



Norwegian University of
Science and Technology

Key Management for Data Plane Encryption in SDN Using WireGuard

Anna Selvåg Braadland

Master of Science in Communication Technology

Submission date: June 2017

Supervisor: Stig Frode Mjøl̄snes, IIK

Co-supervisor: Christian Tellefsen, Thales Group

Norwegian University of Science and Technology

Department of Information Security and Communication Technology



NTNU – Trondheim
Norwegian University of
Science and Technology

Key Management for Data Plane Encryption in SDN Using WireGuard

Anna Selvåg Braadland

Submission date: June 2017
Responsible professor: Stig Frode Mjølsetnes, IIK
Supervisor: Christian Tellefsen, Thales Norway

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: Key Management for Data Plane Encryption in SDN
Using WireGuard

Student: Anna Selvåg Braadland

Problem description:

Through the introduction of Software Defined Networks, it is possible to realize highly dynamic networks. The architecture's main concept is decoupling the data plane and the control plane. The control functions are placed in a central entity in the network, the controller. This entity is defined in software and is provided with a global view of the network for managing flows.

Secure communication between controllers and switches in SDN is a solved problem, commonly using TLS. Protection of the data plane are however not yet fully investigated in the context of an SDN architecture. The architecture may be considerably more dynamic than traditional networks, and this thesis will investigate how this may affect key management. This text will propose a set of requirements for key management in SDN, and apply these to evaluate existing key management schemes.

Responsible professor: Stig Frode Mjølsnes, IIK

Supervisor: Christian Tellefsen, Thales Norway

Abstract

Software Defined Networks (SDNs) decouple the control plane and the data plane, congregating control functions in a designated entity in the network, the controller. The decoupling realizes a highly dynamic network which has benefits such as cost reductions and programmability. The OpenFlow protocol defines the communication between the controller and the network nodes. This thesis aims to construct a key management scheme for data plane encryption in SDN through an OpenFlow channel secured by TLS. Manual operations are usually embedded in key management solutions, but due to the dynamic nature of SDN, manual operations are not realizable in this architecture. Therefore, an automatic scheme needs to be designed.

Consequently, this thesis proposes a scheme for key management where the controller manages and initializes encrypted connections in the data plane. Encryption between the nodes is enabled by the newly developed secure tunneling protocol, WireGuard. The key management procedures are carried out through the secure OpenFlow channel. OpenFlow does not provide functionality for key management operations, and therefore this thesis proposes an extension to the protocol which facilitates this.

Sammendrag

Software Defined Networks (SDN) separerer kontrollplanet fra dataplanet, og samler kontrollfunksjonene i én enhet i nettverket, kontrolleren. Denne oppdelingen gjør nettverket svært dynamisk, noe som muliggjør reduserte driftskostnader og gjør nettverket programmerbart i sanntid. OpenFlow er protokollen som definerer kommunikasjonen mellom kontrolleren og nettverksnodene. Denne masteroppgaven vil konstruere et nøkkelhåndteringssystem for kryptering av dataplanet i SDN gjennom en OpenFlow-kanal sikret av TLS. Manuelle operasjoner er vanligvis en del av nøkkelhåndteringsløsninger, men grunnet SDNs dynamiske natur, er ikke manuelle operasjoner realiserbare i denne arkitekturen. Derfor trengs et automatisk nøkkelhåndteringssystem i SDN.

Denne masteroppgaven foreslår et system for nøkkelhåndtering hvor kontrolleren håndterer og starter kryptert kommunikasjon i dataplanet. Kryptering mellom nodene blir gjennomført av den nylig utviklede tunnelleringsprotokollen WireGuard. Prosedyrene for nøkkelhåndtering er utført gjennom den sikre OpenFlow-kanalen. OpenFlow tilbyr ikke funksjonalitet for håndtering av nøkler, og derfor har det blitt utviklet en utvidelse av protokollen spesifikt for nøkkelhåndtering.

Preface

This thesis is submitted at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU). The thesis is presented as the final project for the MSc program in Communication Technology with specialization in Information Security. The study was performed over 20 weeks, Spring 2017.

I would like to thank my supervisor, Christian Tellefsen, for valuable guidance and support and for being available for addressing problems I encountered during the development. I would also like to thank my responsible professor, Stig Frode Mjølunes, for important feedback and advises in the writing of the thesis.

Furthermore, I would like thank my three golden nuggets without whom this would have been a much lonelier work.

