UNIVERSITY OF LISBON
Faculty of Sciences
Informatics Department

UNIVERSIDADE DE LISBOA

A SOFTWARE DEFINED NETWORKING ARCHITECTURE FOR SECURE ROUTING

Tiago André Raposo Posse

DISSERTATION

MASTERS DEGREE IN INFORMATION SECURITY

2014

UNIVERSITY OF LISBON
Faculty of Sciences
Informatics Department

A SOFTWARE DEFINED NETWORKING ARCHITECTURE FOR SECURE ROUTING

Tiago André Raposo Posse

DISSERTATION

MASTERS IN INFORMATION SECURITY

Dissertação orientada pelo Prof. Doutor Fernando Manuel Valente Ramos e pelo Prof. Doutor Christian Esteve Rothenberg

2014

Acknowledgments

Queria, em primeiro lugar, deixar os maiores agradecimentos à minha família e em especial aos meus Pais, por todo o apoio que me deram durante todo o meu percurso escolar e em especial durante o percurso académico. Por terem sido as pessoas que eu precisava sempre nos momentos certos e pela inegável influência que tiveram para a conclusão de mais esta etapa. Ao meu irmão Rui, pela ajuda em várias decisões críticas na minha vida e principalmente na preparação da minha profissional e, claro, às noites passadas no Bairro ou em casa seja de quem for. Ao meu irmão Carlos, por ser a pessoa que tira a monotonia da minha vida quando estou em casa.

À minha namorada por todas as vivências e o apoio incondicional durante a faculdade. Não existe mais ninguém como tu.

Aos meus amigos Henrique Vaz, Pedro Batista, João Feio, Ricardo Pires e Nani Gama por tempos indescritíveis, pelas saídas, as noitadas, as tardes na ponte e os jogos no C1. Mais importante, pela ajuda que me deram a tornar-me na pessoa que sou hoje e a ver a vida de uma perspetiva diferente, melhor.

Aos meus amigos Rambo, Marta e Mafalda, pela excentricidade que deram à minha vida e por serem pessoas com quem é fácil estar e passar horas na diversão. Força INEM!

Ao João Alves e à Vânia Castanheira. Vocês são uma força da natureza e exemplo de pessoas. Desejo-vos tantas felicidades como as que me proporcionaram.

Aos meus orientadores, Fernando Ramos e Christian Rothenberg, pela liberdade de ações mas prontidão em ajudar e orientação certa que me leveram à escrita deste trabalho de uma vida.

Ao Regivaldo Costa pela ajuda que deu na implementação e teste da aplicação BGP-Sec.

Por fim, à Praxe. Por tudo o que ela me ensinou e me proporcionou, às experiências que nunca conseguiria ter em qualquer outro cenário.

Um quarto de talento, três quartos de esforço!

Contents

List of Figures

List of Tables

Resumo

O tamanho e aceitação que a internet ganhou veio ajudar à inovação e à partilha entre utilizadores, mas em contrapartida aumentou o risco de tanto a infraestrutura da internet como as pessoas que a utilizam serem alvos de ciber-ataques. Esta é apenas uma visão parcial do problema, pois para suportar a crescente utilização da internet a infraestrutura cresceu sem a maturação de vários protocols e algoritmos que executam alguns dos servicos mais básicos com que convivemos todos os dias na internet.

Um dos melhores exemplos é o do Border Gateway Protocol, um protocolo de troca de informação de roteamento que está em uso há mais de 20 anos mas possui vários problemas de segurança conhecidos. O desenho inicial do protocolo, aliado à ineficiência das redes tradicionais impediram a adoção das várias adições de segurança jà propostas para o protocolo. O protocolo não possui atualizações de segurança que o protejam contra os vários tipos de ataques já descobertos, como *prefix hijacking*, *intercepção* e ataques no *plano de dados*. Estes ataques podem ter consequências graves durante períodos de tempo não negligenciáveis, como reportado em [33, 19].

As propostas já existentes, como o *S-BGP*[27], *soBGP*[48] e *Origin Authentication*[12], apesar de eficazes na proteção contra um ou mais ataques contra o BGP, não foram adoptadas na prática devido aos seus elevados requisitos computacionais ou de implementação. Neste trabalho resumimos os problemas para adopção de soluções de segurança em três pontos principais:

1. Algumas soluções requerem poder computacional ou capacidade de memória que nem todos os dispositivos de rede que correm BGP em funcionamento conseguem suportar;

2. A solução requer alterações ao protocolo BGP em funcionamento;

3. A solução não garante benefícios de segurança imediatos ao AS que a adoptar;

A investigação actual tem chegado à conclusão que muitos dos problemas das redes tradicionais surgem devido à necessidade de os dispositivos de rede participarem em protocolos complexos para executar funções de rede que vão além do seu objetivo: encaminhar pacotes [24]. Como consequência, as redes tornaram-se bastante complexas e portanto difíceis de gerir e escalar. A falta de segurança radica também neste problema.

Em alternativa às redes tradicionais, a comunidade científica e a indústria têm vindo a adoptar um novo tipo de redes, as *Software Defined Networks* (SDN). Estas redes sepathe datapathram o plano de controlo do plano de dados, passando toda a lógica e estado de rede para um controlador *logicamente centralizado*, mantendo nos dispositivos de rede apenas a tarefa de encaminhar pacotes. Os controladores SDN implementam funções de rede através de aplicações que executam no próprio ambiente do controlador em vez de obrigar os dispositivos de rede a implementarem esses protocolos. Um desses controladores é o OpenDaylight, que tem o apoio de alguns dos maiores nomes da indústria como a Cisco, IBM, HP e Juniper, e espera-se ser a principal referência no futuro.

Neste trabalho propomos duas aplicações SDNs para o controlador *OpenDaylight*: *RFProxy* e *BGPSec*. O RFProxy é um dos três componentes base da aplicação *RouteFlow*, uma plataforma de serviços de roteamento para SDN. O RFProxy é o único componente da aplicação a executar no controlador e é responsável por gerir e configurar os switches de acordo com as decisões tomadas pelo RFServer. Esta aplicação vem aumentar o número de opções para a utilização do RouteFlow e proporciona uma plataforma de roteamento avançada e eficiente para o OpenDaylight.

A aplicação BGPSec tem como objetivo garantir proteção contra ataques de *prefix hijacking*, onde um atacante tenta redireccionar todo o tráfego destinado a um AS para si. Esta proteção é conseguida através da validação dos dados recebidos do BGP. Ao utilizar uma aplicação para a validação dos anúncios BGP em vez de obrigar os dispositivos de rede a executarem este processamento, o desenho e implementação tornam-se mais simples e permitem um maior conjunto de opções quando comparado com as implementações necessárias em redes tradicionais. A utilização de uma aplicação SDN para este efeito é algo inovador e traz vantagens quando comparada com as redes tradicionais. Em particular, o ambiente SDN permite mitigar os dois primeiros problemas de adopção de uma extensão de segurança, ao passar o processamento para o controlador e a não requerer uma alteração ao protocolo BGP.

As contribuições principais deste trabalho podem ser resumidas da seguinte forma:

1. Implementação e avaliação de um serviço avançado de roteamento em ambiente SDN, nomeadamente ao controlador OpenDaylight;

2. Análise dos problemas de segurança do BGP e das extensões de segurança já propostas para redes tradicionais;

3. Desenho, implementação e avaliação de uma aplicação de segurança para o BGP baseada em SDN;

Abstract

The Internet has evolved from a small group of interconnected computers to an infrastructure that supports billions of devices including computers, smartphones, etc, all with increasing demands in terms of network requirements. The architecture of traditional networks hinders their capability of fulfilling these demands, mainly due to the tight coupling of the data and control planes. Network devices are required to handle and participate in complex distributed protocols to perform network tasks such as routing, making networks very complex and thus affecting their scalability, performance, management and innovation ease.

The Border Gateway Protocol, the *de facto* protocol for routing between Autonomous Systems (ASes) is one of the fundamental protocols for the operation of the internet. However, it was created in a time where the internet was composed of fewer ASes that trusted each other and in the information they provided, which is now unsafe to assume. The internet growth also resulted in an increase in the attacks against the internet routing infrastructure, and several misbehaviors have been detected, either due to attacks against the protocol or misconfiguration. Although several solutions have been presented to solve the security issues of BGP, no proposal has yet been adopted due to three main reasons:

- The solution requires either a computational power or memory size that not all currently deployed BGP speakers will be able to withstand;
- The solution incurs changes to the BGP protocol currently in use;
- The solution does not bring immediate security benefits for the adopting AS;

Software-Defined Networking (SDN) is an emerging network paradigm that aims to solve the problems of traditional networks by decoupling the data and control planes, moving the latter to a logically centralized controller while making network devices execute solely the former. All network tasks and applications run on top of the controller, which abstracts the network and greatly simplifies the development and testing of new applications and protocols. Forwarding rules are installed and removed using OpenFlow, a vendor-independent communications protocol for SDNs.

Several SDN controllers have been developed by different companies and researchers, several of them open-source. One of such kind is the OpenDaylight (ODL) controller,

supported by some of the top names in the IT industry (e.g. Cisco, IBM, HP). The goal of ODL is to create a controller of reference and help accelerate SDN evolution and adoption.

Although the controller is the core component of a SDN, network logic is performed by an application running on top of it. An example is RouteFlow, a routing platform that provides flexible and scalabe IP routing services to a SDN. Routing decisions are made by creating a virtual network that mimics the topology of the physical infrastructure and by analyzing the routing tables of the virtual devices. RouteFlow is composed by three components: RFClient, RFServer and RFProxy, with the latter running in the controller. The first contribution of this work is the implementation and evaluation of the RFProxy module for the OpenDaylight controller.

An SDN architecture provides a new environment to improve BGP security through the creation of an application to run on top of the controller. Such approach mitigates the first two adoption problems mentioned above by offloading the additional processing to the controller and by not requiring changes to the BGP protocol.

The other contribution of this work is the study and analysis of the BGP security problems and traditional solutions, and how to address them in a SDN environment. We implemented and evaluated BGPSec, a security application for the OpenDaylight controller that provides the network with protection against *prefix hijacking* attacks, where a malicious AS tries to direct the traffic destined to an AS onto itself.

Chapter 1

Introduction

1.1 Internet Structure

The Internet is composed of 47744 Autonomous Systems (ASes) [1], geographically distributed across the world. Each AS is represented by a unique numeric identifier (AS number) and is composed of one or more computer networks, all governed by the same administration and employing a single high-level routing policy.

Due to ASes being geographically distributed, the cost of directly connecting all ASes would be unfeasible and as such it becomes necessary that some ASes act as traffic relays, forwarding packets on behalf of their neighbors to other ASes. This requirement is represented in terms of business relationships, a relation that an AS can create with a neighbor through which the pair defines how much traffic an AS will forward on behalf of the other, the cost of such operation and which destinations the AS will forward the traffic to. These relations can be of the following types [45]:

- customer-to-provider: the customer forwards traffic to and from the provider to its customers, free of charge
- provider-to-customer: the provider forwards traffic to and from the customer to all destinations it knows, at an agreed fee
- peer-to-peer: symmetric relation where each peer forwards traffic to and from the other peer to its customers, free of charge

ASes that provide forwarding services traffic to its neighbors are called Transit ASe. They represent 10% of the total ASes in the internet and are responsible for interconnecting the rest of the ASes in the internet, called Stub ASes. Stub ASes have no customers, no peers and only one provider, which acts as its Internet Service Provider (ISP) and thus allows the AS to access the rest of the internet.

ASes are usually classified into three tiers organized hierarchically (figure 1.1): tier-1, tier-2 and tier-3 ISPs [45]. Only a few Tier-1 ISPs exist and they are large national or international ISPs, all with direct access to the internet backbone, resulting in a mesh of interconnected ASes that form the internet core. Tier-2 ISPs are national or regional ASes that purchase internet service to Tier-1 ISPs and thus depend on them for internet access. Tier-3 ISPs are small region ASes that purchase service from both Tier-1 and Tier-2 ISPs and provide internet service to end-customers. Tier-1 ISPs and a group of Tier-2 ISPs compose the transit ASes for the whole internet, passing traffic for the rest of Tier-2 and Tier-3 ISPs that are stubs.

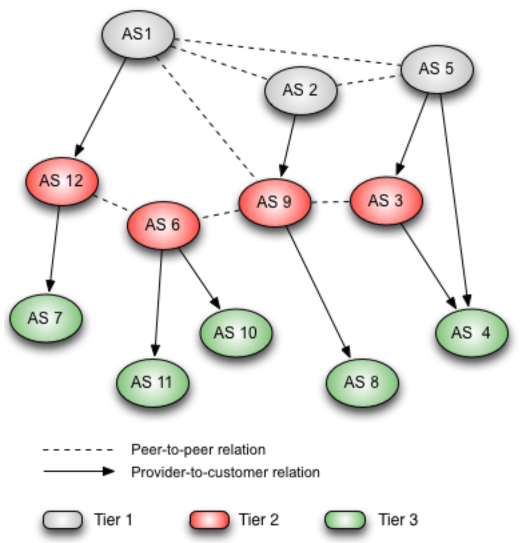

Figure 1.1: ASes create business relations with each other, forming a hierarchical infrastructure for service provisioning. Tier-1 ISPs are larger and form a full mesh with each other and typically have more connections than Tier-2 and Tier-3 ISPs.

Each AS is assigned an AS number, an unique number that uniquely identifies the AS, and a block of IP addresses, called an IP prefix (e.g. 173.50.0.0/16 is an IP prefix that covers all addresses from 173.50.0.0 to 173.50.254.254), that identifies a network destination. Prefixes are regulated by IANA (Internet Assigned Number Authority) and five Regional Internet Registries (RIRs), each responsible for delegating prefixes in a specific region. Large ISPs may apply for an IP prefix to one of the RIRs and then further split its prefix into smaller parts and assign them to client ASes (figure 1.2).

Finally, ASes are mostly driven by economic factors [16]. Smaller ASes tend to peer with each other to reduce traffic sent to the provider, thus reducing operational costs [45][38]. However, peering has its downsides as it requires buying additional equipment and represents higher management costs, and no AS wants to peer with a potential customer and lose the revenue. Peering can be private if done through a dedicated connection between both ASes, or public if done through an Internet eXchange Point (IXP). IXPs are third-party infrastructure that provide a point for several ASes to connect and peer with the participants they want, at a lower cost but with more limited bandwidth.

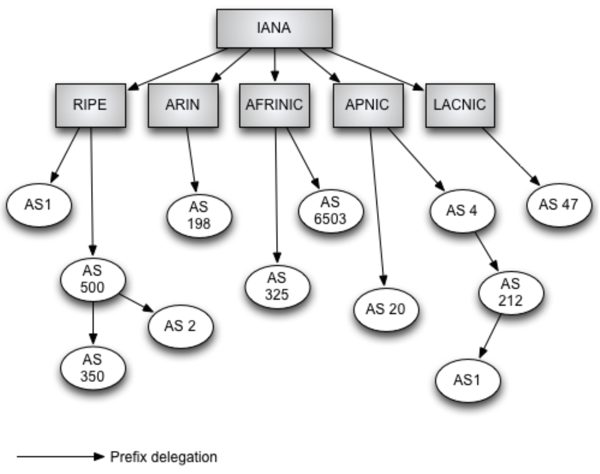

Figure 1.2: IANA acts as root for the entire internet hierarchy and coordinates with the five RIRs to delegate prefix to ASes. ASes can afterwards further delegate parts of their prefix to other ASes

1.2 Traditional Networks

A computer network is composed of a set of terminal devices (computers, etc.) interconnected through one or more network devices (switches, routers). The latter are responsible for forwarding packets from one terminal device to another using a specific communications protocol, usually the Internet Protocol (IP).

