http://dx.doi.org/10.14236/ewic/ICS2018.4

A Two-level Intrusion Detection System for Industrial Control System Networks using P4

Gorby Kabasele Ndonda
Universit Catholique de Louvain, Belgium
gorby.kabasele@uclouvain.be

Ramin Sadre
Universit Catholique de Louvain, Belgium
ramin.sadre@uclouvain.be

The increasing number of attacks against Industrial Control Systems (ICS) have shown the vulnerability of these systems. Many ICS network protocols have no security mechanism and the requirements on high availability and real-time communication make it challenging to apply intrusive security measures. In this paper, we propose a two-level intrusion detection system for ICS networks based on Software Defined Networking (SDN). The first level consists of flow and Modbus whitelists, leveraging P4 for efficient real-time monitoring. The second level is a deep packet inspector communicating with an SDN controller to update the whitelists of the first level. We show by experiments in an emulated environment that our design has only a small impact on communication latencies in the ICS and is efficient against Modbus/TCP oriented attacks.

1. INTRODUCTION

Industrial Control Systems (ICS) have evolved from isolated and proprietary systems to highly interconnected systems relying on IP-based networking and off-the-shelf components. Security-wise, ICS are not prepared well for this evolution. Popular ICS communication protocols such as Modbus (Modbus 2006) were designed under the assumption of isolation and do not contain any security mechanisms. Consequently, we observe an increasing number of attack against ICS. Applying directly existing ICT security to ICS is not possible due to the constraints of the latter (Hahn 2016). Control systems require high availability, so patching and updating equipment is hard, to the extent that many ICS employ outdated hardware and software. Furthermore, ICS operators hesitate to deploy complex authentication or encryption mechanisms due to real-time constraints. Finally, PLCs are resource-constrained devices, which prevents running state-of-the-art host-based intrusion detection systems (IDS) on them.

For these reasons, *network-based* intrusion detection has attracted the interest of researchers and different approaches have been described in the literature. One of them is the usage of whitelisting (Cheung et al. 2007; Barbosa et al. 2013; Lemay et al. 2016). Whitelists describe permitted operations and exploit the fact that almost everything is automated in ICS such that traffic characteristics and topologies are stable (R. R. R. Barbosa and R. Sadre and A. Pras 2012). For example, a *flow whitelist* specifies which hosts are allowed to communicate. Such whitelists require careful preparation: If the list is incomplete, legitimate traffic might be blocked. This can happen for example when the person creating the whitelist does not have full knowledge of all legitimate communication in the network. Barbosa et al. (2013) propose to learn whitelists automatically by observing the communication inside the ICS, with the risk to miss legitimate, but rarely occurring communication flows.

In this paper, we propose a two-level network-based IDS for ICS based on Software Defined Networking (SDN). The first level of the IDS is a whitelist-based filter for the Modbus protocol implemented in P4 on network switches. If a switch cannot find a matching whitelist entry for a packet, it forwards the packet to the second level of the IDS. This second level is a security engine running on a dedicated host. The engine determines whether the packet is malicious or not and instructs the controller to update filters on the switches to accept or block connections. In this way, most of the packet processing happens locally on the switches. Our main contributions are:

- We implemented a packet processor in P4 for Modbus packets. Packet processing runs on the switches and thus eliminates the need of deploying additional IDS equipment (apart from the host running the second-level security engine) or additional software on the PLCs or supervisory computers.

- By using a two-level design, we mitigate the major drawback of existing whitelisting approaches. Instead of directly rejecting suspicious packets, they will be forwarded to the second level for further inspection.
- We show by experiments in an emulated environment the small impact of our design on communication latencies in the ICS, since most of the legitimate traffic will be handled by the switches without intervention by the second-level security engine.

The structure of this paper is as follows: Section 2 gives a short introduction to SDN and P4. Section 3 discusses related works. Section 4 describes our two-level IDS. Evaluation and validation are reported in Section 5. In Section 6, we discuss our findings. This paper is concluded in Section 7.

2. BACKGROUND

In this section we provide a brief introduction to some key concepts necessary for the understanding of our paper.

2.1. Modbus

Modbus (Modbus 2006) is a layer-7 protocol widely used in SCADA networks for the communication between the master server (MTU) and the field devices (PLCs). It uses a client-server model: The client (MTU) sends requests to the server (PLC) to read/write value stored on the latter. Each Modbus request includes a function code specifying what the server has to do. Some function codes are understood by all servers but vendors can add proprietary codes. If there is no error when performing an action, the server echoes the function code in its response, otherwise the most significant bit in the function code field is set in the response.