Finally, thanks to my very best friend, Tuva Eide, for providing support and laughter and for reading every word of this thesis.

Anna Selvåg Braadland

Trondheim, June 12th 2017

Contents

List of Figures	xi
List of Tables	xiii
Glossary	xv
List of Acronyms	xix
1 Introduction	1
1.1 Motivation	2
1.2 Scope and Objectives	3
1.3 Methodology	4
1.4 Novelty	5
1.5 Outline	5
2 Background and Related Work	7
2.1 Software Defined Networking (SDN)	7
2.2 OpenFlow	10
2.3 Cryptography	12
2.3.1 Cryptographic Keys	12
2.4 Key Management	14
2.4.1 Phases in Key Management	14
2.4.2 Architectures	15
2.4.3 Implementations of Key Management Schemes	17
2.5 Applied Technologies	19
2.5.1 WireGuard	19
2.5.2 Open vSwitch	22
2.5.3 Floodlight	23
2.6 Technical Assessment	23
2.7 Related Work	23
3 Key Management in SDN	25
3.1 Characteristics of SDN	25

3.2	Assumptions	26
3.3	Requirements	27
3.3.1	Cryptographic Requirements	27
3.3.2	Operational Requirements	28
4	Data Plane Key Management Scheme (DPKMS)	31
4.1	Topological Design	32
4.1.1	Controller	32
4.1.2	Node	34
4.1.3	Application	34
4.2	Interfaces	35
4.2.1	Application - Controller	35
4.2.2	Controller - OpenFlow Switch	35
4.2.3	OpenFlow Switch - WireGuard	35
4.2.4	Node A - Node B	36
4.3	Detailed Description of the Data Plane Key Management Scheme (DPKMS)	36
4.3.1	Configure WireGuard Interface	37
4.3.2	Add a Peer to the Node	39
4.3.3	Start Communication	40
4.3.4	End Communication	41
4.3.5	Rekeying	44
4.3.6	Revocation of Keys	44
4.3.7	Status Message and Structure	46
4.3.8	Error Message and Structure	47
4.4	Extensions Needed in OpenFlow	48
5	Extending OpenFlow	51
5.1	OpenFlow Experimenter Structure	52
5.1.1	Experimenter in Practice	53
5.1.2	Experimenter Message	53
5.2	Extensions to OpenFlow	54
5.2.1	Experimenter Attributes	55
5.2.2	Experimenter Messages	61
6	Analysis and Discussion	65
6.1	Requirement Analysis	65
6.1.1	Cryptographic Requirements	65
6.1.2	Operational Requirements	67
6.2	Design Decisions	70
6.2.1	Tunnel Between the Nodes	70
6.2.2	Storage of Keys	71

6.2.3	Key Generation	72
6.2.4	Which Entity Initiates Encryption	73
6.2.5	Choice of Control plane Protocol	74
6.2.6	Revocation of Keys	75
6.2.7	Key Distribution	75
6.2.8	How to be informed of the state of the node	76
6.3	Known Limitations	77
6.3.1	WireGuard is in an Experimental Stage of Development	77
6.3.2	One Key Pair is Used as Long-term Keys for All Communication	77
6.3.3	The Extension Only Supports One Key Size	77
6.3.4	The Scheme Does Not Specify Error Handling in the Controller	78
6.3.5	The Scheme Only Includes Two Nodes	78
6.3.6	WireGuard Only Encrypts Layer Three Data	78
6.3.7	Private Keys are Transported Unencrypted Within a Node	78
6.3.8	At One Point the Controller Possesses Private Keys for the Nodes	78
6.3.9	Queuing of Packets Can Cause Delay and Loss	79
7	Conclusion	81
7.1	Future Work	82
	References	83