Over the years, the Internet has grown from a few nodes to thousands of interconnected networks and the number increases each day as more devices are incorporated into the network (e.g. smartphones). This growth has caused an increase in performance and reliability requirements from both network users and devices, as new, more demanding applications and services are created and slowly introduced into networks. In order to cope with such requirements, researchers and the industry have improved existing network protocols so they can deliver better performance, reliability and security. However, these protocols are usually vendor-specific and aim at solving a specific problem, making networks more complex with each added protocol.

Networks are divided into three planes (figure 1.3): management, control and data. Network policies are defined in the first, enforced by the second and executed by the third, resulting in a highly decentralized architecture that has satisfied network requirements for years. However, network control functionalities (control plane) are coupled in network devices along with packet forwarding tasks (data plane), forcing devices to participate in complex distributed protocols to perform several tasks (e.g. routing), exhausting time and resources required for a faster and more reliable packet forwarding. This vertical integration of planes has led to several problems including hard management, poor scalability and flexibility and slow innovation[29].

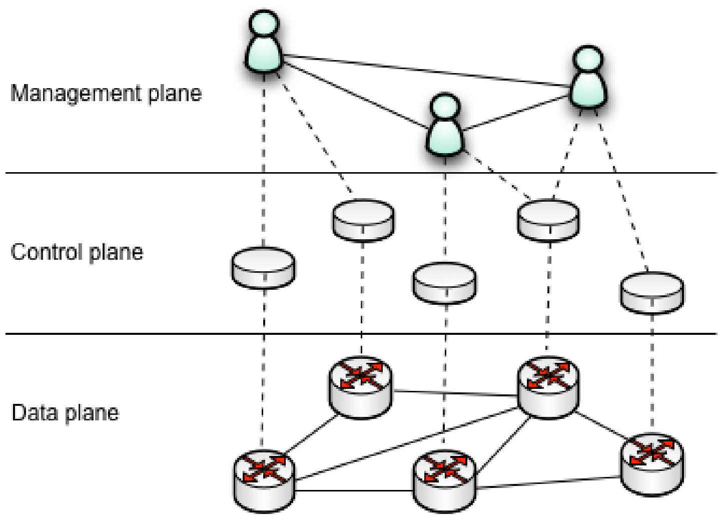

Figure 1.3: Layered view of traditional networks in three planes: management, control and data. The management plane is executed by the network managers, while the data and control plane are integrated in network devices.

First, installing system policies or adapting the network requires network managers to configure each device individually using low-level and often vendor-specific protocols. This is a time-consuming and error-prone process that can render the network slow, insecure or even lead it into an incorrect state. This scenario becomes worse in heterogeneous networks, as network managers are required to deal with several different protocols developed by the different vendors for their devices. Additionally, to implement network functionalities such as load balancing and access control, network managers have been using separate components to compensate for the lack of networking device capabilities, further increasing network complexity and lowering flexibility.

Second, networks become more complex as they scale [14], so it is possible for a network to grow to a point where it satisfies network requirements but cannot be managed due to the necessary time and resources it would take. Additionally, a large network may introduce communication and processing overheads resulting from executed protocols that do not allow it to satisfy the requirements.

Finally, integration of the control plane in network devices results in a closed environment where industry and researchers need to wait for vendors to create new equipment that incorporates new solutions for specific problems. This process typically takes a considerable amount of time and the new equipment is usually too expensive, which in addition to the overall costs of maintaining a network infrastructure means old hardware may be in operation for several years before being updated, hindering innovation to a point where new network protocols and applications take years to be designed, tested and deployed. An example of network stasis is the update from IPv4 to IPv6, which started over a decade ago and is still unfinished.

In conclusion, the incremental approach of providing solutions to cope with network requirements that has tailored traditional networks to their current state is no longer effective. It is becoming increasingly necessary to re-design networks so they can provide the required capabilities to not only address current network problems but also to provide future networks the ability to adapt to new challenges and requirements.

1.3 Software-Defined Networking

Software-Defined Networking (SDN)[40][29] is a network architecture that physically decouples the data and control planes (figure 1.4), moving the latter to a **logically centralized** controller and keeping the former in network devices. By doing so, network control is shifted from the decentralized set of network devices to the controller, tackling the root of traditional network complexity.

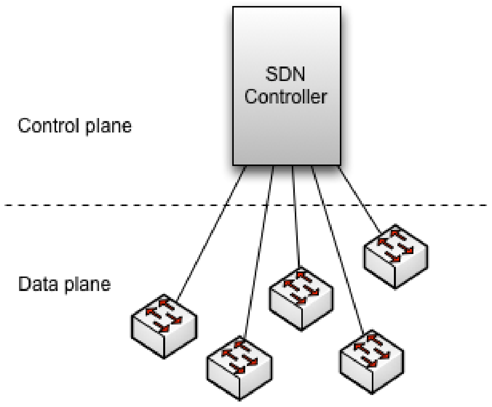

Figure 1.4: Plane decoupling in SDN: the control plane is executed by the controller, that defines how switches execute the data plane

Figure 1.5 shows a representation of the SDN architecture, divided in three planes: data, control and plane. The **data plane** is composed of a set of switches, which become simple packet forwarding devices that do not need to implement and execute complex protocols. They merely process instructions coming from the control plane and forward packets based on those instructions, resulting in simpler device implementations and less use of resources, ultimately leading to faster and more reliable packet forwarding.

The use of a logically centralized controller does not imply a centralized environment, which would create a single point of failure and severely hinder scalability. A controller may in fact be distributed[28][46] for load balancing and fault tolerance.

The **control plane** can be seen as the network brain, holding the architecture together. It is executed by the controller, a software platform that can run on commodity hardware and acts as a Network Operating System (NOS)[25]: it provides high-level programming abstractions and relevant network state to applications, so they can perform network operations without the need to deal with the low-level details of devices. The controller will translate instructions coming from applications and configure devices accordingly in a transparent way, so applications see the network as a single logical entity instead of a decentralized infrastructure. Network state (e.g. network topology, link state, device state, etc.) is made available to applications by means of a logically centralized *Network View*, stored in the controller or in a data store.

Decoupling of the control and data planes requires bi-directional communication between both planes. As such, the *Southbound Interface* (or Southbound API) becomes a critical component of the SDN architecture, as it provides the connecting bridge between the controller and the switches. The most widely accepted standard and the *de facto* protocol for the Southbound API as of today is OpenFlow [34][3], a communications protocol and device-implemented API that provides a clear and uniform way for the controller to interact with heterogeneous network devices without using vendor-specific implementations.

Network behavior is defined in the **management plane**. Implementing new network features and services is done by deploying applications on top of the controller using the provided abstractions and network view, independently of network topology and device implementations. This strategy results in a more simple and straightforward integration of different applications and features.

Communication between applications and the controller is done by means of programming abstractions, which compose the *Northbound Interface* (or Northbound API). At the time of this writing, this API is specific to the controller implementation, as there is no de facto standard for this API. However, it is expected that as SDN evolves, a standard will emerge[29].

The West- and Eastbound interfaces are only present when the network features a distributed SDN controller and are used to provide a way for the different controllers to communicate and coordinate.

A SDN can work in reactive or proactive mode. In **Reactive** mode, switches react to packets they do not know how to process, passing a copy of the packet header to the controller for it to decide how to process it and configure switches with the necessary flows that will be applied to all subsequent packets with the same header. This packet is usually the first packet of a flow, so this operation is commonly called *flow initiation*, done once per flow.

In **Proactive** mode, the controller configures the switches to handle all possible network traffic by installing a pre-determined set of flows. Operating in this mode requires a great understanding of network topology and applications to identify all necessary flows but can help scale the network as it is not required to send the first packet of a flow to the controller.

There are several controllers for SDNs developed with different objectives and following different designs. The most relevant for this work is *OpenDaylight*, as it aims to

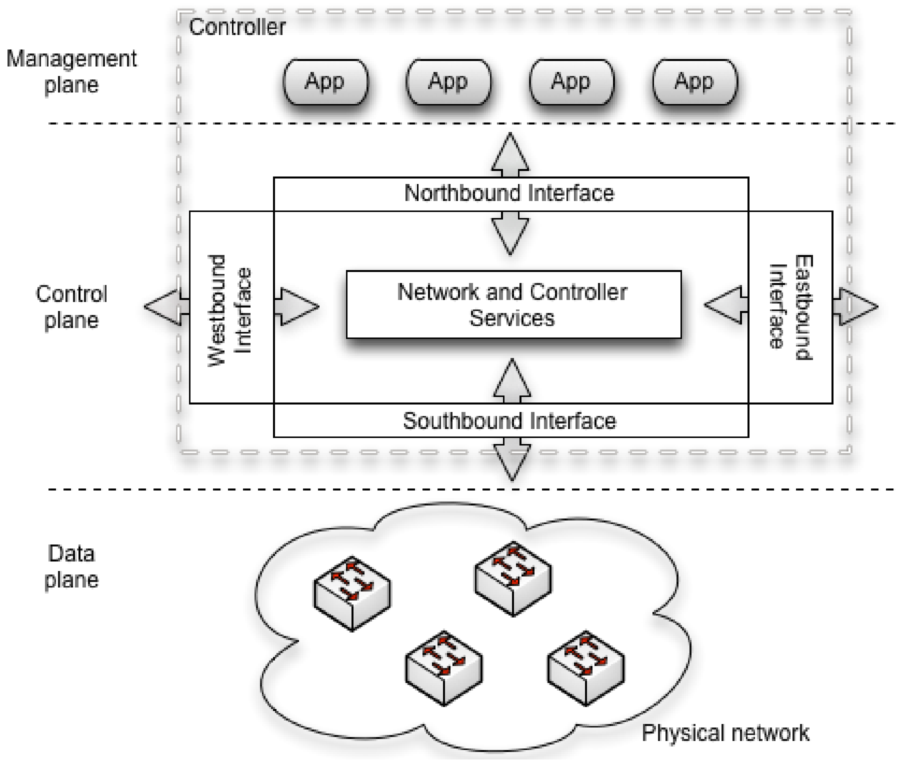

Figure 1.5: Layered view of the architecture of a SDN controller. The management plane is executed by the applications that run on top of the controller and use the northbound interface to interact with it. Controller services represent the control plane and use the west and eastbound interfaces to communicate with other controllers and the southbound interface to instruct the network devices.

be the controller of reference for SDNs and has the support of a large community and some of the top names in the technology industry, such as IBM, HP, Intel, etc. We discuss this and other controllers them further in the next chapter.

1.4 Routing

Routing is a critical part of any network. It is the process of discovering and configuring the paths (or routes) from each network device to all the other devices in the network, providing a way for devices to know where to send a packet so it reaches a specific destination. A path is an ordered sequence of nodes through which the origin node can send a message to the destination.

Due to the large scale of the internet, it would be unfeasible to have all network devices maintain the same routing table, with all participating in the same routing protocol. The hierarchical structure of the internet thus allows dividing routing in two parts: routing between ASes and routing within an AS. Routing devices belonging to an AS use an Internal Gateway Protocol (IGP) for destinations inside the same AS and an External Gateway Protocol (EGP) to route between different ASes (figure 1.6). There are several

IGPs developed and used in networks such as RIP, OSPF, etc., but only one EGP is mainly used in the internet, the Border Gateway Protocol.

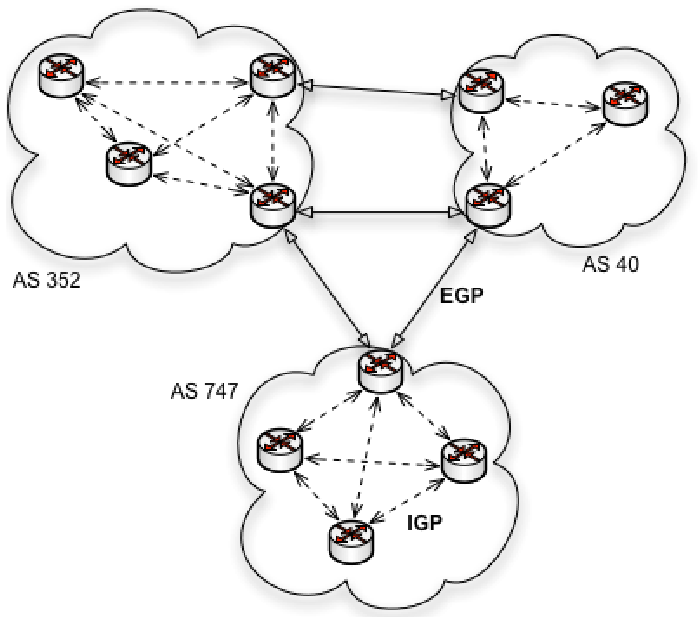

Figure 1.6: Routing between ASes is done through an EGP, while routing inside each AS is done through an IGP

Traditional networks employ this feature through complex distributed protocols in which network devices need to participate to decide how traffic will flow through the network, which incurs in heavy processing overheads on network devices. On the other hand, SDNs implement this feature through applications that are executed on top of the controller. One such kind of app is *RouteFlow*, a flexible and scalable routing platform for SDN that is the target of this work.

1.4.1 Border Gateway Protocol

The Border Gateway Protocol (BGP)[42] is an inter-domain routing protocol and the *de facto* EGP for exchanging routing information between ASes on the internet[45]. The protocol is used so that ASes can interact with each other to share information on how to reach every other destination (IP prefix) in the internet.

BGP speakers (routers running BGP) that are at the edge of an AS are known as *border* routers, establishing BGP sessions (TCP sessions) with border speakers from neighboring ASes through which they exchange UPDATE messages. Speakers that are engaged in a session are called peers.

UPDATE messages contain several fields including a list of prefixes being advertised, a list of prefixes being withdrawn and a list of attributes for each route being advertised, including the *AS_PATH* attribute. This approach, called *path vector*, is a variation of the distance vector algorithm where instead of exchanging a single metric for each route, nodes exchange a set of route-related attributes.

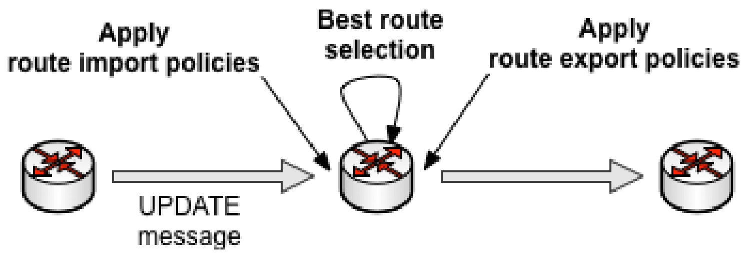

Figure 1.7: When a speaker receives a BGP UPDATE, it applies route importing policies, executes the best selection process and applies route exporting policies, if needed, before re-advertising the message.