2.2. SDN and P4

In SDN, the control plane is moved from the forwarding devices to a logically centralized system called the controller, which allows to make the network programmable. Network operators can create network applications running on top of the controller to dictate how the network behaves (Kreutz et al. 2015). The interface to program the network is called the *northbound* interface and the communication between the controller and the forwarding device occurs via the *southbound* interface. The most popular southbound technology is OpenFlow (Open Networking Foundation 2009). In OpenFlow, each network switch stores one or several tables for look-up and packet forwarding. Table entries are rules installed by the controller and each rule specifies a set of header field values against which packets are matched. Matching application-protocol fields, as it would be needed to monitor Modbus function codes, is not possible because OpenFlow has a fixed protocol parser with a fixed number of header fields in its flow tables. Therefore, in this paper, we will use P4 to program the switches. P4, the *Programming Protocol independent Packet Processor*, is a high-level language whose primary goal is to program protocol-independent packet processors (Bosshart et al. 2014). Unlike OpenFlow, it allows programmers to implement their own parsers. The programmable switch contains one or several match+action tables. A P4 program defines:

- protocol headers and their fields,
- a parser specifying the order of the headers in a packet,
- the table structures, i.e. matching fields,
- the available actions. Actions are composed of protocol-independent primitives.

The P4 program is compiled and uploaded to the switch. Table population is done through a southbound API, allowing the controller to dynamically add or remove table entries.

3. RELATED WORK

3.1. SDN for Intrusion Detection and ICS Security

The usage of SDN to improve or implement security in computer networks has been studied in several publications. We will limit our discussion to work using SDN in IDS design and/or for ICS security.

OpenFlow-enabled switches can be used to collect flow-based traffic statistics on the controller. Some researchers have proposed to use OpenFlow for flow-based intrusion detection in a way similar to what has been done with NetFlow/IPFIX-enabled switches (Sperotto et al. 2010). In contrast to the latter, the OpenFlow approach also allows to implement mitigation actions easily, such as whitelisting (Giotis et al. 2014). However, the flow data does not contain any payload or application-layer protocol headers and therefore limits the data analysis. Therefore, IDS based on OpenFlow monitoring work well for those attacks that have a distinct signature on flow level, such as DDoS attacks (Lee et al. 2017). Shirali-Shahreza and Ganjali (2013) propose to extend OpenFlow such that the controller has full access to the packet payload. Then, they implement an application on the controller to detect port scans. However, they do not

consider that sending full packets to the controller facilitates *data-to-control plane saturation attacks* (Wang et al. 2015). In our solution, the packets are inspected by the IDS and not by the controller, which prevents attacks from reaching the latter. In fact, this design is recommended by the OpenFlow authors (McKeown et al. 2008).

More specific to ICS security, Dong et al. (2015) suggest in a position paper to use SDN for the "dynamic monitoring" of smart grids, without providing further details. A very different application of SDN on ICS security is proposed by da Silva et al. (2015) and Ndonda and Sadre (2017): SDN is used to periodically reconfigure routes in ICS networks in order to make it harder for an attacker to eavesdrop or manipulate entire message exchanges.

3.2. Whitelisting in ICS

Cheung et al. (2007) give three approaches based on behavior modeling. The first one models the correct behavior of the Modbus protocol. The second one specifies allowed communication patterns. The last one is based on a whitelist of all legitimate services in the system. Connection attempts to non-valid services or services unknown to the system are considered as suspicious. Barbosa et al. (2013) capture ICS network traffic over a period of time to build a whitelist of allowed communication flows. When evaluated on real SCADA datasets, legitimate flows absent from the learning generate most of the false alarms, but little is said on how to resolve this problem. Lemay et al. (2016) show that ICS whitelists can be transformed into rulesets for Snort.

Some two-level design have been proposed (Thomas 2001), however they are not as flexible as our design since P4 allows to support new protocols without having to change the software on the switches. Moreover, the P4 community is aiming at runtime management of P4 devices, which would allow to extend device capabilities (not only the table) at runtime.

In this paper, we also follow a whitelist-based approach but unlike existing solutions, a first traffic analysis is done on the switches and a second step happens in a central IDS. In this way, no dedicated monitoring components have to be deployed in the network and no additional software has to be installed on the PLCs or the supervisory hosts.

4. THE TWO-LEVEL INTRUSION DETECTION SYSTEM

Our IDS operates on two levels. The first level consists of the P4 switches which are instructed to parse and match packets against whitelists. In order to reduce the possible number of false positive if the whitelists are not complete, non-matching packets will be forwarded to a dedicated host running the second level of the IDS. In this second level, the packet is analyzed by a security engine using Bro. The security engine interacts with an SDN controller and can instruct it to add new entries to the whitelists on the switches in order to allow connections reckoned as harmless or to block packets from sources identified as malicious.

```
table flow_id {
    reads {    // protocol fields defining a flow
        ipv4.srcAddr : exact;
        tcp.srcPort : exact;
        ipv4.protocol : exact;
        ipv4.dstAddr : exact;
        tcp.dstPort : exact;
    }
    actions {  // possible actions
        _no_op;
        add_miss_tag;
    }
}
```

Figure 1: *P4 definition of the flow table*

Our main motivation for using an SDN enabled network is that it provides an easy management of whitelists and reduces the load on the IDS by performing most of the packet processing locally on the switches.

4.1. First Level: Whitelisting on the Switches

The first level of the IDS consists of matching incoming packets against a *flow whitelist* and a *Modbus whitelist* installed on the network switches. The whitelists are implemented as table lookups in P4. When a new TCP packet arrives, the switch first checks whether the packet's flow signature, consisting of both IP source and destination address, the TCP source and destination port, has a matching entry in the switch's *flow table*. The packet is immediately redirected to the second level if there is no such entry. Figure 1 shows the definition of the flow table in P4.

