An In-depth Look Into SDN Topology Discovery Mechanisms: Novel Attacks and Practical Countermeasures

Eduard Marin
University of Birmingham, UK
imec-COSIC, KU Leuven, Belgium
e.marin@cs.bham.ac.uk

Nicola Bucciol
University of Padua, Italy
nicola.bucciol@icloud.com

Mauro Conti
University of Padua, Italy
conti@math.unipd.it

ABSTRACT

Software-Defined Networking (SDN) is a novel network approach that has revolutionised existent network architectures by decoupling the control plane from the data plane. Researchers have shown that SDN networks are highly vulnerable to security attacks. For instance, adversaries can tamper with the controller's network topology view to hijack the hosts' location or create fake inter-switch links. These attacks can be launched for various purposes, ranging from impersonating hosts to bypassing middleboxes or intercepting network traffic. Several countermeasures have been proposed to mitigate topology attacks but to date there has been no comprehensive analysis of the level of security they offer. A critical analysis is thus an important step towards better understanding the possible limitations of the existing solutions and building stronger defences against topology attacks.

In this paper, we evaluate the actual security of the existing mechanisms for network topology discovery in SDN. Our analysis reveals 6 vulnerabilities in the state-of-the-art countermeasures against topology attacks: *TopoGuard*, *TopoGuard+*, *SPV* and *SecureBinder*. We show that these vulnerabilities can be exploited in practice to manipulate the network topology view at the controller. Furthermore, we present 2 novel topology attacks, called *Topology Freezing* and *Reverse Loop*, that exploit vulnerabilities in the widely used Floodlight controller. We responsibly disclosed these vulnerabilities to Floodlight. While we show that it is difficult to fully eradicate these attacks, we propose fixes to mitigate them. In response to our findings, we conclude the paper by detailing practical ways of further improving the existing countermeasures.

KEYWORDS

Software Defined Networking; Security; Topology Attacks and Countermeasures.

1 INTRODUCTION

Software-Defined Networking (SDN) is a new networking paradigm that is gaining momentum as a technology for developing more dynamic, agile and programmable networks in data centre and enterprise environments [25]. SDN proposes to decouple the control logic (i.e., control plane) from the data forwarding functionality of networking devices, i.e., data plane. One of the key benefits of SDN is that it offers centralised control. The SDN paradigm postulates that the network's intelligence resides in a logically centralised controller, allowing the underlying network infrastructure to become simple forwarding devices [52]. The controller maintains several core services that are responsible for conducting critical functions in the network, e.g., routing or network topology discovery. Another important aspect of SDN is its programmability. Through the use of applications, running on top of the controller, and open standard application programming interfaces, the controller can easily reprogram and collect statistics from networking devices. SDN applications are software-based programs designed to perform high-level tasks in the network, such as balancing load, defining access control list rules or monitoring traffic. The controller typically has a northbound and a southbound interface to communicate with the application and data planes, respectively. The OpenFlow communication protocol, standardised by the Open Networking Foundation (ONF) in 2011, is the de facto open-source standard for the southbound interface [28]. The OpenFlow protocol has received significant attention by the research community [23] and is currently being used in real-world SDN networks, such as Google's B4 network [17].

Using OpenFlow, the controller installs flow rules to instruct switches on how to handle packets. Switches contain several flow tables, each with its own set of flow rules. A flow rule consists of three fields: (i) matching criteria, (ii) action (e.g., drop the packet) and (iii) priority. Every time a switch receives a packet whose headers do not match any of its flow rules, the switch encapsulates the packet in an OpenFlow `packet_in` and forwards it to the controller. The latter then installs a flow rule in the switch through an OpenFlow `packet_out`. From this point onwards, if the switch receives a packet with identical headers, it uses the flow rule previously cached for as long as the flow rule remains valid.

Despite its benefits, SDN broadens the attack surface and introduces new security challenges. Several researchers have shown that it is possible to launch security attacks at the application, control and data planes [18, 24, 27, 41, 44, 45, 50, 54], while other work has proposed countermeasures for improving the security of SDN networks [5, 35, 37, 38, 47, 51, 53]. Among the proposed attacks, *topology attacks* that aim at poisoning the network topology are one of the most dangerous types. Although topology attacks are

well-known by the network security community [22, 42], the consequences of such attacks in SDN networks can be more severe than in traditional networks [2]. In traditional networks, adversaries can only tamper with the topology of a small fraction of the network by convincing a set of switches/routers of a specific (fake) topology event. Instead, SDN relies on the use of a logically centralised controller with full network visibility. As the controller contains all the network topology information, adversaries can influence any part of the network regardless of their location within the network. To further complicate matters, SDN-enabled switches lack sufficient logic and capabilities to implement traditional countermeasures such as dynamic Address Resolution Protocol (ARP) inspection.

Maintaining a genuine network topology view at the controller is of utmost importance. SDN core services and applications require real-time and accurate topology information to perform their tasks correctly. If adversaries compromise the network topology, they can redirect traffic through compromised machines. This allows bypassing middleboxes or conduct Man-in-The-Middle (MiTM) or Denial-of-Service (DoS) attacks. Furthermore, adversaries can impersonate hosts to receive their traffic. This is especially dangerous in the case of a server that handles a large amount of traffic.

Our contribution

This paper demonstrates that securing the SDN topology discovery mechanisms implies not only to design secure topology defences but also to implement the topology services at the controller correctly. Concretely, the contributions of this work are the following:

- We conduct a systematic security analysis of the state-of-the-art defences against topology attacks (see footnote[1]). This resulted in the identification of 6 vulnerabilities in *TopoGuard*, *TopoGuard+*, *Stealthy Probing-Based Verification (SPV)* and *SecureBinder*. We propose and implement attacks against TopoGuard/TopoGuard+ and provide clear evidence of other attacks against SPV and SecureBinder (Section 5).

- We discover important security vulnerabilities within the topology services in Floodlight, one of the major SDN controllers. Following the principle of responsible disclosure, we notified Floodlight about the vulnerabilities we identified. Then we introduce and practically demonstrate two novel attacks, called *Topology Freezing* and *Reverse Loop*, that can severely damage the controller's view of the network. As fully eliminating these attacks would require major changes in the Floodlight controller, we propose practical ways of mitigating such attacks (Section 6).

- Based on our findings, we also discuss possible ways of further hardening the existing topology countermeasures to defend against link fabrication and host location hijacking attacks (Section 7).

[1]We contacted the authors of TopoGuard, TopoGuard+, SPV and SecureBinder to request the source code of their solutions. They all replied to our emails and answered our questions. Unfortunately, the authors of SPV and SecureBinder were unable to share their source code with us.

Organisation. The remainder of this paper is organised as follows: Section 2 gives an overview of related work. Section 3 reviews the SDN topology discovery mechanisms and briefly summarises the topology attacks and countermeasures proposed by other researchers. Section 4 shows the laboratory setup we used for our experiments. In Section 5, we analyse the security of TopoGuard, TopoGuard+, SPV and SecureBinder and exploit weaknesses in each of them to uncover new topology attacks. In Section 6, we introduce *Topology Freezing* and *Reverse Loop*, two novel topology attacks that leverage weaknesses in the way the Floodlight controller implements its topology services. Along with each of these attacks, we propose fixes to mitigate them. Section 7 elaborates on practical ways of further enhancing the existing countermeasures. Section 8 provides concluding remarks.

2 RELATED WORK

Hong et al. [15] and Dhawan et al. [11] were the first to show how adversaries can poison the network topology view at the controller to create fake links between switches (i.e., link fabrication attacks) or impersonate a victim host (i.e., host location hijacking attacks). In response to these attacks, Hong et al. [15] and Dhawan et al. [11] devised TopoGuard and SPHINX, respectively. TopoGuard prevents these attacks by (i) adding an integrity check to topology packets (ii) labelling switch ports to avoid hosts propagating topology packets to the network and (iii) checking pre- and post-conditions (i.e., verifying that a host left the previous network location before moving to the new one). SPHINX proposes a general framework for detecting the occurrence of attacks (not only topology attacks) by validating all network updates. To achieve its goal, SPHINX constructs a flow graph of observed traffic between each pair of endpoints and compares it with past graphs in order to find anomalies. However, neither TopoGuard nor SPHINX can thwart sophisticated topology attacks. For example, adversaries can still perform host location hijacking attacks either by spoofing the victim host's MAC address or by exploiting the time that hosts are in transit, i.e., moving from one network location to another one.

Skowyra et al. found two topology attacks against TopoGuard called port amnesia and port probing [40]. They also proposed Topoguard+, an extended version of Topoguard that additionally checks for suspicious port reset events and tracks the latency of inter-switch links. The latter allows detection of link fabrication attacks by adversaries who relay topology packets using an out-of-band channel. Another approach to detect relay-based link fabrication attacks was proposed by Alimohammadifar et al. [4]. The authors developed a security solution called SPV that periodically injects probing packets to the network to find fake inter-switch links. Jero et al. introduced SecureBinder [19], a security solution that uses a modified legacy version of the 802.1x authentication protocol tp bind together all host network identifiers. However, the previous two solutions can only protect against certain attacks; SPV is only suitable for finding fake links whereas SecureBinder focuses only on preventing host location hijacking attacks.

This paper extensively analyses the security of TopoGuard, TopoGuard+, SPV and SecureBinder and discusses possible research directions to further improve them.

3 BACKGROUND

This section provides the background to understand the current topology discovery mechanisms in SDN. Afterwards, we briefly review the state-of-the-art topology attacks and defences that have been proposed by other researchers.

3.1 Topology discovery mechanisms in SDN

Below, we describe the process for discovering the network topology in SDN. Subsequently, we introduce the Host Tracking Service (HTS) and the Link Discovery Service (LDS), the two main controller core services involved in network topology discovery.

Network topology discovery is the process by which the controller learns about: (i) the network devices (e.g., switches), (ii) the links between switches and (iii) the location of the hosts within the network. The first is achieved when switches establish a TCP connection – ideally with TLS/SSL – and perform the OpenFlow handshake with the controller. All switches have a unique identifier known as DataPath ID (DPID). It is important to note that switches themselves do not support any mechanism to discover links or track hosts. For inter-switch link discovery, the controller relies on the Link Discovery Service (LDS), whereas the Host Tracking Service (HTS) is used for tracking the host's location. Next, we describe the HTS and the LDS in more detail.

The **Host Tracking Service (HTS)** maintains information about hosts (e.g., MAC and IP addresses) and their location within the network, i.e., the DPID and port number of the switch where the host is connected. To discover the host location, the HTS leverages on OpenFlow `packet_in` packets triggered when a host sends a packet for which the switch does not have any flow-rule installed. This causes the HTS to create an entry in the host profile table, binding the host identifiers to its current network location. Whenever a host migrates to a new network location, the host profile table is updated following the procedure previously described. Similarly, if a host disconnects from a switch, the latter notifies the controller by sending an OpenFlow packet containing a `port-down` event. In such a case, the HTS immediately proceeds to remove the corresponding entry in the host profile table.

The **Link Discovery Service (LDS)** discovers and keeps track of the links between switches. In most SDN controllers, the LDS is based on the OpenFlow Discovery Protocol (OFDP) [3]. Figure 1 illustrates the process by which the controller can discover a unidirectional link between two switches (denoted by S1 and S2). First, the controller encapsulates a *Link Layer Discovery Protocol (LLDP)* packet inside an OpenFlow `packet_out` packet and sends it to S1, which in turn forwards the LLDP packet to S2. Subsequently, S2 encapsulates the LLDP packet in an OpenFlow `packet_in` and sends it to the controller, allowing the LDS to discover a unidirectional link from S1 to S2. Following the same approach, the controller can also infer whether a reverse link exists, i.e., from S2 to S1. This procedure is performed regularly in order to account for the dynamics of SDN networks. Existing inter-switch links are removed (i) if a switch detects a port disconnection or (ii) if no LLDP packets are received during a certain amount of time.

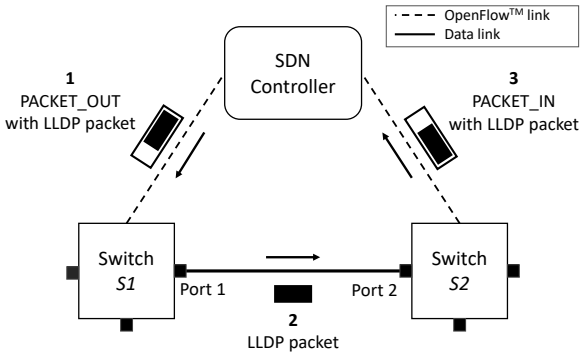


Figure 1: Procedure to discover a unidirectional link from S1 to S2 using the OpenFlow Discovery Protocol (OFDP).

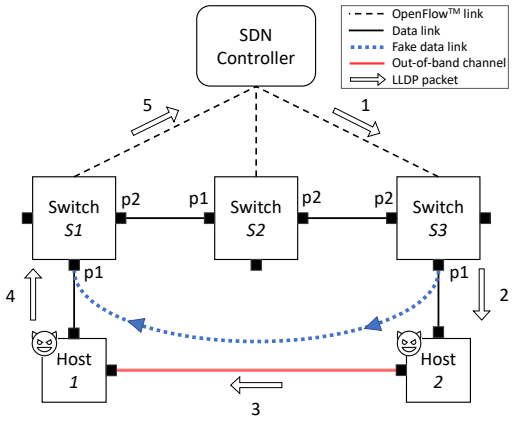
3.2 Existing topology attacks and defences

In this section, we summarise the existing topology attacks and the defences that have been proposed to mitigate them.

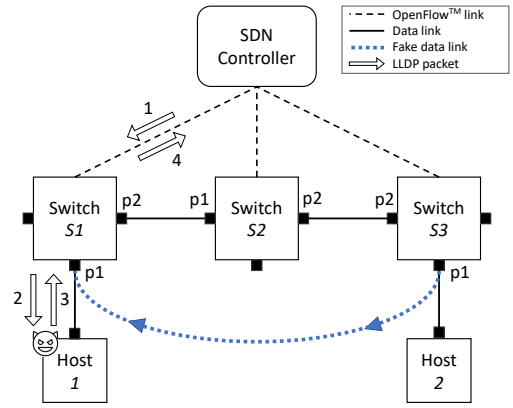
Topology attacks typically assume that adversaries have some knowledge of the network topology and can learn the network identifiers of the victim host(s). These are realistic assumptions. The network topology information can be recovered using standard path tracing tools (e.g., traceroute) or applying reconnaissance techniques (e.g., [41]). Similarly, as SDN networks use the same (insecure) protocols as traditional networks (e.g., ARP or DHCP), it is relatively simple for adversaries to obtain the network identifiers of the victim host(s).