Figure 1.7 shows the process executed by a speaker when it receives an UPDATE message. First, the speaker will apply a route *importing policy* to determine if the route should be filtered or processed and in case it will be processed, the policy might change route attributes. Afterwards, if the speaker did not have a previous route for that destination, it will accept it as the best route and save it in its routing table. If a previously selected best route was already stored, the two are compared using a set of rules (table 1.1) and the winning route is stored in the routing table. Additionally, if the received route wins, the speaker concatenates the AS number of the processing AS to the received AS_PATH and advertises[1] a new UPDATE message using the new AS_PATH. The message attributes and the set of neighbors to which the route is sent is determined by the route *exporting policy*. Both policies and the set of rules are defined by each AS independently and are usually kept private.

Step	Attribute	Controlled by local or neighbor AS
1	Highest LocalPref	local
2	Lowest AS path length	neighbor
3	Lowest origin typ	neither
4	Lowest MED	neighbor
5	eBGP-learned over iBGP-learned	neither
6	Lowest IGP cost to border router	local
7	Lowest router ID (to break ties)	neither

Table 1.1: Steps in the BGP decision process [16]

Additionally, new routes are also propagated using internal BGP (iBGP) to speakers belonging to the same AS. Previously, speakers in the same AS formed a full mesh and propagated routes accordingly, but this approach severely hinders scalability. As such,

[1]The terms advertise and announce are used interchangeably to denote the act of an AS sending an UPDATE message to a set of neighbors.

large networks typically employ a hierarchy of *route reflectors*[13]. Route reflection divides the network in clusters of one or more speakers acting as servers for the remaining clients, which may act as servers for sub-clusters. The servers form a full mesh with each other while clients are not required to be peers. Servers would reflect routes received from servers to all clients and routes received from clients to the remaining clients and servers. There is, however, the problem of misconfiguration, where speakers in a path could be assigned different routes from different reflectors, leading to inconsistencies [20].

One of the worst problems of BGP is its insecure design, which led to several vulnerabilities discovered over the years and exploited through several attacks such as *prefix hijacking*, *interception* and *data-plane* attacks [15]. In particular, a prefix hijacking occurs when a malicious AS advertises a prefix belonging to another AS in order to hijack its traffic and thus gain access to privileged information. We tackle this problem in this work.

1.5 Goals and contributions

In this thesis, we propose two applications for the OpenDaylight controller: RFProxy and BGPSec. The first integrates RouteFlow with the OpenDaylight controller, providing another controller alternative to RouteFlow while providing a more advanced routing suite to OpenDaylight.

Our second main contribution is BGPSec, an application to provide an AS with protection against prefix hijacking attacks in an SDN environment. We discuss the advantages of providing a solution using SDN instead of traditional networks and discuss a possible evolution for this application.

The main contributions can be resumed as follows:

1. Development and evaluation of RouteFlow for OpenDaylight (chapter 3)
2. Analysis of BGP security problems and proposed solutions in traditional networks (chapter 4)
3. Design, implementation and evaluation of a SDN application to provide partial security to BGP (chapter 5)

All the developed software is available open-source in github[2][3]. Additionally, the ODL RouteFlow application was added to the RouteFlow project[8] and in in the initial process of incorporation in OpenDaylight.

[2]https://github.com/routeflow/odl-rfproxy
[3]https://github.com/tiagoposse/sdnbgpsec

1.6 Planning

Figure 1.8 shows the plan to produce this thesis. We identified four main phases:

- Requirements (chapter 2): In this phase, we collected information about SDN and existing controllers, the BGP routing protocol and RouteFlow. Considerable time was also spent on understanding both RouteFlow and the OpenDaylight controller, their architecture, general workflow and code.
- RFProxy Implementation (chapter 3): In this phase, we implemented the RFProxy application for OpenDaylight and performed tests to verify if it was correctly working.
- BGP Security analysis (chapter 4): In this phase, we collected a more complete set of information about BGP security and previous security solutions for traditional networks. We focus on attacks and vulnerabilities and provide clear examples of possible attacks against the internet infrastructure. We also discuss why BGP is still so insecure and why no security proposal has yet been clearly adopted.
- BGPSec Implementation (chapter 5): We design a new application for the OpenDaylight controller with the goal of protecting networks against BGP prefix hijacking attacks and test its correctness and the overheads that it incurs on BGP message processing.
- Dissertation: In this phase we wrote the dissertation.

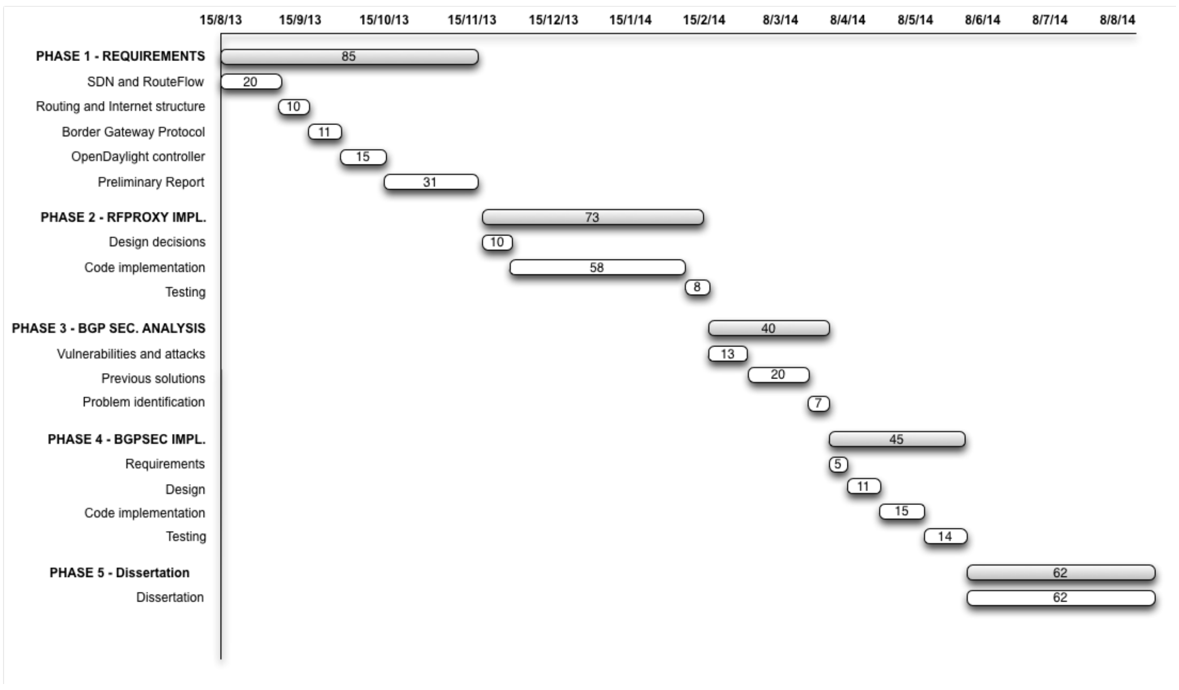

Figure 1.8: Gantt Map

Chapter 2

Background

2.1 OpenFlow

OpenFlow (OF)[34] is an open-source communication interface developed by the Open Networking Foundation[3], adopted as the *de facto* Southbound Interface for SDNs. OF combines a set of common of features present in the flow tables of devices from different vendors and provides a uniform API for developers to define forwarding rules by programming the flow table: the OpenFlow protocol (figure 2.1).

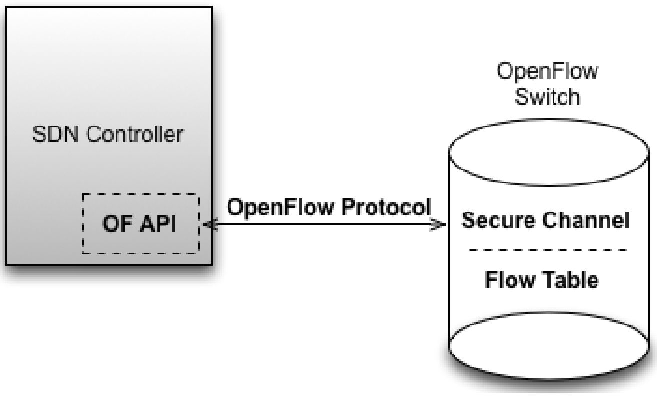

Figure 2.1: Overview of an OpenFlow switch

Each entry in a flow table, called *flow*, is composed of three parts: (1) a matching rule, composed by a set of packet header fields, (2) a list of actions to be executed for matching packets and (3) counters for statistics. Whenever a packet is received by a switch, it will compare the packet headers to the matching rule of each flow until it finds a match, after which it executes the specified actions and updates the flow counters. OF devices implement a minimum set of three possible actions: (1) forward the packet to a specific port, (2) forward the packet to the controller and (3) drop the packet. However, this set can be extended.

When the network is operating in reactive mode (figure 2.2), packets that are not matched with any flow in the switch trigger a *packet_in* message containing the packet headers to the controller. The controller processes the packet and answers with a *flow_mod* message to the switch in order to configure a flow to be applied to subsequent packets, or it directly injects a message in the data plane without changing switch configurations.

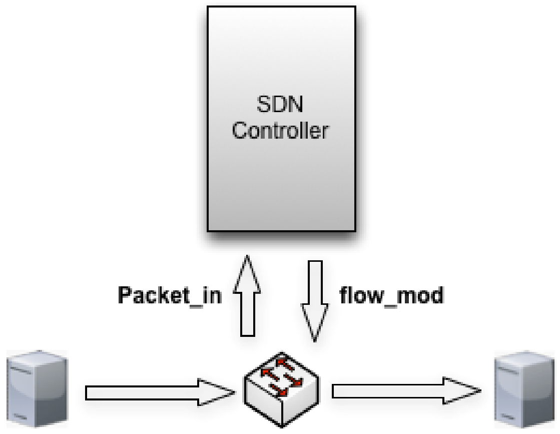

Figure 2.2: Representation of a SDN operating in reactive mode: the first packet of a flow that is not matched by the network device is sent to the controller, which instructs the device of the appropriate action to apply to subsequent packets with the same header

In addition to *packet-in* messages, an OF device will automatically send events representing changes in port or link states to the controller, allowing for quick reactions to changes in devices.

By using the common set of functionalities, OpenFlow works on the vast majority of network devices already deployed, easing its adoption. However, OpenFlow also has its downsides: (1) it requires that switches both understand packet headers and spend execution cycles to extract the necessary information from the packet in order to perform matches. This scenario becomes worse considering that version 1.3 of OpenFlow already features more than 40 header fields, as compared to the 10 header fields of version 1.0. Additionally, adding or removing header fields from the protocol may pose backward compatibility issues.

2.2 SDN controllers

SDNs rely on a controller to operate and manage the network. Most controllers are event-driven, working in compliance with OF: applications register with the controller for events of either implementation or southbound protocol-specific types, so that when the controller generates an event of that type, they may process it. The controller will afterward generate events according to messages received from the data plane and will pass it to interested (registered) applications. This model allows the creation of an execution pipeline (figure 2.3) for each event, with the possibility of defining priorities and dependencies for each application. Additionally, any application may choose to let the next application process the packet, or consume it and thus end the execution pipeline.

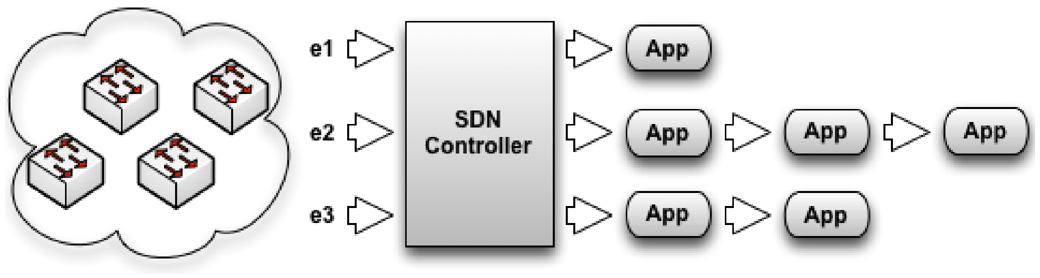

Figure 2.3: Representation of an event-driven SDN controller. Events produced by network devices are captured by the controller and passed to registered applications, forming a separate execution pipeline for each type of event

2.2.1 NOX

NOX[25] is a SDN controller created in 2008, developed in both C++ and Python and the first to introduce the concept of a Network Operating System (NOS) in SDN context.

The objective of a NOS is to provide high-level abstractions of network resources that simplify management and ease network programmability. The NOS itself does not manage the resources, it merely provides a uniform interface for the development of applications that manage these resources. The goal of NOX was to provide these abstractions for developers to be able to program the network by creating and deploying applications for the NOX controller.

NOX was built following the event-driven architecture described above and using a global network view, composed of the switch-level topology, the locations of users, hosts, middleboxes and other network elements; and the services (e.g., HTTP or NFS) being offered. The authors argue that this set of information is complete enough to enable several management tasks to be executed without much effort and also changes slowly enough for the controller to be able to keep it consistent over time without compromising too much scalability. In any case, NOX provides network managers with tools to trade-off scalability for flexibility and vice-versa by increasing or decreasing the amount of information to be gathered from the network.

2.2.2 Beacon

Beacon [21] is an open-source controller whose main goal is to provide a development-friendly environment for developers to easily create high performance applications. For that purpose, Beacon was developed in Java instead of C or C++, providing developers a simpler development environment with faster compilation times and automatic memory management, at the cost of some performance. Its author argues that although Java is slower, the simpler development environment, the cross-platform support and native support for multi-threading were crucial when choosing the implementation language.

The controller is composed by a set of *bundles*, an abstraction of the OSGi framework. Controller applications are defined through bundles and registered as services in the service registry, a component of the OSGi. Other applications can then retrieve instances of registered services to perform other actions.

This modular design is enhanced by a feature that previous controllers did not support: run-time modularity. Previous controllers provided users and developers with either compilation-time modularity, where it was possible to select which applications to build, or start-time modularity, where it was possible to select which applications to launch. Beacon however provides run-time modularity, enabling network managers to add or remove applications without shutting down the controller, therefore easing the way developers interact with it. This possibility allows for several use-cases such as the installation of temporary applications to enhance a property of the controller or the removal of applications experiencing bugs or errors, etc.

Finally, Beacon also operates in an event-driven fashion as explained above and is multi-thread, binding network devices to particular threads.

2.2.3 Floodlight

Floodlight [11] is an open-source, enterprise-class SDN controller based on Beacon. It inherits the same modular design and event-driven architecture, but without using the OSGI framework for module interaction due to performance and deployment reasons.

Service registry and consumption is provided through the controller main module, the Floodlight Provider. The controller also provides run-time modularity just as Beacon, but provides better performance by using the Floodlight Provider instead of the OSGi framework[5] for service provisioning.

Finally, although the controller is written in Java, the northbound API is REST, so applications can be written in any programming language that can use this API.

2.2.4 OpenDaylight

OpenDaylight[1][10] is an open-source SDN controller written in Java, supported by top names of the technological world such as HP, Juniper, IBM, Intel, Linux Foundation and others.

The controller is highly modular (figure 2.4) and features base dynamic modules that perform network tasks and controller services, which are interconnected through the OSGi framework. This framework also composes the northbound interface for the controller, along with a REST API for applications running outside the controller address space.