List of Figures

1.1	Reference model for the scheme to be developed in the thesis, Data Plane Key Management Scheme (DPKMS).	3
2.1	The overall architecture of SDN, illustrating the entities in a network with applications, a controller and nodes. The interfaces connecting the different entities are also shown.	8
2.2	The data plane and control plane in traditional networks.	9
2.3	Illustration of symmetric key cryptography where two parties are using the same key for encryption and decryption.	12
2.4	The operation of a public key algorithm where a key pair, consisting of a public key and a private key, are used for encryption and decryption respectively.	13
2.5	Screenshot of the configuration file showing the simple setup of a WireGuard tunnel in one of the peers.	20
2.6	Screenshot of how a new WireGuard interface (<code>wg2</code>) is added and thereafter how a link is established.	21
2.7	The configuration information in the nodes and the steps for setting up an encrypted tunnel through WireGuard.	21
2.8	Screenshot showing a WireGuard interface and a configured peer through the use of the <code>wg</code> utility.	22
4.1	High level architecture of the model developed in this thesis.	32
4.2	Controller illustrated with its key management extensions which shows two tables, one displaying the peers communicating, while the other contains all configured peers.	33
4.3	State transition diagram explaining states and transitions with regards to the node involved in DPKMS. The circles represent states and the arrows represent the transitions.	36
4.4	Sequence diagram that shows the procedure for configuring a node in the case where an application starts the procedure and specifies the cryptoperiod. <i>WG</i> is short for WireGuard and <i>OF</i> is short for OpenFlow.	38

4.5	Sequence diagram with the general flow in the procedure of adding a peer to the WireGuard interface.	40
4.6	Sequence diagram showing the flow in the procedure of starting the communication followed by one packet being sent from Node A to Node B.	41
4.7	Sequence diagram showing the two different ways of ending the secure communication through WireGuard. Either by redirecting flows through the default port and not through WireGuard, or by completely shutting down all communication to the respective peer.	43
4.8	Sequence diagram showing the rekeying procedure where the controller initiates the rekeying when the cryptoperiod has elapsed.	44
4.9	Sequence diagram displaying how the messages flow in case a third party – for example an intrusion detection system – instructs the controller to revoke the keys for node A. The application is thereafter able to choose how to handle the revocation.	45
4.10	Message sequence diagram showing the flow when the controller issues a <code>get_status_wg</code> to be informed of the state of the WireGuard port, which is returned to the controller in a bitmap in the <code>status_wg</code> message. . .	47
5.1	The messages extending the OpenFlow protocol for key management, displayed as transitions. The messages are shown together with their associated responses displayed in a state transition diagram. The states illustrate the state of the controller that issues the different messages. .	51

List of Tables

2.1	Flow table entries in an OpenFlow switch.	10
4.1	The statuses that can be achieved in the WireGuard interface.	46
4.2	Overview of error messages needed in addition to standard errors issued by the OpenFlow protocol.	48
4.3	Table of all messages needed in the OpenFlow protocol to implement DPKMS together with an explanation of each of the messages.	49
4.4	Table describing the attributes that need to be defined in order to maintain the information needed in DPKMS.	49
5.1	The structure of an Experimenter header.	52
5.2	The structure of the key TLV where each field specifies its length.	55
5.3	The structure of the status TLV for a node containing only one peer in its list of peers	57

Glossary

Application Programming Interface	An interface with clearly defined methods for communication with a program.
Bitstring	A sequence of 0's and 1's to be interpreted by a computer.
Ciphertext	A bitstring produced from a cryptographic algorithm that is not revealing its original content.
Control Plane	The part of the network that controls and manages the traffic forwarding.
Controller	A central entity in a network responsible for managing and controlling the flow of the network while being connected to all network elements.
Cryptographic Primitive	Low level cryptographic algorithms used for designing protocols for ensuring security.
Cryptoperiod	The time span a key is valid for before it needs to be renewed.
Data Plane	Also called forwarding plane, is the part of the network that transport traffic from users.
Datapath	A specific part of the data plane which connects two network entities.
Decryption	The process of turning a ciphertext into readable information with the use of a cryptographic algorithm and a key.
Encryption	Turning plaintext into ciphertext with the use of a cryptographic algorithm and a key.

Flow	A set of packets that are routed the same way through the network in the scope of OpenFlow and SDN.
Key	Or a cryptographic key is a bitstring that is used by an cryptographic algorithm to determine the output; the ciphertext..
Keying Material	Bistrings that are used for generating a key, for example a random bitstring.
Keypair	A public key and a corresponding private key generated by a cryptographic algorithm.
Network Element	An entity in a network able to forward traffic in the data plane and that is connected to a controller that manages the flow.
Perfect Forward Secrecy	A property where revealing long-term keys does not lead to the revelation of used session keys.
Plaintext	Data in its original form, unencrypted, to be transformed in a cryptographic algorithm.
Rekeying	The process of renewing a key after its cryptoperiod has expired.
Software Defined Network	A network architecture where the control plane is decoupled from the forwarding plane and the network is controlled from a central entity.
Tunnel	In the context of networking, a tunnel is a connection that supports another protocol than the underlying protocol does by encapsulating packet sent through the tunnel.