3.2.1 **TopoGuard (NDSS'15).** Hong et al. identified two topology attacks called *host location hijacking* and *link fabrication* which have received significant attention in the last few years [15]. Hong et al. also introduced TopoGuard, a security solution that impedes adversaries from executing both attacks. TopoGuard considers adversaries who can control one or more hosts, i.e., the controller and the switches are fully trusted. TopoGuard also assumes the use of SSL/TLS to protect the control channel between the controller and each of the switches.

Attacks. In a *host location hijacking* attack, adversaries aim to convince the controller that a victim's host moved to another network location. For this purpose, adversaries who control one or more hosts can send packets using the network identifiers (e.g., the IP and/or MAC address) of the victim host. This causes the HTS to update the network location information of the victim's host. This attack can be successfully launched for as long as the victim host remains idle. On the other hand, in a *link fabrication* attack the goal of adversaries is to create fake links between switches. Figure 2 shows several ways for adversaries to create such fake links. For example, adversaries can modify legitimate LLDP packets or even craft valid ones. Another approach consists of relaying LLDP packets between two network locations using either an in-band or an out-of-band channel. In all these cases, the adversary manages to trick the controller into believing that there is a new inter-switch link when the link does not actually exist. All packets that traverse this link will be dropped or intercepted by the adversary.



(a) Link fabrication attack where H2 relays LLDP packets to H1 using an out-of-band channel.



(b) Link fabrication attack where H1 crafts and sends LLDP packets identical to those that originate from S3 port 1.

Figure 2: Link fabrication attack performed using two different methodologies: (2a) relaying LLDP packets over an out-of-band channel (2b) forging LLDP packets. In both cases, the controller believes that there is a unidirectional link from S3 port 1 to S1 port 1.

Countermeasures. Regarding the *host location hijacking* attack, Hong et al. proposed to check pre- and post-conditions before accepting a host migration as valid. The intuition behind this approach is that all genuine host migrations produce a series of events that need to occur sequentially. Essentially, this implies that a host needs to first leave its current network location before being able to connect to a new switch. More specifically, the controller first waits to receive a `port-down` from the switch where the host is initially connected, and then it checks whether the host is still reachable at the initial network location. Only if these conditions are satisfied, the controller accepts the migration as valid and the host can send packets from its new network location. Despite this being a very simple countermeasure, it enables detection of attacks where a host appears to be in two locations simultaneously.

To prevent *link fabrication* attacks, Hong et al. proposed to (i) protect the integrity of LLDP packets and (ii) avoid hosts' participation in the LLDP propagation process. To ensure LLDP integrity, a controller-signed field – computed over the DPID and the port number of the source switch – is added within LLDP packets. This prevents adversaries from crafting fake LLDP packets. To guarantee that LLDP packets are sent only to switches, a port-labelling strategy was designed to identify which type of device is connected to each switch port. This approach considers three possible states: (i) HOST, (ii) SWITCH or (iii) ANY. HOST means that there is a host connected to the switch port, SWITCH refers to the case where a switch is connected, whereas ANY is used if no device is connected. Initially, all switch ports are labelled as ANY. The port label is updated based on the first type of traffic received by the switch on each port. However, it is also important for their approach to be able to "forget" the port type. Recall that SDN networks are expected to be used in dynamic environments where a host can be unplugged and replaced by a switch (or vice versa). This requirement can be satisfied by resetting the port type to ANY every time a `port-down` event is detected, i.e., when the host disconnects from the switch. This port-labelling strategy prevents adversaries who control more than one host from relaying LLDP packets, since these packets are only sent to switch ports.

3.2.2 **TopoGuard+ (DSN'18).** Skowyra et al. presented two new topology attacks called *port amnesia* and *port probing* that can be successfully conducted even in the presence of TopoGuard [40]. Furthermore, Skowyra et al. designed and implemented an extension of TopoGuard, called TopoGuard+, which not only prevents *port amnesia* attacks but also detects link fabrication attacks based on relaying LLDP packets through an out-of-band channel. TopoGuard+ defends against adversaries who control one or several hosts. The controller and the switches are assumed to be trusted.

Attacks. In the *port amnesia* attack, the goal of the adversary is to bypass the port-labelling technique proposed in TopoGuard. Adversaries can disconnect and reconnect the network interfaces of their hosts to reset the switch ports to ANY. This can let the hosts emulate the behaviour of switches to transmit (fake) LLDP packets to the controller.

In the *port probing* attack, the adversary circumvents the mechanisms used in TopoGuard to thwart *host location hijacking* attacks by exploiting the time it takes for a victim's host to migrate to a new network location. This attack leverages the fact that the host's identifiers are not bound to any network location while hosts are in transit. A technique was proposed to stealthily and accurately detect the moment that the victim's host leaves its network location. Even more, the authors demonstrated that a host migration can be maliciously triggered remotely.

Countermeasures. TopoGuard+ extends TopoGuard by including two new modules: (i) the Control Message Monitor (CMM) and (ii) the Link Latency Inspector (LLI). The CMM enables the controller to identify suspicious port-type resets during LLDP propagation. For this purpose, the controller monitors the traffic and raises an alert if `port-up` or `port-down` are received while a LLDP packet is in progress. This makes TopoGuard+ resistant to port amnesia attacks. Nevertheless, the CMM module cannot detect link

fabrication attacks that rely on the use of an out-of-band channel. To defend against such attacks, the LLI module is used. This module detects fake links by keeping track of the latencies of the genuine links between switches.

3.2.3 **Stealthy Probing-Based Verification (ESORICS'18).** Alimohammadifar et al. presented SPV, a stealthy probing-based verification approach for detecting any type of link fabrication attack [4]. Similarly to most existing work, SPV assumes that the SDN controller is trusted and that the control channels between the SDN controller and the switches are protected. In contrast to other works, SPV considers adversaries who can control not only hosts but also a few switches within the network. The authors assume that adversaries can use a *low-bandwidth* out-of-band channel to create fake inter-switch links. However, the authors acknowledge the fact that SPV cannot defend against adversaries who forward all traffic through the out-of-band channel, since this would create a link that actually resembles genuine links in the network.

Countermeasures. To verify the legitimacy of inter-switch links, SPV relies on the use of probing packets that are indistinguishable from normal traffic. For this, SPV listens to the network traffic and maintains a list of reference packets sent by hosts. This also includes the DPIDs of the switches from where these packets were sent. For validating a link, SPV chooses a reference packet at random from the ones previously stored in the list. To guarantee the security of SPV, the reference packet cannot be a packet that has previously been used by any of the two switches involved in this link. SPV proposes using probing packets where some fields match those of normal traffic and some fields are randomised. Specifically, the probing packet takes the *Ethernet_type* and *Payload length* from the randomly-chosen reference packet while the source and destination MAC/IP addresses are chosen at random. Similarly to the OFDP protocol, SPV sends the probing packet to the sender switch which in turn forwards it to the destination switch. Upon receiving the probing packet, the destination switch sends it back to the controller. The core idea of their approach is simple yet effective; if the probing packet returns to the controller, there exists a link between these switches. Otherwise, SPV concludes that the link is fake and removes it from the network topology view at the controller.

Additionally, a mechanism was designed to handle lost probing packets (e.g., due to link failures). In such a case, SPV generates a new probing packet by fetching the first probing packet and using the LineSweep algorithm [21, 32]. The new probing packet, which is just slightly different from the first one, is then sent to the sender switch following the procedure previously described.

3.2.4 **SecureBinder (USENIX'17).** Jero et al. discovered an attack called *Persona Hijacking* that takes advantage of the inherent weaknesses in the identifier binding mechanisms in SDN [19]. Besides proposing a very effective and dangerous attack, Jero et al. introduced SecureBinder, a defence mechanism that can be used to defeat host location hijacking attacks, including the port probing attack introduced in TopoGuard+. SecureBinder assumes that adversaries can control one or more hosts. The rest of the network components are considered to be trusted.

Attacks. *Persona Hijacking* comprises two phases: (i) *IP takeover* and (ii) *flow poisoning*. In the *IP takeover* phase, the goal of the adversary is to break the binding between the IP address and the MAC address of the victim's host. The adversary can successfully launch this attack if it convinces the DHCP server to release the victim's IP address so that it can bind its own MAC address to it. The *flow poisoning* phase is needed only when the DHCP server checks if the IP address is in use before assigning it to a new host. This phase consists of all the necessary steps to break the binding between the victim's MAC address and its network location. Essentially, the adversary exploits a flow rule inconsistency on a switch to redirect traffic to itself. This attack can let adversaries fully takeover and become the owner of the victim's identifiers.

Countermeasures. SecureBinder binds together all hosts' identifiers using a modified legacy version of the 802.1x authentication protocol [34] that additionally checks if the hosts MAC addresses are valid, i.e., within the list of authorised hosts. The controller takes the role of the authenticator, allowing the host (i.e., the supplicant) to access the network after authenticating successfully. The authenticator server, which is connected to the controller, contains a database that binds each host's MAC address with its certificate.

In addition, SecureBinder leverages the SDN architecture to ensure that all binding control traffic is sent directly to the controller (instead of being broadcasted to the network). This prevents adversaries from sniffing the control packets exchanged to establish those bindings and allows the controller to perform several cross-layer checks for validating the bindings when they are updated.

4 LABORATORY SETUP

Figure 3: Our hardware SDN network is composed of three switches (i.e., Raspberry Pi 3 Model B) connected with each other through a linear topology, a controller (i.e., Apple MacBook Pro) and several hosts (not shown in the image) connected to the end of the Ethernet cables. All inter-switch and controller-switch links are 100 Mb/s.

In the next sections, we analyse the security of the state-of-the-art topology defences and the topology core services in Floodlight. For this, we have performed experiments in an emulated environment using Mininet 2.3.0 [43] and in a hardware SDN network. Our hardware SDN network, shown in Figure 3, comprises three Raspberry Pi 3 Model B [12] acting as OpenFlow switches, a controller running on an Apple computer and several hosts that are implemented either in other Raspberries or in a fixed Desktop PC using Linux. To allow multiple SDN-enabled switches to communicate with the controller, we used a traditional L2 Ethernet switch whose only function is to forward the OpenFlow packets from the controller to the switches (and vice versa). We chose to use the Open vSwitch [13] version 2.5.5 LTS as a switch[2], while our controller is based on Floodlight [30]. The choice of Floodlight was motivated by the fact that most existing topology defences are implemented on it. The controller was installed on a 64-bit Ubuntu 14.04 VM with two cores of 2,8 GHz Intel Core i7 and 8GB of RAM.

5 SECURITY ANALYSIS OF THE TOPOLOGY DEFENCES

In this section, we evaluate the security and propose new attacks against TopoGuard, TopoGuard+, SPV and SecureBinder.

5.1 TopoGuard/TopoGuard+

As TopoGuard+ integrates all the security mechanisms used by TopoGuard, we refer to the joint solution as TopoGuard+. One of the main design goals of TopoGuard+ is to preclude any link fabrication attack regardless of its nature. Despite TopoGuard+ mitigating relay-based link fabrication attacks to a large extent, we identified two new vulnerabilities in the *mechanisms to track link latencies* as well as in the *LLDP packet generation*.

To exploit the weaknesses in the mechanisms to track link latencies, we need to overload switches to increase the latency of the inter-switch links. Intuitively, this could be a possible limitation of our attacks since SDN-enabled hardware switches could incorporate mechanisms to defend against overloading by malicious hosts. However, we want to stress that our findings and attacks can be extrapolated to real-world SDN networks for several reasons. First, SDN-enabled hardware switches contain simple CPUs, which restrict their capabilities for parsing and processing packets [10, 46]. Second, SDN-enabled hardware switches have a small flow table space that can only accommodate from hundreds to a few thousand flow rules [10, 20, 26]. For example, a widely used SDN-enabled hardware switch like *Pica8* can only support 8192 flow entries [1]. Likewise, the rate at which flow tables can be updated is limited. As a result, SDN-enabled hardware switches can only handle 100-200 flow rule updates per second [9, 14, 20, 41, 47, 48]. The previous two limitations arise from the fact that SDN-enabled hardware switches achieve wire-speed packet processing using Ternary Content Addressable Memory (TCAM), which is costly and power hungry. Finally, Zhang et al. demonstrated that hosts do not need to directly send packets to switches to overload them [54]. Instead, hosts can trigger the controller into sending a sufficient number of packets to overload switches more effectively.

[2]Open vSwitch supports OpenFlow 1.4 protocol (and earlier).

5.1.1 Insecure mechanisms to track link latencies.

As previously described, TopoGuard+ relies on the LLI module for measuring the latency of the inter-switch links.

In TopoGuard+, LLDP packets contain a fresh encrypted timestamp so that the controller can measure the overall time between sending and receiving an LLDP packet, (i.e., T_{LLDP}). For computing the latency of an inter-switch link (e.g., T_{S1-S2}), the controller subtracts the latencies of the control links (i.e., T_{S1} and T_{S2}) from T_{LLDP} (see Figure 4). The LLI module then compares T_{S1-S2} with a threshold that is determined using an interquartile range of the list of valid latencies (see Algorithm 1). If T_{S1-S2} is within the valid range of latencies, the LLDP packet is processed correctly and T_{S1-S2} is added to the list of valid latencies. Otherwise, TopoGuard+ raises an alert and removes the link if the failure persists over time.

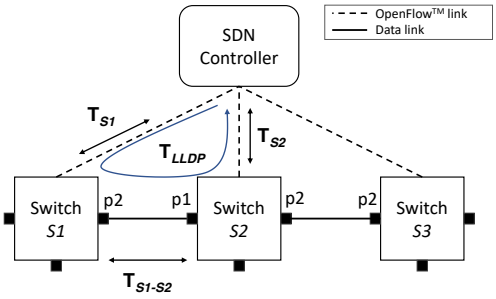


Figure 4: Link Latency Inspector (LLI) module. The latency of the link between S1 and S2, T_{S1-S2}, is obtained as follows: $T_{S1-S2} = T_{LLDP} - T_{S1} - T_{S2}$. T_{LLDP} is the time between sending and receiving an LLDP packet at the controller. T_{S1} and T_{S2} are the control link latencies of S1 and S2, respectively.

Algorithm 1 Procedure to compute the threshold

```
1: if delay != 0 then
2:     q1 ← quartile(latency_list, 25)
3:     q3 ← quartile(latency_list, 75)
4:     interquartile_range ← q3 − q1
5:     threshold ← q3 + 3 ∗ interquartile_range
6: if delay > threshold then
7:     stop processing LLDP packet
8: else
9:     add delay to latency_list
```

Below, we present two attacks against the LLI module where adversaries influence the link latencies to remove genuine links or to create fake ones.

Attack 1. We discovered a new attack against TopoGuard+ that allowed us to remove genuine links between switches. Our attack leverages the fact that the controller removes existing links if their latency is above the threshold in a few LLDP rounds.

