UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Resilient Communications in Smart Grids

Pedro Alexandre Pacheco Pinto Maia

Mestrado em Segurança Informática

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
Prof. Doutor Fernando Manuel Valente Ramos

2018

Acknowledgments

I am grateful to all the persons and entities that helped me throughout this long journey.
I would especially like to thank my instructors Dr. Nuno Neves and Dr. Fernando Ramos for guiding me and also for the time and knowledge that helped me fulfil this work.
I also would like to thank Ricardo Fonseca for all the help and support that he gave me.
To my family of course, this wouldn't be possible without you. Thank you for all the support that all of you gave during all this years.
Also would like to thank the project FP7 SEGRID (607109) for the funding that made this possible.
And last but not least to Vanessa for pushing me and keeping me on track when i even slightly steer out of it. Especially without you this wouldn't be possible.

Resumo

As redes elétricas, algumas já centenárias, foram concebidas para uma realidade bastante diferente da actual. O facto de terem sido desenhadas para transportar e distribuir a energia de forma unidirecional, torna a infraestrutura rígida, causando problemas em termos de escalabilidade e dificulta a sua evolução.

Conhecidas questões ambientais têm levado a que a geração de energia baseada em combustíveis fosseis seja substituída pela geração através de fontes de energia renováveis. Esta situação motivou a criação de incentivos ao investimento nas fontes de energia renováveis, o que levou a que cada vez mais consumidores apostem na microgeração. Estas alterações causaram uma mudança na forma como é feita a produção e distribuição de energia elétrica, com uma aposta crescente na interligação entre as várias fontes ao longo da infraestrutura, tornando a gestão destas redes uma tarefa extremamente complexa. Com o crescimento significativo de consumidores que também podem ser produtores, torna-se essencial uma coordenação cuidada na injeção de energia na rede. Este facto, aliado à crescente utilização de energia elétrica, faz com que a manutenção da estabilidade da rede seja um enorme desafio.

As redes inteligentes, ou *smart grids*, propõem resolver muitos dos problemas que surgiram com esta alteração do paradigma de consumo/produção de energia elétrica. Os componentes da rede passam a comunicar uns com os outros, tornando a rede elétrica bidirecional, facilitando assim a sua manutenção e gestão. A possibilidade de constante troca de informação entre todos os componentes que constituem a *smart grid* permite uma reação imediata relativamente às ações dos produtores e consumidores de energia elétrica. No entanto, com esta alteração de paradigma surgiram também muitos desafios.

Nomeadamente, a necessidade de comunicação entre os equipamentos existentes nas *smart grids* leva a que as redes de comunicação tenham de cobrir grandes áreas. Essa complexidade aumenta quando a gestão necessita de ser feita ao nível de cada equipamento e não de forma global. Isto é devido ao facto de nas redes de comunicação tradicionais, o plano de controlo e o de dados estarem no mesmo equipamento, o que leva a que o seu controlo seja difícil e propício a erros. Este controlo descentralizado dificulta também a reorganização da rede quando ocorrem faltas pelo facto de não existir um dispositivo que

tenha o conhecimento completo da rede. A adaptação rápida a faltas de forma a tornar a comunicação resiliente tem grande importância em redes sensíveis a latência como é o caso da *smart grid*, pelo que mecanismos eficientes de tolerância a faltas devem ser implementados.

As redes definidas por software, ou *Software Defined Networks* (SDN), surgem como uma potencial solução para estes problemas. Através da separação entre o plano de controlo e o plano de dados, permite a centralização lógica do controlo da rede no controlador. Para tal, é necessário adicionar uma camada de comunicação entre o controlador e os dispositivos de rede, através de um protocolo como o Openflow. Esta separação reduz a complexidade da gestão da rede e a centralização lógica torna possível programar a rede de forma global, de modo a aplicar as políticas pretendidas. Estes fatores tornam a SDN uma solução interessante para utilizar em *smart grids*.

Esta tese investiga formas de tornar a rede de comunicações empregue numa *smart grid* resiliente a faltas. Pelas vantagens mencionadas anteriormente, é usada uma solução baseada em SDN, sendo propostos dois módulos essenciais. O primeiro tem como objectivo a monitorização segura da rede, permitindo obter em tempo real métricas como largura de banda, latência e taxa de erro. O segundo módulo trata do roteamento e engenharia de tráfego, utilizando a informação fornecida pelo módulo de monitorização de forma a que os componentes da *smart grid* comuniquem entre si, garantindo que os requisitos das aplicações são cumpridos. Dada a criticidade da rede elétrica e a importância das comunicações na *smart grid*, os mecanismos desenvolvidos toleram faltas, quer de tipo malicioso, quer de tipo acidental.

Palavras-chave: Smart Grid, resiliente, SDN, monitorização, roteamento, engenharia de tráfego

Abstract

The evolution on how electricity is produced and consumed has made the management of power grids an extremely complex task. Today's centenary power grids were not designed to fit a new reality where consumers can also be producers, or the impressive increase in consumption caused by more sophisticated and powerful appliances. Smart Grids have been prepared as a solution to cope with this problem, by supporting more sophisticated communications among all the components, allowing the grid to react quickly to changes in both consumption or production of energy. On the other hand, resorting to information and communication technologies (ICT) brings some challenges, namely, managing network devices at this scale and assuring that the strict communication requirements are fulfilled is a dauting task.

Software Defined Networks (SDN) can address some of these problems by separating the control and data planes, and logically centralizing network control in a controller. The centralised control has the ability to observe the current state of the network from a vantage point, and programatically react based on that view, making the management task substantially easier.

In this thesis we provide a solution for a resilient communications network for Smart Grids based on SDN. As Smart Grids are very sensitive to network issues, such as latency and packet loss, it is important to detect and react to any fault in a timely manner. To achieve this we propose and develop two core modules, a network monitor and a routing and traffic engineering module. The first is a solution for monitoring with the goal to obtain a global view of the current state of the network. The solution is secure, allowing malicious attempts to subvert this module to be detected in a timely manner. This information is then used by the second module to make routing decisions. The routing and traffic engineering module ensures that the communications among the smart grid components are possible and fulfils the strict requirements of the Smart Grid.

Keywords: Smart Grid, resilient network, SDN, network monitoring, routing, traffic engineering

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Smart grid

Smart Grids are composed of the various components of the electrical infrastructure interconnected by a communications network. In contrast to traditional power grids, Smart Grids are able to cope with the dynamism present in today's power grids, which results in a reduction of transmission and distribution losses, increased efficiency in the use of renewable energy sources, allowing large scale energy storage and enabling market-based electricity pricing. Smart Grids enable the use of new, intelligent appliances that are capable of deciding when to consume power based on pre-set costumer preferences. This effectively reduces peak loads and the need for new power plants, while maximising the use of renewable energy. By resorting to smart meters and smart substations, distribution system operators are now able to quickly identify problems and dispatch repair crews to the correct location. The constant communication required in a Smart Grid enables self-healing, self-balancing and self-optimizing distribution, allowing the prediction of cable failures based on real-time data about weather and outage history, by using automated monitoring and analysis tools. Generically, it is possible to separate a Smart Grid network in three parts, as in Figure 1.1.

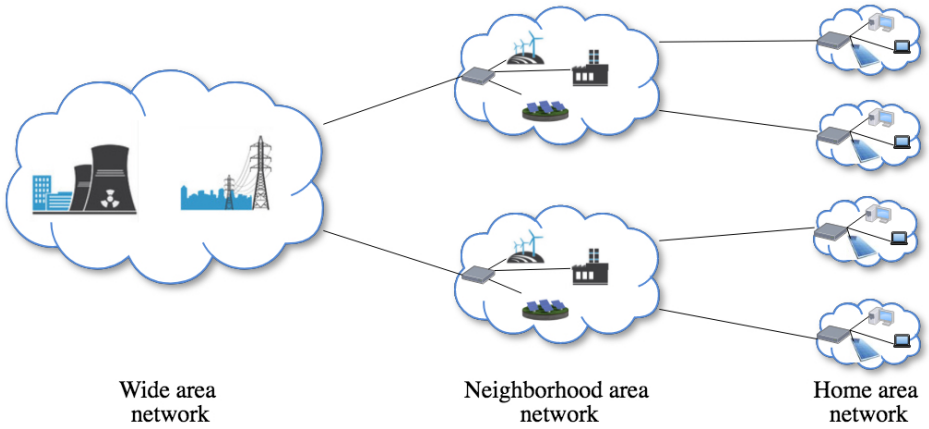

Figure 1.1: Example smart grid communication networks

The Home Area Network (HAN) can use several communications technologies, both wired and wireless, and enables the smart management and consumption monitoring of home appliances. The HAN is connected to the Neighborhood Area Network (NAN) through a gateway. The NAN is composed of all the HANs (represented by its gateway), the sub-stations, the distribution systems, distributed power generation and the NAN gateway. Finally, the Wide Area Network (WAN) connects all NAN gateways, power generation infrastructures, and transmission elements.

1.2 Motivation

Power grids are evolving. The static design of existing electrical infrastructures does not fit the current needs, such as energy production being distributed across a vast area and micro-generation becoming a reality. There is a global demand on the use of renewable energy for environmental protection, which adds new difficulties to power management, as energy generation is not as predictable as in traditional power plants. The need for a dynamic and self-healing infrastructure is growing and smart-grids are the proposed solution for this evolution. Smart Grids are composed of multiple components and assets, as shown in Figure 1.2, such as power generation, substations, circuit breakers, sensors, smart meters, etc. These components bring a new set of capabilities to electrical infrastructures that enable more efficient power delivery and resource usage. To achieve this higher efficiency, these components require the ability to exchange data in order to collect information about their state and then, if necessary, adjust their behaviour. This requirement of continuous data exchange makes the communications network a critical component of Smart Grids [30, 28]. Since the communications network take such an important role in the correct and efficient operation of Smart Grids, making it resilient to faults is crucial.

The communications network inside a substation (in the NAN), where well defined applications typically run, can be made more resilient using traditional techniques (e.g., adding link redundancy). A more challenging problem is enhancing the resilience of the network that interconnects all substations and the control center (the WAN). This network spreads over a large geographical area, and some parts of it may be owned by a telecommunications provider. Due to its scale, such network can be more prone to accidental problems and attacks, requiring advanced solutions to address the uncertainty of the network state. Our work is focused on this particular part of the network: the WAN.

1.3 Conventional communications networks

In conventional communication networks, such as those typically used in traditional grids, the control and packet data planes are bundled together in the same device. In other words,

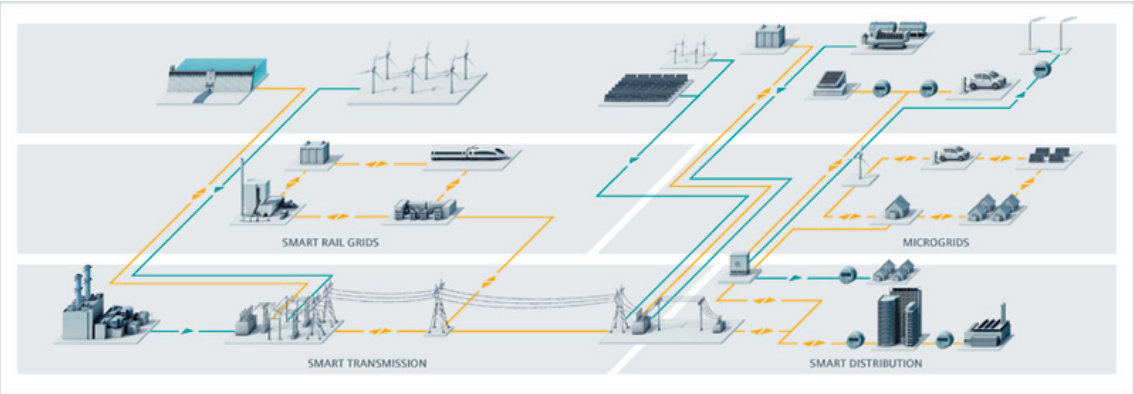

Figure 1.2: Smart Grid Information Flow [1]

the switches and routers deal with two problems: 1) forward packets, and 2) decide how to forward packets. This property makes networks complex to manage and control. Researchers and practitioners have been arguing that this complexity indeed derives from the integration of control functionality in devices that should only be responsible for forwarding packets [37]. Adding to the challenge, Smart Grids run applications with strict requirements, including maximum limits on latency and packet loss. Fulfilling these requirements, entails sophisticated algorithms to be executed in the control plane of each device, increasing the configuration complexity and therefore the probability of errors occurring. These issues mainly derive from the fact that the control plane is decentralized. This means that if multiple components of the network need to agree on a given action, they must run a distributed algorithm, sharing their current view of the network in order to get the required result. These algorithms increase the necessary complexity and computing power, leading to greater costs since more expensive hardware is required.

1.4 Software Defined Networks as an Alternative

Software Defined Networks (SDN) have been proposed as a solution to this problem. By decoupling the data plane from the control plane, and logically centralizing the control plane in a controller, many of the issues that affect conventional networks are mitigated. At the data plane, the devices are only responsible for forwarding packets, generating events when changes occur, and keep the record of simple statistics (such as packet & byte counting). This is specially important in large networks such as the ones in Smart Grids. The control plane now sits in a logically centralized controller, which typically runs in a cluster of high end servers being responsible for all the complex tasks, such as maintaining the network topology, routing decisions, recovering from failures, access control, etc. With the aid of the dataplane devices, the controller builds a global view of the network, making the computation of all those functions more efficient and effec-

tive. Another advantage of SDN is the fact that this shift in the control logic promotes openness, removing existing constrains set by current network equipment, allowing the programmatic configuration of the network, facilitating the management and optimization of network resources dynamically. These characteristics make SDN an interesting solution to achieve the required network dynamism, adaptability and security requirements that are specific of Smart Grids.

1.5 Objectives

The goal of our work is to enable resilient communications between Smart Grid substations at the WAN level, while enforcing network policies across the infrastructure, in order to fulfill the Smart Grid requirements. We want to do so while abstracting the communication infrastructure and technologies used, since different Distribution System Operators (DSO) can use various alternative network implementations. For instance, some DSOs prefer to completely own their WAN infrastructure to have better control, namely for potential failures, while others prefer to rent dedicated lines from telecommunication operators. Our solution thus aids to perform independently from the communication infrastructure topology, infrastructure, and technology employed.