Then, the TCP ports of the packet are used to identify Modbus packet (default 502). Function code and the message length in the Modbus packet are matched against the *Modbus table*. This table contains a list of allowed Modbus function codes and the message length typically associated with them. Checking the code *and* the length makes it harder to perform a buffer overflow attack against a PLC. Packets with a matching table entry are forwarded to their destination, otherwise they are sent to the second level. Because P4 switches are programmable, the whole process is highly flexible and can be extended to other protocols.

4.1.1. Prepopulating the Tables

If the switches were started with empty tables, *every* packet would be forwarded to the second level, which then would (or not) instruct the controller to add table entries on the switches. However, our design also allows the controller to populate the tables beforehand with entries determined manually or by learning. In both cases, missing entries can be added later.

Based on their knowledge of their network, the ICS operators can specify which function codes (and message lengths) are allowed and should be preloaded into the Modbus tables. This way, Modbus messages using those function codes will not be sent to the second level. Similarly, the operator can add connection entries to the flow table to allow certain connections by default. In the basic design, a flow is defined by the IP source and destination address and the TCP source and destination port, therefore prepopulating the flow table only works for TCP connections where both ports are known in advance, which is common for certain ICS equipment. If more flexibility is required, two flow tables have to be used, one for each direction: Entries in the first table do not specify the source port, and entries in the second table do not specify the destination port. In this way, the operator can prepopulate the tables with entries allowing all connections between an MTU and port 502 of a PLC, regardless of the source port used by the MTU.

To prepopulate the whitelists by learning, we follow the approach used by Barbosa et al. (2013): During a learning phase, the traffic in the network is captured and used to build a list of allowed flows. Obtaining a fairly representative set of the traffic is much easier in an ICS network than in an IT network since the former is generally much more static in terms of used protocols and topology (R. R. R. Barbosa and R. Sadre and A. Pras 2012). Nevertheless, the authors reported false positives because of flows occurring too rarely to be captured during a learning phase of reasonable[1] duration.

To automate the distribution of the whitelists, we give the controller the following information: (i) The global unique ID of each switch, (ii) the IP addresses monitored by each switch, and (iii) additional information about the topology for routing.

For every packet in the captured training data, we extract its flow signature and tell the controller to add it to the flow whitelists of those switches that are expected to see the flow according to the routing information and the list of IP addresses. We also extract the function code and the size of Modbus messages. The size depends on the function and can be variable for some functions. For each non-error function code, we learn the message sizes observed in the training data. Thus, Modbus errors will always be reported to the second level. This is important because error messages can be an indication for the *Function Code Scan* attack whose goal is to discover the functions supported by a Modbus server. For each function–length pair, we also store the IP addresses of the hosts using them. Again, the controller will only upload Modbus table entries to switches in charge of handling the traffic from the respective IP addresses.

[1]The underlying assumption of the learning approach is that the system is attack-free during the training phase, which becomes harder to guarantee if the learning phase is very long.

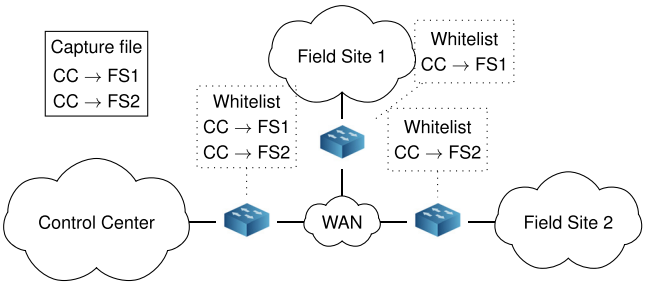

Figure 2: *Whitelist distribution*

Figure 2 shows an example of an ICS network consisting of three subnetworks. We assume that the captured traffic contains message exchanges between a supervisory computer in the *Control Center* sub-network and PLCs in *Field Site 1* and *Field Site 2*, respectively. The controller will push the complete whitelist onto the switch of the *Control Center* and subsets of the list onto the switches of *Field Site 1* and *Field Site 2*. As a consequence, communication attempts between the two field sites will not be accepted by the switches.

4.1.2. Packet Redirection to the Second Level

To redirect a packet to the second level, the switches insert a new header between the IP and the TCP header, set the destination address to the IP address of the second-level host and change the protocol field in the IP header. The new header, to which we refer as the *SCADA Redirection Tag* (SRTag), contains four fields (see Figure 3):

- **Destination Address (DA; 4 bytes)**: the original destination address.
- **Identifier (ID; 2 bytes)**: the ID of the switch.
- **Protocol (1 byte)**: the original value of the protocol field.
- **Padding (1 byte)**

In addition, the packet length and the checksum in the IP header are recalculated.