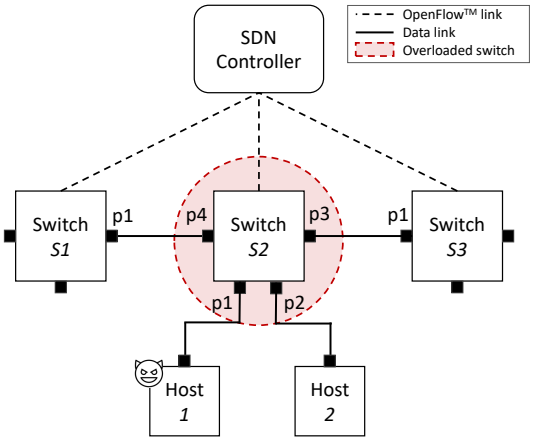
Without loss of generality, let us describe the proposed attack using the network topology shown in Figure 5a. During normal execution, we observed that the latency of the links between switches was approximately 4 ms (see third line in Figure 5b). Nevertheless, we found a way to increase the latency of the links between switches using H1.

Initially, we conducted a series of experiments with different numbers of packets and time between packet bursts then measured how each strategy affected the latency of the inter-switch links. Based on our experiments, we chose the smallest number of packets that can sufficiently overload S2 to drop its links to other switches. Specifically, we injected bursts of 100 packets every 1 s with spoofed source MAC addresses and the destination MAC address of H2. This forced S2 to constantly request new flow rules to the controller, consuming a significant amount of its resources. Figure 5 shows that TopoGuard+ started to report errors due to high latencies (around 140 ms) and shortly after it began to remove the affected links. Even if TopoGuard+ re-discovers these links after our attack, we observed that this causes the controller to lose all the previous information about them.

Attack 2. Recently, Shrivastava et al. were able to perform a relay-based link fabrication attack against TopoGuard [39]. Yet, this is no surprise since TopoGuard was *not* designed to preclude those attacks. In this paper, we demonstrate that TopoGuard+ is also vulnerable despite the effort of their authors to protect against such attacks. In order to execute our attack, adversaries need to increase the latency threshold until it becomes comparable to the latency of their out-of-band-channel. However, this is not an easy task since the latency threshold (i) depends only on the latencies of the links that were previously marked as valid and (ii) is computed using an interquartile range, which helps finding outliers.

Without loss of generality, let us describe our attack using the network topology shown in Figure 5a. An adversary who controls H1 can send a large number of packets to S2, resulting in an abrupt increase in its resource consumption. Consequently, S2 either processes the LLDP packets very slowly or drops them. Using this method, the adversary cannot increase the latency threshold at her will since the latency of the LLDP packets traversing S2 will fall outside the range of valid latencies. Instead, the adversary should opt for carefully overloading S2 over a longer period of time, increasing the time it takes for S2 to process the LLDP packets only slightly each time. By repeatedly doing so, adversaries can delay LLDP packets in such a way that the latency threshold is gradually increased. As TopoGuard+ uses a single latency threshold for the network, adversaries can mount this attack regardless of their location within the network.

5.1.2 **Lack of freshness in LLDP packets.** In TopoGuard+, the controller appends a MAC tag – computed over the DPID and the port number of the source switch – to all LLDP packets. While the authors of TopoGuard+ stated that it is essential to protect the integrity of LLDP packets, their approach lacks freshness. This makes it possible to reuse MAC tags to create valid LLDP packets.



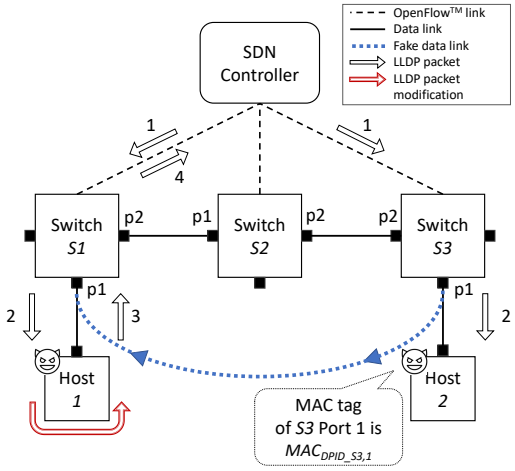
(a) Malicious H1 overloads S2.

```
10:31:38.915 INFO LINK delay between sw 3 and 2 is okay. delay:1ms, threshold:9ms
10:31:38.917 INFO LinkLatencyQueue.size: 395
10:31:38.918 INFO LINK delay between sw 1 and 2 is okay. delay:4ms, threshold:9ms
10:31:38.918 INFO LinkLatencyQueue.size: 396
10:31:38.918 INFO LINK delay between sw 2 and 1 is okay. delay:5ms, threshold:9ms
10:31:38.918 INFO LinkLatencyQueue.size: 397
10:31:38.918 INFO LINK delay between sw 2 and 3 is okay. delay:4ms, threshold:9ms
10:31:53.923 INFO LinkLatencyQueue.size: 398
                                    ...
10:33:24.303 ERROR Detected suspicious link discovery: abnormal delay during LLDP propagation
10:33:24.303 ERROR LINK sw 2 and 3
10:33:24.303 ERROR Link delay is abnormal. delay:82ms, threshold:9ms
10:33:24.303 ERROR Detected suspicious link discovery: abnormal delay during LLDP propagation
10:33:24.303 ERROR LINK sw 3 and 2
10:33:24.303 ERROR Link delay is abnormal. delay:142ms, threshold:9ms
10:33:24.303 ERROR Detected suspicious link discovery: abnormal delay during LLDP propagation
10:33:24.303 ERROR LINK sw 2 and 1
10:33:24.303 ERROR Link delay is abnormal. delay:145ms, threshold:9ms
10:33:24.304 ERROR Detected suspicious link discovery: abnormal delay during LLDP propagation
10:33:24.304 ERROR LINK sw 1 and 2
10:33:24.304 ERROR Link delay is abnormal. delay:143ms, threshold:9ms
                                    ...
10:33:54.081 INFO Inter-switch link detected: Link [src=00:00:00:00:00:00:00:02 outPort=4,
dst=00:00:00:00:00:00:00:01, inPort=1]
10:33:54.081 INFO LinkLatencyQueue.size: 403
10:33:54.081 INFO Inter-switch link detected: Link [src=00:00:00:00:00:00:00:01 outPort=1,
dst=00:00:00:00:00:00:00:02, inPort=4]
10:33:54.081 INFO LinkLatencyQueue.size: 404
10:33:54.081 INFO Inter-switch link detected: Link [src=00:00:00:00:00:00:00:03 outPort=1,
dst=00:00:00:00:00:00:00:02, inPort=3]
10:33:54.082 INFO LinkLatencyQueue.size: 405
10:33:54.082 INFO Inter-switch link detected: Link [src=00:00:00:00:00:00:00:02 outPort=3,
dst=00:00:00:00:00:00:00:03, inPort=1]
10:33:54.088 INFO LinkLatencyQueue.size: 406
```

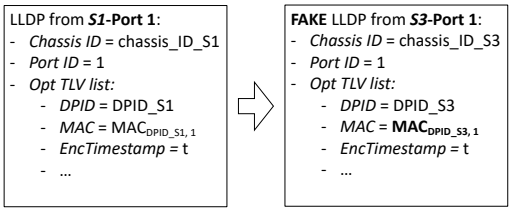
(b) Floodlight log console. Initially, the latencies of the inter-switch links (i.e., S1-S2 and S2-S3) are valid and hence they are added to the latency list. Subsequently, LLI detects a number of links with higher latencies and stops processing the corresponding LLDP packets. After a few LLDP iterations (approximately 20 s later), the controller removes the links between S1-S2 and S2-S3 and then re-discovers the links as new links.

Figure 5: Attack against the Link Latency Inspector (LLI) module proposed by TopoGuard+. (5a) Network scenario. (5b) Log console.

Attack. We tested the feasibility of this attack using our hardware SDN network comprising two hosts, three switches and a controller running TopoGuard+[3] (see Figure 6). Our goal was to create a fake unidirectional link from S3 to S1 using two malicious hosts (H1 and H2) that can communicate over an out-of-band channel. Note that we assume that the ports of the switches where H1 and H2 are connected are initially set to *HOST*.



(a) H2 sends the MAC tag of S3-Port1 inside the LLDP packet to H1 such that the latter can forge valid LLDP packets as if they originated from S3 port 1.



(b) H1 modifies the LLDP packet that it receives from S1. It keeps all the fields excluding the Chassis ID and the DPID, which are easily derivable, and learns the valid MAC tag from H2.

Figure 6: Link fabrication attack against TopoGuard+ where adversaries manage to forge valid LLDP packets.

As in the port amnesia attack proposed in TopoGuard+, the first step of our attack is to disconnect and reconnect the network interfaces of both hosts (e.g., by unplugging the cables). This causes the controller to reset their port type to *ANY*. For our attack to succeed, the resetting needs to be done before an LLDP round starts so that the CMM module does not flag these events as suspicious. Then, H2 waits to receive an LLDP from the controller, which contains a valid MAC for the corresponding DPID and port number of S3, i.e., the source switch. At this point, H2 sends the valid MAC tag to H1 over the out-of-band channel. Due to the lack of freshness, the MAC tag needs to be exchanged only once. Upon learning the MAC, H1 can successfully send LLDP packets as if they were originating from S3 port 1. This triggers the controller into believing that there is a real unidirectional link from S3 to S1. We want to highlight that, as the process of creating this link does not involve relaying LLDP packets, the LLI module is not capable of detecting our attack.

[3]We downloaded the source code of TopoGuard+ from its public repository in Git [49] on Oct 2nd, 2018.

5.2 Stealthy Probing-Based Verification (SPV)

By conducting a thorough analysis of SPV, we found two weaknesses in the way the probing packets are generated and sent to the switches. These vulnerabilities mainly stem from the fact that network traffic is not completely random in practice. Therefore, any defence that uses probing packets is likely to have similar issues to the ones we identified in SPV.

5.2.1 **Insufficient obfuscation in the probing packet.** SPV periodically verifies the legitimacy of all inter-switch links by sending probing packets that resemble normal traffic. Yet, probing packet generation is a very fragile task since ideally it requires (i) having a large amount of network traffic sent by hosts and (ii) obfuscating the probing packets at different levels.

Attack. To validate a link while concealing the probing packet from adversaries, SPV should ideally send the probing packets at random times. However, as SPV uses newly discovered links before verifying them, it is likely that one of the first packets sent to this link is a probing packet. Clearly, it is undesirable to use an unverified link for a long time. This observation reduces the randomness of the process and increases the chances that adversaries can identify the probing packets. To further improve their chances, adversaries who control a large set of malicious hosts can inject enough packets in the network to influence the generation of probing packets. These packets can contain `Ethernet_type` and `Payload size` fields that are unique in the network. In both cases, whenever adversaries receive a packet that is likely to be a probing packet, they can simply forward it over the out-of-band channel. As discovered links are used before SPV verifies them, even if adversaries fail to infer some probing packets, they can still have a sufficiently large time window to perform their attacks.

The previous two observations show that it is very difficult to create probing packets that resemble normal traffic regardless of how these are created.

5.2.2 **Use of the LineSweep algorithm.** Another important weakness of SPV originates from the use of the *LineSweep algorithm* for generating new probing packets when the initial probing packet fails to arrive to the destination. The LineSweep algorithm takes the initial probing packet and slightly modifies it such that the new probing packet is not completely different than the initial one. Concretely, it chooses new IP and MAC addresses that are adjacent to the ones used in the first probing packet, e.g., in the same subnet.

Attack. The use of the LineSweep algorithm can expose useful information to infer what packets are used for validating a link. In a large network with many hosts, the probability of two hosts connected to the same switch having adjacent IP and MAC addresses is very low. Following this reasoning, adversaries can initially drop the traffic they receive – including the first probing packet – and wait until the second probing packet is sent. As the subsequent

probing packets will have almost identical IP and MAC addresses, adversaries can identify and relay these probing packets using their out-of-band channel. This way, adversaries can extend the time window the controller uses this link for routing purposes before it is removed from the network.

5.3 SecureBinder

SecureBinder thwarts host location hijacking attacks using a slightly different version of a legacy 802.1x authentication protocol. We acknowledge that SecureBinder clearly enhances the security of the actual binding mechanisms in SDN. Nevertheless, we identified two potential weaknesses in the way the modified legacy version of the 802.1x authentication protocol is used.

5.3.1 **Disconnect the good and connect the bad.** SecureBinder is an extension of a standard legacy 802.1x authentication protocol that allows hosts to authenticate to the network. Before a host is authenticated, the switch port where the host is connected accepts only "authentication traffic" (i.e., 802.11x frames). In the 802.1x protocol used by SecureBinder, hosts are authenticated only once each time they change their location in the network (instead of in every packet). The choice of extending a legacy 802.1x protocol was motivated by the fact that per-message authentication would incur a large overhead in the controller. However, we observed that this decision comes with important security implications.

Our hypothesis was that, if adversaries can connect a (malicious) host to the network location where the victim's host is located without triggering a `port-down`, they can bypass SecureBinder and join the network without needing to authenticate themselves to the authentication server. To force the victim's host to move to another network location, adversaries could follow an approach similar to the one used in the port amnesia attack [40]. Crucial to the proposed attack is that during the authentication procedure the host does not establish any cryptographic session key with either the controller or the switch. As a result, the packets transmitted by hosts after completing the authentication protocol are neither encrypted nor authenticated.

According to SecureBinder, it should be impossible to disconnect and connect a host without triggering a `port-down` and a `port-up` event even when the adversary has *specialised equipment* and *physical access* to the devices. We envision a scenario where the physical hosts can contain several Virtual Machines (VMs) and a virtual switch for routing the packets to/from each VM. In such a case, the virtual switch (inside the physical host) should always inform the corresponding physical switch when a VM is (dis)connected so that the VM can authenticate to the controller using SecureBinder. The way this is done depends on how the virtual switch is designed and programmed. However, it is important to note that all the events generated by the VMs should be treated as if they were originating from a physical host connected to a physical switch.

Attack. We conducted several experiments to investigate if it is possible to disconnect a (victim) host from its switch/port and connect a (malicious) host fast enough such that the switch does not notice about this disconnection. According to the IEEE 802.3 standard, if twisted pairs Ethernet connections are used between switches and hosts, a signalling protocol is used where a link integrity pulse is sent by these devices every 16 ± 8 ms [16]. In other words, a switch infers that a host is no longer connected if it does not receive such a pulse in 24 ms. If adversaries can remotely detect or trigger a (victim) host disconnection and immediately connect a (malicious) host to the switch, they could benefit from the valid ongoing communication session initiated by the victim's host.