1.6 Contributions

To achieve these objectives we propose, design and implement two solutions, based on SDN, to manage the Smart Grid. The first contribution is a solution to securely monitor a Smart Grid network. Its core is an algorithm that gathers information from the network to detect ongoing attacks to the monitoring system itself. To address switch performance limitations, namely the rate of control traffic SDN switches can send to the controller, the algorithm includes logic to balance between impact performance on the switch and the effectiveness of attack detection.

The second contribution is a solution for resilient communications. We propose an algorithm that uses information gathered by the monitoring module to guarantee resilient communications between each pair of WAN nodes. In addition, it enforces the strict Quality-of-Service (QoS) required by Smart Grid applications. To minimize network disruption, the solution includes a backup route algorithm that, in the event of an acidental failure (e.g. a switch or link failure) re-routes packets immediately sent to a backup route reducing, or even eliminating, disruption time.

We have implemented and evaluated our solutions. Through the evaluation we were able to conclude that our solution quickly adapts to the network conditions, but takes some time to detect attacks on links with low throughput. The rerouting after the detection of

an attack is executed in less then 2ms, which is a negligible time for the requirements of a smart grid network.

1.7 Document Structure

The remaining of this document is organized as follows:

- Chapter 2 describes related work, covering topics such as software defined networks, network routing, traffic engineering, and monitoring solutions.
- Chapter 3 explains our solution, details the design of the different modules and its implementation.
- Chapter 4 presents the evaluation of our solution.
- Chapter 5 concludes our work, reviewing what has been achieved.

Chapter 2

Related Work

Traffic engineering, network management and monitoring are subjects that have been well studied by the networking community. Many of the limitations are also well known. The introduction of new approaches to computer networking, such as SDN, open new ways to solve some of the issues of existing infrastructures. In this chapter we provide, in Section 2.1, a brief introduction to SDN, regarding its architecture and the specific control protocols used, and also a description of some of its uses. We describe some attacks that these solutions are vulnerable to and that can be leveraged by an attacker to either mislead the system or control the traffic flow. We also present in Section 2.2 some of the most common network routing algorithms and their specific use cases. This information allowed us to understand the algorithms that best fit Smart Grids. Afterwords, in Section 2.3, we introduce several traffic engineering algorithms describing their characteristics and also some of their limitations, such as achieving a high link utilization. We then describe, in Section 2.4, various techniques for network monitoring, their advantages and disadvantages, and detail some of the existing security issues. Identifying such issues allowed us to get a better perception of the type of attacks that exist to monitoring systems and apply that knowledge while designing our solution. Finally, in Section 2.5, we describe some tools to emulate and test networks, which we use in our evaluation to simulate the WAN of a Smart Grid.

2.1 Software Defined Network

SDN is an approach to computer networking where the control plane is logically centralised in a controller, and is physically separated from the data plane. This addresses some of the limitations of current network infrastructures, since coordination among the multiple data forwarding devices is no longer required. The core benefit of centralization of control is direct programming of the network from a vantage point.
In order to separate the network's control logic from the data forwarding plane, a well-defined interface between the switches and the SDN controller is necessary. Openflow [3]

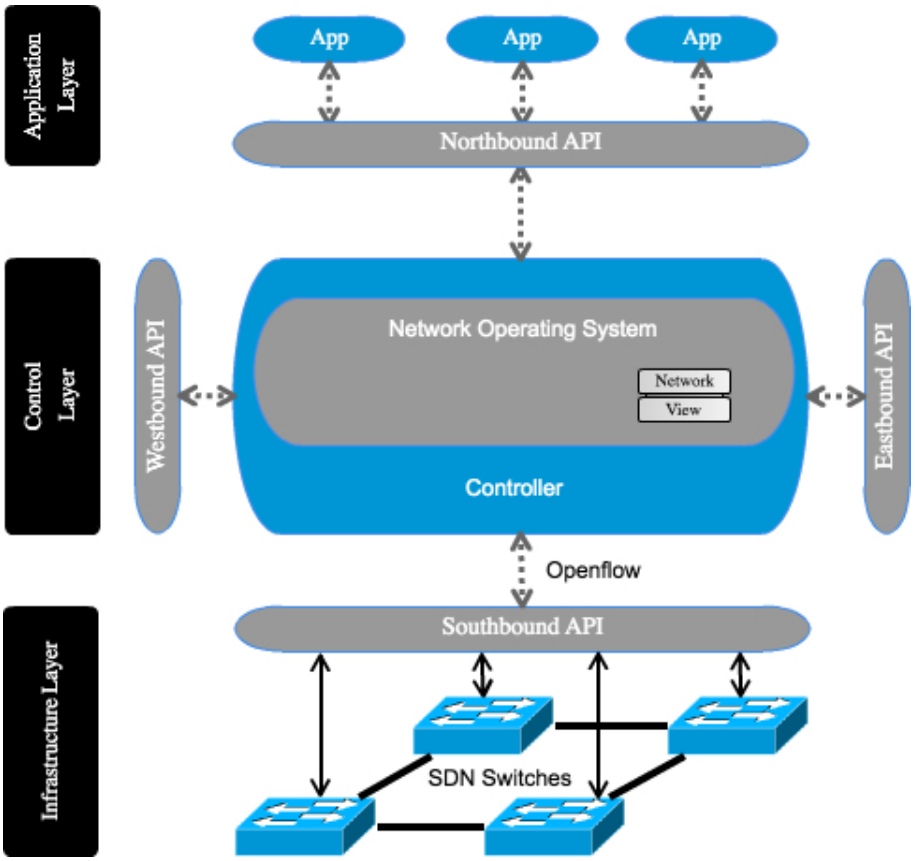

Figure 2.1: General SDN architecture

is nowadays the most commonly used specification for this interface.
A SDN can be decomposed in the Network Operating System (NOS) and a set of network applications that define the policies to be implemented by the SDN switches(see Figure 2.1). The NOS offers services to the network applications and abstracts all the low level communication required to control the traffic forwarding on the SDN switches. This separation releases the network application programmers of the low level details of the interface specification, and abstracts future changes to the specification.

2.1.1 Architecture

The SDN architecture, as defined by the Open Network Foundation (ONF) [4], is a three-layer stack, composed of the infrastructure layer, the control layer and the application layer. The aim of this layered structure is to provide well defined interfaces that enable the development of software to control the connectivity provided by a set of network resources, and the packet traffic that flows through them. The remaining of this section details each component of the common SDN structure as seen in Figure 2.1.

The infrastructure layer, as in traditional networks, is responsible for forwarding packets. These packets are transmitted according to the rules sent by the controller through a

communication channel that the device established with the controller. This communication channel can be either out-of-band or in-band, depending on whether it is established through a dedicated interface or not. The network devices in this layer communicate with the controller using the Southbound API (typically, Openflow [3]). The control layer is the controller. This entity is responsible for maintaining a global view of the network and updating it according to the events received from the infrastructure layer. This Network View is then made available to the applications using the Northbound API (typically a REST interface).
The controller is also responsible for making any changes requested by the Applications Layer effective in the devices on the infrastructure layer. The Eastbound and Westbound API only exist when distributed control is employed (which is the common way in production). Finally, the Applications Layer is where the network logic resides. Here, applications use the network view provided by the controller to enforce the global objectives required by the network administrators. Examples of such applications include firewalls, routing, traffic engineering, among others.

2.1.2 Openflow

Openflow is the most commonly used communications open standard between the control and the infrastructure layers. It defines a set of instructions that enable direct access to the network devices, allowing full control on how data packets are forwarded. These instructions are sent through a secure channel and are processed and stored on a table in the Openflow enabled switch (see Figure 2.2).

Openflow uses the concept of flow to identify traffic based on pre-defined rules, which are composed of match fields and actions, as defined in the Openflow specification [3]. When an Openflow enabled switch needs to know the appropriate action to take with a packet, it starts by parsing the packets and then searches its flow tables for a matching flow entry. When a match is found, the action defined in that flow entry is executed on the packet. Multiple actions can be applied to the packet such as adding, removing or changing specific header fields, drop the packet, forward the packet to a specific switch port, send the packet to the controller for further analysis, etc. The expected behaviour of a switch and the dataplane protocols supporting it, are also defined by this specification.

2.1.3 SDN Controller

A SDN Controller logically centralizes the network intelligence. The SDN controller runs in commodity server technology and can be placed anywhere on the network. It is responsible for managing the flow entries on the switches and take the appropriate actions to guarantee the correct flow of traffic. To accomplish this, it runs multiple applications that listen to switch events, such as link up/down. These events are sent by the switches

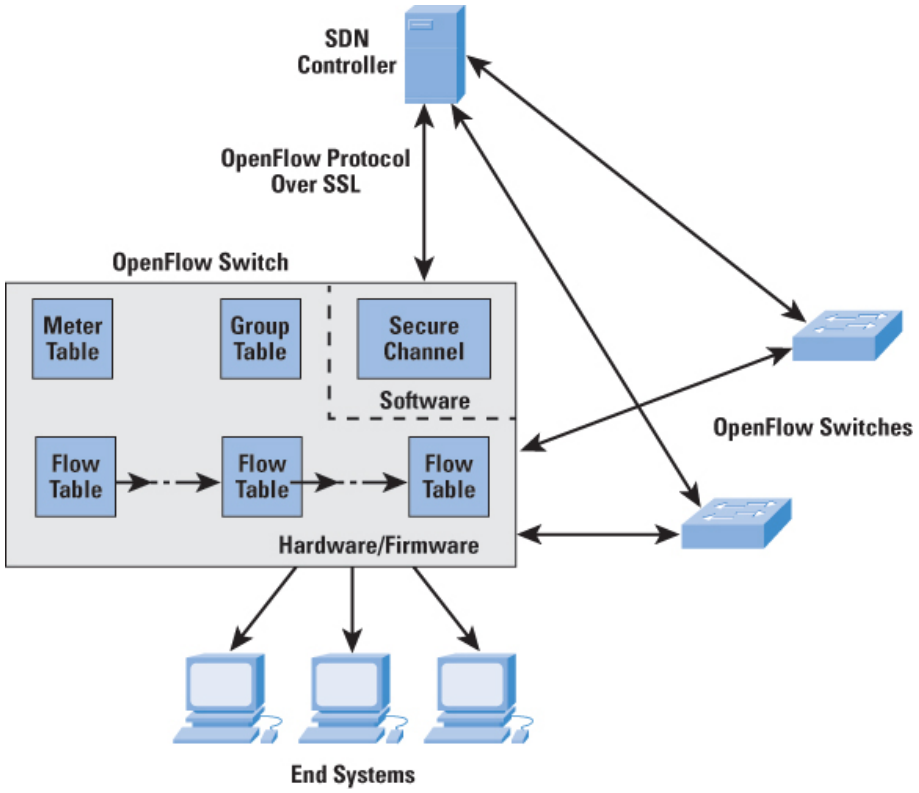

Figure 2.2: Openflow Switch [2]

to the controller using Openflow messages.
As mentioned, the SDN controller is subdivided in two parts, the NOS and a set of network applications. The NOS uses its southbound interface to perform all the low level communication with the SDN switches, and employs it northbound API to provide services to the network applications. The Northbound API provides ways for network applications to obtain information from the switches and also to define policies that will be used by the switches to process packets. The abstraction provided by the controller allows a faster development of network control applications. This makes SDN an ideal solution to create custom network behaviours that follow the applications requirements.
Bellow we present a few examples of SDN controllers.

NOX

NOX [5] was the first Openflow controller (2008). It was developed in both C++ and Python and has an event-driven programming model. It included in its core a group of applications that enable the creation of a single network view based on its observations of the network.

Opendaylight

Opendaylight [6] is an open source project supported by several major network companies like IBM, Cisco, Juniper and VMWare. It presents a new SDN controller layered structure implemented in Java which aims to provide a 'universal' SDN platform. Its Service Abstraction Layer allows the management of devices, using multiple APIs and protocols such as Openflow, OVSDB, NETCONF, SNMP, among others.

ONOS

ONOS [7] is also an open source project hosted by The Linux Foundation that was specifically designed to meet the needs of large operators while offering the flexibility to create and deploy new dynamic network services with simplified programmatic interfaces.

Floodlight

Floodlight is an open source Apache licensed controller developed in Java that has been designed to have high-performance. The overall functionality comes from its modules, which use its core to communicate with switches. The implementation is multi-threaded and event based. In this work, we choose Floodlight for the following reasons:

- It offers a modular loading system making it simple to extend and enhance;
- Can handle mixed Openflow and non-Openflow networks making it ideal for an incremental update of the network to Openflow switches;
- It uses the asynchronous event based multithreaded library Netty, which improves performance on large networks, making it ideal for time critical applications;
- It has a very active community of developers.

2.2 Network Routing

In general, routing aims to answer the question: "What is the best way to get from point A to point B". Network routing answers that question for every packet that enters the network, but dynamically since links go up and down, routers fail, and so the best route from A to B is constantly changing. When such actions take place, the existing routes have to be updated.
In traditional networks the control plane is distributed among all devices. This lack of a centralized view means that a routing protocol must be executed in the control plane of every device, so that each can update their own view of the network and know how to forward packets. Routing protocols can be organized in one of two classes. Some routing protocols, such as Open Shortest Path First (OSPF) [14], and Intermediate System to

Intermediate System (IS-IS) [12] are link-state routing protocols [42]. Routing protocols in this class, map the connectivity of all the network in the form of a graph in each node. This requires any change in the network to be flooded to every node, whereas in a centralized setting (as in SDN) only the controller is required to run the Dijkstra algorithm, on the new network topology. One advantage of this class of protocols is that they support setting static weights to each link, such as cost, that are taken in consideration when choosing the best route.
The second class of routing protocols is distance-vector [42]. Protocols of this class only inform their neighbours about changes in the network. These protocols do not create a graph of the network, but instead they keep vectors with the distance in hops that a route takes to reach each node. These protocols use a distributed approach and have scalability issues. Example of such protocols are the Routing Information Protocol (RIP) and the Border Gateway Protocol (BGP).

2.3 Traffic Engineering

Communication networks are dynamic environments that require constant adjustments to improve user performance and making more efficient use of resources. Traditional routing protocols enable connectivity but they make it difficult to manage rapid changes in the network. This is the goal of Traffic Engineering (TE) [15, 19, 18]. The typical goal is to distribute load across the network to avoid hotspots. There are multiple ways to achieve this kind of optimization. One commonly used is Equal-Cost Multi-Path routing (ECMP) [16], where traffic is split among the multiple routes that have the same cost from point A to point B (multiple shortest paths). This technique can be used for instance in OSPF networks, producing better results than using OSPF alone (restricted to a shortest path).