[1]https://github.com/opendaylight/controller

Applications register and use services provided by other modules through the northbound interface and communicate in an event-driven way. OF messages are converted to events and passed to modules that register as listeners for that type of message. It is also possible to define listener dependencies, in order to generate a chain of message delegation for a type.

The southbound interface supports a set of protocols including OF, SNMP, OVSDB, etc., which are abstracted to applications through the Service Abstraction Layer (SAL). The SAL deals with the different southbound interfaces and provides a uniform API to be used by northbound applications.

The goal of Opendaylight is to be the reference open-source SDN controller in order to ease the adoption of SDNs. This is the reason why we chose to work with this controller.

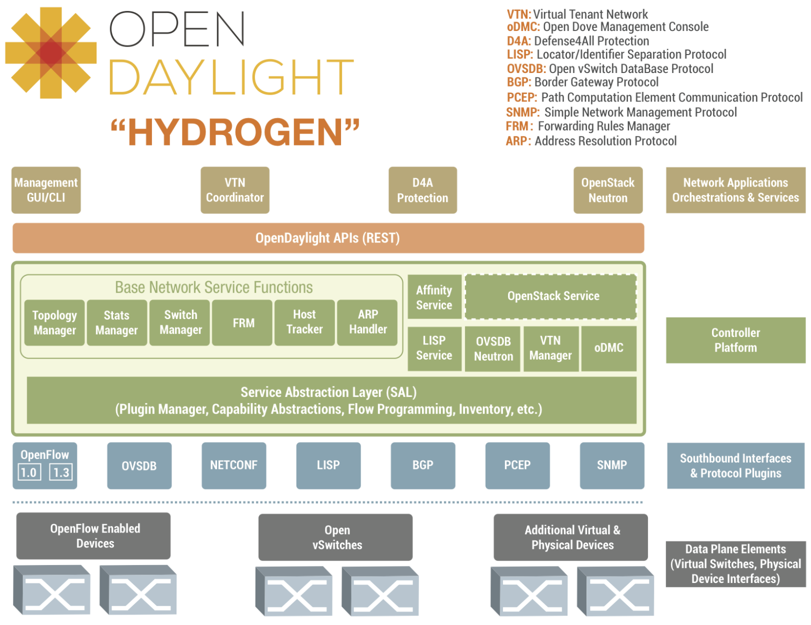
OPEN DAYLIGHT
"HYDROGEN"
VTN: Virtual Tenant Network
oDMC: Open Dove Management Console
D4A: Defense4All Protection
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase Protocol
BGP: Border Gateway Protocol
PCEP: Path Computation Element Communication Protocol
SNMP: Simple Network Management Protocol
FRM: Forwarding Rules Manager
ARP: Address Resolution Protocol
Management GUI/CLI
VTN Coordinator
D4A Protection
OpenStack Neutron
Network Applications Orchestrations & Services
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
FRM
Host Tracker
ARP Handler
Affinity Service
OpenStack Service
LISP Service
OVSDB Neutron
VTN Manager
oDMC
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
OpenFlow 1.0 1.3
OVSDB
NETCONF
LISP
BGP
PCEP
SNMP
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 2.4: OpenDaylight controller architecture[4]

2.3 RouteFlow

RouteFlow[2][43][8] is a routing platform that provides flexible and scalable IP routing services for SDNs. The platform creates a virtual network that mirrors the underlying network infrastructure and makes routing decisions by analyzing the changes on the routing and ARP tables of the virtual hosts.

Virtual hosts are created using the *process-based virtualization* and *network namespaces* features of Linux to provide lightweight virtualization. A container, or virtual host, is attached to an independent set of system resources, including a network interface, ARP

[2]https://github.com/routeflow/routeflow

and routing tables, etc., that are not accessible to other virtual hosts. This type of OS-level virtualization is more scalable and uses less resources than normal full-system virtualized hosts.

Figure 2.5 shows the overview of the RouteFlow architecture, divided in three components [47]: RFClient, RFServer and RFProxy. Each virtual host runs a daemon, called **RFClient**, responsible for 1) registering the virtual host in the controller as a virtual plane resource; 2) manage the virtual host network ports; 3) detect changes in ARP and routing tables and 4) convert these changes to OpenFlow flows.

RFServer runs as a standalone application, written in Python. It is responsible for managing the virtual environment by creating and removing virtual machines, communicate with RFClients and request RFProxy to install/remove flows. It holds the control logic of all the application and as such it is the central component of RouteFlow.

RFProxy runs as an application in the controller and is responsible for receiving packets from the network infrastructure, inform RFServer of changes in the network and configure network devices.

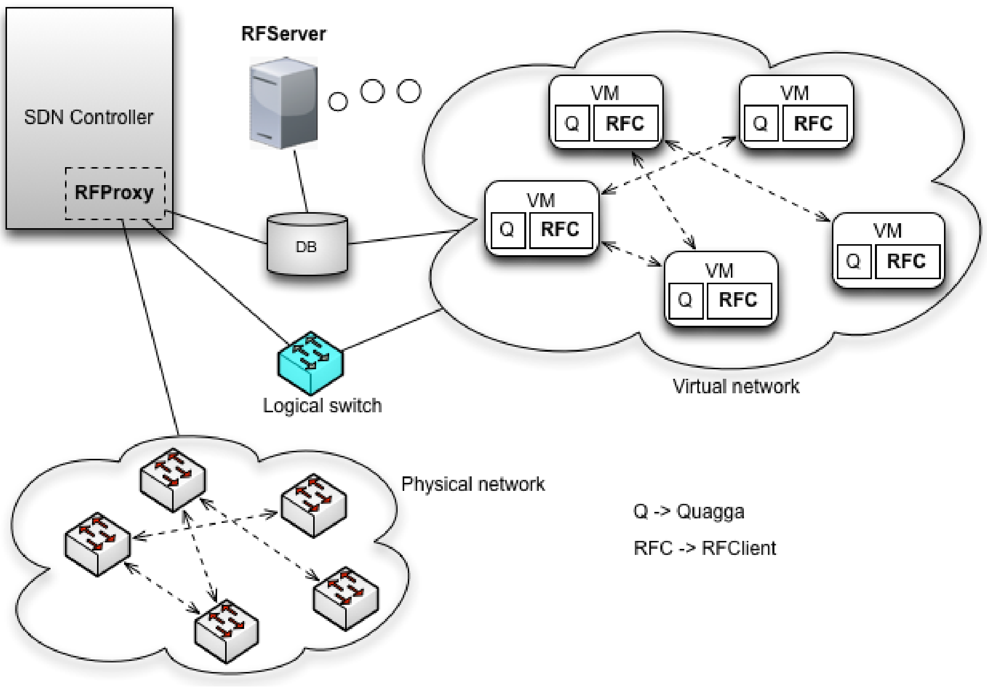

Figure 2.5: Representation of the RouteFlow architecture

Routing decisions are made based on the virtual network by observing its behaviour: the routing engine (Quagga[2]) produces the forwarding information base (FIB) according the configured routing protocol (e.g. RIP, BGP). RFClient then captures changes in routing and ARP tables produced by Quagga, translates them into OpenFlow tuples and passes them to the RFServer, which adapts this FIB to the specified routing logic and instructs the RFProxy to configure flows in the physical datapaths using OpenFlow.

2.4 Mininet Hi-Fi

Mininet Hi-Fi[30] is a container-based emulation platform that allows the user to run and experiment with custom network topologies. It aims at providing a framework that allows the production of realistic and reproducible network experiments using commodity hardware.

It uses the same process-based virtualization as RouteFlow to emulate the network, providing a more scalable and dynamic environment than full-system virtualized hosts.

Mininet Hi-Fi is an enhancement to Mininet [26] that provides performance fidelity. To achieve this goal, it enforces a minimum limit on CPU and network bandwidth by using three Linux features: Control Groups, CPU Bandwidth Limits and Traffic Control. All three features can be configured to provide a system-wide behavior that matches hardware and provides high-fidelity results [30].

Additionally, Mininet Hi-Fi employs real-time monitoring to check for network and processing delays. In case any delay occurs and any switch or host is deemed overloaded, fidelity in the execution is lost and the experiment needs to be reconfigured to ensure realistic results.

Mininet Hi-Fi has been the evaluation platform of choice by the SDN research community, providing a realistic platform to test new applications.

Chapter 3

Routing Proxy for SDN

At the time of this writing, RouteFlow has been developed for the POX, NOX, Ryu and Floodlight controllers, but not for OpenDaylight (ODL). As ODL is the SDN controller of reference, we chose to develop it for ODL to provide a better routing platform and thus help this platform to evolve. The application is available open-source in https://github.com/routeflow/odl-rfproxy.

3.1 Architecture

Figure 3.1 depicts the architecture and general communication pattern of RFProxy, divided in three modules: the main module, which we generally call *proxy*, the Inter-Process Communication (IPC) module and the RFProtocol.

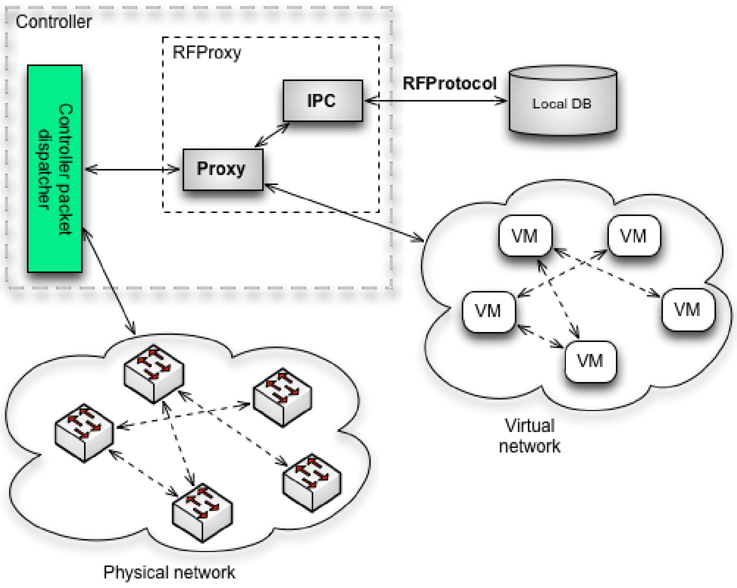

Figure 3.1: Detailed view of RFProxy in a network scenario

The proxy is responsible for communicating with the OpenDaylight controller, processing packets coming from both physical and virtual networks and processing events for adding or removing switches/switch ports. The IPC module is responsible for reading and writing from the database and execute actions based on received messages coming from the RFServer. The RFProtocol describes the messages exchanged between the RFServer and the RFProxy implementation.

Communication between the virtual topology and RFProxy is made through a virtual switch that has to be set up separately, usually when starting up RFServer. Communication between the RFProxy and RFServer is done through a *NoSQL* database using the RFProtocol for message format specification.

Figure 3.2 shows an overview of the controller running RFProxy:

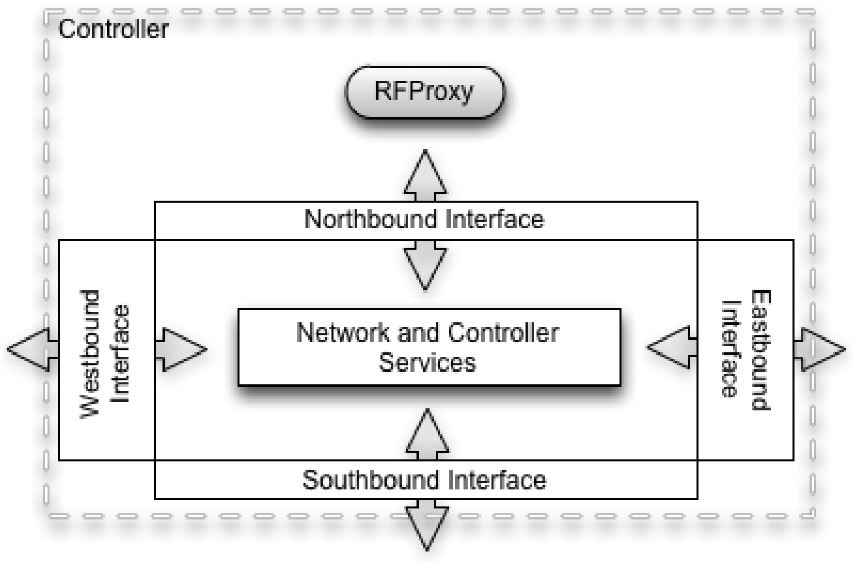

Figure 3.2: Overview of the RFProxy application

3.2 Implementation

In this section we present the details of how RFProxy works, considering the architecture explained above. RFProxy was written in Java, the programming language for the northbound API of OpenDaylight. The code was divided into Java classes to simplify programyming and organization.

In compliance with the OpenDaylight controller, RFProxy features an *Activator* class, where dependencies and services used by the application are specified in order to register and consume all necessary modules with the OSGi framework of OpenDaylight.

We now describe the main components used in our implementation:

- RFProxy implements the *IInventoryListener* and *IListenDataPacket* interfaces. The first allows the application to listen to events generated by the addition or removal of switches/switch ports, while the second allows the application to listen to *packet_in* events.

- MongoIPCMessageService: responsible for the communication between the RF-Server and RFProxy. The component creates a thread that runs an infinite loop checking for new messages in the database that are destined to RFProxy. It does so by checking the *read* field for all entries in the database.

- RFProtocolFactory: responsible for the creation of RFProtocol messages. Each message is represented by a different class and code that is instanced by the RFProtocolFactory when receiving a message from the database.

- RFProtocolProcessor: responsible for processing messages received from RFServer. The component works on top of objects created by the factory and performs actions based on the received action and parameters.

- AssociationTable: responsible for storing the association between virtual and physical network devices. Table 3.1 shows an example of this table that stores the id and port number that was registered for a physical or virtual switch. When a port is registered but cannot currently be associated with its correspondent, only its respective half of the table is filled. As such, a full table entry represents a client-dapatath association.

Datapath	Virtual switch
dp_id, dp_port	vm_id, vm_port

Table 3.1: The association table consists of two-field entries that list the mapping between datapath ports and virtual switch ports

The RFProxy acts as a protocol translator, converting OF messages into RFProtocol messages and vice-versa and communicates with the RFClients (*clients*) to map the physical to the virtual network (figure 3.3).

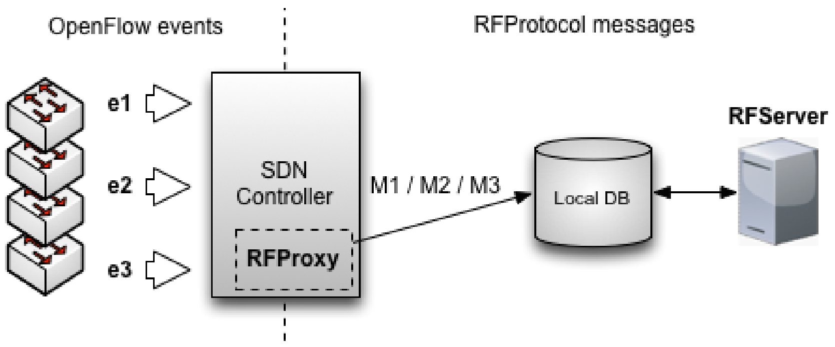

Figure 3.3: RFProxy processes events (e1, e2, e3) and generates a RFProtocol message (M1, M2, M3) for each event, which are inserted in the local database for the RFServer to process.