We started our experiments by running the command `ifconfig eth0 down && ifconfig eth0 up` on a Mininet host to measure the average time between disconnecting and re-connecting a network interface. This test resulted in a delay of only 8 ms, which indicates that it could be possible to circumvent SecureBinder in order to add a malicious host in the network. (Recall that switches can only detect that a host is no longer connected after 24 ms). However, we observed that Mininet switches always detect the port disconnection and send a `port-down` to the controller. In contrast to real OpenFlow switches, Mininet switches do not implement any signalling protocol with the host and thus they do not check the port liveness before notifying the controller about the `port-down`. In other words, Mininet emulates the disconnection of a host from a switch without considering the delays introduced by the signalling communication protocol between the host and the switch.

Subsequently, we used Wireshark to measure the average time between a `port-down` and a `port-up` on the control plane. Similarly to the previous test, this experiment also resulted in a delay of 8 ms. Motivated by the results obtained in Mininet, we tested our hypothesis using our hardware SDN network with two Raspberry Pi 3 acting as a switch and host, respectively. However, in this case the previous commands took 68 ms, whereas the average time between the `port-down` and `port-up` was around 1630 ms on average.

While our preliminary results indicate that SecureBinder is capable of detecting our attack, we suggest that topology defences should not base their security solely on the fact that `port-down` and `port-up` events are all genuine (i.e., they are generated if and only if a host really (dis)connects from/to a switch). As a future work, we plan to further investigate how to accelerate the process of disconnecting and reconnecting a network interface.

5.3.2 **Insecure low level bindings.** SecureBinder provides a weak binding between the hosts' MAC address and their network location. More specifically, the main limitation of SecureBinder is that it does not bind the authentication traffic to the switch/port where the host is connected.

Attack. Adversaries can intercept the authentication traffic from/to a victim host and replay it in a different network location. If adversaries replay the victim's authentication traffic to a switch whose path to the controller is faster, they can convince the controller that the victim's host is at their location. This attack can be possible (i) if the adversary manages to insert an Ethernet hub in the network or (ii) when the victim host and the adversary have two VMs running on the same physical host.

Figure 7 shows a realistic network scenario where this attack could be mounted. This network configuration is known as *in-band SDN* and is widely used in practice [8, 36]. Unlike out-of-band SDN configurations, where all switches can directly communicate to the controller, only a few switches interact with the controller in an in-band SDN configuration.

In an in-band SDN configuration, adversaries can intercept the (plain) authentication traffic from/to H1 (e.g., using an Ethernet hub) and replay it to S2 using H2. If the link between S1 and S2 is slower than the path between S2 and the controller, the authentication traffic sent by the adversary using H2 is received first. This causes the controller to believe that H1 is connected to S2. To improve the effectiveness of this attack, the adversary could overload S1 to deliberately increase the latency of the link between S1 and S2. The main advantage of our attack is that, if H2 succeeds in convincing the controller that H1 is connected to S2, H2 not only receives all the traffic to H1 but also prevents H1 from receiving its traffic. Even if the genuine authentication traffic sent by H1 is eventually received and validated by the controller, the previously installed flow rules can lead to inconsistencies in the data plane.

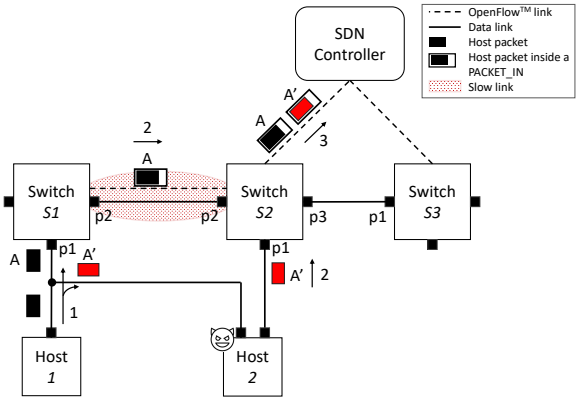


Figure 7: Possible attack scenario against SecureBinder using an in-band SDN network.

6 IMPLEMENTATION ATTACKS

In 2018, Nehra et al. showed that most SDN controllers lack security mechanisms to protect the network topology information [29]. Motivated by their study, we manually inspected the source code of the Floodlight controller to find potential ways of tampering with the network topology view at the controller[4]. Our analysis resulted in the identification of two new attacks called *Reverse Loop* and *Topology Freezing*. These attacks do not require the controller or the switches to be compromised, and assume that the control channel between switches and the controller is protected using TLS/SSL.

Although these attacks are specific to the Floodlight controller, these (or similar) attacks are also likely to exist in other major SDN controllers. Most controllers rely on the standard OFDP protocol, each with minor message field variations [29]. In addition, the OFDP protocol is not well defined and lacks security mechanisms to protect packet integrity and confidentiality. This results in ad hoc insecure implementations.

[4]We downloaded the Floodlight controller version 1.2 from the official Git repository on Sept 5, 2018 [31].

6.1 Reverse Loop

We unveil a new attack – which we call *Reverse Loop* – that exploits a weakness in the way the LDS handles the *LINK-TYPE* field inside the LLDP packets. Before describing our attack, let us first briefly explain what the purpose of the LINK-TYPE field is. Suppose that S1 and S2 are connected to each other and the controller does not know yet about the existence of a link between them. Initially, the controller sends an LLDP packet with the LINK-TYPE field set to '0x01' to S1 which in turn sends it to S2. Once the controller infers a unidirectional link from S1 to S2, it immediately checks if the reverse link (i.e., from S2 to S1) exists by sending an LLDP packet to S2. However, in this case the LINK-TYPE field is set to '0x02' (instead of '0x01'). Note that the controller regularly repeats this procedure with all switches to collect information about existing or new inter-switch links.

The core idea behind the *Reverse Loop* attack is to extend the duration of an LLDP round as much as possible to exhaust the controller resources, potentially leading to crashes. To illustrate how the *Reverse Loop* attack works, let us give an example using the network topology shown in Figure 2b. Essentially, the adversary proceeds in the same way as when a fake link is created from S3 to S1. Specifically, the adversary (i.e., H1) starts by sending a maliciously crafted LLDP packet containing the headers associated with S3. The controller then proceeds as expected and sends an LLDP packet with the LINK-TYPE field set to '0x02' to S1 to validate if a reverse link exists from S1 to S3. Subsequently, the adversary transmits the LLDP packet with the LINK TYPE field set to '0x01'. Crucially, we found that re-sending the initial LLDP packet (with the LINK TYPE field set to '0x01') triggers the controller into checking if the reverse link exists indefinitely. We then discovered an even more powerful variant of our attack that induces the controller into continuously computing the topology instance. This variant leverages the fact that the LDS re-computes the topology instance every time the latency of a link changes. (Note that from Floodlight version 1.2 onwards, LLDP packets contain a time-stamp that is used to determine the link latency).

To test the practicality and severity of the *Reverse Loop* attack, we conducted a series of experiments where we instructed H1 to send LLDP packets with a slightly modified time-stamp to force the controller into recomputing the topology instance. We tested our attack in Mininet using a tree topology with depth 4 and fanout 3, comprising 81 hosts and 40 switches. Figure 8 illustrates the resource consumption of the Floodlight process inside the controller before and during the attack. To measure the CPU performance, we used the `top` command on the Linux shell. This test demonstrates that computing the topology instance is a very demanding task that can cause the controller to crash. While we did not evaluate it, the *Reverse Loop* also results in more LLDP packets being sent by the controller, which can negatively impact the network bandwidth.

We highlight that the *Reverse Loop* attack can facilitate the creation of fake links using an out-of-band channel when TopoGuard+ is deployed. As the controller always sends an LLDP packet to check for the reverse link whenever it receives a *valid* LLDP packet, our attack can indicate whether the LLI module accepted the delayed LLDP packet. Recall that the controller checks for the reverse link only if the latency of the LLDP packet is within the valid range.

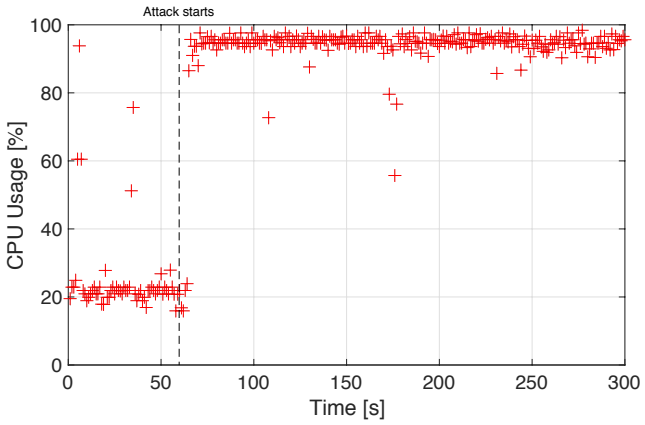


Figure 8: CPU usage of the Floodlight controller during the Reverse Loop Attack. In this example, the attack starts after approximately 60 s.

The fundamental reason this attack is possible is because LLDP packets lack integrity protection that guarantees that they have not been tampered with, and originate from the sender switch. Protecting the integrity of LLDP packets would mitigate such attacks to a large extent, but this would require major changes in the Floodlight controller. Additionally, we argue that the use of an anomaly detection system could also help to keep track of the number of times a certain LLDP packet is sent.

6.2 Topology Freezing

Our second attack – which we call *Topology Freezing* – affects the module responsible for computing the topology instance. It can be launched to "freeze" the current topology instance, preventing the controller from updating part of the network topology view.

Essentially, our attack is based on the following observation: When two links are created that originate from the same "origin" switch/port (e.g., S1 port 1) but end in two different network locations (e.g., S4 port 1 and S5 port 1), the LDS accepts both links and treats the "multi-link" port as a broadcast domain port (see Figure 9). As a result, the LDS removes the "origin" port and its links from the topology graph construction. Yet, we made the crucial observation that these links are still being used by the controller to compute the shortest path (using an outdated topology instance), causing systematic runtime exceptions.

To show the consequences of performing our attack, we implemented it in Mininet using the network topology shown in Figure 9. As we configured the network so that the shortest path between two hosts is determined based on the number of hops, H3 and H6 initially communicate through Link A. (Note that our attack can be executed regardless of the metric used to compute the shortest path between a pair of hosts). To execute our attack, we used H4 and H5 to create two fake links between (i) S1-Port1 and S4-Port1 and (ii) S1-Port1 and S5-Port1. From this moment onwards and for as long as S1-Port1 contained two links, the controller was unable to fully update the topology instance and the forwarding module, as shown in Figure 10.

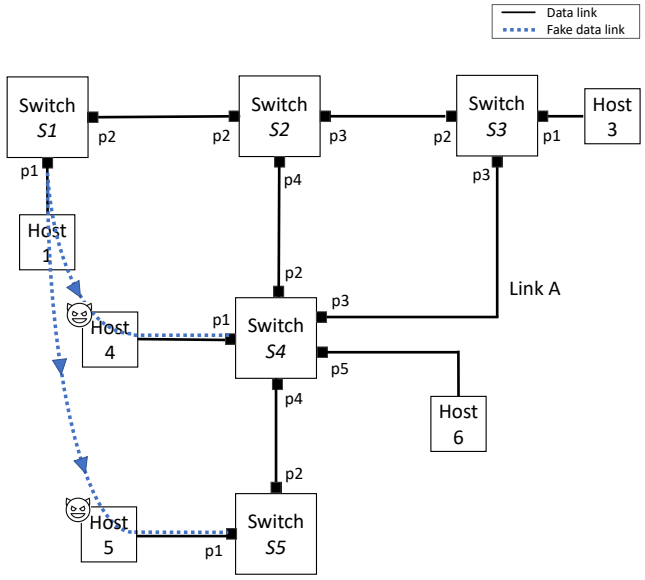


Figure 9: Topology Freezing Attack. H4 and H5 can successfully create two fake links originating from S1 Port1.

```
ERROR [n.f.t.TopologyManager] Error in topology instance task thread
java.lang.NullPointerException: null
at net.fl[..].topology.TopologyInstance.dijkstra(TopologyInstance.java:624)
at net.fl[..].topology.TopologyInstance.yens(TopologyInstance.java:1068)
at net.fl[..].topology.TopologyInstance.computeOrderedPaths(TopologyInstance.java:790)
at net.fl[..].topology.TopologyInstance.compute(TopologyInstance.java:168)
at net.fl[..].topology.TopologyManager.createNewInstance(TopologyManager.java:1015)
at net.fl[..].topology.TopologyManager.updateTopology(TopologyManager.java:208)
at net.fl[..].topology.TopologyManagerUpdateTopologyWorker.run(TopologyManager.java:179)
at net.fl[..].core.util.SingletonTaskSingletonTaskWorker.run(SingletonTask.java:69)
at java.util.concurrent.ExecutorsRunnableAdapter.call(Executors.java:511)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
```

Figure 10: Log file in the Floodlight controller when freezing the network topology.

Subsequently, we removed Link A from the network topology and observed that H6 can no longer receive packets from H3. This is because the controller keeps using an old topology instance and hence uses Link A (which no longer exists) to carry the packets between H3 and H6. It is interesting to see that even if the controller learns about the existence of a new legitimate link between two switches, it cannot update the topology instance and thus cannot use the link. Therefore, adversaries can execute this attack to freeze the network topology and act maliciously without being noticed by the controller. We want to stress that this attack can complement and increase the impact of existing SDN attacks. To maximise its outcome, adversaries need to carefully choose the switch/port from where they launch the attack. Updating the topology instance is done sequentially based on the switches' DPID, starting from the switch with the lowest DPID. If the multi-link port is created on the switch with the lowest DPID, the rest of the network is no longer updated. The best strategy is therefore to perform the attack from the switch with the lowest DPID. As the Floodlight controller assigns the DPID to each switch based on its MAC address, finding the switch with the lowest DPID is straightforward.

Fixing this vulnerability is not easy. The most intuitive solution to defend against it would be to check if the link exists (e.g., by pinging the other host) before using the link to send real traffic.

7 PRACTICAL COUNTERMEASURES

To defend against topology attacks, SDN networks could simply use a static network configuration similar to traditional networks (e.g., limiting the number of MAC addresses in a switch port) [2]. However, this is a tedious and error-prone approach that does not account for the dynamics of SDN networks. Re-designing the existing topology discovery mechanisms would provide better security and scalability guarantees [6, 33]. Yet, this is not a viable option due to its difficult adoption. In this section, we propose simple yet effective ways of extending the existing countermeasures to defend against the attacks we found in Section 5.

Link fabrication attacks by forging LLDP packets. Protecting the integrity of LLDP packets is crucial to prevent adversaries from sending forged LLDP packets. As shown in Section 5, some of the defences that were proposed to solve this problem lack freshness, allowing replay attacks. For example, TopoGuard proposed to compute a MAC tag over the DPID and port of the source switch using a cryptographic key that is only known to the controller. To provide integrity protection with freshness, Alharbi et al. presented a solution where the cryptographic key is updated in every LLDP round [3]. The main limitation of their approach is that it requires the controller to keep track of the keys used in each LLDP round. To overcome the limitations of the previous solutions, we suggest computing the MAC tag over the DPID, the port and a time-stamp using a single cryptographic key (instead of using a different cryptographic key in each LLDP round)[5]. This can easily be implemented since LLDP packets already contain a time-stamp field for calculating the link latency. Our modification protects against adversaries who (i) tamper with the DPID or the port to conduct link fabrication attacks or (ii) alter the time-stamp field inside the LLDP packets (e.g., to execute a *Reverse Loop* attack).