2.3.1 Traditional Networks TE

Today's networks typically use Multiprotocol Label Switching (MPLS) for TE. MPLS is an advanced routing scheme that provides extensions with respect to forwarding and path controlling. Each MPLS packet has a special label that the router uses for a lookup in its forwarding table. This enables routing not only based on the destination IP, but also on this label [20], allowing traffic to be sent through different paths even for the same destination addresses. By using MPLS with Constraint-Based Routing (CBR) [38] and reserving resources with the Resource Reservation Protocol (RSVP) [19], it is possible to increase substantially the efficiency of current networks.

2.3.2 Software Defined Networks TE

Despite the improvement that MPLS-TE brought to traditional networks, it is still far from ideal. Even with the most recent technologies and techniques, network providers are unable to fully leverage their investments, with the busier links still having an average utilization around 40-60% [39]. One of the main reasons that contribute to this low utilization is the poor efficiency of the distributed resource allocation that is used in MPLS-TE, as no entity has a global view, and ingress routers greedily select paths for their traffic. This leads to configurations where efficiency is suboptimal.
SDN offers a solution to this problem. The controller has a complete knowledge of the network and therefore it is capable of taking decisions that are globally optimal. This centralized control also eases the required frequent updates to the network data plane, necessary to maintain high utilization, since the execution of the complex distributed protocols used in traditional networks is no longer necessary. SDN enabled providers to use their links for long periods of time at near 100% utilization [40] and improve the network global efficiency.

2.4 Network Monitoring

Network monitoring is the process of continuously measuring a computer network for multiple metrics, such as availability, delay, jitter, bandwidth, error rate, and loss rate. These metrics are fundamental to TE as it is based on these values that routing decisions are made. Network monitoring therefore is fundamental in allowing network traffic to be managed efficiently. Besides traffic engineering [39], network monitoring has multiple other applications such as anomaly detection [43], costumer accounting [32], Service Level Agreements (SLA) checking, identification of applications [29], forensic analysis [50], among others.
Network monitoring has such an important role in communication networks that multiple efforts from the research community have been made in order to provide solutions that are fine-grained, scalable and accurate. In the rest of this section we describe some protocols and technics used in monitoring systems.

2.4.1 Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) [13] is an Internet-standard protocol created in 1988, still widely used to collect information about managed devices. Devices that support SNMP keep track of multiple parameters such as the amount of forwarded traffic, including the number and rate of packets (and bytes) forwarded, the packet error rate, etc. The devices then expose their management data in the form of variables to which the Network Management Station (NMS) can access through GET requests. The

NMS periodically queries the network devices to obtain the desired monitoring information about the network. Accurate monitoring information demands frequent queries to multiple parameters, which may degrade the network device efficiency due to CPU overhead. The information available in each network device is defined in a Management Information Base (MIB), which is vendor/model dependent. However, vendors tend to implement SNMP counters only for aggregate traffic, making SNMP not suitable for the fine-grained statistics required by traffic engineering. Earlier versions of this protocol allowed an attacker easy access to all the information provided by a device, but in later versions, starting in the version three, authentication, authorization, access control, and privacy, was added, limiting the attackers actions. Nevertheless, an attacker that is able to inject or drop traffic on a given link is able to remain unnoticed if one uses only this protocol to monitor the network.

2.4.2 Active Probing

Another approach to network monitoring is active probing. In this approach, a special packet is sent through the network, enabling information about the network status to be extracted, either by the receiver or by the sender. Tools such as ping, traceroute, tracert, and others implement some form of active probing. As such they can be useful to collect different sorts of information for many of the applications referred before, specially to monitor delay, loss rate, jitter, and available bandwidth. However, security concerns arise because an active attacker may be able to easily identify the probe packets and treat them differently from the normal traffic, to evade the monitoring system. For example, after having identified the packets that are used for latency measurement in systems such as OpenNetMon [25] and SLAM [51], an active attacker can delay the delivery of the probes to fool the link latency estimates calculated by the monitoring system. A smart attacker would then be able to control traffic routing, for example to direct packets to locations under his control.

2.4.3 Packet Sampling

The first packet monitoring solutions were based on specially designed equipment installed on the network [26]. These machines would do passive monitoring by copying the entire contents of packets for further analysis. This approach relieved network devices from the monitoring overhead and enabled fine-grained monitoring, but it also put a lot of pressure in the monitoring machines and was very expensive to deploy. For scalability reasons, only the initial part of the packet [31], such as the protocol headers and the initial contents of the application data, were captured and stored. With the advent of high-speed switched networks this approach became infeasible, and packet sampling techniques started being employed [21] [22].

Packet sampling consists in randomly capturing only a subset of the packets (typically at a rate of 1:1000 or 1:10000), under the assumption that the subset is representative of the whole network traffic. This subset is then used to measure different properties of the traffic. Choosing a representative subset is specially challenging in communication networks given that even if a certain traffic exhibits a temporal cycle (daily, weekly), traffic engineering stations can blur that cycle by spliting or simply changing it from the ”normal” path. The sampling frequency is a specially important factor because it translates into the accuracy of the estimation procedure. With a low sampling frequency some network behavior might be missed, which can be harmful when performing anomaly detection. In the opposite side, high sampling frequency produces vast amounts of data that need to be processed, creating overhead in the devices, which can lead to packet loss. In any case, this technique is widely used, with protocols such as Netflow and sFlow.

Trajectory Sampling

One of the problems of the traditional sampling approaches is the lack of information on the exact paths packets follow. To solve this problem, in trajectory sampling the same set of packets is sampled, in both ingress and egress points. This allows the same packets to be tracked as they flow through the network. Existing implementations require specialized hardware to be integrated in network devices. This technique enables also per-flow monitoring, which supports an even more fine-grained monitoring, and it has been used and studied in multiple works [46, 36].

2.4.4 SDN Monitoring

Flow-based measurements such as NetFlow and sFlow provide generic support for different measurement tasks, but consume too much resources (e.g., CPU, memory, bandwidth) [33]. The centralized architecture of SDN and the per flow statistics supported by Openflow are two major advantages when developing a monitoring system. The programability enabled by SDN enables a customised dynamic measurement data collection, making it possible to have the right amount of accuracy while minimizing resource consumption.

Approaches like OpenSketch [53] suggest the redesign of switches APIs to accommodate for customized measurement functionality. With the proposed changes, their solution supports the collection of measurement data on a per flow basis and additional flow and data filtering. To accomplish this, switches execute a hashing function over the defined packet fields and count the number of packets that match a set of defined rules. This information is then stored within the switch in TCAM (Ternary Content-Addressable Memory) memories. Despite having advantages, such functionality would require an upgrade or replacement of all network nodes. Others, such as OpenNetMon [25], focus solely on a few

metrics like delay, throughput and packet loss, not considering security or anomaly detection. It makes use of Openflow functionality to periodically query the switches and obtain flow statistics to calculate the flow throughput and error rate, this makes it prone to traffic inflation/deflation attacks, since an attacker that controls a link on the monitored path can inject or drop packets making the throughput measurements incorrect. For latency measurement, OpenNetMon sends a probe on a specific vlan and measures the time the probe takes to arrive at the end switch of the monitored path. This allows a attacker that controls a link on that path to influence the latency measurement, since the attacker can easily identify the probe packet and delay it, tricking the monitoring system to measure a latency value different from the real one.
OpenSample [48] goes a bit further in regard to the granularity of monitoring information it analyses. By using sFlow, OpenSample is able to get more detailed information of the current network traffic, but does not focus on security, and is vulnerable to the attacks previously described. Similarly to OpenNetMon, SLAM also uses a recognizable probe to measure latency. Instead of measuring the latency of a path, SLAM measures the latency on a switch to switch basis, and accounts for the switch internal processing time. It focuses solely on measuring latency, not accounting for other important measurements such as error rate or throughput. It suffers from the same security issues as OpenNetMon.
FlowSense [52] uses a passive technique to estimate network performance. In FlowSense, *packet-in* and *flow-remove* messages are used to estimate per flow link utilization. The communication overhead in this case is low, and it does not suffer from the security issues of active probing, but the estimation is not as accurate as with the active approaches.
Despite the multiple works in this area, to the best of our knowledge none has considered the resilience aspect in network monitoring, which is our main goal.

2.5 Tools

We have used some tools to realistically emulate multiple topologies of a smart grid network and to evaluate our solution. These tools are described in this section.

2.5.1 Mininet

Mininet [8] is a software tool able to realistically emulate networks. It runs a collection of end-hosts, switches, routers and links on a single Linux kernel. Some advantages of using mininet are as follows:

- By using lightweight container-based virtualization they are able to make a single system to emulate a large-scale network with hundreds of nodes, with each device behaving like the real hardware;

- It allows the creation of custom topologies, from a single switch to Internet-like topologies and also the execution of regular software inside each container;
- Packet forwarding is customisable because mininet switches can use the Openflow protocol.

Despite being fast and flexible, it also has some limitations:

- Since it uses lightweight virtualization, the Linux kernel is shared by all virtual hosts. This means that it is not possible to run software from other operating systems;
- All virtualized systems execute on the same host, so it is necessary to limit traffic rates, otherwise performance issues may arise with the system becoming the bottleneck, not the emulated network under analysis.

2.5.2 Scapy

Scapy [9] is a powerful packet manipulation framework that is able to change packets in real time. It supports a wide number of protocols and is able to capture, decode, match requests and replies and send them on the wire. It also offers the possibility of creating packets from scratch and injecting them on the network.

2.5.3 Iperf

Iperf [10] is a tool for active measurements of throughput of IP networks. It implements a client and a server, and supports various protocols such as UDP and TCP. It can measure packet loss, delay jitter, and bandwidth/throughput.

In the next chapter we describe our solution which was designed to fill the gaps previously identified while maintaining the requirements of Smart Grid networks. We will describe how our solution coupes traffic inflation/deflation attacks and also how we are able to perform trajectory sampling without the need of adding additional hardware for existing SDN switches.

Chapter 3

Design and Implementation

Our goal is the design and implementation of a SDN based solution to resiliently route smart-grid traffic at the WAN level. The solution consists of two applications. The first is a monitoring solution responsible for querying network switches to gather information about the current state of the network. The second application uses that information to create resilient routes along the WAN network. The solution aims to prevent several types of attacks, so we start this chapter by defining them.

3.1 Attack taxonomy

Given that traffic engineering decisions are based on information provided by the monitoring system, an adversary might target the monitor to compromise routing decisions. In order to design a resilient solution, we have identified multiple attack vectors that might be exploited.

3.1.1 Generic adversary capabilities

Any device from the smart-grid network can be used as an attack vector. Even components that are not under our control pose a threat. In this section, we describe the capabilities of an adversary that aims to mislead our system.

Traffic eavesdropping: is the capability of an adversary to observe and record packets that pass through a link. One example of such attack is when an adversary has physical access to the wire that is used to connect two devices and is able to passively wiretap it.

Traffic injection: is the capability of an adversary to insert unaccounted packets into a link. One example of such attack is when an adversary has control of a device that is inside the network, which is employed to generate and inject packets.

Traffic interception: is the capability of an adversary to control the packet flow that traverses some resource, allowing him to manipulate any packet. One example of this kind of attack is when the adversary controls a device in the path between two SDN switches

(e.g., the forwarding devices used in a dedicated link provided by a communications operator) and uses that device to manipulate any packet that traverses it.

Traffic rerouting: is the capability of an adversary that controls two or more links to drop a given number of packets that are traversing a link and injects those same packets on any other link controlled by the adversary. One example of such attack is when the adversary controls two or more devices forming the path between two SDN switches under our control, and uses those two points to redirect packets. This particular attack is depicted in Figure 3.1, where the attacker drops packets flowing through the top path and injects them in another link under his control, on the bottom path.

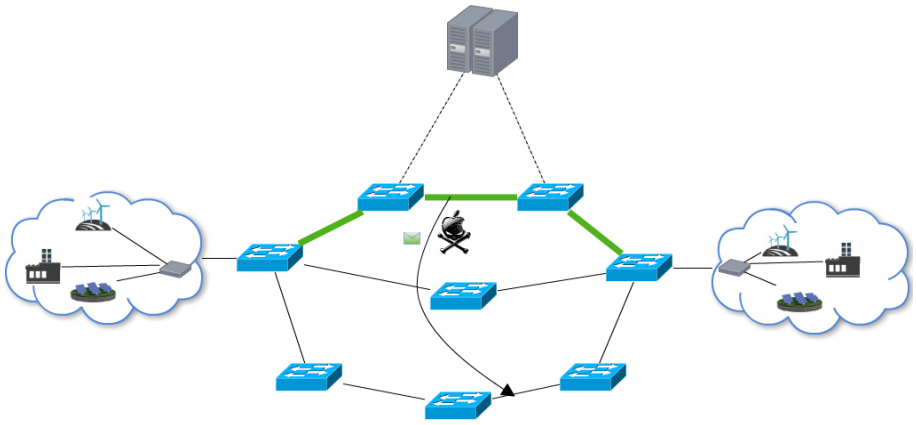

Figure 3.1: Example of a packet reroute performed by the adversary

3.1.2 Attacks on latency estimation

Latency is one of the fundamental parameters used in traffic engineering. Hence, protecting its estimation is of crucial importance. If not properly protected, it can become the target of various kinds of attacks, performed by adversaries with access to the link being monitored.

The common SDN monitoring applications measure the latency of a link by recording the time a given packet leaves a link entry-point and when that same packet arrives at the link exit-point. The latency of the link is then calculated by subtracting the entry-point time from the exit-point time. The packet used for the estimation can be a special packet generated by the controller (a "probe" packet), or it can be a normal packet that is flowing through the link. We refer to this mechanism as *probing*, in both scenarios.

Below, we describe two adversary capabilities that are specific to the probing environment:

Probe spoofing: the capability of an adversary to make a monitoring SDN application accept forged probes for estimating link delay values

Probe anticipation: the capability of an adversary to predict the timing of probe reception by a monitoring SDN application.

Probe spoofing capability

An adversary can only successfully spoof probes if the monitoring SDN application cannot distinguish between a fake probe and a genuine. This capability presumes the adversary already knows about the probe generation and filtering mechanisms, e.g., by having access to the source code or by deducing the mechanism behaviour from repeated observation of transmitted probes.