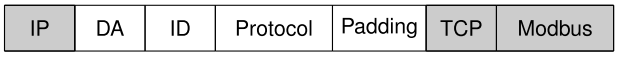

Figure 3: *SRTag header*

4.2. Second Level: The Security Engine

The second level of the IDS consists of the security engine and *Bro* co-located on the same host. Bro is an event-based network security monitor. An event is triggered by a single packet matching a specific format or by a sequence of packets, such as a TCP three-way-handshake. Bro supports events for many protocols, including Modbus, and provides an interface allowing other applications to capture generated security events.[2] Our security engine communicates with Bro as well as with the controller. Since the security engine is reachable by the switches it could be possible for an attacker to launch an attack against it, for example by flooding it with packets. In order to protect the controller from such a *control plane saturation attack*, the controller is located on a different host. In this way, the network operators can still control the network if the engine is under attack. The security engine performs the operations described below.

4.2.1. Packet analysis

The security engine analyzes packets sent by the switches in order to detect and repel attacks. Our solution favors passive monitoring and notification in form of alerts over active measures. In an ICS, a loss of communication due to a message wrongly blocked could be catastrophic. It is unrealistic to assume that the prepopulated whitelists are complete and cover all corner cases, such as Modbus messages only generated in emergency situations. Therefore, our IDS will forward all packets that have successfully passed the analysis to their original destination, unless the analysis reveals that the packet is clearly malicious or malformed. Independently from this, the network administrator is always informed about newly appearing flows or function codes. When forwarding a packet, a special 64-bit tag is added to the packet to inform the switches along the path about the safety of the packet. The tag value is randomly chosen and made known to the switches by the controller. The following tests are performed by the security engine:

- Malformed Modbus messages where the packet size (less headers) does not match the message size, are dropped.
- Modbus messages requesting sensitive operations (such as a reset command) will be dropped if they do not originate from an authorized host. In practice, this is implemented by providing the engine the IP addresses of the MTUs. However, even if the message comes from an MTU, an alert is generated.
- Modbus messages whose source or destination addresses are not in the range of IP addresses of any field site or control center trigger an alert. Again, those address ranges have to be provided by the network operator.

In addition to forwarding the packet, a copy is sent to Bro for further inspection. The security engine listens for events raised by Bro, and depending on those it will take further actions that are described below.

4.2.2. Blocking flows

The security engine counts the number of malicious events received from Bro for each combination of source IP, destination IP, destination port and transport protocol. This counter is also increased when a Modbus error message is seen. We introduce a threshold called *max block request*. When the counter reaches this threshold, the security engine requests the controller to block further packets matching that particular signature. A block request is also issued when Bro sends an event that clearly indicates an attack, such as a SYN flooding attack. Note that the controller will only install a rule to block traffic on the switches if it does not overlap with an entry included in the original prepopulation list (see Section 4.1.1). Since we assume that all flows in the original list are legitimate, blocking them could seriously harm the operation of the plant controlled by the ICS. In any case, an alarm is raised to notify the operator about the situation. Flow blocking is implemented on the switches in form of a *blocking table* containing the source IP, destination IP/port and transport protocol of the flows to block. This table is checked before the flow table and if there is a matching entry for the packet, it is dropped.

4.2.3. Adding new entries to the whitelists

Provided that a non-error message has passed all security checks of the packet analysis, the engine can instruct the controller to add the function-length combination entry permanently to the Modbus whitelists on the affected switches. The underlying assumption is that this particular combination was missed when the whitelist was built. By adding an entry to the whitelist, the delays caused by the redirection to the second level are eliminated for further messages of the same type.

The addition of new entries in the flow whitelist is similar to the addition of new entries in Modbus whitelist. Again, the underlying assumption is that

[2] Other IDS, such as Snort, could be used; the only requirement is that the security engine can read their output.

this particular flow signature was missed when building the whitelist. Note that the controller will only modify the whitelists of the switches along the path relevant for the flow. Additions can be (a) disabled for maximum security, (b) done immediately, or (c) done only after a certain amount of unsuspicious packets have been observed for a flow. Option (a) is useful in situations where the ICS operator wants to closely monitor applications that are known to be particularly vulnerable. For example, some PLCs host HTTP servers for configuration and it could therefore make sense to never whitelist connections to port 80, even at the expense of the additional delay caused by the redirection. In our prototype, we whitelist a flow as soon as we observe a successful TCP three-way handshake, provided that the packets have successfully passed the packet analysis. SYN flooding attacks can therefore not fill the whitelists.

5. EVALUATION

This section describes our evaluation of the two-level IDS. We first present the evaluation scenario and the test setup, followed by the results obtained for different test cases.

5.1. Scenario and Adversary Model

In our evaluation, we emulate a scenario where a malicious host under control of an attacker is present in an ICS network. The attacker targets the PLCs. As they do not authenticate the commands they receive, they represent convenient primary targets for attacks.

The emulated ICS is shown in Figure 4. It consists of an MTU and six PLCs located in two field sites. The security engine and the controller are located in the north. Furthermore, the network contains a human machine interface (HMI) and the host controlled by the attacker. All hosts are connected through P4 switches to a core network. The switches are represented by boxes in the figure. The core network (the “diamond” in the center) consists of four routers, represented as cylinders. Typical for large ICS installations, the core network has redundant links in order to improve reliability.