Link fabrication attacks using out-of-band channels. We note that the security mechanisms implemented in the LLI module can be enhanced. One possible improvement would be for the LLI module to distinguish between LLDP packets received from existing links and those from new links. We suggest that the LLI module allows for higher tolerances in the latencies of the existing links in order to avoid attacks where adversaries aim at removing genuine links. However, even with these improvements, the LLI module can only mitigate these attacks. This is because the LLI module cannot defend against adversaries who use sophisticated hardware to relay packets, as acknowledged by Skowyra et al. [49]. The only possible way to effectively preclude relay attacks is the use of distance bounding protocols [7]. Distance bounding is a security technique through which it is possible to determine an upper-bound on the physical distance between two parties, i.e., the prover and the verifier. By combining physical properties of the communication channel with a cryptographic challenge-response protocol, distance bounding protocols can detect adversaries who use specialised hardware to relay packets over an out-of-band channel. As a future work, we plan to investigate how to apply the ideas behind distance bounding protocols in SDN networks.

Host location hijacking attacks. Unfortunately, the HTS only relies on unencrypted and unauthenticated OpenFlow `packet_in` packets to determine the hosts' location in the network. The lack of strong identifier bindings makes it possible to hijack the location of a victim's host. SecureBinder was introduced to protect SDN networks against host location hijacking attacks through a modified version of the 802.1x authentication protocol. While SecureBinder mitigates host location hijacking attacks to a large extent, it is based on a legacy 802.1x protocol that authenticates hosts only once each time they connect to a new switch. This can be sufficient if the controller was always capable of securely inferring a port disconnection from a host. As a future work, we plan to investigate the possibility of developing a middle-ground solution where the controller authenticates a host with a certain probability every time it sends a packet to the network (instead of only once). Several articles have already shown that it might not be suitable for SDN networks to perform security operations in all packets due to scalability issues. Instead, these articles propose to apply security mechanisms only to (i) a set of packets selected at random [5] or (ii) per flow [55].

8 CONCLUSIONS

This paper presented an in depth evaluation of the security of the topology discovery mechanisms in SDN. We conducted a security analysis of the state-of-the-art topology defences which included TopoGuard, TopoGuard+, SPV and SecureBinder. Our analysis revealed that, even if TopoGuard/TopoGuard+ is used, adversaries can still create fake links or remove genuine links. In addition, we provided clear evidence that SPV and SecureBinder are both likely to be vulnerable to attacks. Our work also uncovered weaknesses on the Floodlight controller which enabled us to identify and demonstrate two new topology attacks called *Reverse Loop* and *Topology Freezing*. These attacks can be mounted for various purposes ranging from performing DoS attacks to causing misleading behaviours as a first step before launching a more sophisticated attack. Finally, we elaborated on possible ways of further improving the existing countermeasures to defend against the new attacks we discovered.

ACKNOWLEDGEMENTS

The authors would like to thank our shepherd, Seungwon Shin, and the anonymous reviewers for their constructive and valuable comments that helped us to improve our paper. We also want to thank Allarna Janson for her help and support. During his research stay at the University of Padua, Eduard Marin was supported by a Travel Grant of the Research Foundation - Flanders (FWO file number V413318N). This work was supported in part by the Research Council KU Leuven (C16/15/058), the FWO (SBO project SPITE) and the European Commission (LOCARD project, Grant Agreement no. 832735)

[5]The cryptographic key should be updated regularly to prevent certain types of cryptanalytic attacks.

REFERENCES

[1] Pica8: Flow scalability per broadcom chipset. https://docs.pica8.com/display/picos2102cg/Flow+Scalability+per+Broadcom+Chipset [Online; accessed 7-Dec-2018].

[2] A. Abdou, P. C. van Oorschot, and T. Wan. Comparative Analysis of Control Plane Security of SDN and Conventional Networks. *IEEE Communications Surveys Tutorials*, pages 3542–3559, 2018.

[3] T. Alharbi, M. Portmann, and F. Pakzad. The (in)security of Topology Discovery in Software Defined Networks. In *Local Computer Networks (LCN)*. pages 502–505, 2015.

[4] A. Alimohammadifar, S. Majumdar, T. Madi, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. Stealthy Probing-Based Verification (SPV): An Active Approach to Defending Software Defined Networks Against Topology Poisoning Attacks. In *European Symposium on Research in Computer Security (ESORICS)*. pages 463–484, 2018.

[5] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran. Lineswitch: Tackling control plane saturation attacks in software-defined networking. *IEEE/ACM Transactions on Networking (TON)* 25, 2, pages 1206–1219, 2017.

[6] A. Azzouni, R. Boutaba, T. Mai Trang Nguyen, and G. Pujolle. sOFTDP: Secure and Efficient Topology Discovery Protocol for SDN. In *CoRR*, Vol. abs/1705.04527. arXiv:1705.04527 http://arxiv.org/abs/1705.04527

[7] S. Brands and D. Chaum. Distance-bounding Protocols. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. pages 344–359, 1994.

[8] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang. The CrossPath Attack: Disrupting the SDN Control Channel via Shared Links. In *USENIX Security Symposium*. pages 19–36, 2019.

[9] H. Chen and T. Benson. The Case for Making Tight Control Plane Latency Guarantees in SDN Switches. In *Symposium on SDN Research (SOSR)*. pages 150–156, 2017.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. In *ACM SIGCOMM Conference*. pages 254–265, 2011.

[11] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Network and Distributed System Security Symposium (NDSS)*. pages 8–11, 2015.

[12] Raspberry Pi Foundation. 2018. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ [Online; accessed 7-Dec-2018].

[13] Linux Fundation. Open vSwitch version 2.5.5 LTS. http://openvswitch.org/releases/openvswitch-2.5.5.tar.gz [Online; accessed 7-Dec-2018].

[14] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. Erran Li, and M. Thottan. Measuring Control Plane Latency in SDN-enabled Switches. In *Symposium on SDN Research (SOSR)*. pages 25:1–25:6, 2015.

[15] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Network and Distributed System Security Symposium (NDSS)*. pages 8–11, 2015.

[16] IEEE. 2016. IEEE Standard for Ethernet. *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, 1–4017. https://doi.org/10.1109/IEEESTD.2016.7428776

[17] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *ACM SIGCOMM Conference*. pages 3–14, 2013.

[18] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy. BEADS: Automated Attack Discovery in OpenFlow-Based SDN Systems. In *Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. pages 311–333, 2017.

[19] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow. Identifier Binding Attacks and Defenses in Software-Defined Networks. In *USENIX Security Symposium*. pages 415–432, 2017.

[20] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In *Symposium on SDN Research (SOSR)*. pages 6:1–6:12, 2016.

[21] H. Kim and H. Ju. Efficient method for inferring a firewall policy. In *Asia-Pacific Network Operations and Management Symposium (APNOMS)*. pages 1–8, 2011.

[22] J. King and K. Lauerman. 2014. ARP poisoning attack and mitigation techniques. https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11_603839.html [Online; accessed 7-Dec-2018].

[23] R. Kloti, V. Kotronis, and P. Smith. 2013. OpenFlow: A security analysis. In *IEEE International Conference on Network Protocols (ICNP)*. pages 1–6.

[24] D. Kreutz, F. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *ACM SIGCOMM workshop on Hot topics in software defined networking*. pages 55–60, 2013.

[25] D. Kreutz, F. Ramos, P. Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. In *ArXiv e-prints, 2014*. https://doi.org/10.1109/JPROC.2014.2371999

[26] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization. In *ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. pages 199–212, 2014.

[27] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A Porras. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Network and Distributed System Security Symposium (NDSS), 2017*.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, pages 69–74, 2008.

[29] A. Nehra, M. Tripathi, M. Singh Gaur, R. Babu Battula, and C. Lal. TILAK: A token-based prevention approach for topology discovery threats in SDN. *International Journal of Communication Systems*, pages e3781, 2018.

[30] Big Switch Networks. 2018. Floodlight. http://www.projectfloodlight.org/floodlight/ [Online; accessed 7-Dec-2018].

[31] Big Switch Networks. 2018. Floodlight Git repository. https://github.com/floodlight/floodlight [Online; accessed 7-Dec-2018].

[32] P. P. Lin, P. Li, and V. L. Nguyen. Inferring OpenFlow rules by active probing in software-defined networks. In *International Conference on Advanced Communication Technology (ICACT)*. pages 415–420, 2017.

[33] F. Pakzad, M. Portmann, W. Lum Tan, and J. Indulska. Efficient topology discovery in openflow-based software defined networks. *Computer Communications* 77, pages 52–61, 2016.

[34] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote Authentication Dial In User Service (RADIUS), 2000.

[35] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig. SDNsec: Forwarding Accountability for the SDN Data Plane. In *International Conference on Computer Communication and Networks (ICCCN)*. pages 1–10, 2016.

[36] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. Automatic bootstrapping of OpenFlow networks. In *IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*. pages 1–6, 2013.

[37] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Annual Network and Distributed System Security Symposium (NDSS), 2013*.

[38] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *ACM SIGSAC conference on Computer and communications security (CCS)*. pages 413–424, 2013.

[39] P. Shrivastava, A. Agarwal, and K. Kataoka. Detection of Topology Poisoning by Silent Relay Attacker in SDN. In *Annual International Conference on Mobile Computing and Networking (MobiCom)*. pages 792–794, 2018.

[40] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry. Effective topology tampering attacks and defenses in software-defined networks. In *International Conference on Dependable Systems and Networks (DSN)*. pages 374–385, 2018.

[41] J. Sonchack, A. Dubey, A. J. Aviv, J. M. Smith, and E. Keller. Timing-based Reconnaissance and Defense in Software-defined Networks. In *Annual Conference on Computer Security Applications (ACSAC)*. pages 89–100, 2016.

[42] A. Steinhoff, A. Wiesmaier, and R. Araújo. The State of the Art in DNS Spoofing. In *International Conference on Applied Cryptography and Network Security (ACNS), 2006*.

[43] Mininet Team. 2018. Mininet. http://mininet.org [Online; accessed 7-Dec-2018].

[44] K. Thimmaraju, L. Schiff, and S. Schmid. Outsmarting network security with SDN teleportation. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. pages 563–578, 2017.

[45] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi. Cross-App Poisoning in Software-Defined Networking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. pages 648–663, 2018.

[46] A. Wang, Y. Guo, F. Hao, T.V. Lakshman, and S. Chen. Scotch: Elastically Scaling Up SDN Control-Plane Using vSwitch Based Overlay. In *ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. pages 403–414, 2014.

[47] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu. Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. pages 3–16, 2018.

[48] H. Xu, Z. Yu, X. Yang Li, C. Qian, L. Huang, and T. Jung. Real-time update with joint optimization of route selection and update scheduling for SDNs. In *International Conference on Network Protocols (ICNP)*. pages 1–10, 2016.

[49] Lei Xu. TopoGuard+ git repository. https://github.com/xuraylei/TopoGuard_Plus [Online; accessed 7-Dec-2018].

[50] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu. Attacking the brain: Races in the SDN control plane. In *USENIX Security Symposium*. pages 451–468, 2017.

[51] L. Xue, X. Ma, X. Luo, E. W.W. Chan, T. T.N. Miu, and G. Gu. LinkScope: Towards Detecting Target Link Flooding Attacks. *IEEE Transactions on Information Forensics and Security (TIFS), 2018*.

[52] S. Hassas Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. In *IEEE Communications Magazine*. pages 136–141, 2013.

[53] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *ACM SIGCOMM Conference*. pages 351–362, 2010.

[54] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai. Control Plane Reflection Attacks in SDNs: New Attacks and Countermeasures. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2018.*

[55] P. Zhang. Towards rule enforcement verification for software defined networks. In *IEEE Conference on Computer Communications (INFOCOM).* pages 1–9, 2017.

AMON: an Automaton MONitor for Industrial Cyber-Physical Security

Giuseppe Bernieri
bernieri@math.unipd.it
Department of Mathematics
University of Padua
Padua, Italy

Mauro Conti
conti@math.unipd.it
Department of Mathematics
University of Padua
Padua, Italy

Gabriele Pozzan
gabriele.pozzan@studenti.unipd.it
Department of Mathematics
University of Padua
Padua, Italy
CyBrain Srl
Vicenza, Italy

ABSTRACT

The rapid evolution towards the *Industry 4.0* improves the performances of Industrial Control Systems (ICSs). However, due to the unmanageable re-engineering cost of pre-existing industrial devices, insecure protocols continue to be used to manage these systems. In this scenario, legacy protocols, such as the Modbus/TCP, are still largely used to control a range of industrial processes alongside with modern technologies. Consequently, hybrid industrial infrastructures with both legacy and innovative devices require novel security and prevention methodologies.

In this work, we present AMON (Automaton MONitor): an Intrusion Detection System (IDS) based on Deterministic Finite Automata (DFA) for Modbus/TCP traffic monitoring. AMON combines DFA with the Longest Repeating Subsequence (LRS) algorithm, commonly used in bioinformatics, to model the traffic and identify anomalies. In order to address the challenges presented in hybrid scenarios, we extend AMON to work with the Constrained Application Protocol (CoAP), used for the Industrial Internet of Things (IIoT). We show preliminary results in a simulated industrial network and discuss possible implementation of the developed detection system to secure hybrid industrial infrastructures.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; • **Computer systems organization → Sensors and actuators**.

KEYWORDS

Intrusion Detection System, Anomaly Detection, Cyber-Physical System, Industrial Security

1 INTRODUCTION

Supervisory Control and Data Acquisition (SCADA) systems allow the management of a wide variety of industrial facility processes. Threats to ICS infrastructures can cause serious economic and human damages. *Stuxnet* [10] represents the most famous malicious worm targeting ICSs. Discovered in 2010, it was designed to target Programmable Logic Controllers (PLCs) in nuclear plants in order to modify the behavior of the centrifuges and damage them. Another example of threat to ICS networks is the *Industroyer* worm [4], which caused an outage in the Ukraine's power grid in 2016. This malware installed a backdoor to communicate to a Command & Control server and exploited the trusting nature of the protocol to execute its payload by simply issuing commands. Last but not least, the *Triton* worm [9], discovered in 2017, targeted Safety Instrumented Systems (SISs) of a petrochemical processing plant. SISs are special PLCs designed to keep the physical processes in a safe state preventing incidents.