If the probes are invariable and the adversary knows about their content, spoofing probe packets is trivial. If the probes are variable but the SDN application does not check the validity of their contents (or does so incorrectly), then the adversary can spoof probes by forging packets with arbitrary contents. Specifically, the adversary may be able to insert contents in such a way that it effectively manipulates the SDN application into making erroneous delay calculations (e.g., if the contents include a timestamp that is used for calculating latencies, and the adversary inserts a fake timestamp). Importantly, wrong latency estimations can result in undesirable changes to packet routing (e.g., sending packets to locations controlled by the attacker).

If the probes are variable and the SDN application strictly checks the validity of their contents, i.e., if the contents of each probe received are validated by matching an exact byte sequence, then the adversary can still spoof probes if he can predict the contents of future legitimate probes (e.g., if each probe includes a sequence number that is incremented in every transmission, an adversary with eavesdropping capabilities can easily predict the next sequence number).

A different situation occurs when actual user generated traffic is used as probe. In this case, the SDN application mirrors to the controller part of the normal traffic to be used as probes, as they traverse the switches that interconnect the link being probed. In this case it becomes harder to spoof probes because their content is generated by the user application. However, if the adversary can first discover which data packets are being probed and reverse engineer the user application, he may be able to guess the contents of future packets, and therefore spoof probes, in case the application has predictable behaviour for at least a subset of the network traffic. As an alternative, the adversary could attempt to introduce packets in the network to be transmitted through the link.

Probe anticipation capability

An adversary with this capability can successfully predict the timing of probe reception by a monitoring SDN application (with a small error margin). If the attacker is able to guess the probe transmission frequency and the propagation delay between the adversary's point of injection and the controller, he will be able to anticipate probes.

The probe transmission frequency may be static and known by the adversary if he or she obtains it via source code disclosure or deduced from repeated eavesdropping of transmitted probes. The transmission frequency may also be dynamic and dependent on

certain properties of the data traffic passing through the link (e.g., it may vary according to the current throughput). In this situation, the adversary may be able to deduce the frequency from continuously monitoring eavesdropped data traffic and by applying the same algorithm used by the SDN application. A similar strategy could be employed in the scenario where the probes are normal data packets that are mirrored to the controller.

Attacks Forms

An adversary can attack the latency estimation using the previously described capabilities in the following ways:

Probe poisoning: An adversary with the capability of probe spoofing and probe anticipation sends probes at specific (shorter than normal) times, making the monitoring system estimate lower link delay values. If the monitoring system uses the probe to transport a timestamp, it is also possible to force the monitoring system to estimate longer link delay values.

Probe flood: An adversary with the capability of probe spoofing continuously sends spoofed packets, leading the monitoring system to calculate incorrect delay estimations, as these packets arrive with high probability at unexpected times.

Probe replay: An adversary with traffic eavesdropping and traffic injection capabilities eavesdrops legitimate probes and injects them at a later time. Since the probes suffer higher delays the monitoring system will estimate larger link delay values.

Probe disturbance: An adversary with traffic interception capability intercepts all packets and selectively delays either the data traffic or the probes, making the calculated link latency values inconsistent with the real propagation time of data packets.

3.1.3 Attacks on throughput estimation

Another fundamental parameter in traffic engineering solutions is throughput. To obtain this estimate, a monitoring SDN application can leverage the capability of SDN devices to report traffic statistics to the controller in order to estimate traffic throughput at any given moment in a link. For example, Openflow-enabled switches can report periodically the total number of bytes transmitted/received in some port. As that same report can include the elapsed time since the previous request, two consecutive reports from the same switch enables the SDN application to calculate the average (transmission/reception) throughput of the switch port. This mechanism is also prone to attacks.

Throughput inflation: An adversary with traffic injection capability can add traffic on a link under his control, increasing the real throughput on that link. As a result, the

monitoring application would be presented with a lower entry throughput (to the link), when compared to the exit throughput.

Throughput deflation: An adversary with traffic interception capabilities can drop packets that flow on a link under his control. In this case, the monitoring application would be presented with a higher entry throughput, when compared to the exit throughput.

In either case, depending on how the monitoring application acts, inflation and deflation attacks could negatively influence the decisions made by other modules, including those of a traffic engineering application.

3.2 Threat model

We consider that the link between two SDN switches under our control can be composed of an arbitrary number of other components, such as routers and switches, which may or may not be SDN-capable. An adversary may want to attack the monitoring system of the network to influence routing decisions, by using any of the previously described capabilities. We limit the adversary actions to the WAN data plane. Attacks in the NAN and the HAN, and in the switch-controller connections are out of scope. With respect to the latter, we assume that the communication channel used between the SDN switches and the controller is secure (e.g., using TLS, as per the Openflow specification), and that the SDN switches themselves are not compromised by the attacker, meaning that they operate accordingly to their specification.

3.3 Monitoring

The monitoring module consists of various submodules, as represented in Figure 3.2, that gather information about the current state of the network. Multiple techniques are used in order to effectively obtain the most up-to-date information about the network, while ensuring correctness of that same information, and while minimising the performance impact. The monitoring module periodically queries all switches to extract information on its counters and port status so that it can build the global network state. As it is possible to control the monitoring period, this method has low overhead. However, this technique has vulnerabilities that make it prone to several attacks to monitoring [35]. For protection, we have devised an algorithm that verifies if there are ongoing attacks to mislead the monitoring which resorts to random trajectory sampling at its core.

Trajectory sampling is a method of packet sampling where packets that flow through a given link (or subset of links) are sampled within a measurement domain. The main difference from normal packet sampling is that in trajectory sampling one also “follows”

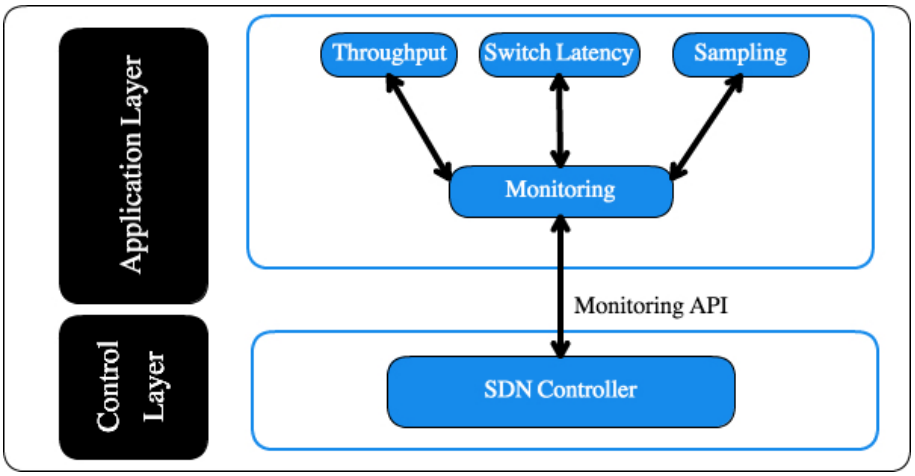

Figure 3.2: Monitoring architecture

the packets along the paths. For this to be possible, it is not enough to just randomly sample each link, as one would not be able to derive the precise path that a sampled packet has followed through the measurement domain, from the ingress point to the egress point. For this purpose, a deterministic hash function is executed for each packet that traverses the network device. This allows tracking a packet as it flows throughout the network, and usually requires specialized hardware to be integrated in network devices. This technique enables fine-grained, per-flow monitoring, and as such has been used in multiple works [46, 36]. With this technique we are able to verify if the values that we receive from different switch counters are correct, denying the attacker a way to mislead the system.

In the following sections we give details on the design of the monitoring submodules, with a focus on how they ensure the correctness of the monitoring information.

3.3.1 Control plane latency

In order to detect attacks and have a higher degree of confidence in the information obtained from the switches, our solution always probes the two switches that are the endpoints of the link that is currently under evaluation. As mentioned before, SDN has the control plane separate from the data plane, which means that when the controller communicates with the switches there will be a delay between the issuing of the message and its delivery. In other words, when the controller sends the message to add a new entry to a flow table, this message will take some time to arrive at the target switch and to be actually applied. Even though this amount of time is normally very small in local networks, this may not be true in Smart Grids given the diversity of communication technologies that may be employed. So it becomes important to accurately estimate the delays that control messages experience.

The ControllerLink submodule measures the latency from the controller to every data-

plane device. This information is then made available to the other monitoring submodules so that they can correctly perform their time calculations. This will make monitoring more accurate and will cause less performance impact on both dataplane and control plane.

3.3.2 Throughput

The Openflow specification [3] defines a set of messages that the controller can use to obtain data from the network devices. We use this functionality to periodically collect the total number of bytes transmitted/received by some port. As explained, by subtracting the values from two consecutive reads and dividing by the number of seconds of the monitoring period, one can calculate an estimate of the throughput for that given device port. In order to detect possible deviations due to congestion on the device-to-controller-link or (possible) attacks to the estimation, the measurement is made on both ports of the link and then compared. If the measurements are too far apart then a second measurement is made to increase the probability of detecting throughput inflation or deflation attacks, or simply to confirm that the difference was due to a normal increase in the jitter induced by the network.

This technique causes the least overhead on the dataplane devices because we are only querying the switch registers, but is vulnerable to attacks. Our solution copes with this problem using the sampling technique described next.

3.3.3 Sampling

As mentioned before, several attacks might corrupt monitoring measurements if they affect a specific set of switches. To address this problem we resort to a novel variant of trajectory sampling. The main advantage of our solution is that it does not require complex hardware in the switch for its materialization. Our solution requires only conventional Openflow switches.

The goal is to compare the same set of packets that pass through the source switch of a given link, with the ones that arrive at the destination switch, during a specific period of time. As such, it allows us to verify if packets were modified, injected or dropped. This supports the detection of an ongoing attack on that link under the following conditions:

1. If the number of sampled packets on the destination switch is greater than the packets on the source switch, then packets were injected;

2. If the number of sampled packets on the destination switch is less than the packets on the source switch, then packets were dropped;

3. If there is not an exact match on the content of every packet sampled at the source switch when compared to the same packets sampled on the destination switch, then

packets were modified;

Detection can only be effective if the attacker cannot predict which packets are being sampled. Therefore, our solution samples random links during random periods of time at random intervals. Given the sampling randomness, an attacker cannot determine when a sampling is being performed on any given link, and therefore he/she cannot mislead our monitoring system, allowing any attack to be eventually detected. To perform trajectory sampling in an SDN, the packets being sampled need to be duplicated during the sampling period at the source and destination switch of a link, and sent to the controller for further processing. This can degrade the performance of the dataplane, so it is important that the algorithm responsible for performing trajectory sampling is dynamic and adapts to the current network and switch performance conditions. The sampling submodule will set the sampling time accordingly to the current throughput of the link, so that the amount of packets sampled does not negatively impact performance. As such, if a link is experiencing high throughput, then the sampling frequency is reduced.

3.3.4 Loss Rate

The loss rate is the number of packets that enter a link but do not arrive at the destination, divided by the total number of packets that entered the link in that period. This information can be directly obtained from the previously described sampling algorithm by comparing the packets that arrived at the destination switch versus the ones that traversed the source switch. Since the packets that traverse the source switch are compared with the same packets that arrive at the destination switch, any packet that was changed by the attacker is considered as loss. Thus, the loss rate takes into account both the packets that were dropped due to accidental network failures, and the packets that are dropped or tampered by an attacker.

3.3.5 Monitoring API

All the information that is gathered by the monitoring submodules are made available to all other SDN applications through the monitoring Application Programming Interface (API). SDN applications interested in obtaining information about any link can either obtain it as notifications or on demand. The routing and traffic engineering module we describe next is an example of an application that uses this API to react to network changes and push novel forwarding rules to the SDN switches. By opting for this modular design, we allow other SDN applications to use this module as its secure monitoring base.

3.4 Traffic Engineering

The traffic engineering module and its submodules are shown in Figure 3.3. This module uses the information provided by the monitoring module to continuously evaluate if the smart-grid requirements are being met. In the following sections we describe the role of each submodule and how they enable resilient routing.

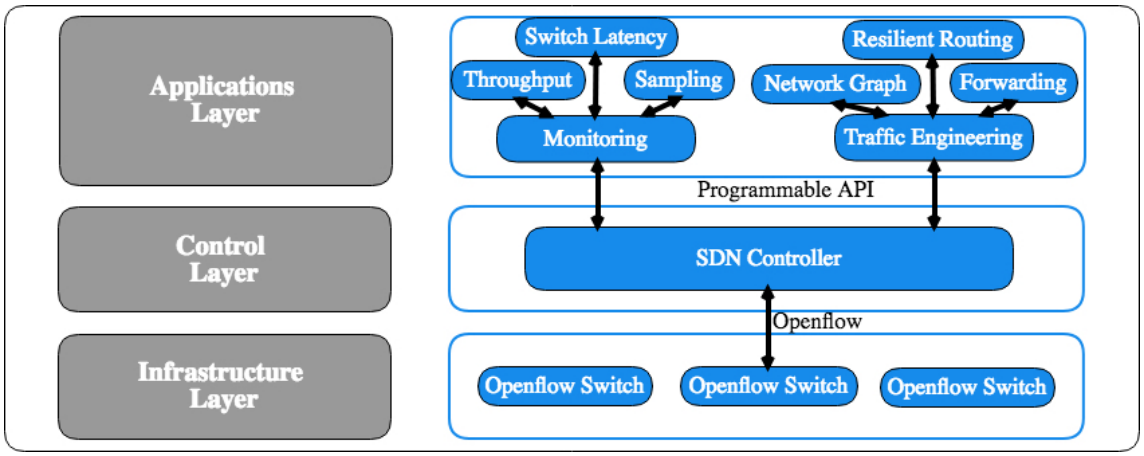

Figure 3.3: Trafic engineering module architecture.

3.4.1 Smart Grid Requirements

Different smart grid applications have different requirements when it comes to bandwidth, loss rate and latency. The smart grid requirements submodule is responsible for storing such information and also for verifying if a given route with specific bandwidth, loss rate and latency characteristics comply with the specific needs. This information is used by the Resilient Routing submodule to decide if a given route can or cannot be used for a specific smart grid application.

3.4.2 Network Graph

One of the advantages of SDN is the centralization of the network global view in the controller. This eases the process of updating a route in case if certain network events such as link failures, as it avoids the need for complex distributed algorithms to be executed. The Network Graph submodule is responsible for building that global network view in the form of a graph. This graph is constantly updated with the current network topology, allowing operations such as obtaining the shortest path between two points of the network by simply running the Dijkstra algorithm. Not having to resort to complex and time consuming distributed algorithms to re-route traffic around failed links is especially important in this context, as several smart grid applications are sensitive to latency and

packet loss.