5.2. Setup

To evaluate the performance of our IDS in the above scenario, we use the network emulator Mininet (Lantz et al. 2010). It emulates P4 switches using *bmv2*, a P4 implementation in C++. We use the Modbus library *pymodus (v1.3.2)* to emulate Modbus servers (the PLCs) and a Modbus client (the MTU). Each link in the network has a 10Mbps bandwidth and a delay of 5ms. Figure 4 shows the

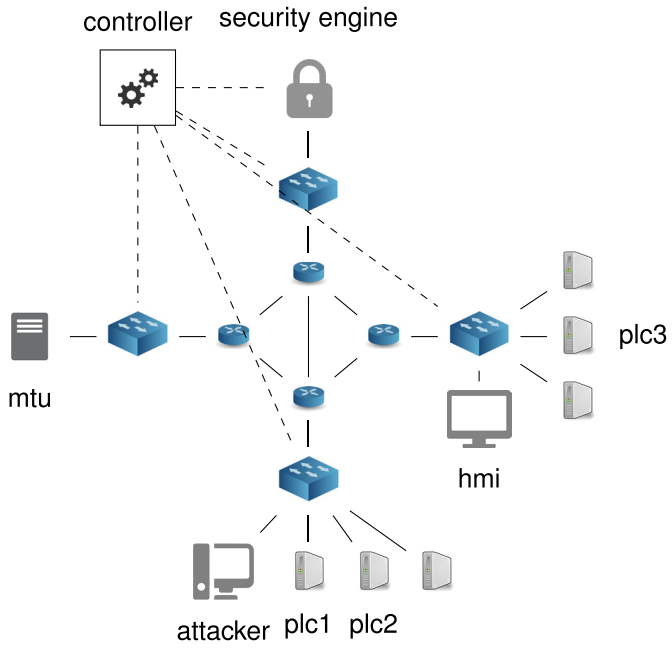

Figure 4: *Setup topology*

topology; the dashed lines represent the control-plane communication between the SDN-Controller and the P4 switches. We use Bro version 2.5.2. All the experiments were performed on the virtual machine provided by Mininet. The VM runs on a Dell XPS 13 with Ubuntu 16.04. The MTU sends requests to all PLCs every two seconds in order to obtain status information or to update process parameters. Four function codes are used in the communication between the MTU and each PLC.

5.3. Whitelist Management

Our first test shows the (undesired) effect of incomplete whitelists. We first capture a specific number of packets from the network (without any IDS or attacks running) and use the captured traffic to build the whitelists for the switches. Obviously, the shorter the taken sample, the less complete the whitelists. In a basic whitelist approach with static whitelists, an incomplete whitelist will result in a large number of false positives: every packet without an entry in the list will be blocked. In our two-level IDS, only the first packet of a flow or Modbus message type (i.e. a combination of Modbus function code and message length) will raise an alert. After the first alert, our IDS will add a new entry to the whitelist, in this way ensuring the functioning of the ICS.

Figure 5 shows the resulting percentage of false positives obtained for the basic whitelist approach and our IDS when using different capture file sizes. In total 937 packets were monitored.

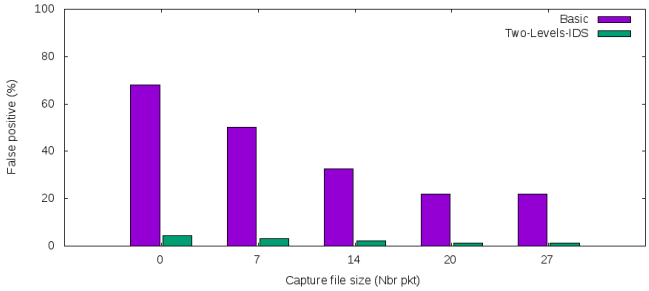

Figure 5: *Number of false positive for different capture sizes*

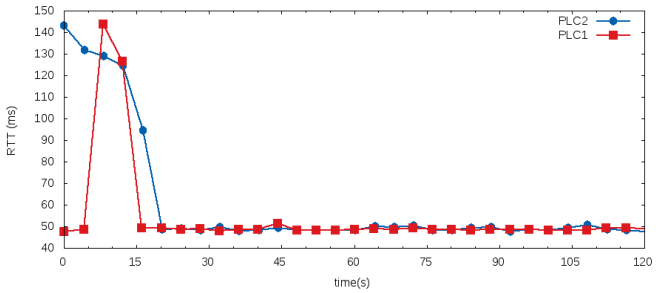

Figure 6: *Round-trip time experienced by two flows. The flow to PLC1 has no matching Modbus entry at the beginning. The flow to PLC2 has no matching flow entry nor matching Modbus entry.*

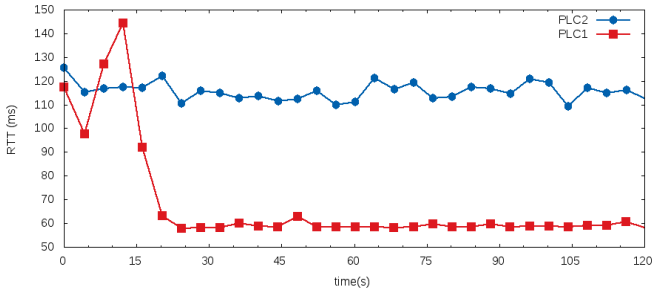

Figure 7: *Round-trip time experienced by two flows. The flow to PLC2 is never added to the whitelist.*