Nowadays, the *Industry 4.0* paradigm created an hybrid industrial scenario where legacy and novel systems coexist. This situation opens up a series of challenges for the security of such systems. The Modbus/TCP protocol [16] is a clear example of this kind of challenges: designed with the idea of an isolated and trusted network, it lacks basic security controls and is still largely used by modern ICS operators. In order to protect the industrial control networks, vendors provide specific solutions for intrusion detection. However, industries need novel security solutions addressing monitoring of hybrid scenarios where legacy protocols work coupled with IIoT ones, such as the CoAP [17]. Therefore, it is important to develop novel detection techniques because the importance of industrial systems makes them likely targets of *Advanced Persistent Threats (APTs)*: well funded, prolonged attacks which cover all the attack surface of their targets, from social engineering to zero-day exploits.

As contribution, in this work we:

- conceive a novel methodology based on DFA and the LRS algorithm to model multi-periodic traffic;
- present AMON: a novel security framework able to recognize normal patterns of industrial Modbus/TCP and CoAP communications, predicting data exchanges, and alerting in case of suspicious and potentially dangerous deviations;
- build a hybrid ICS network simulation scenario with Modbus/TCP and CoAP protocols using *Mininet*[1];

[1] http://mininet.org

- evaluate AMON with experiments on different threat scenarios, such as *Denial of Service* (*DoS*), *Buffer Overflow*, and *Man In The Middle* (*MITM*) attacks.

The remainder of this paper is organized as follows. Section 2 describes the background with the protocols used in this work. Section 3 analyzes related work considering Modbus/TCP and DFA in security contexts. In Section 4, we describe our proposed methodology which we evaluate in Section 5. Section 6 concludes the paper.

2 BACKGROUND

In this section we give an overview of the industrial protocols used in this work.

2.1 Modbus and Modbus/TCP

Modbus [15] is an application layer protocol largely used in industrial networks. Conceived in 1979, it became the *de facto* standard for industrial communication. The protocol was designed for serial networks but is nowadays available on the TCP/IP stack with reserved port 502. The basic unit of a Modbus message is the Protocol Data Unit (PDU) which is independent from the layer over which it is communicated and consists of a **Function Code (FC)** which is a 1 *byte* identifier for the operation requested and an optional **Data** field used by the server to perform the operation requested by the client. The Modbus/TCP implementation includes a Modbus Application Protocol Header (MBAP) which adds some information:

- **Transaction Identifier (TI)**: a unique number which defines a particular query/response couple;
- **Protocol Identifier**: used to identify Modbus/TCP packets for intra-system multiplexing;
- **Unit Identifier**: used to identify entities when the network consists of both serial and TCP/IP channels;
- **Length**: indicates the size of the Modbus/TCP packet.

The rest of the fields could vary depending on the FC of the packet.

The Modbus protocol was built to work on serial networks in which every entity was considered trusted and the isolation of the network was considered enough to keep malicious actors out, this leads to considerable vulnerabilities. Some of these issues carry over in Modbus/TCP: the main problems we want to highlight are the *lack of authentication* for masters or slaves and the *lack of encryption* in the communication.

2.2 CoAP

The CoAP [17] is a service layer protocol used in IIoT scenarios and designed to support the communications of resource constrained entities such as sensors, microcontrollers, and embedded devices [8]. Messages are exchanged over UDP and the communications follow a request/response pattern modeled after HTTP GET, POST, PUT, DELETE methods. Each message starts with a fixed header which holds the following information:

- **Type**: indicates the type of message, *Confirmable* (CON) messages which must be acknowledged or rejected, *Non-Confirmable* messages which must never be acknowledged, *Acknowledge* (ACK) messages, and *Reset* (RST) messages used for rejections.
- **Code**: indicates the class and details of the message using the format *c.dd*. The class can be "0" for request messages, "2" for success responses, "4" for client error responses and "5" for server error responses.
- **Message Id**: unique value used to match requests with responses.

Following the header there is a variable length **Token** value which is used to match requests with responses along with the Message Id. This is followed by one ore more **Options** and an optional **Payload**. It's worth noting that the data of a response to a GET request could be held in two different places: in the option *Uri-query* (code "15") if the response is piggybacked on an ACK message or in the payload section if the response is an independent message.

3 RELATED WORK

In this section, we present a literature review of the methodologies conceived to use DFA for security, stressing those applied to ICSs.

3.1 Pattern DFA

A DFA [13] is a finite state machine which accepts an alphabet of symbols and produces only one result for each sequence. It can be defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where $\mathbf{Q}$ is the set of states, $\mathbf{\Sigma}$ is the alphabet, $\delta : \mathbf{Q} \times \mathbf{\Sigma} \Longrightarrow \mathbf{Q}$ is the *transition function*, $\mathbf{q_0} \in \mathbf{Q}$ is the initial state and $\mathbf{F} \subseteq \mathbf{Q}$ is the set of final states.

In [6], *Goldenberg and Wool* describe an approach using DFA for anomaly detection in ICSs. We refer to this work as *Pattern DFA*. The *Pattern DFA* models the periodic pattern of communication between a Human Machine Interface (HMI) and a PLC. It is built automatically after a training phase during which query and response packets are captured and analyzed and it differs from a basic DFA because *it does not have final states* since its intent is to check that there are no variations in the expected periodic pattern of packets. Moreover the results and output of the DFA are related to the *transitions* that the packet traffic produces (and not so much to the states).

We will next summarise how *Goldenberg and Wool* describe the anatomy of this DFA (which states and transitions define it) and how they automatically generate it from captured data.

Anatomy: each *Pattern DFA* has its own alphabet Σ built during the initial training phase. Its symbols are defined by parts of the Modbus/TCP packet:

$$(Q, FC, RN, BC), \tag{1}$$

where:

- **Q**: is the query/response flag.
- **FC**: is the Function Code.
- **RN**: is the Reference Number (this value is only specified in query packets and will always be "0" for responses).
- **BC**: is the byte count.

The DFA states represent the situation after the *Pattern DFA* receives query or response packets and each one have *normal* and *exception* transitions as defined by the transition function δ:

- **Normal**: this transition is triggered when the packet received is the next one in the expected pattern. If the DFA is

in state s_i, after this transition it will be in state s_{i+1}. This transition can have subcases: a *Mismatch* happens when the TI of a response does not match the one of the query, an *Oversized packet* is a packet longer than 252 bytes.

- **Retransmission**: the packet received is the current packet in the expected pattern. This causes a self loop transition and is not considered an alert situation.
- **Miss**: the packet received is part of the pattern but it is not the expected one. This is often caused by packets being dropped and is not considered an alert situation. If the received packet is relative to state s_j in the pattern, the following state will be s_{j+1}.
- **Unknown**: the packet received is not part of the expected pattern. This situation causes an *unknown alert* to be raised. The next state will be reset to s_0.

Additionally, the system determines if the master/client IP address changed. If so, the security system will raise a *master/client IP changed* alert.

While exceptional states do not usually raise alerts, the system keeps track of the number of exceptional packets for each category and uses this data during the automatic generation of the DFA.

Generation: the system automatically generates a *Pattern DFA* after it captures and analyzes a number of packets during the training phase. The algorithm used for the generations is described in detail in [6].

Goldenberg and Wool discuss how this approach is not suitable to accurately model multi periodic traffic (i.e., communications in which there are different query/response patterns running at different speed). Using a single DFA to model such a scenario would require a large amount of states and it would lead to many false positives since the different patterns are most probably not in lockstep.

To tackle this representation challenge, they propose a multi-DFA model in which packets rejected as *unknown* by the standard *Pattern DFA* are passed along to a second level DFA which models the other pattern, and so on.

3.2 Statechart-based DFA

Kleinmann and Wool in [12] discuss an extension to the *Pattern DFA* which involves a *Statechart DFA* to model multi periodic traffic. The scenario upon which they base their solution is that of a multi-threaded HMI which runs different patterns of queries with different scheduling frequencies.

The main idea of this solution is to use a different *Pattern DFA* to model the traffic produced by each thread and use a *Pattern Selector* ϕ to differentiate the traffic among the DFA. Whenever a packet *pkt* is received the system calls the function $\phi(pkt)$ and selects the DFA based on the result: if $\phi(pkt) = \emptyset$ then the packet is not part of any DFA alphabet and the system raises an **unknown** alert. If $\phi(pkt) = \{A\}$ then the packet is part of the alphabet of the *Pattern DFA A* which is selected to run it. Finally, if $\phi(pkt) = \{A1, A2, \ldots\}$ then the packet is part of multiple DFA alphabets and the system gives it to the DFA for which its time of arrival is closest to the expected one.

In order to identify the correct DFA among a set of candidates, the time of arrival of a packet must be compared to the expected arrival time of the next packet for each DFA. Therefore, the *Pattern DFA* is extended to keep track of time intervals between states. This extension closely resembles the one we propose in Section 4.2 although it must be noted that in our proposal the interval effectively *augments the symbols* of the *Pattern DFA*'s alphabet and is used to detect timing anomalies.

Training Phase: the *Statechart DFA* is built by splitting the training data into different channels and using the algorithm described in [6] on each of them to generate the respective *Pattern DFA*.

The main challenge of this phase, in our opinion, is the **splitting** operation, since there is no way of automatically discerning a channel from another by simply looking at the packets' contents.

We propose a solution to this problem in Section 4.3.

3.3 Further Works

Markman et al. in [14] describe the structure of the communication of a SCADA system for a water control facility as a series of bursts of queries interspersed with silence. Their system checks the time interval between queries and divides the bursts based on a threshold (packets with small intervals are part of the same burst). They argue that these bursts have *semantic meaning* (i.e., they are meant to contain a certain pattern of queries and are not the result of a buffering process).

Byres et al. in [3] use the *attack tree* representation technique to detail possible real-world threat scenarios for SCADA systems based on Modbus/TCP. According to this work, the attacks we will analyze in the following sections fall under the category of *support goals* which means that they are steps taken in order to achieve some other goal. This highlights how AMON can be useful for an early detection of potential threats and how it can help identifying attacks before the final payload execution.

Faisal et al. in [5] propose a *specifications based* IDS. Specifications are lists of allowed behaviors which are taken from design documents and manuals. They consider this approach valid for Modbus/TCP communications because of the simplicity of the protocol. This approach is indeed optimal whenever a precise specification for the behavior of a network is available and AMON can enforce particular specifications if given a specially crafted training dataset. However, this level of precision is not always realistic. Moreover, in very complex scenarios which employ lots of different Modbus/TCP functions, the precision of the approach could decrease.

Kirat and Vigna in [11] use Longest Common Subsequence (LCSS) algorithms (a category closely related to LRS algorithms) to collect evasion behavior signatures from malware. Their technique involves calculating a *diff* of two system call traces, one taken from a sandboxed environment (which is characterized by evasion behavior) and one taken from a normal environment (with the execution of some kind of payload). This information allows to identify the point where the malware behavior diverges. In their approach, they experience some problems relative to the fact that the LCSS is not always the most meaningful one, we address similar issues as detailed in Section 3.2.

Hajji et al. in [7] use a DFA to detect anomalous behavior in Europay-Mastercard-Visa (EMV) transactions. The DFA described in this work is built over a Transition State Graph which models a secure transaction. This is similar to the approach of the *Integrity*

DFA (cfr. Section 4.4) in the sense that DFA can work to "parse" some data (the fields of a Modbus/TCP packet or a series of operations and states in a transaction) and alert deviations from a defined path.

Becchi and Crowley in [1] discuss the difficulties of implementing signature based intrusion detection by translating regular expressions in DFA and propose an hybrid deterministic and non deterministic finite automaton model. This approach is different from the one we used since we attempt to build a system able to adapt automatically to different scenarios by using an anomaly based technique.

Branch et al. in [2] use a time-dependant DFA to detect *DoS* attacks. A time-dependant DFA defines time constraints on the state transitions and treats a symbol which is in the correct place in a sequence but does not respect the time constraints as a symbol not part of the sequence (thus resetting the DFA to its first state). This idea is similar in principle to the extensions we made to the *Pattern DFA* (cfr. Section 4.2) since it takes time intervals into consideration. However, our approach extends the alphabet of the DFA in order to keep its functions as simple (and fast) as possible. Another fundamental difference is in the fact that the time-dependant DFA described in [2] represent signatures for specific attacks while we take an anomaly detection approach and thus use DFA to represent normal traffic patterns in the connection.

We believe AMON can stand out among these works because of its anomaly based approach united with a very detailed modeling of the *safe* state of the communication which takes into consideration aspects like *data variations* and *packet exchange intervals*.

4 AMON

In this section, we present AMON: a DFA based IDS for industrial network traffic monitoring. AMON is built by extending the *Pattern DFA* (cfr. Section 3.1) and *Statechart DFA* (cfr. Section 3.2). This extension is done both by augmenting the capabilities of the two approaches and by combining them with the *Integrity DFA* described in Section 4.4.

In the following Subsections we first describe the system and adversary models for our work. Subsequently, we present our extensions and the new *Integrity DFA* in detail for the Modbus/TCP protocol. Finally, we show how to translate these concepts to CoAP and how to combine everything in the final IDS scheme.

4.1 System and Adversary Models

In this section, we describe the system and adversary models used for the development of AMON.

System Model: we consider ICS networks where a centralized SCADA system monitors physical processes of industrial plants. We design an hybrid network configuration where a gateway acts as *protocol translator* between legacy protocols and IIoT-ready ones. This hybrid system model allows to face upcoming industrial network implementations, where the legacy devices are coupled with recent IIoT-ready devices. Legacy industrial components are used along with innovative ones to contain re-engineering costs. We assume that AMON runs on secured hardware and cannot be compromised.

Adversary Model: we assume the attacker to have some degree of access to the network, this could be caused by direct physical access to a control room or by machines being wrongly exposed to the internet. Some of these intrusion vectors are detailed in [3].

We consider the following attack scenarios in this work:

- A *MITM* enabled by an ARP spoofing attack which can masquerade a malicious host. In this case, the attacker has direct connectivity to the central switch in the control room zone network and is able to inject packets.
- A *DoS* attack performed by flooding a network node with requests. In this case, the attacker obtains control over a HMI machine and is able to directly query the server.
- A *Buffer Overflow* attack exploited by sending packets in which the *Length* field is mismatched with the actual packet size.

4.2 Extended Pattern DFA

The *Pattern DFA* described in Section 3.1 will detect different anomalies in the communication like unusual or unusually formed queries, sequences of packets which do not reflect the normal behavior or an entity which is no longer working (because it will not send its share of the packets). However, there are some attacks and anomalies which would not be noticed:

- A *DoS* attack which attempts to flood the server by sending a high number of packets while keeping the expected pattern.
- A *MITM* attack performed via ARP spoofing.
- Integrity violations (e.g., response to a query with wrong *TI*).