By maintaining this up-to-date global view of the network our solution is able to quickly react to changes in the network topology causing less impact on latency and packet loss.

3.4.3 Resilient Routing

The Resilient Routing submodule is the core of the traffic engineering module. This submodule is responsible for combining the information provided by the network graph submodule, smart grid requirements submodule, and the monitoring module to make routing decisions. It continuously checks the information provided by the monitoring module and cross-references it with the needs of the smart grid requirements submodule in order to confirm that every smart grid application is running within its pre-defined limits. If a given requirement is not being met or if a link goes down on a specific route, it uses the information of the Network Graph submodule to quickly calculate the best new route between the affected points, taking in consideration the number of hops and link statistics. Our solution considers as best route the shortest path (the path with the least number of hops) that is compliant with the smart grid application requirements.

The existing separation between the control plane and the data plane in SDN is beneficial in multiple ways, but has a few drawbacks. For very time sensitive smart grid applications, the time it takes for a switch to send the information to the controller that a link has gone down, added to the time the controller takes to process that information and propagate the new route to the dataplane devices, may be incompatible with the requirements of specific smart grid applications. To cope with this issue, our solution is designed so that in the case of such event the data-plane devices already have the information on where to re-route the traffic. This is accomplished by means of backup routes. This route is installed in switches but is only used when the dataplane device detects that the normal route for a given packet is no longer valid, such as when the egress port is down due to link failure.

Taking as example the network depicted in Figure 3.4, while calculating the route between A and B, the resilient routing module has chosen the route composed by A-$S1$-$S2$-$S3$-B. This selection was made because at the time this was the shortest path that was compliant with the smart grid applications requirements. Besides the normal route, the controller also calculates backup routes and installs them on each dataplane device so that if a link (or switch) goes down on the normal route, the dataplane device that detects that failure will quickly divert all packets to the backup route.

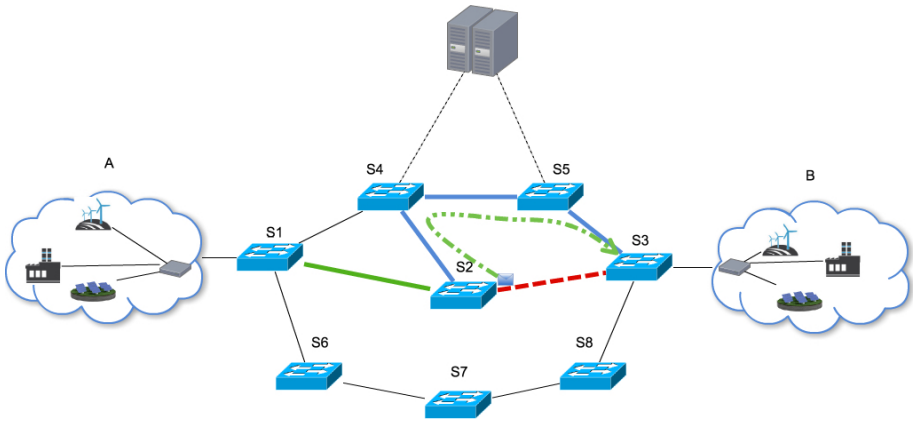

Figure 3.4: Backup route in action

As an example lets assume that the doted line between $S2$ and $S3$ represents a link failure detected by $S2$. In this case accordingly to the backup route, $S2$ will send the packets to $S4$, which will detect that these packets are using the backup route. Therefore, it will forward the packets to $S6$ and then to $S3$ before final delivery to B, as denoted by the chained line.
More complex cases are also covered, as when the backup route requires the packets to return back a few links before being transmitted through an alternative path (as is visible in 3.5).

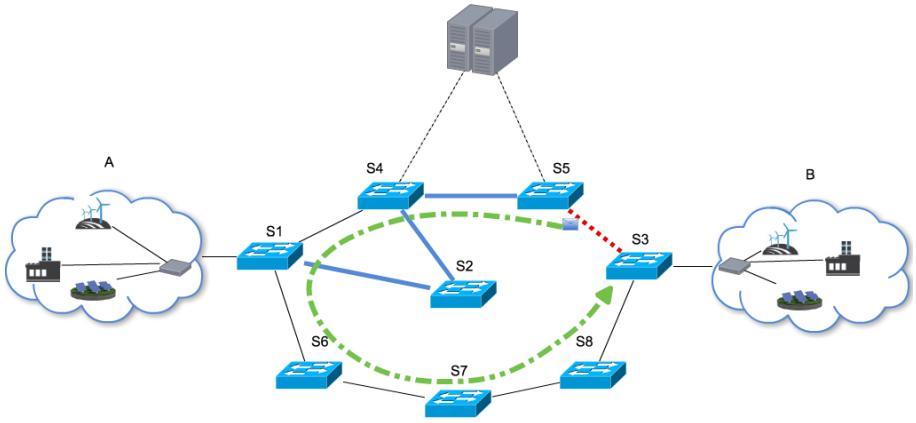

Figure 3.5: Complex backup route in action

In this example, the link between $S5$ and $S3$ has failed and since $S5$ had no lateral path to forward the packets that lead to B, packets will return back to $S1$ and then will follow the backup route composed by $S1$-$S6$-$S7$-$S8$-$S3$-B. Therefore, a packet that was sent from A would take the path A-$S1$-$S4$-$S5$-$S4$-$S1$-$S6$-$S7$-$S8$-$S3$-B as denoted by the chained line. This will guarantee that if there is an alternative path, no packet will be lost (starting from the moment the dataplane device $S5$ detects that its link to $S3$ is no longer up). After the controller is notified that the link is failed, it will recalculate the routes and

push the new forwarding rules to the data plane devices. This will result in potentially novel normal and backup paths.

3.4.4 Forwarding

After the routes are generated (both normal and backup) they need to be converted to data plane rules, to update the switching devices. The Forwarding submodule is responsible for this operation and for maintaining the list of routes published in the dataplane devices. This design allows a clear separation between the routing algorithm and the southbound protocol, making the solution easier to adapt or evolve in the future.

3.5 Implementation

This section describes the implementation details of each module and submodule of our solution for resilient Smart Grid communications.

3.5.1 Switch Latency

The typical way to measure network latency is through the use of the ICMP protocol. More specifically, the sender would measure the time to transmit an ICMP Request and then receive the ICMP Reply returned by the receiver (using the ping tool). This period corresponds to the Round Trip Time (RTT), which divided by two gives a rough estimate of the current link latency. In SDN, a similar result can be attained by means of $Packet\text{-}Out$ messages. These messages are employed by the controller to send packets through the data plane to a destination output port. If such message is set with output port equal to $OFPort.CONTROLLER$ then the dataplane device will transmit that packet back to the controller, mimicking the behaviour of the ICMP Request/Reply messages.

The latency result from this technique includes the latency between the controller and each switch. To obtain the latency between switches, we need to measure the latency between the controller and each SDN switch to remove this latter component. For this purpose, this submodule periodically sends a Packet-Out message containing a sequential number to each switch, as depicted in Figure 3.6, and saving the time when this message was sent. When the message arrives at the dataplane device it will be returned to the controller (the controller sets it to be sent to $OFPort.CONTROLLER$).

When the packet is received, the controller will compare the timestamp of the received packet to the timestamp of when that specific $Packet\text{-}Out$ message was sent. By subtracting the first from the last, and divide the difference by two, it is possible to obtain the estimate of the one-way latency in the communication between the controller and that dataplane device. This process can be viewed on Figure 3.6 in which:

1. The controller sends M_1 to the dataplane device and saves a timestamp $t1$;
2. M_1 takes λ_1 time to reach the dataplane device;
3. When M_1 arrives at the dataplane device, it is processed and the packet M_{1R} is returned to the controller through the port $OFPort.CONTROLLER$;
4. M_{1R} takes λ_2 time to reach the controller;
5. The controller saves the timestamp $t2$ of the M_{1R} packet

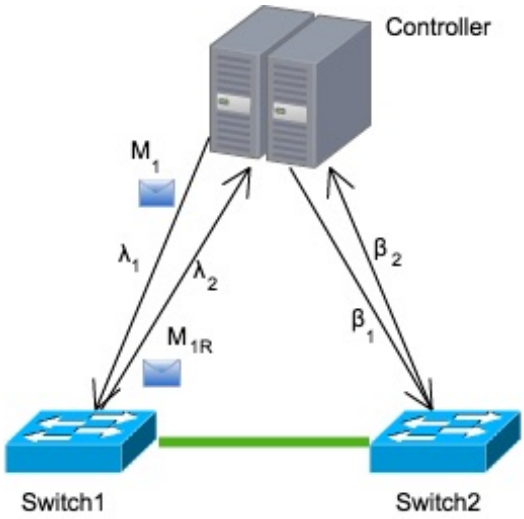

Figure 3.6: Method to Measure the Latency.

6. The controller performs a lookup on a list for that dataplane device, searching for the sequencial number included in M_{1R};
7. The latency is then calculated by performing (t_2 - t_1)/2

Security considerations

The link connecting the controller to the dataplane device is assumed correct in our threat model. Nevertheless, the fact that a TLS [24] channel is used to protect the communication, guarantees that an attacker is unable to spoof packets. The attacker is also prevented from antecipating probe packets because they are not sent at a constant frequency. Another possibility for the attacker would be to identify the probe packets from their size, but the padding added by the TLS tunnel solves this problem.

3.5.2 Throughput

The throughput submodule periodically requests traffic statistics from all dataplane devices. These statistics include the amount of received/transmitted traffic on any given port

since the last time that port became active. By comparing the results of two consecutive requests to the same port, one can get the throughput estimate for that period. To detect throughput inflation or throughput deflation attacks, the calculated throughput on each source switch port is compared with the corresponding destination switch port of every link, and if it exceeds a pre-defined value then that link is double checked with an additional measurement. This extra check serves the purpose of minimizing the false positive rate. The following points elaborate on the finer details of the throughput calculation and discrepancy checking.

Estimating data throughput

The throughput submodule sends an Openflow *port-statistics* request and records the information on every port of every datapath device. Each port-statistics reply includes the duration of the port activity and the number of bytes received/transmitted since the last time the port became active. Using the difference between the number of bytes received/transmitted in two consecutive statistics requests, the submodule obtains the amount of bytes received/transmitted by the measured port in the time interval between the two requests. The submodule then calculates the time interval from the difference between the port activity durations of both requests[1]. Finally, we estimate the reception/transmission throughput by dividing the amount of bytes by the interval.

Checking for discrepancies in data throughput

For each datapath link it is expected, in ideal conditions, for its exit throughput τ_d to (approximately) match its entry throughput τ_s whenever both values are estimated closely in time. However, if a significant portion of the traffic traversing a link is dropped, or if a significant amount of foreign traffic is injected into the link, then τ_s and τ_d should differ by a larger than usual margin.

The submodule detects a throughput discrepancy if $(|\tau_s - \tau_d|/\tau_s) * 100 > \tau_x$, where τ_x is a configurable value that represents the percentage difference having in consideration the total throughput measured at the source switch.

If a discrepancy is detected, the submodule performs a rapid double check on both entry and exit throughput values of the link by immediately sending a port-statistics request to each link's datapath, and recalculating both throughput values upon receiving the replies. If there is no discrepancy in the newly calculated values, then our application signals a mild traffic interference event (e.g., caused by a sudden spike in user traffic throughput). However, if the discrepancy persists in the second calculation, then our application signals a stronger traffic interference event (e.g., eventually caused by an attack).

[1] We use the time difference between the activity durations announced by the datapath device, instead of the time difference between the arrival instants of the replies, since the activity durations are set closer in time to the update of traffic counters. This enables more accurate results.

Security considerations

By using the technique described in the previous sections, the submodule is unable to identify attacks on the datapath link in which an adversary with packet interception capabilities affects the user traffic without changing its perceived throughput. By simply modifying packets, without dropping them or adding additional ones, the adversary can potentially fool the aforementioned check for throughput discrepancy. Such an attack cannot be reliably detected by the throughput submodule since its function does not extend to packet content verification.

3.5.3 Sampling

To extend the capabilities of the throughput submodule to enable the detection of the above attacks, and also to measure both packet delay and loss, we employ our version of trajectory sampling. As explained, this submodule performs, at random periods (within specified limits), trajectory sampling to a group of active (unidirectional) links chosen randomly. The same set of packets have to be sampled (mirrored) in both the source and the destination dataplane devices, requiring the controller to add rules to both switches, taking in consideration the link latency to connect to each switch, but also the delay of the sampled link. By sampling the same packets we are able to determine if any packet was changed, dropped, or inserted in the link, allowing the detection of ongoing attacks. This submodule has some configuration parameters that should be set to reduce the overhead and optimize performance. Table 3.1 contains a small description of each parameter.

Sampling on Openflow switches

According to the Openflow specification, a flow entry contains a set of instructions (or "actions") that are executed when a packet matches that entry. Examples include sending the packet to a specific output port ($output$ action), sending the packet to the controller ($controller$ action), sending the packet to another switch table ($Goto - Table$), among others.

To perform sampling without adding latency to existing communications, the packet needs to be duplicated, with one copy sent to the controller and the other following the normal path to its destination. Since a flow entry only supports a set of instructions, we cannot add more than one output instructions (one to put the packet in the normal path, and the other to send it to the controller). To circumvent this problem we added one instruction to output one copy to the controller, and one $Goto - table$ instruction to force the packet to be processed in a second flow table. This second flow table contains the flow entries added by the other SDN applications that will transmit this copy on the normal path to its destination.

This technique allows the duplication of a packet, ensuring that a copy is forwarded to the

Parameter	Value
numberOfSamplesPerCycle	Number of unidirectional links that should be sampled in each sampling cycle
cycleTimeInMiliSeconds	Average sampling cycle period in milliseconds
maxSamplingDelayInMiliSeconds	Maximum delay to start the sampling process on the chosen unidirectional link in milliseconds
hashAlgorithm	Hashing algorithm used to summarize packets
maxSamplingPeriodInMiliSeconds	Maximum sampling period in milliseconds
defaultSamplingPeriodMiliSeconds	Target average sampling period in milliseconds
defaultLatencyInMiliSeconds	Default latency for a link with unknown latency in milliseconds
defaultCatchWindowInMiliSeconds	Default duration for the catch window in milliseconds
defaultCatchWindowOverheadPercentage	Target overhead percentage of the catch window
catchWindowIncrementMiliSeconds	Number of milliseconds that the catch window should be increased when at least one of the samples was not received in the destination switch
catchWindowDecrementMiliSeconds	Number of milliseconds that the catch windows should be decremented when the overhead at the destination switch is superior to the value set in $defaultCatchWindowOverheadPercentage$

Table 3.1: Sampling configuration parameters description.

controller while the original packet continues through its normal path.