5.4. Delay Measurements

In our second experiment, we measure the round-trip times (RTT) of request/response pairs between the MTU and PLC1 resp. PLC2. Note that we define RTT as the total time seen by the MTU between sending the request and receiving the response from the PLC. For the communication with PLC1, we have prepopulated the tables with matching flow entries, but without matching Modbus entries for two function codes used by the MTU. For PLC2, we do not preload any matching entries onto the flow tables nor onto the Modbus tables. The IDS is configured such that it will install new flow entries or Modbus entries. The measured RTT over two minutes is shown in Figure 6. The high peak in the RTTs for PLC1 reflects the fact that the message exchanges result in Modbus packets redirected to the security engine. At around time $T = 10$ seconds, the IDS has learned all combinations of function codes and message lengths used between the MTU and PLC1 and added them to the whitelists. After that, the RTT drops to its normal value. It should be noted that both the request packet as well as the response packet are affected: although they contain the same function code, they differ in length. We are aware that a real-time control system would suffer from the redirection, however we expect that a high percentage of the flows will be prepopulated so the number of flow redirected is kept to a minimum. Also, we think it is better to handle unexpected flows with delay instead of completely blocking them. For PLC2, the effect is even more pronounced because it does have neither a matching flow entry nor a matching Modbus entry. Therefore, the handshaking packets of the TCP connection are also affected by the redirection until the IDS gets a validation from Bro to add a new entry to the flow table.

Next, we instruct the security engine to never add whitelist entries for PLC2 to simulate a “sensitive” service that the network operator wants the IDS to continuously monitor. Figure 7 shows the different RTT experienced when communicating with the two PLCs. Unlike the previous experiment, no flow entry is added for PLC2 and therefore every packet is redirected to the second level.

5.5. Throughput Measurements

We measure throughput by sending data from the MTU to the HMI in three configurations : (1) both are connected via a 10Mbps link with a delay of 30ms, (2) the flow is added to the whitelist after the TCP handshake (3) each packet is redirected to the IDS.

We use *iperf (v2.0.5)* to generate traffic for 10 seconds from the MTU to the HMI. Figure 8 shows the achieved throughput. The results for configuration 1 and 2 are quite similar, except at the beginning due the redirection of the three-way handshake. In configuration 3, the throughput is clearly affected by the redirection and stays at around 1Mbps. This is the cost of not adding a flow entry to the whitelist. Note that only the flow whitelist is used in this experiment because the traffic is not Modbus.

5.6. Blocking Scan Attacks

To test the protecting capabilities of the IDS against attacks, we study a scenario where an attacker was able to infiltrate one of the field sites and performs attacks. In Modbus, when a server is not able to perform the action requested by the client, it will send a function error code equal to the one

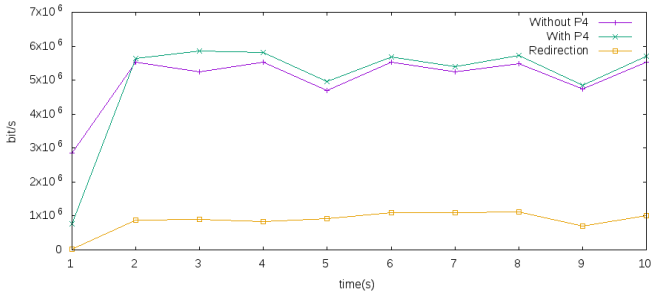

Figure 8: *Impact of the switches on the throughput*

Table 1: *Function Code Discovery*

Max Block Request	# of Function Codes Discovered
1	7
3	9
5	11
∞	19

in the request incremented by 128 along with an exception code set to 01 to specify the nature of the error. During a *Function Code Scan* attack, an attacker sends Modbus requests to a server (i.e. a PLC) to discover the function codes supported by the server. Depending on whether or not it receives this exception code, the attacker knows if the function code is supported by the server. As stated earlier, function error codes are never inserted in the Modbus whitelist so these events are captured by the IDS. We set the *max block request* threshold that is used to block flows (see Section 4.2) to 1, 3, 5 and ∞, respectively. Table 1 shows the number of function codes that an attacker could discover. The *pymodbus* server that we use only supports the 19 most common function codes Modbus (2006). Since there are gaps between those codes, an attacker can only discover 9 of them before it is blocked when the threshold is set to 3. We have used the *Function Code Scan* as an example to demonstrate the blocking capability of the IDS. Other Modbus scanning attacks, like a scan of the register addresses[3] supported by a PLC, can be blocked, too, with this mechanism.

5.7. Blocking Sensitive Function Codes

We test how the IDS reacts to the use of sensitive function codes by an attacker. In this experiment, the attacker sends a Modbus request to a PLC with a sensitive function code. The function code is the *Force Listen Mode* function. When receiving such a request, a Modbus server will enter in the so-called *Listen Mode*, meaning that it will stop sending responses when receiving a request from any client.

