urbina, Alvaro A. Cardenas, and Nils Ole Tippenhauer. 2019. *Hide and Seek: An Architecture for Improving Attack-Visibility in Industrial Control Systems*. Springer, Cham, Cham, Switzerland, 175–195. https://doi.org/10.1007/978-3-030-21568-2_9

[7] White House. 2016. Federal Cybersecurity Research and Development Strategic Plan. (2016).

[8] Celine Irvene, David Formby, Samuel Litchfield, and Raheem Beyah. 2018. HoneyBot: A honeypot for robotic systems. *Proc. IEEE* 106, 1 (jan 2018), 61–70. https://doi.org/10.1109/JPROC.2017.2748421

[9] Robert M Lee, Michael J Assante, and Tim Conway. 2016. *Analysis of the Cyber Attack on the Ukrainian Power Grid*. Technical Report. SANS Industrial Control Systems.

[10] Samuel Litchfield, David Formby, Jonathan Rogers, Sakis Meliopoulos, and Raheem Beyah. 2016. Rethinking the Honeypot for Cyber-Physical Systems. *IEEE Internet Computing* 20, 5 (sep 2016), 9–17. https://doi.org/10.1109/MIC.2016.103

[11] Luciana Obregon and Barbara Filkins. 2015. *Secure Architecture for Industrial Control Systems Secure Architecture for Industrial Control Systems Secure Architecture for Industrial Control Systems 2*. Technical Report. SANS Institute. https://www.sans.org/reading-room/whitepapers/ICS/secure-architecture-industrial-control-systems-36327

[12] Andrés F. Murillo Piedrahita, Vikram Gaur, Jairo Giraldo, Alvaro A. Cardenas, and Sandra Julieta Rueda. 2018. Virtual incident response functions in control systems. *Computer Networks* 135 (apr 2018), 147–159. https://doi.org/10.1016/J.COMNET.2018.01.040

[13] Juan Enrique Rubio, Cristina Alcaraz, Rodrigo Roman, and Javier Lopez. 2017. Analysis of Intrusion Detection Systems in Industrial Ecosystems. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications*, Vol. 4. SCITEPRESS - Science and Technology Publications, Madrid, Spain, 116–128. https://doi.org/10.5220/0006426301160128

[14] Richard William Skowyra, Andrei Lapets, Azer Bestavros, and Assaf Kfoury. 2013. Verifiably-safe software-defined networks for CPS. In *Proceedings of the 2nd ACM international conference on High confidence networked systems - HiCoNS '13*. ACM Press, New York, New York, USA, 101. https://doi.org/10.1145/2461446.2461461

[15] Alexandru L Stancu, Simona Halunga, Alexandru Vulpe, George Suciu, Octavian Fratu, and Eduard C Popovici. 2015. A comparison between several Software Defined Networking controllers. In *2015 12th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services, TELSIKS 2015*. IEEE, Nis, Serbia, 223–226. https://doi.org/10.1109/TELSKS.2015.7357774

[16] The Open Networking Foundation. 2015. OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06). (2015). http://www.opennetworking.org

[17] Changhoon Yoon, Taejune Park, Seungsoo Lee, Heedo Kang, Seungwon Shin, and Zonghua Zhang. 2015. Enabling security functions with SDN: A feasibility study. *Computer Networks* 85 (jul 2015), 19–35. https://doi.org/10.1016/j.comnet.2015.05.005