All network events are converted into messages according to their type and sent to the RFServer (*server*) for processing. This processing will generate an action to be executed by RFProxy, which may include the configuration of OF flows in the physical infrastructure. This communication is made through a database.

When a client is started, it sends a *mapping packet* to the RFProxy containing information about its mapping. When both the physical and virtual elements of the mapping are connected and registered to the server (figure 3.4), the client sends a message to RFProxy update its local mapping table. Using this table, the proxy can forward received messages to its respective correspondent.

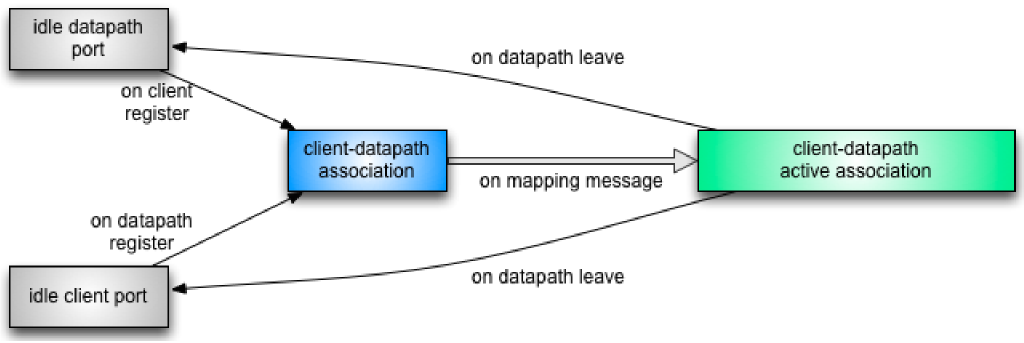

Figure 3.4: How mapping between the physical and virtual networks occurs in RouteFlow. Each client or datapath is assigned an idle status until its correspondent is registered, leading both to a client-datapath association status. The server will ask the clients to send a mapping message to the RFProxy, which will inform the server and update its association table with the active association.

3.2.1 Messages

RouteFlow makes use of an IPC based on a *NoSQL* database for easy integration with the main programming languages. The database used is MongoDB, as it is the default database used in RouteFlow. Each database entry identifies a message from either RFServer or RFProxy to the other, containing a set of parameters, a destination field and a *read* field that indicates whether the message has already been processed.

Figure 3.5 shows a representation of an IPC message flow between RFProxy and RFServer:

- DatapathDown: used to inform RFServer that a physical switch has been disconnected from the controller;
- DatapathPortRegister: used to register a new physical switch port in RFServer;
- DataPlaneMap: used to inform RFProxy of a successful mapping between a physical and a virtual switch port;

- RouteMod: used to trigger the configuration of a new route in the physical infrastructure;
- VirtualPlaneMap: used after receiving a mapping packet from a RFClient to inform RFServer of a new possible mapping;
- PortRegister: implemented according to RFProtocol but not used as it is not yet used for operation by RF-Server
- PortConfig: implemented according to RFProtocol but not used as it is not yet used for operation by RF-Server

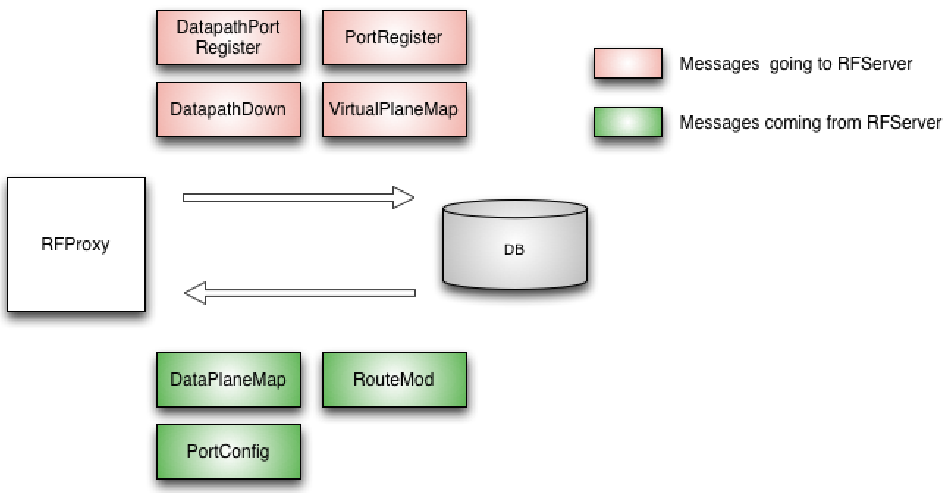

Figure 3.5: Representation of RFProtocol messages sent and received by the RFProxy through the local database

3.2.2 Processing

There are several separate workflows that the RFProxy application can execute, depending on the events received.

Figure 3.6 illustrates how RouteFlow processes events for adding switches.

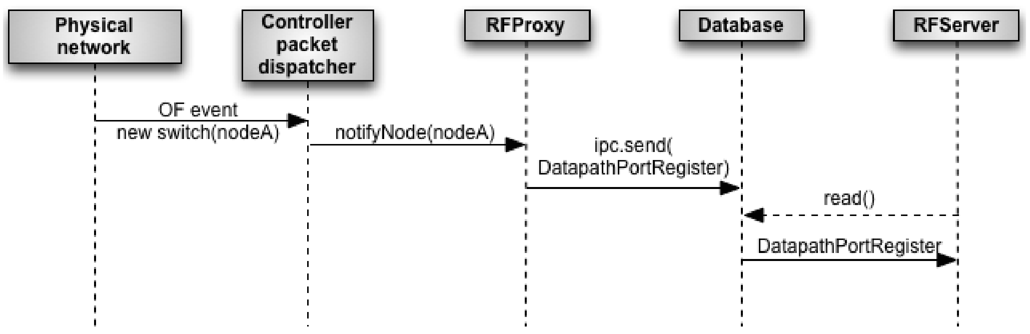

Figure 3.6: Processing of an event to add a physical device that connected to network

When RFProxy receives a message from the network, it verifies whether it is a *mapping packet* coming from a RFClient or if a normal message. In case it is the latter, RFProxy forwards the message to the correspondent virtual or physical device if the mapping between both has already taken place. If it is a mapping packet (figure 3.7), it is processed and passed to RFServer as a *VirtualPlaneMap*, containing the required parameters to perform a virtual to physical network mapping. If a normal packet (that is, not a mapping packet) comes from either a virtual or physical switch, the proxy will check the Association Table for the corresponding network element and forward the packet to it.

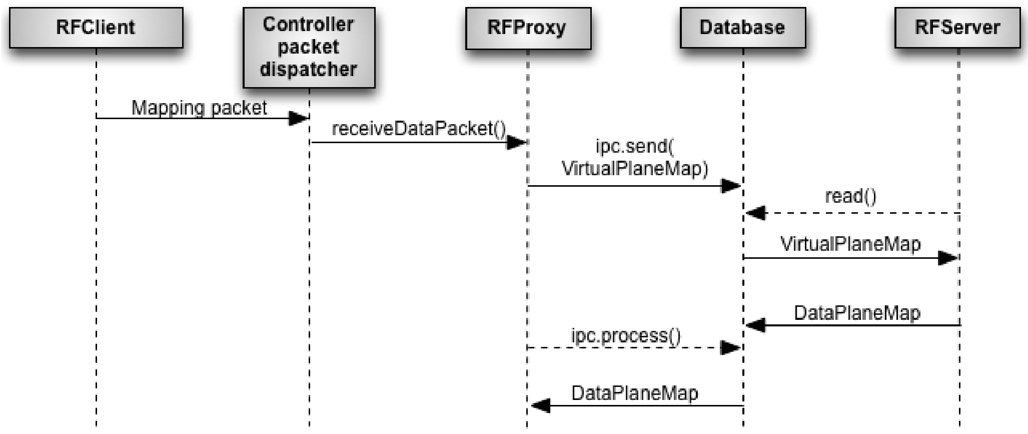

Figure 3.7: Processing of mapping packets

As for events received through the IPC, for each type of event RFProxy generates a specific RFProtocol message that is inserted in the database for RFServer to process. RFServer will read it and insert a response in return, to be executed by RFProxy. Figure 3.8 shows the execution flow of RouteFlow for RouteMod messages.

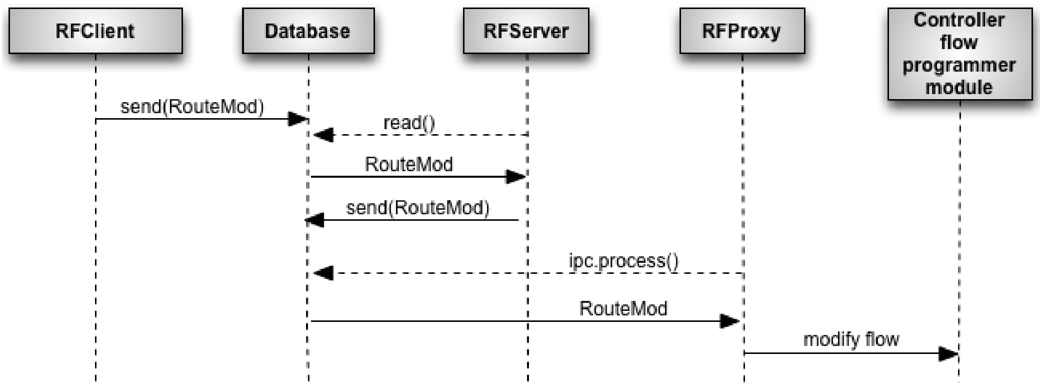

Figure 3.8: Processing of flow_mod messages

Chapter 4

Security in BGP

4.1 Introduction

When BGP was designed, the internet was a secure place where all entities trusted each other. As the internet grew, more and more cases of incorrect BGP announcements were detected[33, 19, 41]. Since BGP was not designed to be secure, vulnerabilities started to be discovered and attacks were successfully executed. Several works have summarized the security limitations of BGP[15, 37, 12, 35] to the following:

- There is no mechanism to protect the *integrity*, *freshness* and *authentication* of messages.
- There is no way to *verify* the authority of an AS to advertise a prefix.
- Paths are not *authenticated*.

This set of limitations allows attackers to tamper with normal BGP behaviour through BGP speakers or by attacking BGP sessions. Attacks against BGP sessions include message insertion/forgery, deletion, modification, eavesdropping and replaying [45]. In a forgery or insertion attack (figure 4.1a), the attacker inserts a forged, malicious message into a BGP session with the intent of causing session failures or the insertion of wrong routing information into the network. In a deletion attack (figure 4.1b), the adversary intercepts and removes a message that is being exchanged by two peers, possibly leading to inaccurate routing tables. For example, if a BGP update message containing a best path for a peer is intercepted, the peer will accept a worst path as the best.

Eavesdropping (figure 4.2a) consists in passively listening data exchanged between speakers in order to gain access to sensitive data such as policy and routing information. In a modification attack (figure 4.2b), an adversary removes a message from a session and reinserts it with erroneous information, which can lead to session failures and wrong routing information being inserted in the network. Finally, replay attacks (figure 4.2c) are performed by intercepting and storing a message exchanged in a BGP session and re-send

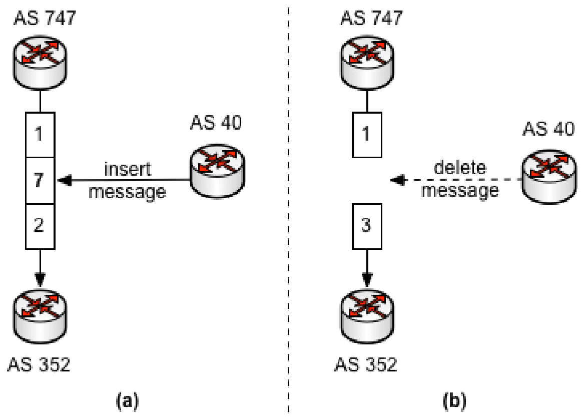

Figure 4.1: Representation of insertion/forgery (a) and deletion (b) attacks against BGP speakers

it afterwards. This attack can be used to re-introduce withdrawn routes into the network, withdraw valid routes or even to cause denial of service by replaying messages in bulks.

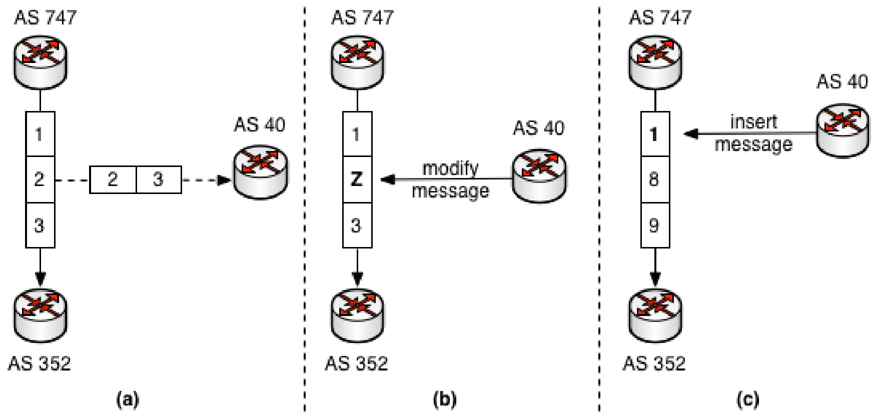

Figure 4.2: Representation of eavesdropping (a), modification (b) and replay (c) attacks against BGP speakers

As for speakers, an attacker may use either authorized (valid BGP speakers) or unauthorized speakers to attack the network infrastructure. While attacks coming from unauthorized speakers can be prevented using data origin authentication and data integrity mechanisms, attacks from authorized speakers are complicated to prevent since an authorized speaker can run flawed software, be compromised, be accessible by a malicious person or it may be misconfigured, causing involuntary disclosure of erroneous information. Real examples of attacks performed by authorized speakers include prefix hijacking, interception or data-plane attacks, described below. There are several other attacks against BGP including de-aggregation, contradictory announcements, manipulation of route exports, etc., but we do not detail them here.

In a **prefix hijacking attack**, a malicious or misconfigured speaker announces a forged UPDATE message containing an IP prefix that belongs to another AS (figure 4.3a).

Neighbors that receive this message will believe that the information is correct and run the best path algorithm using the received path. If the new path is elected the best, the AS will forward the traffic for that destination to an incorrect AS (figure 4.3b). Since current BGP implementations do not prevent a speaker from announcing prefixes it does not originate, these attacks are easy to perform and hard to detect. Independently of whether the hijack is malicious or unintentional, it will result in partial or complete redirection of traffic, in addition of easing the execution of other types of attacks including denial of service (DoS), man-in-the-middle (MITM), spamming and interception of password reset messages[45]. An example of this type of attack happened in 2009, when a Pakistani ISP hijacked traffic belonging to Youtube for over three hours due to an incorrect advertisement[33].