Openflow flow tables management

To perform sampling, our submodule has to control the insertion, deletion and modification of all entries in the switches' forwarding tables. Otherwise, the sampling process could be compromised, if other SDN applications inserted flow entries that would overlap with the ones from the sampling submodule.

To cope with this issue while being transparent to SDN applications, we have developed a proxy submodule that intercepts all updates to the flow tables. Specifically, we proxy the $writeMessage$ function, and are thus able to reserve the first and last flow tables to the sampling submodule, in a transparent way to the applications. These two flow tables are essential to perform sampling on the source and destination switches. We employ the last table on the source switch to send packets being sampled to the controller.

Then, in the destination switch, the first table is used to send a copy of the sampled packets to the controller.

The algorithm for sampling in the source switch is thus as follows:

1. A new packet arrives at the source switch;
2. The packet is processed by the flow entries of the other SDN applications following the logic set by them;
3. Since any $output$ instruction was replaced by the same $output$ instruction and an additional $Goto\text{-}Table$ instruction, the packet will be sent to the normal output port and a copy of it will be forwarded to the last table, with additional metadata information containing the port through which the packet was transmitted;
4. If a sampling rule on the last table matches the metadata information, then the packet is copied to the controller.

The algorithm for sampling at the destination switch is as follows:

1. A new packet arrives at the destination switch;
2. The switch will look for a matching flow entry on the first table, which is reserved for sampling;
3. If a sampling flow entry is found, a copy of the packet is performed and forwarded to the controller, while the other copy is sent to the second flow table to be processed by the flow entries of the other SDN Applications. If the packet does not match, then the default flow entry is applied sending the packet to the second table.

Sampling the same traffic on both switches

The goal of trajectory sampling is to enable the same set of traffic to be sampled in both switches. In our case, as we want to use only normal Openflow switches, we cannot resort to hashing techniques for this purpose, as is common. The algorithm we propose tries to sample the same (small) window of packets from the switches where sampling is occurring. The window has to be very small, as otherwise the switch CPU would not be able to cope with the rate at which packets are received (that is why sampling is needed in the first place). The challenge is therefore one of timing: when to instruct the various switches to start and stop sampling. Our solution takes into consideration the measurements made by the monitoring module, namely, the latency between the switches being sampled and the controller, and also the latency of the link itself. By using this information it is possible to calculate the moment when the controller should send the

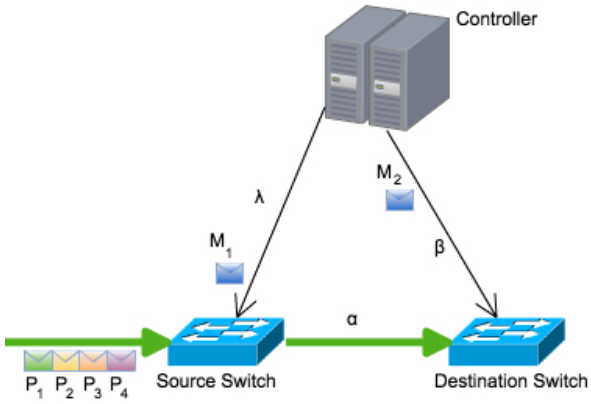

Figure 3.7: Sampling example

messages to add and remove sampling flow entries to each switch, so that a similar slice of traffic is mirrored to the controller.

One challenge to the effectiveness of this scheme is network jitter (inter-packet delays). If the inter-packet delays in the links being monitored are different, the result is that some (or all) of the packets sampled at the source switch may not to be sampled at the destination switch. See Figure 3.7 for an example. A network with high jitter may cause the timing of M_1 and M_2 to change, causing P_1, P_2, P_3 and P_4 to be sampled at the source switch but not at the destination switch.

To address this challenge we have added a $safety\_window$. This safety window is an extra period of time in which the sampling flow entries are active at the destination switch. So, if the algorithm calculates that the rules in the destination switch should be applied between the instants t_1 and t_2, with the $safety\_window$ the flow entries will be applied between the instants $t_1 - (safety\_window/2)$ and t_2 + ($safety\_window/2$) making the value of the actual sampling period equal to t_2 - t_1 + $safety\_window$.
Since jitter may vary, the safety window is dynamic, decreasing when all sampled packets at the source switch were also sampled at the destination switch, and increasing otherwise, up to a maximum value that is set in the configuration file. This limit is a security measure because an attacker could perform a Denial of Service attack (especially in links with high throughput) by forcing an increase in the $safety\_window$, causing a high volume of traffic to be forwarded to the controller.

To calculate the moments t_1 and t_2 when the controller should send flow entry messages M_1 and M_2 (as per Figure 3.7 above) we use Algorithm 1.

To calculate when the sampling flow entries should be removed, we use a similar algorithm. The main difference is that instead of sending the delete flow entries messages M_1' and M_2' in a way that M_2' is applied $(safety\_window/2)$ seconds before the slice of traffic to be sampled arrives at the destination switch, we make it arrive at the destina-

$diff = \lambda + \alpha - (\beta + (safety\_window/2))$;
if $diff > 0$ **then**
  $t_1 = 0$;
  $t_2 = diff$;
else
  $t_1 = (\beta + (safety\_window/2)) - (\lambda + \alpha)$;
  $t_2 = 0$;
end

Algorithm 1: Add sampling flow entries algorithm

tion switch $(safety\_window/2)$ after the slice of traffic to be sampled has traversed the switch. We present Algorithm 2 next.

$diff = \lambda + \alpha + (safety\_window/2) - \beta$;
if $diff > 0$ **then**
  $t_1 = 0$;
  $t_2 = diff$;
else
  $t_1 = (\beta - \lambda + \alpha + (safety\_window/2))$;
  $t_2 = 0$;
end

Algorithm 2: Delete sampling flow entries algorithm

Sampling a link with unknown latency

In the previous section we have detailed how the sampling submodule calculates the instants of time when it should send the add/delete flow entry messages. At is base, it uses the latency between switches and the controller and the link latency between switches. Since this last parameter is obtained through the trajectory sampling process itself, when the system starts this value is not available. To overcome this issue, we set a configurable predefined value. In the beginning, there is a high probability of this value not being accurate. This is not a problem as the $safety\_window$ will increase automatically, and that eventually makes the same set of packets start being sampled in both switches. When this happens, the monitoring submodule is able to calculate the latency on the sampled link, and use that latency in the sampling procedure.

Sampling process

At this point we are able to obtain the same slice of sampled packets on both switches. When each Openflow message that carries a packet arrives at the controller our submodule is notified, and a hash of the packet, including all headers and the payload, is obtained using the SHA-256 algorithm. The hash of each packet is then kept in a list with the

corresponding timestamp. When the sampling period ends the list of hashes from packets received from the source switch is compared with the one from the destination switch, and the controller updates the information about each link according to Algorithm 3:

for $hash$ *in* $source\_switch\_messages$ **do**
  if $hash$ *exists in* $destination\_switch\_messages$ **then**
    Add $latency$ to the $link\_information$;
    Increment $correct\_samples\_count$;
    if *count(*$hash$*)* $>$ *1* **then**
      Decrease by one the count for that $hash$ in $destination\_switch\_messages$;
    else
      Remove $hash$ from $destination\_switch\_message$;
    end
  else
    Increment $missing\_samples\_count$;
  end
end
for $hash$ *in* $destination\_switch\_messages$ **do**
  if $hash$ *exists in* $source\_switch\_messages$ **then**
    Save packet as duplicate
  else
    Save packet as foreign
  end
end
$error\_rate$ = ($missing\_samples\_count$ / $count(source\_switch\_messages)$)$*100$

Algorithm 3: Compare sampling messages

Where:

- **latency** is the time it took for the packet to arrive at the destination switch;
- **correct_samples_count** is the number of packets that were sampled at both source and destination switches;
- **missing_samples_count** is the number of packets that were sampled at the source switch but not at the destination switch;
- **foreign** is the number of packets that were sampled at the destination switch that do not match any of the packets sampled at the source switch;
- **duplicate** is the number of duplicate packets that were sampled at the destination switch.

This information is then used in order to set a $LinkSuspicionLevel$, which is then made available through the monitoring API.

3.5.4 Network Graph

The network graph submodule represents the network in a graph form. It supports different weights for physical links and tunnels, and provides path calculation functions. This submodule contains a Dijkstra algorithm implementation allowing any other module to find the K-shortest paths between two given nodes. This is then used by the Resilient Routing submodule to obtain the shortest paths between two network devices.

3.5.5 Resilient Routing

The resilient routing submodule is the core of the routing and traffic engineering module. It listens for events provided by the monitoring module and other network events, such as link up/down events and updates in the network graph.

Routes consist of an ordered list of graph nodes that are interconnected through a link. This submodule maintains information on the $number\ of\ hops$, $latency$, $error\ rate$, $minimum\ available\ bandwidth$ and $route\ suspicion\ level$.

When this submodule receives an update on a given link, it will retrieve information to understand if that link is being used in any route, and update the information on those routes. After updating the route information, it also checks if any parameter is not compliant with the $smart\ grid\ application$ that is flowing through that route. If it is not, then it will search for a new route that meets the $smart\ grid\ application$ requirements. If such route exists, the new route will be applied, from the last switch to the first switch (by applying the flow entries in this way we reduce packet loss during the convergence period).

3.5.6 Forwarding

When the resilient routing submodule creates the best routes, the forwarding submodule uses this information and converts them to flow entries, which are then applied to the switches.

3.6 Summary

In this section we have described the design and implementation of our solution. The different components of the monitoring module complement each other and fill the previously identified security gaps. The routing module uses that information to chose among the existing possibilities the shortest path that is compliant with the specified application's restrictions. Additionally by only using functionalities defined in the Openflow specification our solution can be used with standard Openflow compliant SDN switches.

Chapter 4

Evaluation

The main goal of our evaluation is to answer three questions. First, how fast can the monitoring application get to a stable state? By stable we mean the amount of time required for the monitoring algorithm to adapt to the current network conditions and correctly perform trajectory sampling on a given link. Second, how long does it take for the system to detect that there is an ongoing attack? Finally, how quick is the solution in re-routing once an attack has been detected?

4.1 Test Environment

In order to generate traffic we have used iperf 2.0.5. We have set iperf to generate UDP traffic at varying speeds, 1Mbps, 10Mbps and 100Mbps for the evaluation. The topology used is depicted in Figure 4.1, and it is a small scale representation of a real topology used by power grid operators. This allows us to evaluate both monitoring and resilient routing solution in a similar setting to where they will be used in practice, given our resource constraints.

The test machine is a MacBook Pro with one quad-core 2.8Ghz Intel Core i7 and 16Gb of RAM. The controller is Floodlight version 1.1 and the network is emulated using mininet 2.2.1 (described in Section 2.5), with each of them running separate virtual machines, each with 2 cores and 4Gb of RAM. We have guaranteed in all experiments that the CPU and memory resources were not the bottleneck, in order for the emulation to present results with the required fidelity. Virtual machines were interconnected through a virtual interface.
The software environment for the controller virtual machine was Ubuntu 16.04.3 LTS with Java(TM) SE Runtime Environment 1.8.0_141 64bits. The Mininet virtual machine was running Ubuntu 16.04.1 LTS.

The algorithms were adjusted with the values described in Table 4.1. For our evaluation we have chosen SHA-256 since it is, nowadays, considered a secure hashing al-

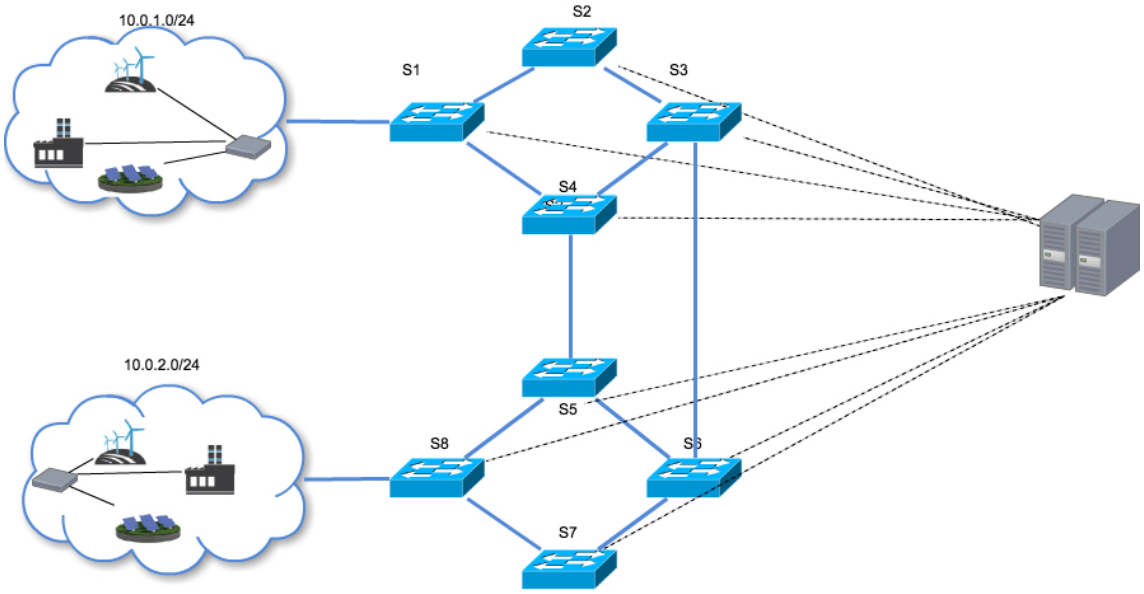

Figure 4.1: Evaluation network

gorithm. The $maxRouteLatencyMiliseconds$, $maxRouteErrorRate$ and $minimum AvailableBandwidthBits$ parameters were defined based on the requirements of the SEGRID project[1]. The $numberOfSamplesPerCycle$, $cycleTimeInMiliSeconds$, $max SamplingPeriodInMiliSeconds$ parameters control the number of packets that are sampled and therefore have a direct impact on the performance requirements of the solution. For example, by increasing $numberOfSamplesPerCycle$ and decreasing $cycleTimeIn MiliSeconds$ one is effectively increasing the number of sampled packets across the network and therefore the probability of detecting an ongoing attack at the expense of performance. In the same way, increasing the value of $maxSamplingPeriodInMiliSeconds$ will lead to a faster sampling configuration time. The values we chose for these parameters reflect the best values for our test environment, considering the performance of the hardware used. The parameters $defaultCatchWindowOverheadPercentage$, $defaultLatencyInMiliSeconds$, $defaultLatencyInMiliSeconds$, $defaultCatchWin dowInMiliSeconds$ and $defaultSamplingPeriodMiliSeconds$ are used as a starting point and are adjusted by the monitoring application. These are the configurations that cause the least impact on the operation of the application and the chosen values should suit most environments.