[3]Register addresses are roughly equivalent to I/O ports or memory addresses in Modbus PLCs.

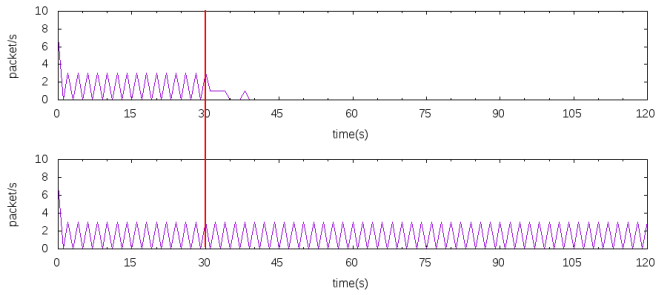

Figure 9: *Number of packets transmitted during a* Force Listen Mode *attack against a PLC. In the above plot, the attack is successful and the PLC stops communicating.*

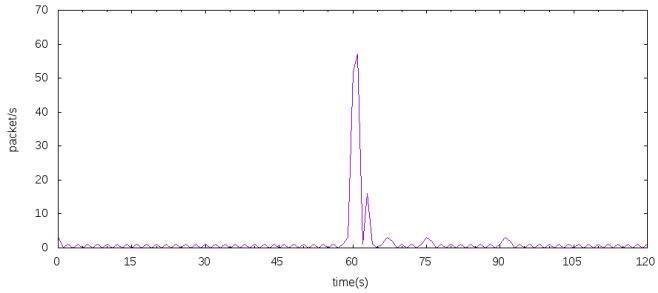

Figure 10: *Number of packets received by a PLC during a SYN Flooding attack*

Figure 9 shows the number of packets exchanged between the MTU and a PLC per second. In the upper plot, the attacker successfully performs the attack at time $T = 30s$. At that point, the MTU stops receiving responses from the PLC. After a few attempts, it closes the TCP connection. In the lower plot, the IDS detects the use of the sensitive function code by the attacker and blocks the flow. The MTU can continue communicating with the PLC.

5.8. Blocking SYN Flooding Attacks

In our last experiment, the attacker performs a SYN flooding attack against a PLC. To this end, it sends 60 SYN packets/s during 10 seconds, but does not respond to the SYN+ACK packet. Figure 10 shows the number of packets received by the PLC. The attack is clearly visible over the baseline of requests sent by the MTU every two seconds.

Since, those flow are incomplete and not in the whitelist, the SYN packets will be redirected to the security engine. The redirection of those packet allows Bro to detect the attack and raise an event that will be received by the security engine. The latter will then issue the controller to block request. For this experiment, Bro was configured to raise an event if it sees more than 30 incomplete connection attempts per 5 seconds. The flow tables are not impacted by this attacks since incomplete TCP handshakes do not cause installation of new flow entries.

6. DISCUSSION

IDS Load: A traditional network-based IDS monitors all the traffic in a network. Since the traffic in an ICS network is periodical and highly repetitive, most of the time the IDS monitors the same traffic again and again. This puts an unnecessary load on the IDS. With our design, the monitoring task is shared across the network and the IDS only monitors suspicious packets or critical events.

Middleboxes Placement in SDN: The placement of middleboxes for security purposes, such as firewalls, is a common issue in non-SDN environments (Sherry et al. 2012). Such middleboxes are used to prevent hosts from communicating with each other if they are not supposed to, so they must be placed such that the traffic between any pair of hosts is passing through them. It is challenging to ensure this property in a highly distributed system such as an ICS (Igure et al. 2006). By programming the switches, our IDS is able to monitor all traffic from a central point thus eliminating the problem of middleboxes placement. Moreover, it allows to elegantly maintain the whitelist compared to other two-level solutions (Thomas 2001).

Configurability and Extensibility: Using P4 makes our design flexible and extensible. It can be extended, with minimal effort, to support other ICS protocols, such as DNP3 or any vendor-specific proprietary protocol. It is an important advantage given that it is common for some ICS to use proprietary protocol. Plus, unlike other distributed IDS (see Section 3), the programmability of the switches enables us to analyze protocol messages directly on the switches. Moreover the centralized design of the IDS makes it easier to configure and update the whitelists and IDS policies for the entire network. The main limitation of our approach is that it requires that the application-layer messages analyzed by the IDS are unencrypted. This is not uncommon in ICS network as ICS operators are still hesitating to move to protocols with encryption because they shy away from the burden of key management and the higher resource consumption on the PLCs.