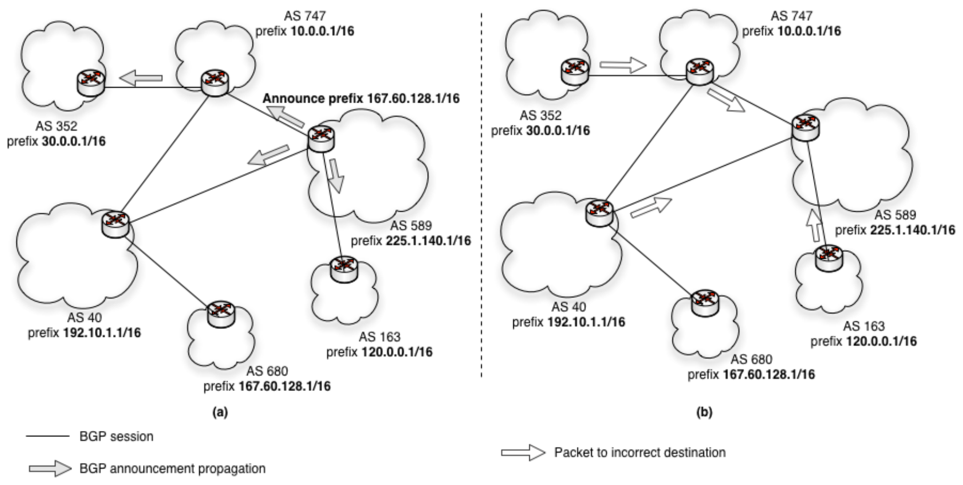

Figure 4.3: In (a), AS 589 maliciously announces a prefix that it does not own to its neighbors, that in (b) start to incorrectly sent traffic to AS 589 for the announced prefix, resulting in traffic hijack.

Additionally, prefix hijacking attacks can be classified as complete or subprefix hijacking attacks[32], depending on the sizes of the announced and original prefixes. In a subprefix hijack, an AS announces a more specific prefix (prefix with a bigger network section) owned by another AS. For instance, if an attacker announces a 24-bit prefix when the original AS announced a 16-bit prefix, the first will be chosen due to the longest prefix matching rule of BGP. Sub-prefix hijacks can be more effective as more specific prefixes are chosen independently of the value of other parameters considered by the path selection process[22].

In an **interception attack**, the attacker AS announces a malicious path for a correct destination (figure 4.4a), that is, a path ending in the AS that owns the destination prefix but that goes through the attacking AS. By doing so, the attacker may trick several ASes into accepting this new path as best, so that it gains access to the information destined to another AS (figure 4.4b).

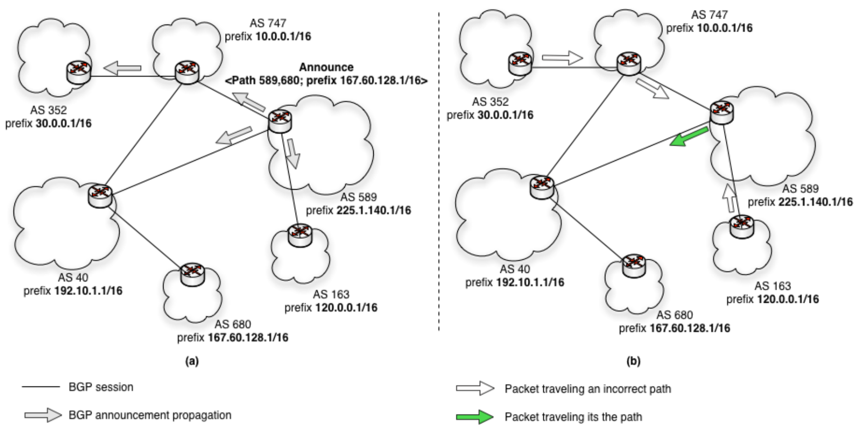

Figure 4.4: In (a), AS 589 maliciously announces a path that ends in the rightful owner of a prefix but traverses an incorrect path, so that in (b) its neighbors start incorrectly sending traffic to the AS, that will send it through its correct route. Although the traffic reaches its destination, AS 589 gains unauthorized access to data.

In a **data-plane attack**, a malicious AS announces a path and forwards traffic through a different, more profitable route (figure 4.5). Since BGP does not validate whether an AS forwards traffic through the path it announces, this attack is hard to prevent.

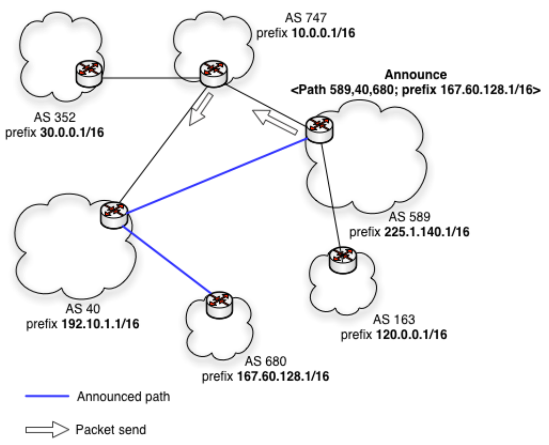

Figure 4.5: The attacker announces the blue path but incorrectly forwards through a different path

In conclusion, BGP is not secure by design and thus has several vulnerabilities that have been exploited over the years, resulting in partial outages of services caused by the different attacks.

4.2 Security Goals

In [45], the authors propose five security goals that should be guaranteed by any major BGP security extension:

- AS Number authentication: it must be verifiable that an entity using an AS number is its rightful owner;
- BGP Speaker Authentication: it must be verifiable that a BGP speaker uses an AS number that was assigned to the AS it belongs to;
- Data Integrity: it must be verifiable that a BGP message has not been illegally modified during transit;
- Prefix Origin Verification: it must be verifiable that an AS is the rightful owner of a prefix;
- AS PATH Verification: It must be verifiable that an AS_PATH was originated and traversed through the AS sequence in the right order and that it does not violate exporting policies.

A security solution that implements these security goals would prevent all known attacks. However, guaranteeing these goals requires the usage of cryptographic mechanisms that incur a large computational and storage overhead on network devices, resulting in a delay of BGP convergence and packet processing times.

4.3 Previous solutions

Several protocol additions were proposed over the years, some were *full* proposals that aim to fully secure BGP when all ASes implement the solution [27, 39, 48] and others were *partial* proposals to guarantee protection against a set of attacks [44, 49, 50]. We now describe some of these solutions following the work of [23], which organizes the solutions from weakest to strongest: simple BGP (described in the Background section), Origin Authentication, Secure Origin BGP, Secure BGP and Data-plane verification. Additionally, we describe Defensive filtering as it can be crucial to securing the internet and we propose it as future work.

4.3.1 Origin Authentication

Origin Authentication (OA)[12] consists in validating the delegation of prefixes to ASes to verify if an AS only announces a prefix that was correctly assigned to it by IANA or the rightful owner of the prefix. This is done through the use of *origin authentication tags* (OATs), usually cryptographic proof that the AS is the rightful owner of a prefix.

An OAT is composed of a delegation path, a set of delegation attestations and an ASN ownership proof. A delegation path is an ordered set of ASes that describes a delegation chain from the last AS to the origin. Each delegation attestation in the set validates one of the hops in the sequence, so every attestation would have to be validated in order to guarantee origin authentication. An ASN ownership proof is a certificate produced by IANA attesting that a set of AS numbers are assigned to a specific organization.

To ease revocation and distribution of attestations and ownership proofs, received updates are validated using an external entity, usually a trusted repository or database. This eliminates the need for a new message to distribute attestations and reduces the burden of the protocol in ASes.

In conclusion, OA guarantees that an AS cannot falsely claims that it is the rightful owner of a prefix, thus preventing prefix hijacking attacks. However, it is still possible to perform interception and other attacks, as the protocol does not guarantee path verification, BGP speaker authentication and AS Number authentication.

4.3.2 Secure Origin BGP

Secure Origin BGP (soBGP)[48] is a security extension for the BGP protocol with the goal of guaranteeing IP prefix ownership and that any announced path physically exists in the internet. It implements a web-of-trust model for authenticating AS public keys and a hierarchical structure for verifying prefix ownership.

The security mechanisms of soBGP are based on three certificates: *Entity*, *Policy* and *Authorization*, used to guarantee the existence of an entity, provide information about an AS and to guarantee origin authentication, respectively. All three types of certificates are distributed to the ASes through a newly defined message created to transport security information, the SECURITY message.

Initially, a set of trusted public entities sign a number of root Entity certificates and deliver them to a set of ASes using secure communication channels, usually out-of-band. These certificates are trusted by all ASes and thus can be used by the owning ASes to further sign Entity certificates, forming a web-of-trust.

Using Policy certificates, each speaker builds a topology map of possible paths to each prefix. When receiving an UPDATE, speakers use the map to verify that the received path physically exists. However, announcing a path that is correct but unavailable is not detected by soBGP. An unavailable path is a path were at least one of the ASes is a neighbor of, but has not announced a path to the next AS in the sequence and thus will not pass traffic to it, leading to a blackhole.

soBGP provides the possibility for partial deployment since there are security benefits when few ASes have adopted the protocol. Adopting ASes exchange certificates directly through multihop BGP sessions or through some other mechanism, in order to be able to validate IP prefixes and AS Numbers of the adopting ASes and guarantee partial path plausibility.

Considering the security goals described above, soBGP guarantees Prefix Origination Verification, AS Number Authentication, Data Integrity (through the use of IPSec) and BGP Speaker Authentication, but does not guarantee AS_PATH Verification (as only path plausibility is guaranteed).

In conclusion, soBGP web-of-trust model provides a more flexible and dynamic delegation and verification of certificates. However, it is questionable whether ASes should be able to sign certificates on behalf of other ASes, as IP prefixes and AS Numbers are currently managed and assigned by the infrastructure described in section 1.1. Additionally, it does not satisfy all security goals, it requires a change to BGP (and therefore requires re-deploying speakers) by adding the SECURITY message and it provides no immediate benefits to an AS that adopts the protocol.

4.3.3 S-BGP

S-BGP[27] is a secure, scalable and deployable architecture for authentication and authorization of BGP information that addresses most of the security problems associated with BGP. The architecture is based on two PKIs (Public Key Infrastructures), public entities that will be responsible for the distribution and validation of digital certificates to organizations. Both PKIs parallel the existing IP prefix delegation infrastructure and use it to delegate certificates.

Each AS receives a set of digital certificates from the PKIs that bind a public key, an AS number and an IP prefix to the AS. Using this set of certificates, the AS creates additional certificates: one to bind an AS number to an IP prefix, one for each speaker belonging to the AS to authorize its announcements and one for each route (route attestations). All certificates except the last are distributed using out-of-band mechanisms while route attestations are distributed in UPDATE messages for path validation. Route attestations are carried through a new, optional path attribute that can be carried through speakers that do not implement S-BGP (easing partial deployment). However, in order for the approach to work, every speaker would have to be replaced by a new, updated speaker that implements S-BGP and thus knows how to process the new field.

The protocol suffers from several deployment issues that have prevented its adoption:

- The memory size required to store certificates is too big for most currently deployed commercial routers.
- The existing internet infrastructure would also need to be changed in order to be able to provide services for the creation and validation of digital certificates.
- Replacing current speakers in order to deploy speakers that implement S-BGP is unfeasible as it would cost too much and would possibly affect internet stability.
- signing and verifying all certificates and attestations is computationally expensive, resulting in increase of BGP convergence times by a non-negligible amount.

S-BGP provides prefix origination verification and AS number verification through address attestations, speaker authentication through router public key certificates, AS path verification through route attestations and data integrity through IPSec. In sum, it guarantees all BGP security goals and provides protection against most attacks, including prefix hijacking and interception attacks. As such, S-BGP guarantees all properties of previous solutions, but it does not implement a mechanism to verify if an AS respects its routing policy when forwarding traffic, so it does not prevent data-plane attacks.

4.3.4 Data-plane verification

Data-plane verification[15, 49] is a defense mechanism that prevents an AS from announcing a path and forwarding traffic through another path. ASes may violate their exporting policies in order to send traffic through a more profitable path, instead of through its provider.

In [49] the authors offer a solution for this problem by allowing an AS to act as a *verifier* for its packets, allowing it to contact an AS (the *prover*) in the announced path for that destination to verify that the packet traversed it and its predecessor through tokens in the message. By chaining this verification, the verifier could detect misbehavior. The actions to take when such behavior occurs is left for each AS to decide.

This solution requires out-of-band pre-distribution of shared secrets between the verifier and the receiver, which may consume large storage space of each router but allows cryptographic operations to be avoided for fast verification.

4.3.5 Defensive Filtering

Defensive filtering is a defense mechanism which limits the BGP announcements made by stubs. Each transit AS keeps a list of the IP prefixes owned by their direct customers that are stubs. Whenever the customer announces an IP prefix, the provider checks the prefix list and in case the customer is not the owner for that prefix, the announcement is dropped/ignored.

Additionally, this mechanism is orthogonal to other protocols, that is, it can be used along with other security protocols like S-BGP.

Finally, if all transit ASes would implement this mechanism, all attacks performed by stubs would be prevented, leading to a big increase in internet security.

4.4 BGP Security adoption issues

Although several proposals have been considered for adoption and there are real efforts for the implementation of a RPKI (Resource Public Key Infrastructure)[31], which would allow an easier deployment of protocols such as S-BGP, several issues slowed the adoption of a security protocol. Reasons may vary from protocol to protocol, but we summarize the three main deployment issues:

- The solution requires either a computational power or memory size that not all currently deployed BGP speakers will be able to withstand;
- The solution incurs changes to the BGP protocol currently in use;
- The solution does not bring immediate security benefits for the adopting AS;

The first item describes the problem in adding cryptographic mechanisms and other processing needs to current BGP speakers due to their computational power and memory requirements. Some BGP speakers across the internet do not meet those requirements and thus they would need to be replaced so the solution could be implemented.

In addition to having less memory and computational power than it would be required, speakers only have the standard BGP protocol installed and ready to use. As such, if a new and enhanced protocol was to be deployed, it would have to be replaced in all the deployed infrastructure.

Finally, most solutions provide security benefits only after a set of ASes adopt the protocol and each AS provides security for the other ASes instead of for itself.

Chapter 5

Securing BGP in SDN

The ease of innovation and deployment of new network protocols and architectures offered by SDN provides a clear alternative path to improve internet security. In this section we propose BGPSec[1], an SDN application that will run on the OpenDaylight controller prevent prefix hijacking attacks by providing prefix origination verification to the network.

The design of this solution aims to tackle two of the three main issues that prevented the deployment of BGP security solutions so far, by

1. Offloading the security task to the controller without switch intervention;

2. Requiring no changes to the BGP protocol;

Indeed, these two issues are solved by the adoption of a SDN architecture. First, security is provided through an application running on top of the controller instead of as a decentralized protocol running in switches. Hence, the additional processing is performed by the controller. Which, by residing in normal servers, typically has higher computational power and memory space than network devices. Second, by adding these features through a separate application, it is not required to change the protocol.

To prevent prefix hijacking we leverage on the existing RPKI infrastructure [31] and automate the process of real-time verification of IP prefix ownership. The RPKI is an infrastructure that results from the effort of the IETF SIDR group to build a trusted mapping from an IP prefix to the AS that is authorized to announce it. Currently, the RPKI only covers around 5% of existing ASes [7], but it is growing at a steady pace. For example, the number of objects in the RPKI has doubled in the last 18 months[9]. The use of an RPKI offeres the means to solve a part of current internet attacks, including the mentioned in the beginning of the previous chapter [33, 19, 41].