[1]This work was funded by the SEGRID project

Parameter	Value
numberOfSamplesPerCycle	10
cycleTimeInMiliSeconds	1000
maxSamplingDelayInMiliSeconds	100
hashAlgorithm	SHA-256
maxSamplingPeriodInMiliSeconds	1000
defaultSamplingPeriodMiliSeconds	30
defaultLatencyInMiliSeconds	1
defaultCatchWindowInMiliSeconds	30
defaultCatchWindowOverheadPercentage	50
catchWindowIncrementMiliSeconds	10
catchWindowDecrementMiliSeconds	10
maxRouteLatencyMiliseconds	100
maxRouteErrorRate	5
minimumAvailableBandwidthBits	1048576

Table 4.1: Configuration parameters value used in evaluation

4.2 Monitoring

In order to evaluate the monitoring algorithm it is important to measure both its accuracy and its efficiency.
We consider that it has adapted to the network conditions when all the packets that were sampled at the source switch of a link, are also sampled at the destination switch. This will allow us to determine how long it takes for the system to reach operating conditions. These metrics can be obtained by measuring, respectively, the:

- **Number of correct samples** and the **Number of missing samples** - The number of correct samples is the number of packets that were sampled both *at the* source and the destination switches of the unidirectional link being sampled, whereas the number of missing samples is the difference between the number of samples obtained at the source switch and the correct samples. These values allow us to verify the accuracy of our monitoring solution.

- **Difference of the number of sampled packets at the source and destination switches** - The difference in the number of sampled packets in the source and destination switches of the unidirectional link. This metric will allow us to measure the overhead of our monitoring solution.

In the rest of this chapter we will use these metrics to evaluate the accuracy and efficiency of the monitoring solution.

4.2.1 Monitoring startup time

The monitoring startup time is defined as the time elapsed between the first sample and the sample in which there are no missing packets. In other words, the elapsed time between the first sample and the solution reaching a stable state. When the monitor starts it does not have an actual latency value, and therefore the value of the variable $defaultLatencyInMiliSeconds$ is used for the initial calculations.

In Figure 4.2, Figure 4.3 and Figure 4.4, we show, from the moment the monitoring solution starts, namely the number of packets received, missing packets, and correct packets, for throughputs of 1Mbps, 10Mbps, and 100Mbps, respectively. The number of packets received represents the number of packets that were sampled at the destination switch during that specific sampling cycle. By comparing this value with the number of correct packets (which represent the number of matching packets that were sampled at both source and destination switches), one can obtain the overhead (in number of packets) that the solution is generating. If these two values are equal then the overhead is zero. The number of missing packets represent the packets that were sampled at the source switch but did not have a matching sample at the destination switch. This can occur in the event of packets being changed, dropped or delayed in transit, or due to the the solution not having adapted to the current network conditions. The results show that the solution is able to reach a stable state in a very short time interval. The results also show that the solution maintains a average overhead close to the one defined in the configuration parameters.

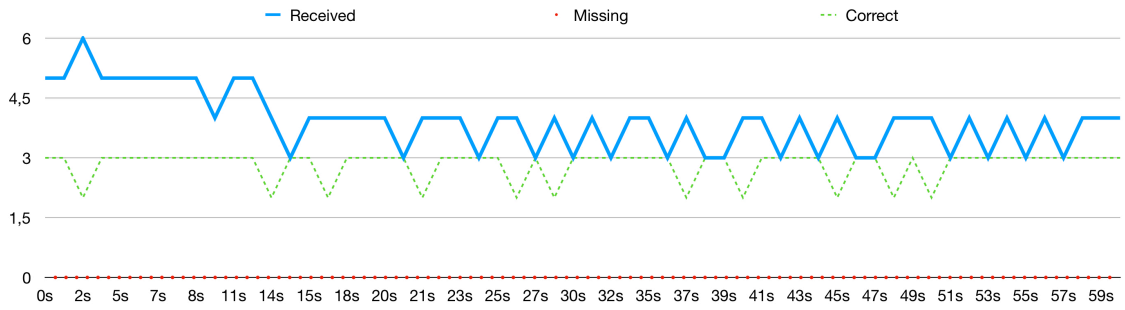

Figure 4.2: Monitoring startup time at 1Mbps

Similarly, Figure 4.5, Figure 4.6, and Figure 4.7 show how the catch window, used to cope with the diferences between the link latency and the latency information currently known by the controller, evolves throughout the first 60 seconds.
In these figures one can observe that the module is optimizing the catch window for the monitored link. This optimization occurs at second 3, and again at second 14, for the 1Mbps evaluation, and is due to the overhead on the destination switch to have exceeded

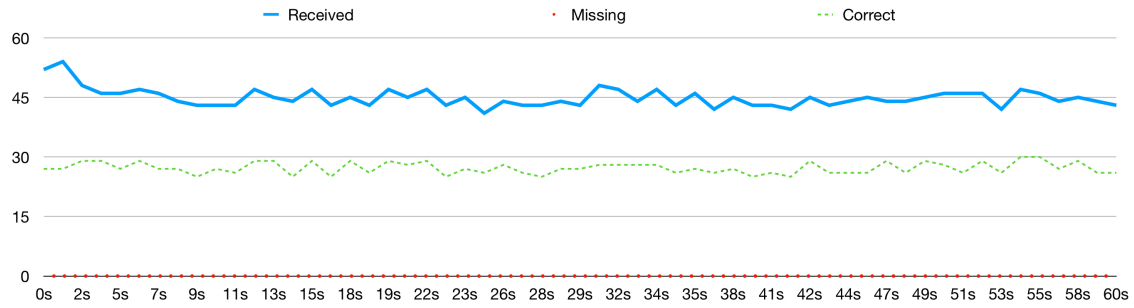

Figure 4.3: Monitoring startup time at 10Mbps

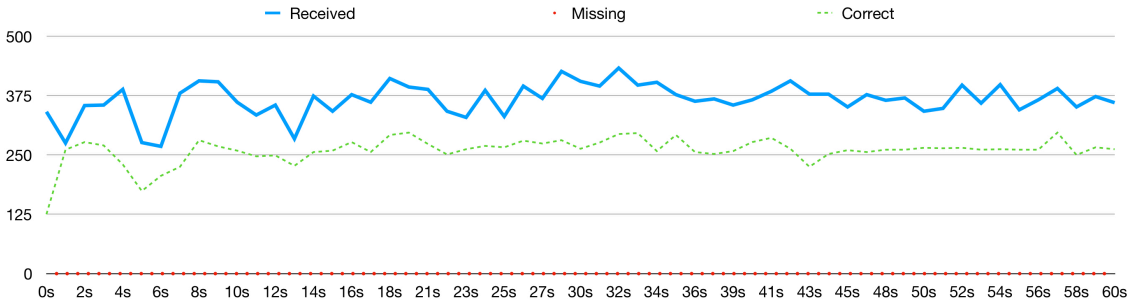

Figure 4.4: Monitoring startup time at 100Mbps

the configured $defaultCatchWindow$- $OverheadPercentage$. (in this case, it reached 200% and 100%, respectively, which is higher then the configured 50%). After this optimization it is clear that the number of received packets is closer to the number of correct packets, thus the overhead is smaller.

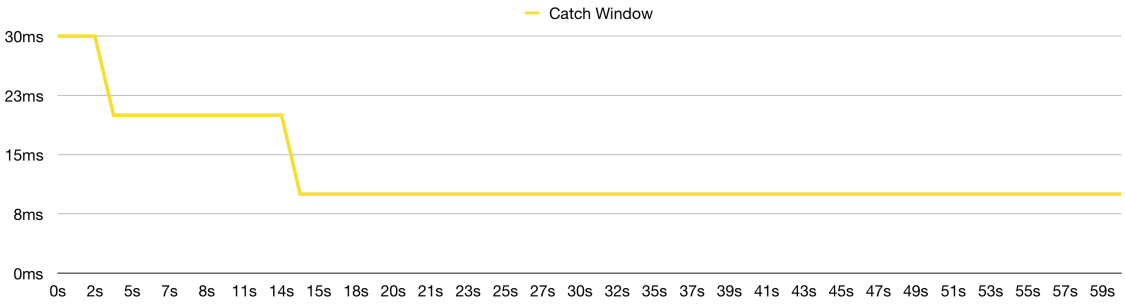

Figure 4.5: Monitoring startup catch window at 1Mbps

At 10Mbps and 100Mbps we obtain similar behaviours, as Figure 4.6 and 4.7 demonstrate.

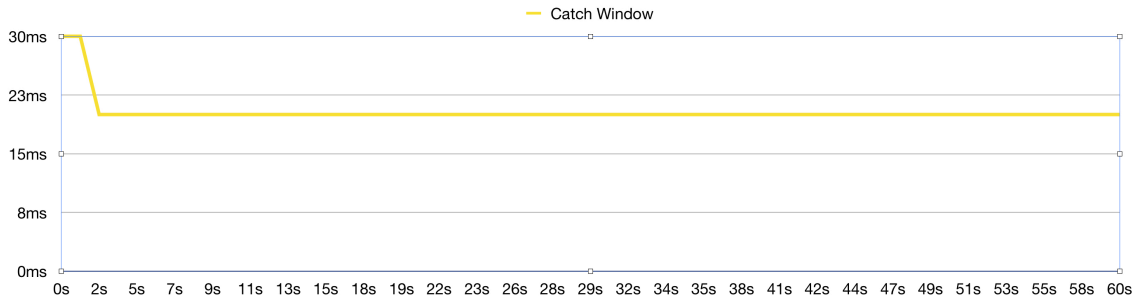

Figure 4.6: Monitoring startup catch window at 10Mbps

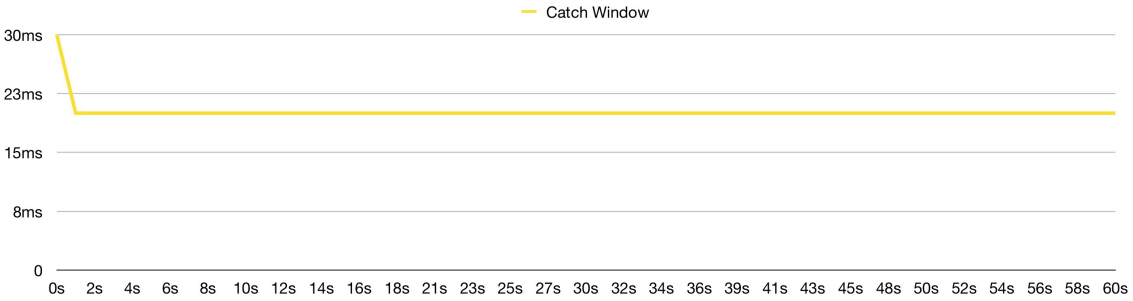

Figure 4.7: Monitoring startup catch window at 100Mbps

4.2.2 Failure detection time

The second metric we consider in our analysis is the amount of time our solution takes to detect a failure. A failure can be either malicious, as when an attacker influences the flow of packets through the network, or accidental, such as when a link goes down due to some hardware failure. Our solution abstracts this diference, since in either case the impact is the same. The time to detect such failures is directly related to the number of samples per second that the module was configured to perform on the network. Since the links being sampled are selected at random, and the instant in which they are sampled is also random, the detection time varies. Moreover, since the module is configurable, if required one can reduce the cycle time, and consequently reduce the time of detection, at the expense of performance.

For this evaluation we considered an attacker that has control of a network link, and that starts to drop 5% of the packets in that link. We have set the maximum acceptable error rate for any link to be 5% (emulating an application that has an error rate limit of 5%) not tolerating higher drop rates. We have run 15 iterations at 10Mbps and 100Mbps, allowing for the link to reach a stable state before introducing the failures. We record the instant the attacker starts dropping the packets, the instant when the module detects the failure

and also the number of samples until the detection. Figure 4.8 shows that at 10Mbps the module needs, in average, 3 seconds to detect the failure, with a standard deviation of around 2 seconds.

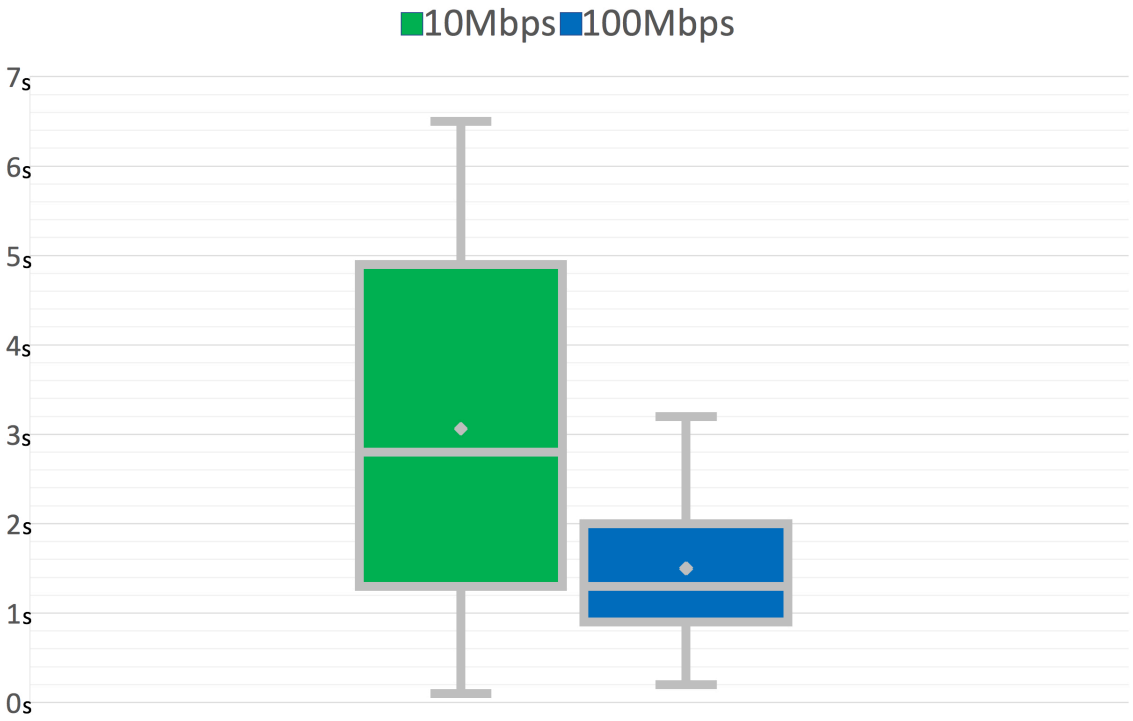

Figure 4.8: Experiment until detection at 10Mbps and 100Mbps

At 100Mbps the module is able to detect the attack faster, with an average of 1,5 seconds to detect the failure, and a standard deviation of 1 second.