Attacks Against the IDS: An attacker can attempt to perform a DoS attack against the security engine by generating a large number of random packets redirected by the switches to the engine. If the attack is successful, the IDS loses its ability to analyze and forward non-whitelisted packets. To protect the IDS, one could implement a caching mechanism between the IDS and the switches similar to the one proposed by Wang et al. (2015), although this would increase the delay for each packet. However, it should be noted that even a complete crash of the security engine does not impede the normal functioning of the network; all flows and function codes that were already whitelisted on the switches before the attack will continue working. Furthermore, since the security engine is not co-located with the controller, the network can still be manually managed.
If the capability of the IDS to add new table entries is enabled by the operator (see Section 4.2), an attacker can also attempt to fill up the flow tables[4] on the switches with new entries. To perform this attack, the attacker has to generate a large number of legitimate-looking flows with new flow signatures. We do not consider this attack as very critical since the IDS would still be able to process and forward non-whitelisted packets.
Another issue is the security of the control plane. The communication between the controller and the switch must be secured in any case.Since this is not specific to our IDS it is not discussed here.

Availability of P4-programmable Switches and Migration Costs: As described in Section 4, our design requires that P4-programmable switches are deployed in the network. P4 is still a relatively young technology and we are currently only aware of one manufacturer of such switches. Another issue is the migration cost in existing ICS infrastructures. Adding one or more traditional firewalls to an existing network is certainly cheaper than our solution which requires replacing existing switches by programmable ones. However, our solution is much more convenient to manage and also more flexible: a SDN-approach allows to centrally manage the entire IDS and the P4-programs can be easily adapted to new protocols or protocol changes.

7. CONCLUSIONS

Industrial Control Systems are bound to many constraints that make them hard to protect them. Security solutions should require minor changes in the system and only have little impact on its reliability and real-time capabilities. In this paper, we have proposed a two-level IDS architecture leveraging Software Defined Networking. The first level runs on the network switches and consists of whitelists for flows and Modbus/TCP messages implemented in P4. Packets without matching whitelist entries are redirected to a security engine that runs on a dedicated host. This engine analyzes the packets and decides to forward or to drop them. Moreover, it can instruct the SDN controller to add new entries to the whitelists on the switches or to block malicious traffic. We have shown in experiments that the IDS is capable to detect and prevent transport-protocol

[4]Filling up the Modbus tables is not possible because there are only 128 non-error function codes and the number of legitimate function-code/message-length combinations is limited in Modbus.

attacks as well as attacks on application-protocol level, that is Modbus. Furthermore, whitelisted benign traffic is directly processed on the switches, without impact on the throughput. The two-level design allows operators to use restrictive whitelists without having to fear that false positives, i.e. unexpected but benign traffic without matching whitelist entries, cause catastrophic failures in the ICS. As future work, we plan to add support for other ICS protocols, such as DNP3, and to extend the capability to detect more elaborated attacks.

REFERENCES

R. R. R. Barbosa, R. Sadre, and A. Pras. Flow whitelisting in scada networks. *International Journal of Critical Infrastructure Protection*, 6(3): 150 – 158, 2013.

P. Bosshart et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

S. Cheung et al. Using model-based intrusion detection for scada networks. In *Proceedings of the SCADA Security Scientific Symposium*, 2007.

E. G. da Silva et al. Capitalizing on sdn-based scada systems: An anti-eavesdropping case-study. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 165–173, May 2015.

X. Dong et al. Software-defined networking for smart grid resilience: Opportunities and challenges. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, CPSS '15, pages 61–68. ACM, 2015.

K. Giotis et al. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments. *Computer Networks*, 62:122–136, April 2014.

A. Hahn. *Operational Technology and Information Technology in Industrial Control Systems*, pages 51–68. Springer International Publishing, 2016.

V. M. Igure, S. A. Laughter, and R. D. Williams. Security issues in scada networks. *Computers & Security*, 25(7):498 – 506, 2006.

D. Kreutz et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

B. Lantz, B. Heller, and N. McKeown. 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, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

S. Lee et al. Athena: A framework for scalable anomaly detection in software-defined networks. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 249–260, June 2017.

A. Lemay, J. Rochon, and J. M. Fernandez. A practical flow white list approach for scada systems. In *Proceedings of the 4th International Symposium for ICS & SCADA Cyber Security Research 2016*, ICS-CSR '16, pages 1–4. BCS Learning & Development Ltd., 2016.

N. McKeown et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

Modbus. Modbus Messaging on TCP/IP Implementation Guide V1.0b , 2006.

G. K. Ndonda and R. Sadre. A low-delay sdn-based countermeasure to eavesdropping attacks in industrial control systems. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2017.

Open Networking Foundation. Openflow switch specification,version 1.0.1 (wireprotocol 0x01), 2009.

R. R. R. Barbosa and R. Sadre and A. Pras. A first look into scada network traffic. In *2012 IEEE Network Operations and Management Symposium*, pages 518–521, April 2012.

J. Sherry et al. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

S. Shirali-Shahreza and Y. Ganjali. Efficient implementation of security applications in openflow controller with flexam. In *High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on*, pages 49–54. IEEE, 2013.

A. Sperotto et al. An overview of ip flow-based intrusion detection. *IEEE Communications Surveys & Tutorials*, 12(3):343–356, 2010. security netflow.

D. Thomas. Using open source to create a cohesive firewall/ids system. 2001.

H. Wang, L. Xu, and G. Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *Proceedings of 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 239–250, Washington, DC, USA, 2015. IEEE Computer Society.