To account for some of these possibilities we extended the *Pattern DFA* alphabet (Eq. 1) with an indicator of frequency. A symbol in the alphabet will thus be a 5-tuple of the form:

$$(Q, FC, RN, BC, \Delta t), \tag{2}$$

where, for query packets, Δt is a measure of the time interval from the preceding queries (we expect responses to immediately follow the specific queries so the value is always "0" for them). This value must be rounded to account for a level of variation which is expected in the normal communications of a physical network.

For each *Pattern DFA* transition we define a **Wrong query interval** variant which is indicated by the "*" symbol (e.g., a *normal** packet is a normal packet with wrong query interval). Moreover, we keep a $TI : RN$ map for each query and we use it to set the *RN* value for the corresponding response. This allows to correctly identify responses when queries on different registers but with the same *FC* are present in the pattern. We extend the check on IP address also to MAC address adding new *master/slave MAC address changed* alerts (*MstMAC*, *SlvMAC*). This ensures detection of ARP spoofing attacks.

4.3 Extended Statechart DFA

The main issue we want to tackle with this extension is the **splitting** of the packets into different channels to feed them to the correct *Pattern DFA* during detection. We propose an LRS-based algorithm to extract multiple repeating patterns from the packets' stream captured during the training phase. The *Statechart DFA*

function pseudocode can be found in Algorithm 1, the LRS function pseudocode in Algorithm 2, their symbols are detailed in Table 1.

Algorithm 1 Statechart DFA Generation

```
1: procedure SC_Gen(Seq, Pattern_List)
2:     Sub ← LRS(Seq)
3:     if (Len(Sub) > 0) then
4:         Cleared_Sub ← CS(Seq, Sub)
5:         Find_Sub(Sub, Pattern_List)
6:         Find_Sub(Cleared_Sub, Pattern_List)
7:     else if (Len(Seq) > 0) then
8:         Add Seq to Pattern_List
```

Algorithm 2 Longest Repeating Subsequence

```
1: procedure LRS(Seq)
2:     n ← Len(Seq)
3:     RST ← Empty_Table(n+1, n+1)
4:     IT ← Empty_Table(n+1, n+1)
5:     for (i ← 1 to n + 1) do
6:         for (j ← 1 to n + 1) do
7:             if (Seq[i] = Seq[j] AND i ≠ j) then
8:                 RST[i][j] ← RST[i-1][j-1]
9:                 Append(RST[i][j], Seq[i])
10:                IT[i][j] ← IT[i-1][j-1]
11:                Append(IT[i][j], i)
12:            else if (Len(RST[i][j-1]) > Len(RST[i-1][j])) then
13:                RST[i][j] ← RST[i][j-1]
14:                IT[i][j] ← IT[i][j-1]
15:            else
16:                RST[i][j] ← RST[i-1][j]
17:                IT[i][j] ← IT[i-1][j]
18:    LRS ← FNO(IT)
19:    Return LRS
```

Algorithm 3 Find Non Overlapping

```
1: procedure FNO(IT)
2:     Vertical ← FF_Vert(IT)
3:     Horizontal ← FF_Hor(IT)
4:     Remove Horizontal indexes from Vertical
5:     Return Vertical
```

Algorithm 1 starts by finding the LRS in all of the traffic captured during the training phase. The actual LRS in a long sequence of symbols could have overlapping parts as shown in the following example:

*ABCD***ABCD***ABCD*

In this example the LRS is *ABCDABCD*, the bold characters highlight the symbols which overlap (albeit at different indexes on the respective subsequences).

We want to avoid this occurrence to improve the overall performance of the algorithm (if a simple subsequence like *ABCD* was repeated n times we would need to run the LRS algorithm for $n-1$ times in order to let the base pattern emerge). We also want to avoid another problem: if a long sequence of repeating characters is interspersed with the symbols from another repeating sequence, the LRS will be prefixed with repeating occurrences of the first character of the subsequence. An example is the sequence in which the subsequence *ABCD* is interspersed with elements of the subsequence *EFGH*:

*ABCD***E***ABCD***F***ABCD***G***ABCD***H**

If this sequence is repeated up to three times, the LRS algorithm will extract the subsequence *ABCDABCDABCDABCD* which can then be reduced to *ABCD*. However, if the sequence is repeated four or more times the LRS algorithm will start prefixing the results with a number of occurrences of the first character *A*. This is indeed expected behavior for a generic string but does not adapt well to the query/response communication patterns we want to model.

To handle this problem we use the *Find Non Overlapping* (*FNO*) function which is able to cut the overlapping parts of a subsequence by looking at its index table *IT*. This function is described in Algorithm 3: in the *IT* table the indexes relative to the first occurrence of the complete LRS in the last row partially overlap with the ones relative to the first occurrence in the last column: by removing the overlapping indexes we will then remove the overlapping part of the subsequence.

After this step, if the LRS is not empty we run the algorithm again using it as an input in order to find potential nested patterns. Finally, we clear the original sequence from the subsequence's symbols in order to let other patterns emerge and we look for LRS in this cleared subsequence (*Cleared_Sub*).

A sequence which has no LRS and is not empty is considered a pattern and is added to the *Pattern_List*.

Table 1: Definitions for Algorithm 1, 2, and 3.

LRS	Longest Repeating Subsequence Algorithm
Len	Returns length of sequence
CS	Removes Sub's states from Seq
Seq	Initial sequence of states
Pattern_List	Global list of patterns
RST	Repeating Subsequence Table
IT	LRS Index Table
FNO	Find Non Overlapping subsequence
FF_Vert & FF_Hor	Find first vertical or horizontal occurrence of the subsequence in the IT table

4.4 Integrity DFA

We designed the *Integrity DFA* (shown in Figure 1) to parse the fields of each incoming packet. This Section describes it in detail.

Training Phase: the *Integrity DFA*'s training phase consists of checking the FCs used during normal communication and storing them in a list we will call *Accepted_FCs*. Moreover, response packets are analyzed in order to learn the average variation of the data which is returned: in many physical scenarios (e.g., water distribution systems) we expect to be able to predict a variation threshold for the responses of subsequent queries (i.e., a water tank can not be emptied instantly). An average data variation measure is saved for each FC, RN couple (*FCRN*), this allows us to distinguish between queries with the same FC on different registers.

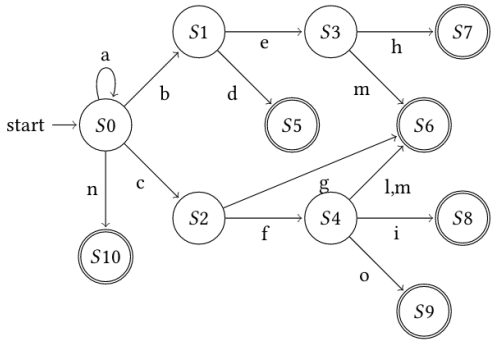


Figure 1: The *Integrity DFA*.

Anatomy: in order to perform some integrity checks we update a *Transactions* map ($TI : FC$) for every query. Moreover, we keep track of the latest data for every *FCRN* identifier in order to calculate the data variation for each response. The states of the *Integrity DFA* are:

- **S0**: starting state, a packet arrived. The DFA will remain in this state if non Modbus/TCP packets arrive (transition **a**).
- **S1**: the packet is a Modbus/TCP query.
- **S2**: the packet is a Modbus/TCP response.
- **S3**: the TI of the query is valid (i.e., this is not a retransmission).
- **S4**: this is the response to a pending query.
- **S5**: this query does not have a unique TI and is not valid $\Longrightarrow$ **Alert**.
- **S6**: this packet presents some anomalies:
  - the FC of this packet is not part of the accepted FCs (i.e., is not in *Accepted_FCs*) $\Longrightarrow$ **Alert**.
  - the FC of this response packet doesn't match the one in the query packet $\Longrightarrow$ **Alert**.
  - the TI of this response doesn't match any pending query $\Longrightarrow$ **Alert**.
- **S7**: this query packet is accepted.
- **S8**: this response packet is accepted.
- **S9**: this response packet presents an *anomalous data variation* $\Longrightarrow$ **Alert**.
- **S10**: there is a mismatch between the size indicated in the *Length* field of the packet and its actual size $\Longrightarrow$ **Alert**.

A detailed description of the transitions is in Table 2.

4.5 CoAP integration

The previous Sections described the application of the Extended *Pattern*, *Statechart*, and *Integrity DFA* to the Modbus/TCP protocol. As part of our work we extended these approaches to CoAP: in order to do so a simple translation is required (we found that only one Modbus/TCP field relevant to the DFA has no correspondence in CoAP).

The field translations are listed in Table 3, the symbol $\emptyset$ is used when there is no correspondence for a particular field, to indicate a particular option by name we use the notation *Options[Option-Name]*.

Table 2: *Integrity DFA* Transitions.

a	Not a Modbus/TCP packet
b	This is a query packet: save TI and FC in *Transactions* map
c	This is a response packet
d	The TI of this query packet is already present in the *Transactions* map
e	This is not a retransmitted query
f	The TI of this response is in the *Transactions* map
g	The TI of this response is not in the *Transactions* map
h	The FC of this query is part of the accepted ones
i	The FC of this response corresponds with the one saved in the *Transactions* map.
l	The FC of this response does not correspond with the one saved in the *Transactions* map.
m	The FC of this query is not part of the accepted ones
n	The size of the packet and the *Length* field are mismatched
o	The difference between the data of this response and the last seen data for its *FCRN* identifier is greater than expected

Table 3: Modbus/TCP to CoAP translations.

Modbus/TCP	CoAP
Transaction Identifier	Message Id + Token
Function Code	Code
Reference Number	Options[Uri-Path]
Length	$\emptyset$

4.6 Analysis and integration

AMON is the result of the combination and extension of the *Pattern DFA*, *Statechart* and *Integrity DFA*, in this Section we explain the reasons for this choice.

Combination: as seen in the previous sections the extended *Pattern DFA* and *Integrity DFA* can detect different threats in a Modbus/TCP communication and could be employed by themselves as IDS. However, because of their focus on different aspects of the communication (i.e. patterns and packet contents), we think they should be employed together in order to compensate and synergize their behavior. In Table 4, we show that there are some cases in which the combination of the analysis of the two automata generates new results.

The *Integrity DFA* is able to identify a packet classified as *Normal* by the *Pattern DFA*. Such packet could still have some suspicious aspects, like a strange TI, a mismatch between the Length field value and the actual content or some strange variation of response data. These scenarios are marked with (*) in Table 4. A packet accepted by the *Integrity DFA* could actually be seen as *Miss* or *Unknown* packet by the *Pattern DFA*. These scenarios are marked with (**) in Table 4.

Workflow: on a practical side, the training phase for AMON consists of the combination of the training phases of the modules with some additions necessary for the extensions described in Section 4.2.

Table 4: *Ext. Pattern* and *Integrity DFA* synergies.

Ext. Pattern DFA	*Integrity DFA*
Normal, Oversized, Mismatch	Will detect TI problems(*). Will detect data length problems(*). Will detect anomalous data variations(*).
Retransmission	A response packet will end in S6. A query will end in S5.
Miss	A response could end in S6. A query could be accepted(**).
Unknown	Based on the reason for the unknown transition the *Integrity DFA* could end in every state: S5 and S6 would be integrity alerts. S9 and S10 would be accepted(**).

The resulting training data includes the extended pattern description (cfr. 4.2), the set of authorized FCs and the expected data variations.

After the training phase, the detection phase starts: AMON analyzes each packet sequentially first through the extended *Pattern DFA* and then through the *Integrity DFA*. Each module produces a result state which is collected and analyzed to detect anomalous situations (cfr. Figure 2). The *Detector* module looks at *precise sequences* as well as *percentages of states* in order to decide if a situation warrants an alert. In order to find relevant sequences of states, we applied a LRS algorithm to find the longest repeating substring (which differs from a subsequence because it requires all the characters to be sequential) in the result states' sequence collected during some attack tests. More details can be found in Section 5.

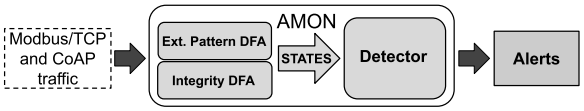


Figure 2: AMON Detection Workflow.

5 EVALUATION

This section describes the tests we ran to validate AMON against a series of attacks and the scenario in which these attacks took place.

5.1 Simulation Scenario and Tools

Testing security solutions for industrial processes is a delicate issue since actual real world attack implementations could endanger critical assets and infrastructures. For this reason, we created a test environment using Mininet, a tool which allows the simulation of a network in a virtualized environment. With this approach it is possible to avoid using expensive testbeds. Moreover, a new network configuration can be easily and quickly prototyped.

For this work, we developed a simulation environment for an hypotetical industrial scenario: a water tower which gradually empties by filling up two consumers with different consumption rates. This scenario is implemented with an hybrid network configuration which involves a server and a client configured to communicate respectively with the CoAP and Modbus/TCP protocols. The CoAP server mimics a PLC connected to three sensors, one for the water tower and the other for the consumer tanks. A gateway stands in between the server and the client and enables the communication by keeping a local database of the server's values, which is periodically updated through CoAP requests, and by providing a Modbus/TCP interface to the client. This configuration models a realistic scenario in which legacy components (i.e., the Modbus/TCP infrastructure) are used along with innovative ones to contain re-engineering costs: new sensors able to communicate over CoAP are installed in the system while the legacy Modbus/TCP HMI-PLC configuration allows communication by installing the Modbus/TCP-CoAP gateway. All the elements of the system are implemented in Python: the CoAP server and the gateway client use the CoAPthon library [18], the Modbus/TCP client and gateway server use the uModbus library[2].

AMON runs on a fourth host which passively receives a copy of each packet directed to or coming from the server through Switch Port for Analysis (SPAN) also known as port mirroring. This is configured through the Open vSwitch[3] *ovs-vsctl* tools. We represent the network topology conceived for the tests in Figure 3.

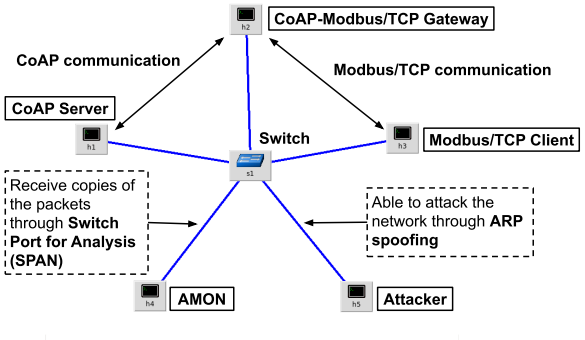


Figure 3: Network configuration for the attack tests.

The normal traffic pattern for this network consists of a polling cycle of CoAP requests which every second queries for the values of the three tanks ($t1, t2, t3$) in order to keep the gateway local database updated. The Modbus/TCP client sends queries for the values of $t1$ and $t2$ (every 2 seconds) and for the value of $t3$ (every 10 seconds).

5.2 Attack Tests

To test the validity of our approach, we implemented a series of attacks which explicitly attempt to sidestep the Extended *Pattern DFA* and *Integrity DFA* detection capabilities.