[1]https://github.com/tiagoposse/sdnbgpsec

5.1 Architecture

The architecture of the application we propose is represented in figure 5.1. The application is divided in two modules: database synchronization and packet verification. The first is responsible for synchronizing a local database with the global RPKI so that verifications can be done locally, avoiding the overhead of accessing the RPKI for each request. This module also provides an HTTP API for database lookups. The packet processing module is responsible for listening and processing BGP announcements and perform prefix verification in the local database.

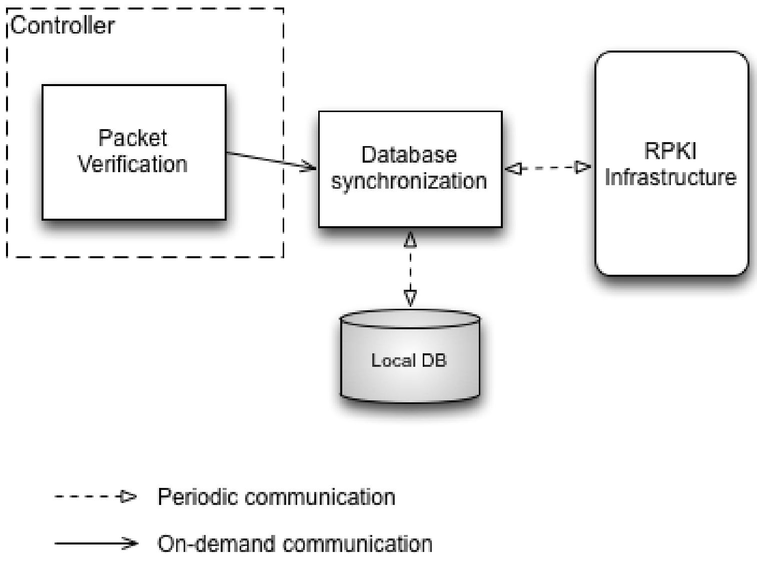

Figure 5.1: Overview of the BGPSec application

The application point of entrance is the packet processing module. Upon startup, BGPSec registers itself as a service in the OSGi framework (figure 5.2) for other modules to use, providing an interface with methods to validate packets.

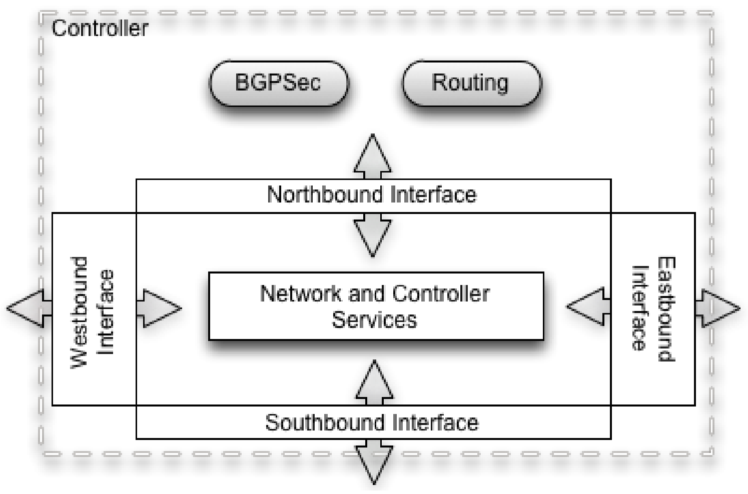

Figure 5.2: Overview of the BGPSec application

5.2 Implementation

The database synchronization feature is implemented using the RIPE-NCC RPKI validator tool [6], which acts on the global RPKI managed by the RIRs. The tool automatically accesses and downloads resources from the RPKI and stores them in a local database for faster access. Additionally, the validator tool has a sixty minutes time-to-live (TTL) on its database entries, after which it queries the global RPKI in order to update the database. With the objects added in the past 18 months, the update rate is roughly 1,2 updates every 2 hours, we decided one hour would be effective at keeping the database in sync, especially considering the default for the RPKI TTL is 3 hours.

The packet processing module is implemented in Java and has the following components:

- Handler: responsible for verifying if the message received is a BGP UPDATE and, if so, extracting the NLRI information and passing it to the MessageParser for validation.
- MessageParser: responsible for parsing and verifying the correctness of BGP UPDATE messages, called the RPKI to validate origin information

When the application receives a packet (Figure 5.3), it retrieves the contained BGP information and verifies if the message length is within bounds, after which the message is parsed and the cache is checked for the AS number and IP prefix pair contained in the message. If no result is found, the application performs an HTTP call to query the RPKI validator tool and returns whether the pair is a *VALID*, *INVALID* or *UNKNOWN* ROA. For a valid or an unknown result, the application finishes processing the packet and returns the result to the calling application. We allow unknown results to be return because the RPKI is still incomplete. In case of invalid result, the application returns invalid to the calling application, causing it to drop the packet. After the result is received, it is stored in the cache.

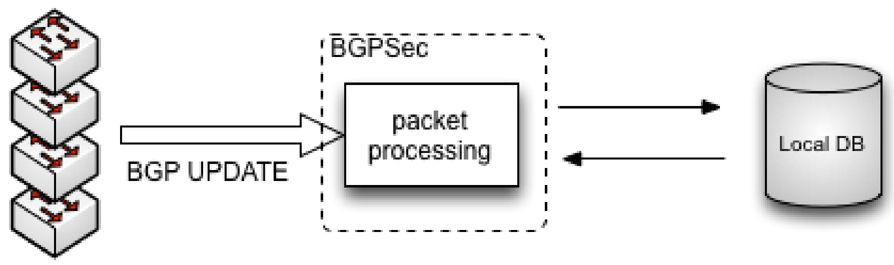

Figure 5.3: BGP UPDATE messages received by speakers are passed to the controller, which verifies its validity by checking the cache first and the local database if needed.

An IP prefix and AS number pair will be valid in case it is verified by one or more

ROAs in the RPKI or invalid in case there is at least one ROA that states the prefix as invalid. The result will be unknown if no ROA contains information about the pair.

Chapter 6

Evaluation

In this chapter we present and analyze functionality and evaluation tests made to both the RFProxy and BGPSec applications.

Tests were performed on an Intel Core 2 Duo CPU, running Ubuntu 12.04 (recommended by RouteFlow) with a 2.93 GHz CPU and 4 GB of memory.

6.1 RFProxy

We divide our tests in two parts to both assess the functionality of our implementation and provide statistics for evaluation purposes. For functionality tests, we run the available example topologies provided by the RouteFlow organization: *rftest1* and *rftest2* and use Mininet Hi-Fi to emulate a network using the same topology of each test to provide a data-plane layer for the controller to work on top of.

For evaluation purposes, we designed and ran a small, a medium and a large topology and tracked CPU and memory usage and the average time it takes for a flow to be configured following a packet_in event.

6.1.1 Testing

The rftest1 and rftest2 tests are provided along with the RouteFlow code for the RFServer and RFClients. Each test consists in a script and a set of configuration files that will setup the database, start the virtual hosts, define the mapping between clients and physical hosts and provide topology information for mininet to create an emulated network.

The first test creates a simple topology with two hosts connected by a single switch, while the second creates a more complicated topology (figure 6.1) using a set of four hosts and four switches.

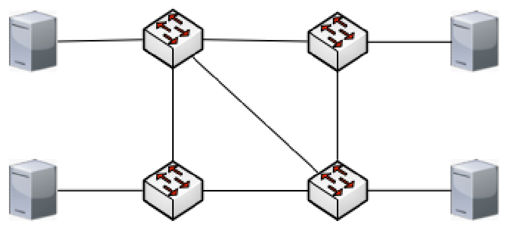

Figure 6.1: Small topology. The topology consists of 4 switches and 4 virtual hosts

Both tests were successful and resulted in a correct routing configuration where every host in the network was able to reach every other host.

6.1.2 Evaluation

Evaluating the performance of our RFProxy implementation required the use of the third-party tools *cbench*[1] and *YourKit* java profiler. Cbench provides a configurable benchmark platform that creates switches and uses them to send a variable amount of messages to the controller. YourKit provides statistics about CPU and memory utilization of a java application.

For the first test, we used the same topology of rftest2, while for the second test and third tests, we used a topology with 16 interconnected datapaths (figure 6.2), but mapped to different amounts of virtual hosts, taking advantage of RouteFlow ability to map several datapaths to a single virtual host. In this case, each virtual host represents four datapaths (figure 6.3). Finally, for the third test, we used the same 16 switches mapped to 8 virtual hosts (figure 6.4), to analyze the effect of growing the virtual network and compare the CPU and memory usage.

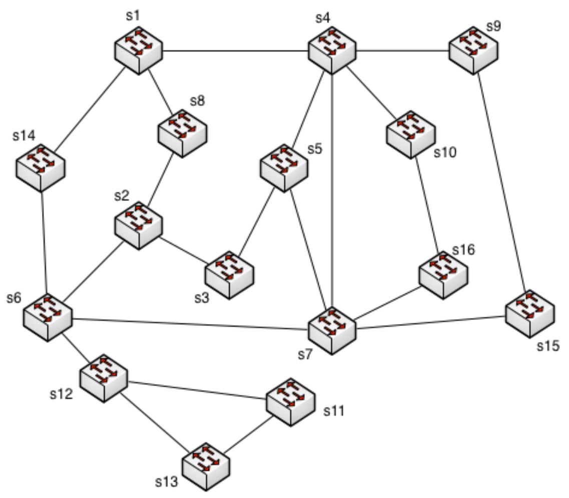

Figure 6.2: Large topo. Topology of the physical network for the second and third tests to RFProxy, composed of 16 interconnected switches.

[1]https://github.com/andi-bigswitch/oflops/tree/master/cbench

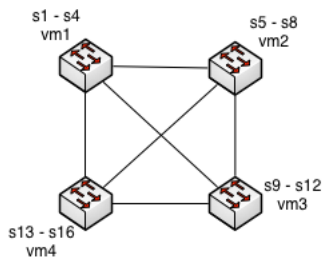

Figure 6.3: Topology of the virtual network for the second test, composed of 4 interconnect virtual hosts.

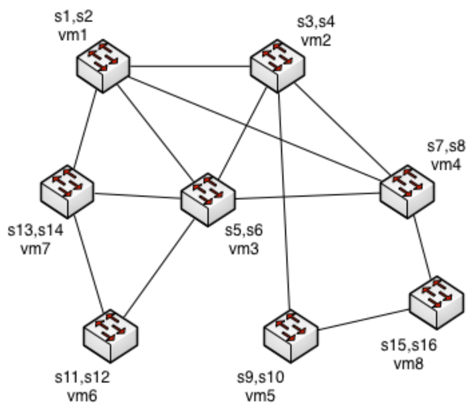

Figure 6.4: Topology of the virtual network for the third test, composed of 8 interconnect virtual hosts.

As we said, we used cbench to simulate the different scenarios and perform tests. Each test result depicts the moment from when cbench started, after which the network is subject to mapping by RouteFlow and the test begin as soon as a switch is correctly mapped. In Figures 6.5, 6.6 and 6.7 we present the results for the three tests made to RFProxy:

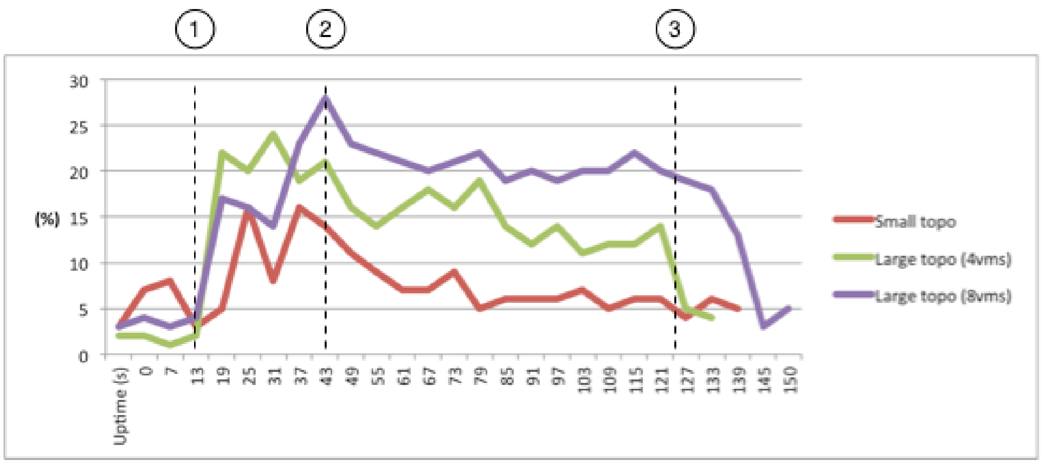

Figure 6.5: CPU utilization of the RFProxy tests.

The marks in the figure refer to the following events:

1. Cbench starts sending packets
2. The setup of the topology finishes
3. Test finish;

These results show that as we increase the number of datapaths and virtual hosts, the CPU usage increases as RFProxy is subject to higher loads, as expected. We were unable to test with more than 8 virtual hosts, as the computer did not withstand the interrupt load of the virtual hosts. In fact, it can be observed that the largest topology finishes the mapping between networks after the other two, due to the higher load caused by the virtual machines.

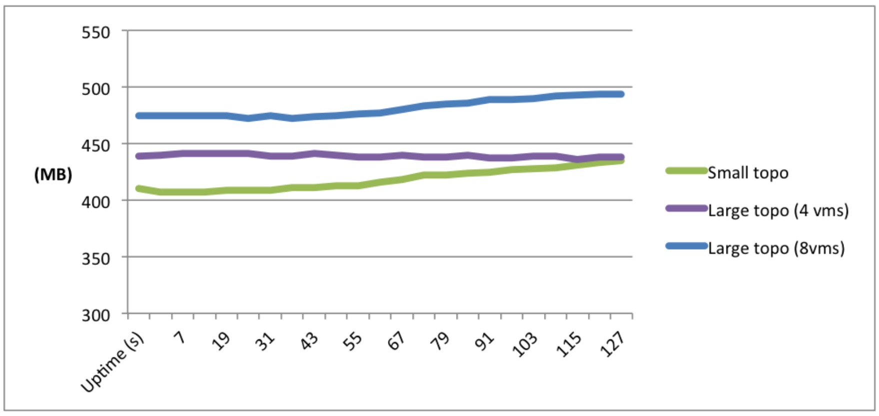

Figure 6.6: Memory utilization of the RFProxy tests.

Results for memory utilization are as expected, with the smaller topology achieving the lowest memory utilization, while the larger topology with 8 virtual hosts uses the most memory.

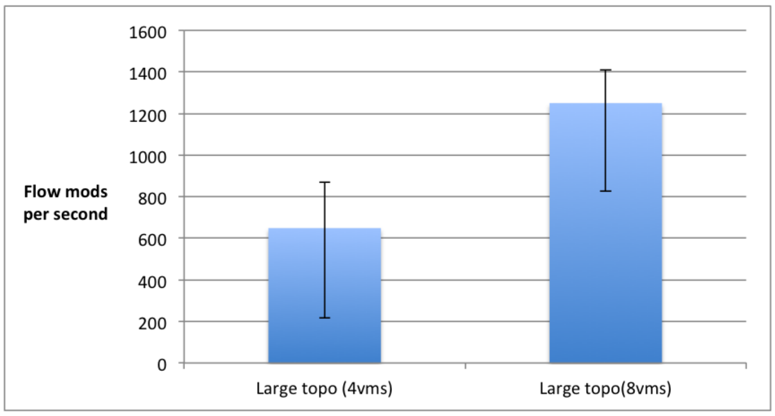

Figure 6.7: Comparison between the results of throughtput (latency mode) of the three tests.