The higher number of packets that flow in a 100Mbps link results in a higher number of packets dropped, when the link has the same drop rate of 5%. Such a higher number of dropped packets accelerates the process of detection of failures. Increasing the sampling duration in links that are experiencing less throughput will improve the effectiveness of the algorithm at detecting failures. This value may be considered high for some critical Smart Grid applications, and therefore should be addressed in future work.

4.3 Routing

Once a failure has been detected by the monitoring module it is important to recalculate a new valid route (if one exists) as fast as possible. This will minimize the impact of the failure, which is specially important in critical services. To evaluate our re-routing algorithm, we measure the time elapsed from detection of the failure up to the time the rules are pushed to the dataplane devices.

Using the evaluation network described previously, we have measured the recovery time from a failure we ran each experiment 20 times. Figure 4.9 shows the average re-route time.

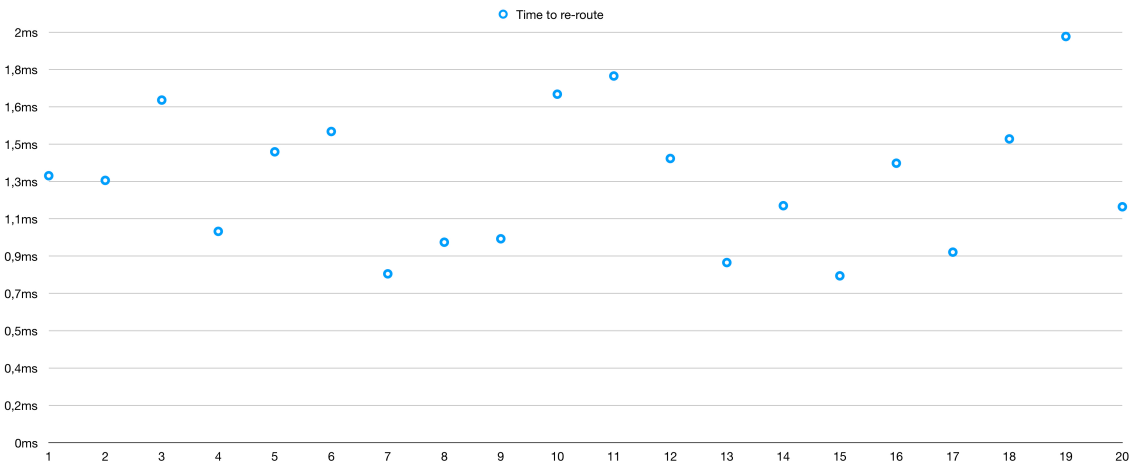

Figure 4.9: Time to re-route on failure

As we can verify, the solution takes an average 1,3ms (with a maximum of 2ms) to re-route traffic, in the topology present in Figure 4.1.
Since the number of switches managed by the controller directly impacts the calculations necessary to obtain a route, and therefore its performance, we also evaluate how the routing algorithm scales. For this part of the evaluation we automated the process of creating ring type networks on mininet and in each iteration we added a new ring composed of 5 switches, creating a chain like network. We recall that we have chosen to use this topology since it is commonly used in Smart Grids. Figure 4.10 shows the increase in recovery time from a failure, as more dataplane devices are added to the network. By increasing

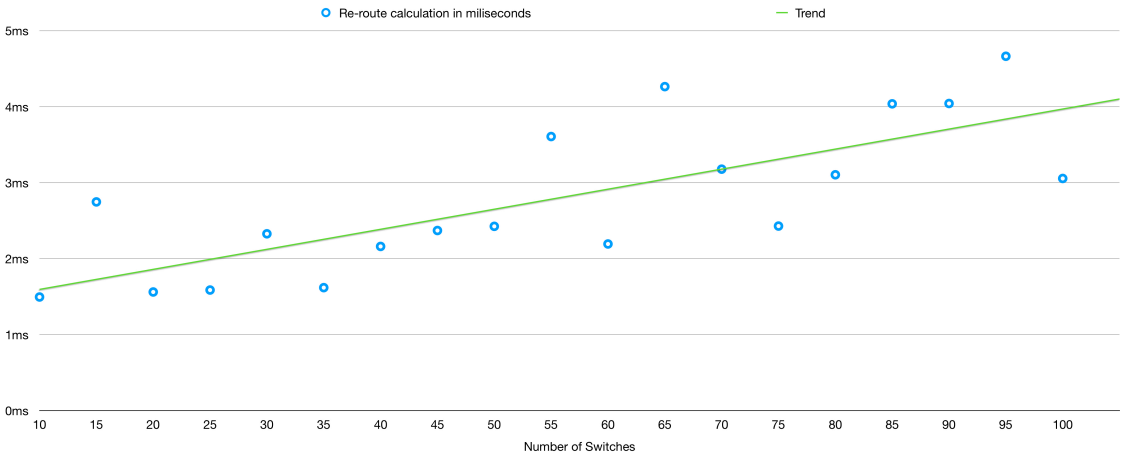

Figure 4.10: Reroute time after a failure on networks with different number of switches.

the number of switches in the network from 10 to 100, there is an increase in the recovery time, from around 1,5ms to around 4,3ms. Still, this is within the limits of the most time sensitive Smart Grid applications.

4.4 Summary

The evaluation shows that our solution has the ability to quickly adapt to network conditions specially in links with throughput higher then 10Mbps. Error detection also benefits from a higher throughput as Figure 4.8 shows. The slower failure detection time in links with lower throughput is a point of improvement that should be considered in future work.

Chapter 5

Conclusions

In this work we have proposed a solution for resilient communications in Smart Grids. Our approach is composed of two major components: a secure monitoring module and a resilient routing module. The monitoring module is a SDN application that periodically probes each switch of the network to update its current view. The main conceptual contribution is an algorithm that takes advantage of Openflow functionalities to obtain information on ongoing attacks to the monitoring system. The solution includes adaptable trajectory sampling, cryptographically secure verification of the sampled packets, and employs randomness on critical actions, resulting in a secure base for the resilient routing module. The solution was designed as a standalone component making it possible to be used with other routing/traffic engineering modules, tailored for their specific requirements. Importantly the monitoring algorithm was developed in compliance with the Openflow specification, and requires no special hardware features. The second component is a resilient routing module, that aims to minimize the recovery time from failures by creating and installing backup routes in network switches.

We have evaluated our solution by means of network evaluation, using mininet, assuming real Smart Grid topologies. The main conclusions are that SDN is suited for being used in Smart Grids, it is possible to use SDN to perform secure monitoring using common SDN switches, and that having a centralized view of the network is beneficial in latency sensitive networks.

Bibliography

[1] http://www.pennenergy.com/articles/pennenergy/2015/05/smart-grids-and-energy-storage-the-road-to-dynamic-grids.html. [Visited 14 July 2016].

[2] https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-59/161-sdn.html. [Visited 14 July 2016].

[3] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf. [Visited 14 July 2016].

[4] https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf. [Visited 14 July 2016].

[5] https://github.com/noxrepo/nox. [Visited 14 July 2016].

[6] https://www.opendaylight.org/. [Visited 14 July 2016].

[7] https://onosproject.org/. [Visited 14 July 2016].

[8] http://mininet.org/. [Visited 14 July 2016].

[9] https://scapy.net/. [Visited 14 July 2016].

[10] https://iperf.fr/. [Visited 14 July 2016].

[11] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview. [Visited 14 July 2016].

[12] D. Oran, OSI IS-IS Intra-domain Routing Protocol, RFC 1142, February 1990.

[13] J. Case, M. Fedor, M. Schoffstall and J. Davin, A Simple Network Management Protocol (SNMP), RFC 1157, May 1990.

[14] J. Moy, OSPF Version 2, RFC 2328, April 1998.

[15] D. Awduche, J. Malcolm, J. Agogbua, M. Dell and J. McManus, Requirements for Traffic Engineering Over MPLS, RFC 2702, September 1999.

[16] C. Hopps, Analysis of an Equal-Cost Multi-Path Algorithm, RFC 2992, November 2000.

[17] D. Thaler and C. Hopps, Multipath Issues in Unicast and Multicast Next-Hop Selection, RFC 2991, November 2000.

[18] D. Awduche, A. Hannan and X. Xiao, Applicability Statement for Extensions to RSVP for LSP-Tunnels, RFC 3210, December 2001.

[19] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan and G. Swallow, RSVP-TE: Extensions to RSVP for LSP Tunnels, RFC 3209, December 2001.

[20] E. Rosen, A. Viswanathan and R. Callon, Multiprotocol Label Switching Architecture, RFC 3031, January 2001.

[21] P. Phaal, S. Panchen and N. McKee, Cisco Systems NetFlow Services Export Version 9, RFC 3954, October 2004.

[22] P. Phaal, S. Panchen and N. McKee, InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, RFC 3176, September 2004.

[23] D. Oran, A Border Gateway Protocol 4 (BGP-4), RFC 4271, January 2006.

[24] T. Dierks and E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246, August 2008.

[25] N. Adrichem, C. Doerr, and F. Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *Proceedings of the Institute of Electrical and Electronics Engineers Network Operations and Management Symposium*, May 2014.

[26] J. Apisdorf, K. Claffy, K. Thompson, , and R. Wilder. Oc3mon: Flexible, affordable, high performance statistics collection. In *Proceedings of the Large Installation System Administration*, September 1996.

[27] S. Chowdhury, M. Bari, R. Ahmed, and R. Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *Proceedings of the Institute of Electrical and Electronics Engineers Network Operations and Management Symposium*, May 2014.

[28] S. Clements and H. Kirkham. Cyber-security considerations for the smart grid. In *Proceedings of the Institute of Electrical and Electronics Engineers Power & Energy Society General Meeting*, July 2010.

[29] M. Collins and M. Reiter. Finding peer-to-peer file-sharing using coarse network behaviors. In *Proceedings of the European Symposium on Research in Computer Security*, September 2006.

[30] V. Delgado-Gomes, J. Martins, C. Lima, and P. Borza. Smart grid security issues. In *Proceedings of the International Conference on Compatibility and Power Electronics*, September 2015.

[31] N. Duffield. Sampling for passive internet measurement: A review. In *Proceedings of the Statistical Science*, August 2004.

[32] N. Duffield, C. Lund, and M. Thorup. Charging from sampled network usage. In *Proceedings of the Association for Computing Machinery Special Interest Group on Data Communication*, November 2001.

[33] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. In *Proceedings of the Association for Computing Machinery Special Interest Group on Data Communication*, October 2004.

[34] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational ip networks: methodology and experience. In *Proceedings of the Institute of Electrical and Electronics Engineers Transactions on Networking*, June 2001.

[35] S. Goldberg and J. Rexford. Security vulnerabilities and solutions for packet sampling. In *Institute of Electrical and Electronics Engineers Sarnoff Symposium*, April 2007.

[36] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and Jennifer Rexford. Path-quality monitoring in the presence of adversaries: The secure sketch protocols. In *Proceedings of the Institute of Electrical and Electronics Engineers Transactions on Networking*, July 2014.

[37] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. In *Proceedings of the Association for Computing Machinery Special Interest Group on Data Communication*, October 2005.

[38] H. Hodzic and S. Zoric. Traffic engineering with constraint based routing in mpls networks. In *Proceedings of the Electronics in Marine*, January 2008.

[39] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the*

Association for Computing Machinery Special Interest Group on Data Communication, August 2013.

[40] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the Association for Computing Machinery Special Interest Group on Data Communication*, August 2013.

[41] D. Kreutz, F. Ramos, and P. Verissimo. Software-defined networking: A comprehensive survey. In *Proceedings of the Institute of Electrical and Electronics Engineers*, January 2015.

[42] J. Kurose and K. Ross. Computer networking: A top-down approach. Pearson, March 2012.

[43] A. Lakhina, M. Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of the Association for Computing Machinery Special Interest Group on Data Communication*, October 2004.

[44] B. Min and V. Varadharajan. Design and analysis of security attacks against critical smart grid infrastructures. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, October 2014.

[45] M. Moshref, M. Yu, R. Govindan, and Amin Vahdat. Dream: Dynamic resource allocation for software-defined measurement. In *Proceedings of the Association for Computing Machinery Special Interest Group on Data Communication*, August 2014.

[46] V. Sekar, M. Reiter, W. Willinger, H. Zhang, R. Kompella, and D. Andersen. Csamp: a system for network-wide flow monitoring. In *Proceedings of the USENIX Networked Systems Design and Implementation*, April 2008.

[47] V. Sekar, M. Reiter, W. Willinger, H. Zhang, R. Kompella, and D. Andersen. Csamp: a system for network-wide flow monitoring. In *Proceedings of the USENIX Networked Systems Design and Implementation*, April 2008.

[48] J. Suh, T. Kwon, C. Dixon, W. Felter, and J. Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *Proceedings of the International Conference on Distributed Computing Systems*, June 2014.

[49] C. Vineetha and C. Babu. Smart grid challenges, issues and solutions. In *Proceedings of the International Conference on Intelligent Green Building and Smart Grid*, April 2014.

[50] Y. Xie, V. Sekar, D. Maltz, M. Reiter, and Hui Zhang. Worm origin identification using random moonwalks. In *Proceedings of the Institute of Electrical and Electronics Engineers Security and Privacy*, May 2005.

[51] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. Madhyastha. Software-defined latency monitoring in data center networks. In *Proceedings of the Passive and Active Measurement*, March 2015.

[52] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. Madhyastha. Flowsense: monitoring network utilization with zero measurement cost. In *Proceedings of the Passive and Active Measurement*, March 2013.

[53] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proceedings of the USENIX Networked Systems Design and Implementation*, April 2013.