We considered two categories of attacks:

- **Malicious client**: the attacker obtains the control of a client and is able to query the server directly. The attacks and results are described in Subsections 5.2.2 and 5.2.3.
- ***MITM* with *ARP spoofing***: a simple ARP spoofing attack can deviate the client/server communications through a malicious host. This could lead to the obfuscation of physical parameters of the system's sensors or to the execution of arbitrary operations by the actuators. The attacks and results are described in Subsections 5.2.4, 5.2.5, and 5.2.6.

[2]https://github.com/AdvancedClimateSystems/uModbus/
[3]http://www.openvswitch.org/

We conducted each attack separately against the Modbus/TCP and CoAP links.

To evaluate these tests we check the **capability** of detecting an ongoing attack, the **time** required, the emergence of a particular **state sequence** which can be used as an attack signature by the *Detector* (and its coverage of the complete sequence of states produced) and finally the emergence of particular **percentages** of states (to use along with the attack signature to get more attack details). We represent *Integrity DFA* states with the concatenation of the states traversed by the packet, for example the state S0S1S3S7 indicates an accepted query.

5.2.1 Normal Traffic. We tested the behavior of the system under normal traffic to ensure the avoidance of false positives. The state percentages for this test are listed in Table 5. The results highlights that the Extended *Pattern DFA* only produces *normal* states with a small percentage of delayed *normal** packets. The *Integrity DFA* accepts every query (*S0S1S3S7*) and response (*S0S2S4S8*).

Table 5: Normal Traffic.

State	Modbus/TCP	CoAP
normal	99.24%	99.61%
normal*	0.76%	0.39%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this test covers 96.42% of the states produced: *S0S1S3S7, normal, S0S2S4S8, normal*. This is the expected pattern of states since it shows an accepted/normal query followed by an accepted/normal response.

5.2.2 Standard DoS. In this attack, a malicious client floods the server with a specific query in order to impede its functions. The query is one of the accepted queries collected during the training phase (requesting the value relative to the tank *t1*) in order to avoid the raising of *unknown* and *integrity* alerts.

Modbus/TCP link attack: the state percentages for the attack on the Modbus/TCP link are presented in Table 6. The Modbus/TCP states show a high percentage of *miss** statuses since the query flooded is part of the accepted pattern. The CoAP traffic shows increased delays caused by the ongoing attack.

Table 6: Standard DoS results: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	50%	61.11%
normal*	0.04%	38.89%
miss*	49.96%	0%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this attack covers 99.93% of the states produced: *S0S1S3S7, normal, S0S2S4S8, miss**. This sequence of states is semantically correct for the *Standard DoS* attack in this scenario and can be used to quickly identify it.

CoAP link attack: the CoAP link attack is somewhat different from the Modbus/TCP one because in this case it's not the CoAP client running on the gateway which performs the attack but a third malicious client. We chose this configuration because we consider the gateway as part of the safe system (malicious exploitation of the gateway would lead to different possible attacks). The state percentages for the CoAP link attack are listed in Table 7. This attack affects the quality of the CoAP link communications (but not so greatly the Modbus/TCP communications) and produces a wider range of different states.

Table 7: Standard DoS results: CoAP.

State	Modbus/TCP	CoAP
normal	97.83%	12.79%
normal*	2.17%	7.04%
miss*	0%	14.93%
miss	0%	10.66%
retransmission*	0%	48.61%
mismatch	0%	5.97%
S0S1S3S7	50%	80.81%
S0S2S4S8	50%	19.19%

The LRS of states emerged during this attack covers 49% of the states produced: *S0S1S3S7, MstMAC, retransmission**. This sequence of states shows that the IDS is able to recognize the external nature of the attack (the *MstMAC* state) and can be used to identify it.

5.2.3 Smart DoS. A skilled attacker could attempt to fool the *Pattern DFA* by performing a DoS attack using the normal communication pattern to flood the server.

Modbus/TCP link attack: the state percentages for this attack on the Modbus/TCP link are listed in Table 8. As for the *Standard DoS*, the attack disturbed the CoAP communication link. The Modbus/TCP link states are divided between *normal* and *normal** because the pattern is respected, the percentage of delayed packets is vastly increased with respect to the normal traffic. We use this information to complement the LRS of states described below. Specifically, we recognize a *Smart DoS* attack only after that at least 20% of the states are *normal**.

Table 8: Smart DoS results: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	71.5%	62.5%
normal*	28.5%	37.5%
S0S1S3S7	50.02%	50%
S0S2S4S8	49.98%	50%

The LRS of states emerged during this attack covers 86% of the states produced: *S0S1S3S7, normal, S0S2S4S8, normal*, S0S1S3S7, normal, S0S2S4S8, normal*. This is not a very unusual sequence of states even for a normal communication so we combine it with percentage information to decide if an alert should be raised.

CoAP link attack: as in the *Standard DoS*, we conducted this attack from a third malicious host. The state percentages for the CoAP link attack are listed in Table 9.

Table 9: Smart DoS results: CoAP.

State	Modbus/TCP	CoAP
normal	90.91%	13.94%
normal*	9.09%	4.85%
miss*	0%	70.91%
miss	0%	3.64%
retransmission	0%	1.21%
mismatch	0%	3.03%
S0S1S3S7	50%	85.45%
S0S2S4S8	45.45%	14.55%
S0S2S4S9	4.55%	0%

The LRS of states emerged during this attack covers 69% of the states produced: *normal, MstMAC, miss**. This sequence of states shows how the attack actually disrupts the normal pattern of communication (i.e., the obfuscation does not work) as we have a large number of *miss* states.

5.2.4 Traffic sniffing. In this attack the MITM simply receives and forwards all the packets of the communication without modifying them.

Modbus/TCP link attack: we present the state percentages for this attack on the Modbus/TCP link in Table 10. The attack causes delays in the communication (increased *normal** rate), but no other alert states.

Table 10: Traffic Sniffing: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	91.0%	99.05%
normal*	9.0%	0.95%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this attack covers 78% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal*. This sequence of states correctly identifies the ongoing attack since it highlights an accepted query received from a different client (*MstMAC*) and an accepted response from a different server (*SlvMAC*). This sequence is common to most ARP spoofing attacks, so we will use other details to differentiate them.

CoAP link attack: the state percentages for the CoAP link attack are listed in Table 11. The attack causes more delays on this link as the higher percentage of *normal** packets shows.

Table 11: Traffic Sniffig: CoAP.

State	Modbus/TCP	CoAP
normal	97.56%	72.84%
normal*	2.44%	27.16%
S0S1S3S7	50%	50%
S0S2S4S8	50%	50%

The LRS of states emerged during this attack covers 64% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC*. This sequence of states is very similar to the Modbus/TCP link one and the same observations we made apply also to this attack.

5.2.5 Buffer overflow attempt. Depending on its implementation, a server could have *Buffer Overflow* vulnerabilities and an attacker could try to exploit them by appending raw bytes to a packet. This will result in a mismatch between the length indicated in the packet fields and the actual size of the packet. This attack was only tested on the Modbus/TCP link since on CoAP packets there are no equivalent *Length* fields.

Modbus/TCP link attack: we list the state percentages for the attack on the Modbus/TCP link in Table 12. The Extended *Pattern DFA* states show the disturbance caused by the ongoing attack. The *Integrity DFA* states correctly highlight the presence of packets bigger than expected (*S0S10*), the presence of *integrity alerts* (*S0S2S6*) is caused by the fact that the overflown queries' TI are not saved, causing the relative responses to raise the alerts.

Table 12: Buffer Overflow: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	45.38%	98.89%
normal*	1.87.0%	1.11%
retransmission	4.42%	0%
miss*	4.72%	0%
retransmission*	4.32%	0%
miss	39.29%	0%
S0S1S3S7	1.47%	50%
S0S2S4S8	1.47%	50%
S0S10	48.53%	0%
S0S2S6	48.53%	0%

The LRS of states emerged during this attack covers 64% of the states produced: *S0S10, SlvMAC, S0S2S6, MstMAC, normal, S0S10, SlvMAC, miss, S0S2S6, MstMAC, miss*.

5.2.6 Replay. A way to perform a *Replay* attack is to change the data fields of a packet before forwarding it to the client. In this way, the packet will be accepted by the client and it will respect the pattern.

Modbus/TCP link attack: the state percentages for the attack on the Modbus/TCP link are listed in Table 13. This attack raises a few *Anomalous data variation* alerts when the replay begins. We can use this small percentage to identify and differentiate it from a simple *Traffic Sniffing*. We identify a *Replay* attack if we see between 0% and 10% of *S0S2S4S9* states.

Table 13: Replay: Modbus/TCP.

State	Modbus/TCP	CoAP
normal	92.41%	99.69%
normal*	7.59%	0.31%
S0S1S3S7	50%	50%
S0S2S4S8	48.71%	50%
S0S2S4S9	1.29%	0%

The LRS of states emerged during this attack covers 78% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal*. This is equal to most of the ARP spoofing states sequences so we must use percentage information to raise a precise alert.

CoAP link attack: the state percentages for the CoAP link attack are listed in Table 14.

Table 14: Replay: CoAP.

State	Modbus/TCP	CoAP
normal	97.12%	64.14%
normal*	2.88%	35.86%
S0S1S3S7	50%	50%
S0S2S4S8	47.12%	8.48%
S0S2S4S9	2.88%	1.52%

The LRS of states emerged during this attack covers 69.5% of the states produced: *S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal**. The interesting part of this sequence is the higher impact of the attack on the timings of the packets, hence the presence of a *normal** state.

5.2.7 Attack Detection. The presence of anomalous states in the DFA results can provide the evidence of malicious activities in the communications. We developed the AMON's *Detector* module to identify specific ongoing attacks, by keeping a list of LRS signatures and percentages of states. This module is responsible for raising the alerts and it is fundamental to filter the noise of the intermediate level of information provided by the DFA states.

Table 15 and Table 16 summarize the signatures, the state percentages (not necessary for every attack, the symbol $\emptyset$ is used when the signature is sufficient to identify the attack), and show the interval of time between the beginning of the attack and the raising of the alert that we observed during the tests. These results show how AMON is able to combine the data produced by various network communications anomalies into effective alerts in a quick and responsive way.

Table 15: Modbus/TCP: Signatures and Timeliness.

Signature	Percentages	Δt	Alert
S0S1S3S7, normal, S0S2S4S8, miss*	$\emptyset$	18ms	Standard DoS
S0S1S3S7, normal, S0S2S4S8, normal*, S0S1S3S7, normal, S0S2S4S8, normal	$normal* > 20\%$	74ms	Smart DoS
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal	$\emptyset$	1.3s	Traffic Sniffing
S0S10, SlvMAC, S0S2S6, MstMAC, normal, S0S10, SlvMAC, miss, S0S2S6, MstMAC, miss	$\emptyset$	57ms	Buffer Overflow
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal	$0\% < S0S2S4S9 < 10\%$	2s	Replay

Table 16: CoAP: Signatures and Timeliness.

Signature	Percentages	Δt	Alert
S0S1S3S7, MstMAC, retransmission*	$\emptyset$	295ms	Standard DoS
S0S1S3S7, MstMAC, retransmission*	$\emptyset$	62ms	Smart DoS
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC	$\emptyset$	622ms	Traffic Sniffing
S0S1S3S7, SlvMAC, normal, S0S2S4S8, MstMAC, normal*	$0\% < S0S2S4S9 < 10\%$	2.7s	Replay

6 CONCLUSION

In this work, we developed AMON, an IDS framework based on DFA for Modbus/TCP and CoAP traffic monitoring. The results obtained implementing AMON in a simulated hybrid industrial scenario show the usefulness of the conceived security system. In future work, we will extend AMON considering real IIoT testbeds implementations. Moreover, we will develop active features and evolve AMON to an Intrusion Prevention System.

ACKNOWLEDGMENTS

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the LOCARD project (G.A. no. 832735). This work is also supported by a grant of the Italian Presidency of the Council of Ministers and by CyBrain Srl.

REFERENCES

[1] Michela Becchi and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *2007 ACM CoNEXT conference*. ACM, 1.
[2] Joel Branch, Alan Bivens, Chi Yu Chan, Taek Kyeun Lee, and Boleslaw K Szymanski. 2002. Denial of service intrusion detection using time dependent deterministic finite automata. In *Proc. Graduate Research Conference*. 45–51.
[3] Eric J Byres, Matthew Franz, and Darrin Miller. 2004. The use of attack trees in assessing vulnerabilities in SCADA systems. In *Proceedings of the international infrastructure survivability workshop*. Citeseer, 3–10.
[4] Anton Cherepanov. 2017. WIN32/INDUSTROYER: a new threat for industrial control systems. *White paper, ESET (June 2017)* (2017).
[5] Mustafa Faisal, Alvaro A Cardenas, and Avishai Wool. 2016. Modeling Modbus TCP for intrusion detection. In *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 386–390.
[6] Niv Goldenberg and Avishai Wool. 2013. Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems. *International Journal of Critical Infrastructure Protection* (2013), 63–75. https://doi.org/10.1016/j.ijcip.2013.05.001
[7] Tarik Hajji, Noura Ouerdi, Abdelmalek Azizi, and Mostafa Azizi. 2018. EMV Cards Vulnerabilities Detection Using Deterministic Finite Automaton. *Procedia Computer Science* 127 (2018), 531–538.
[8] Markel Iglesias-Urkia, Adrián Orive, and Aitor Urbieta. 2017. Analysis of CoAP implementations for industrial internet of things: a survey. *Procedia Computer Science* 109 (2017), 188–195.
[9] Dragos Inc. 2017. TRISIS Malware: Analysis of Safety System Targeted Malware. https://dragos.com/blog/trisis/TRISIS-01.pdf.
[10] Stamatis Karnouskos. 2011. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011-37th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 4490–4494.
[11] Dhilung Kirat and Giovanni Vigna. 2015. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 769–780.
[12] Amit Kleinmann and Avishai Wool. 2015. A statechart-based anomaly detection model for multi-threaded SCADA systems. In *International Conference on Critical Information Infrastructures Security*. Springer, 132–144.
[13] Mark V Lawson. 2003. *Finite automata*. Chapman and Hall/CRC.
[14] Chen Markman, Avishai Wool, and Alvaro A Cardenas. 2017. A New Burst-DFA Model for SCADA Anomaly Detection. In *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy*. ACM, 1–12.
[15] Modbus. 2004. MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3.
[16] Modbus. 2004. Modbus messaging on TCP/IP implementation guide. *v1.0b* (2004).
[17] Zach Shelby, Klaus Hartke, Carsten Bormann, and B Frank. 2014. RFC 7252: The constrained application protocol (CoAP). *Internet Engineering Task Force* (2014).
[18] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. 2015. CoAPthon: Easy development of CoAP-based IoT applications with Python. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 63–68.