In latency mode, each emulated switch maintains exactly one outstanding new flow request, waiting for a response before soliciting the next request. Latency mode measures the OpenFlow controller request processing time under low-load conditions, providing an indicator on how many transactions (packet_in + flow_mod) were generated in one second.

In RouteFlow, packet_in messages generate ARP requests and wait for the corresponding responses before sending flow_mods to the switches, adding a delay. The results show that for the large topology with four virtual hosts, the average number of flows_mod messages per second was 648, translating to $1 \setminus 648(*1000) \approx 1.54ms$ of latency. For the large topology with eight virtual hosts, the average number of flow mods per second was 1410, translating to $1 \setminus 1410(*1000) \approx 0.71ms$ of latency. We conclude that a larger virtual topology reduces the load on virtual hosts and thus improves the number of flow_mod responses (and ARP requests/responses) that each virtual host can process. In this case, the latency reduction factor was significant, approximately 2.17.

6.2 BGPSec

As with the RFProxy implementation, we divided our tests in two parts, one to assess BGPSec functionality and the other to evaluate its performance and overhead.

6.2.1 Testing

To test whether the application is working correctly, we introduced code snippets into RFProxy to call BGPSec, so that it processes and validates BGP messages before passing them to the virtual hosts for routing. We used the same topology as with the first test of RFProxy (figure 6.1), and used the bgpsimple[2] tool to inject BGP packets into the network. We injected an invalid, a valid and an unknown packet into the network and checked the results of the validations and whether the routes were configured in case the result was valid. We could verify that the invalid packet was dropped, while the valid and unknown packets were correctly parsed and the corresponding routes were configured.

6.2.2 Evaluation

We consider the overheads that the application may introduce in the environment and determined three metrics to evaluate: CPU and memory usage and average time per packet processing (the time it takes for the application to completely process a packet).

The original dataset used for tests was obtained from the RIPE NCC Routing Information Service[3], which stores real BGP data collected from several locations around the world.

[2]https://code.google.com/p/bgpsimple/
[3]http://www.ripe.net/data-tools/stats/ris

For our first evaluation test we used a single speaker to advertise 1000 UPDATE messages to the network. For the second evaluation test we used a single speaker to send 5000 UPDATE messages. Finally, for the last evaluation test we used three speakers, each sending 5000 UPDATE messages against the network.

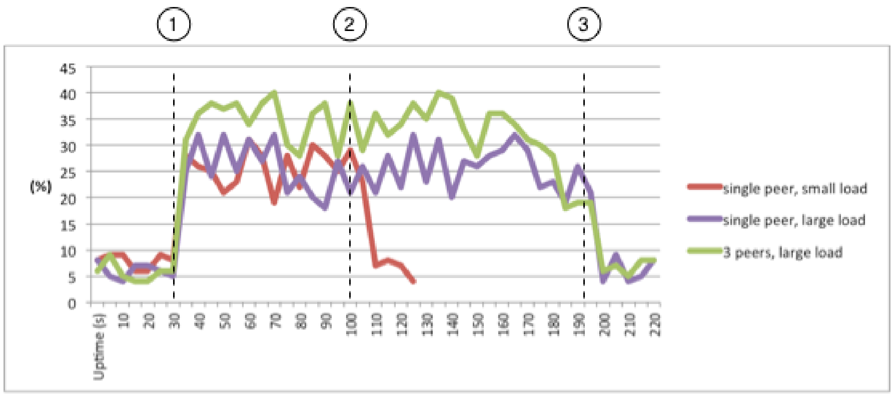

Figure 6.8: CPU utilization of the BGPSec tests.

The marks refer to the following events:

1. All tests start;
2. The singler-peer, small load test finishes;
3. The remaining tests finish;

Since we used a single advertising peer for tests one and two, the CPU usage (figure 6.8) is similar between them, as opposed to the third test. The use of additional peers caused an increase of CPU usage to a peak of 40%, while the first and second tests peeked at 32% and 31% respective, proving that additional peers will cause more load on the controller. The gradual drop of CPU usage in the third test is caused by the three peers not finishing at the same time, leading to a slight drop each time a peer finished its packet injection.

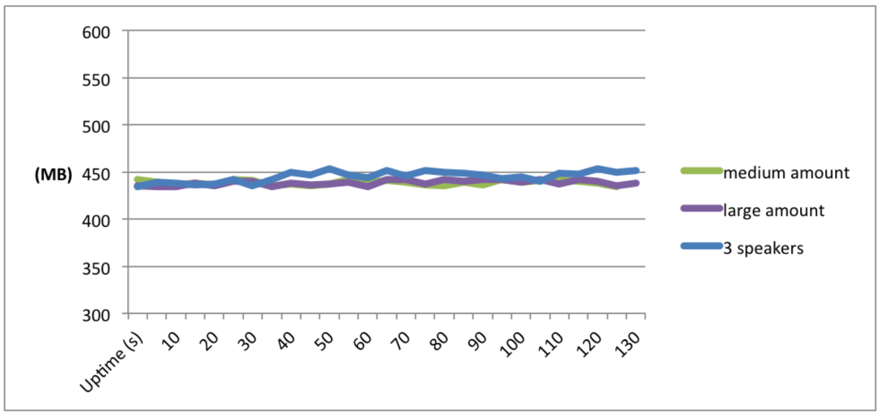

Figure 6.9: Memory utilization of the BGPSec tests.

The memory utilization results (figure 6.9) show no significant variation, hence memory is not the bottleneck. Indeed, as long as there is room for the RPKI data in the in-memory database, processing poses no additional overhead.

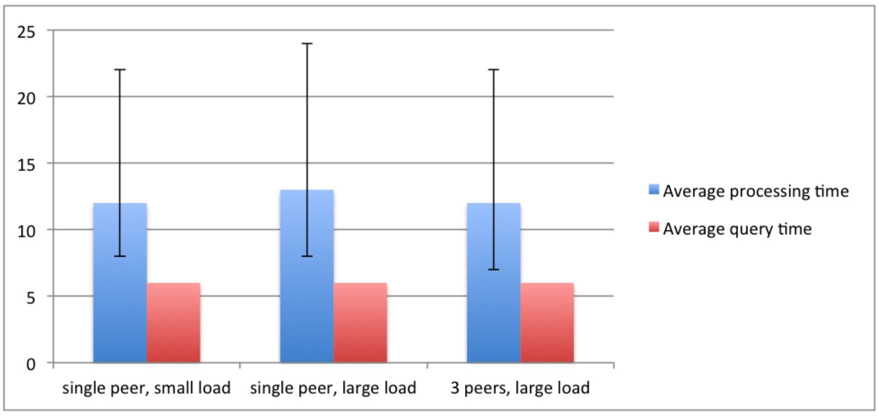

Figure 6.10: Comparison between the processing time results of the three tests.

As can be seen in figure 6.10, an average of 6ms was spent querying the RPKI to validate the BGP announcement just received. This represents, on average, half of the total processing time. Anyway, the the scale of the total processing time is still relatively insignificant between 10 and 15ms

Chapter 7

Final Remarks

7.1 Conclusions

In this work we propose two applications for the OpenDaylight SDN controller: RF-Proxy and BGPSec. RFProxy is a component of the RouteFlow routing platform which is responsible for integrating the platform with the controller and manage the physical infrastructure. To prove the feasibility and provide an evaluation of the application, we used a set of third-party tools to generate switch traffic in the network. The drawback for RFProxy is the CPU usage increase when the network is created and RouteFlow is trying to route and converge the hosts. However, this increase in CPU is temporary and will drop when the network converges, to a point where it is negligible.

We studied several BGP security issues and vulnerabilities and provided a clear view of several attacks and security solutions for the protocol. We further provide reasons for why these solutions have not yet been adopted based on our study and use these reasons as guidelines to propose a SDN security application: BGPSec. This application provides protection against IP prefix hijacking by using the existing global RPKI managed by the RIRs, enabling on-demand local verification of BGP information received from this trusted repository. Although the solution incurs minimal processing overheads, the main issue today is the completeness of the RPKI, which features only around 5% of existing ASes. Even though this number is still low, its rate of increase gives reason to suspect this solution will become more important in the future.

7.2 Future Work

As a future work, RFProxy should be extended to support the RouteFlow monitor component, a new feature that will eventually be implemented by RouteFlow, as well as the *PortRegister* and *PortConfig* messages. Additionally, recent more recent RouteFlow implementations include support for multiple controllers, which our implementation did not account for. *ElectMaster* and *ControllerRegister* messages should be implemented to support the existence and management of several controllers with RFProxy.

Additionally, we would like to extend BGPSec to also allow network administrators to define BGP preference rules based on the verification result instead of automatically dropping/accepting routes. This provides a more flexible route management. Finally, studying the impact of using a cache would also be interesting.

We are in the process of submitting RFProxy to the OpenDaylight project as an official project for the Service Provider edition.

Bibliography

[1] CIDR report as of 28 of July 2014.

[2] GNU Quagga Project. `http://www.quagga.org`.

[3] Open Networking Foundation. `https://www.opennetworking.org/`.

[4] Opendaylight technical overview. `http://www.opendaylight.org/project/technical-overview`.

[5] OSGi framework. `http://www.osgi.org/Main/HomePage`.

[6] RIPE NCC RPKI Validator. `http://www.ripe.net/lir-services/resource-management/certification/tools-and-resources`.

[7] RIPE Network Coordination Centre Statistics. `http://certification-stats.ripe.net/`.

[8] RouteFlow platform. `https://sites.google.com/site/RouteFlow`.

[9] RPKI Spider Deployment Growth. `http://rpkispider.verisignlabs.com/growth.html`.

[10] OpenDaylight: A Linux Foundation Collaborative Project. `http://www.opendaylight.org`, 2013.

[11] Project Floodlight. `http://www.projectfloodlight.org/floodlight/`, 2013.

[12] William Aiello, John Ioannidis, and Patrick McDaniel. Origin Authentication in Interdomain Routing. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 165–178, New York, NY, USA, 2003. ACM.

[13] T. Bates, R. Chandra, and E. Chen. BGP Route Reflection - An Alternative to Full Mesh IBGP, 2000.

[14] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[15] K. Butler, T.R. Farley, P. McDaniel, and J. Rexford. A Survey of BGP Security Issues and Solutions. *Proceedings of the IEEE*, 98(1):100–122, 2010.

[16] M. Caesar and J. Rexford. BGP Routing Policies in ISP Networks. *Netwrk. Mag. of Global Internetwkg.*, 19(6):5–11, November 2005.

[17] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[18] Danny Cooper, Ethan Heilman, Kyle Brogle, Leonid Reyzin, and Sharon Goldberg. On the Risk of Misbehaving RPKI Authorities. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 16:1–16:7, New York, NY, USA, 2013. ACM.

[19] J. Cowie. Rensys blog: China's 18-minute mystery. `http://www.renesys.com/blog/2010/11/chinas-18-minute-mistery.shtml`.

[20] Rohit Dube. A Comparison of Scaling Techniques for BGP. *SIGCOMM Comput. Commun. Rev.*, 29(3):44–46, July 1999.

[21] David Erickson. The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[22] Christoph Goebel, Dirk Neumann, and Ramayya Krishnan. Comparing Ingress and Egress Detection to Secure Interdomain Routing: An Experimental Analysis. *ACM Trans. Internet Technol.*, 11(2):5:1–5:26, December 2011.

[23] Sharon Goldberg, Michael Schapira, Peter Hummon, and Jennifer Rexford. How Secure Are Secure Interdomain Routing Protocols. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 87–98, New York, NY, USA, 2010. ACM.

[24] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[25] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[26] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[27] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway Protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, 2000.

[28] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[29] Diego Kreutz, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *To appear in Proceedings of the IEEE*, 2015.

[30] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[31] M. Lepinski and S. Kent. An Infrastructure to Support Secure Internet Routing, February 2012.

[32] Qi Li, Mingwei Xu, Jianping Wu, Xinwen Zhang, Patrick P. C. Lee, and Ke Xu. Enhancing the Trust of Internet Routing with Lightweight Route Attestation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 92–101, New York, NY, USA, 2011. ACM.

[33] M. A. Brown. Rensys Blog, "Pakistan hijacks YouTube.". `http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml`.

[34] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innova-

tion in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[35] S. Murphy. BGP Security Vulnerabilities Analysis, January 2006.

[36] Marcelo R. Nascimento, Christian E. Rothenberg, Rodrigo R. Denicol, Marcos R. Salvador, and Maurício F. Magalhães. RouteFlow: Roteamento Commodity Sobre Redes Programáveis. *XXIX Simpósio Brasileiro de Redes de Computadores - SBRC'2011.*

[37] Ola Nordström and Constantinos Dovrolis. Beware of BGP Attacks. *SIGCOMM Comput. Commun. Rev.*, 34(2):1–8, April 2004.

[38] Ricardo V. Oliveira. *Understanding the Internet As-level Structure*. PhD thesis, Los Angeles, CA, USA, 2009. AAI3384046.

[39] P.C. van Oorschot, Tao Wan, and Evangelos Kranakis. On Interdomain Routing Security and Pretty Secure BGP (psBGP). *ACM Trans. Inf. Syst. Secur.*, 10(3), July 2007.

[40] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. White paper, Open Networking Foundation, Palo Alto, CA, USA, April 2012.

[41] T. Paseka. Cloudflare blog: Why google went offline today. `http://blog.cloudflare.com/why-google-went-offline-today-and-a-bit-about`, November 2012.

[42] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771, IETF, March 1995.

[43] Christian E. Rothenberg, Marcelo R. Nascimento, Marcos R. Salvador, Carlos N. A. Corrêa, Sidney C. De Lucena, and Robert Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking," ser. *HotSDN '12. ACM*, pages 13–18.

[44] Lakshminarayanan Subramanian, Volker Roth, Ion Stoica, Scott Shenker, and Randy H. Katz. Listen and Whisper: Security Mechanisms for BGP. In *NSDI*, pages 127–140. USENIX, 2004.

[45] P.C. van Oorschot T. Wan and E. Kranakis. A Selective Introduction to Border Gateway Protocol (BGP) Security Issues. In *NATO Advanced Studies Institute on Network Security and Intrusion Detection*, Oct 2005.

[46] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[47] Allan Vidal, Fábio Verdi, Eder Leão Fernandes, Christian Esteve Rothenberg, and Marcos R. Salvador. Building upon RouteFlow: a SDN development experience. *XXXI Simpósio Brasileiro de Redes de Computadores - SBRC'2013*, 98, May 2013.

[48] R. White. Securing BGP through secure origin BGP (soBGP). *BUSINESS COMMUNICATIONS REVIEW*, 33(5):47–53, 2003.

[49] Edmund L. Wong, Praveen Balasubramanian, Lorenzo Alvisi, Mohamed G. Gouda, and Vitaly Shmatikov. Truth in Advertising: Lightweight Verification of Route Integrity. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 147–156, New York, NY, USA, 2007. ACM.

[50] Zheng Zhang, Ying Zhang, Y. Charlie Hu, and Z. Morley Mao. Practical Defenses Against BGP Prefix Hijacking. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 3:1–3:12, New York, NY, USA, 2007. ACM.