List of Acronyms

AH Authentication Header.

API Application Programming Interface.

CA Certification Authority.

DH Diffie-Hellman.

DoS Denial of Service.

DPKMS Data Plane Key Management Scheme.

ECDH Elliptic Curve Diffie Hellman.

ESP Encapsulating Security Payload.

IKE Internet Key Exchange.

IP Internet Protocol.

KDC Key Distribution Center.

KTC Key Translation Center.

LTE Long Term Evolution.

OVS Open vSwitch.

OXM OpenFlow Extensible Match.

PKI Public Key Infrastructure.

PoC Proof of Concept.

QoS Quality of Service.

RA Registration Authority.

RBG Random Bit Generator.

RPC Remote Procedure Call.

RTT Round Trip Time.

SA Security Association.

SDN Software Defined Network.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TLV Type Length Value.

TTL Time To Live.

UDP User Datagram Protocol.

VPN Virtual Private Network.

Chapter 1

Introduction

Software Defined Networking (SDN) is a paradigm which enables simpler and faster innovation for network architectures. It differs from traditional networks where control functions are defined in hardware within the nodes, as it defines control functions in software. This shift allows for a simplified and programmable approach to networking, enabling the rearrangement of the virtual network topology in real time. SDN introduces a centralized approach where a designated entity – the controller – has congregated the control functions traditionally found in the network nodes. This allows the controller to have a global view of the network and the ability to automate operations to improve performance. Applications can access the controller through open APIs and can express granular policies. Innovating network processes will thereafter become simpler because networks can be easily orchestrated by multiple vendors.

OpenFlow is the most commonly used protocol standardizing the communication between the controller and the nodes in SDN. It can be run securely over TLS, ensuring encryption and authentication of the nodes. Furthermore, it decides the possibilities of configurations that can be done to nodes in SDN, and has the ability to be extended in order to introduce new functionality.

Network security is a topic which is increasingly gaining interest. Encryption can be applied in different layers in the network, and the focus of this thesis is to encrypt the traffic between the nodes in the network. Typically, secure network tunnels are configured manually, raising the bar for implementing security in the network.

The demand for flexible solutions enabling encrypted data transmission is putting pressure on network development. Traditional networks are built up of static hardware appliances, which limit innovation and opens up for new approaches such as SDN. This thesis aims to make use of the benefits of SDN in order to facilitate encrypted data transmission between nodes in a network. A difficult problem for securing data transmission relies on management of keys which commonly includes manual

operations. Manual operations in the data plane of SDN cannot be realized, seeing as the architecture is highly dynamic. Therefore, an automatic key management scheme is essential in order to facilitate encryption of the data plane in SDN.

The problem to be solved in this thesis is *how to achieve a key management scheme for data plane encryption in SDN through an OpenFlow channel secured by TLS*. The thesis aims to develop a scheme for key management in SDN, hereafter referred to as DPKMS. The focus lies in automating processes where the controller centralizes and initializes the key management functions. Additionally, a protocol extension to OpenFlow will be formulated in order to include support for key management.

1.1 Motivation

Due to the increased focus on digitalization on a global scale, network security is gaining importance. At the core of network security is cryptography; hiding messages with the use of algorithms which enables only parties with specific information, i.e. keys, to read it. A challenge when it comes to encryption is the way in which these keys are managed, that is, how to efficiently and securely handle processes related to their lifecycle.

SDN architecture is currently being introduced to a growing number of data centers and enterprises worldwide [SDN16]. The architecture brings new possibilities to security research and this thesis aims to develop a key management scheme for encrypting the data plane in SDN, utilizing these benefits. A key management scheme implemented in SDN would imply encryption between nodes connected to a controller. Following is a presentation of scenarios in which it would be beneficial to enable secure traffic in this part of the network.

Today, SDN is being deployed in closed data centers which isolates data traffic within the network, for instance in order to offer cloud services. However, if the SDNs were to send data over unsecured networks such as the Internet, it would be useful to encrypt traffic from the nodes connected to the insecure network. Moreover, if two secure SDNs were located in different geographical areas, traffic transmitted through insecure transport networks should be encrypted.

Another possible scenario is SDN in wireless networks, which is still in an early stage of development [BQCM⁺16, CGMP12]. Encryption of wireless transmission between nodes is strongly recommended seeing as the medium allows for easy eavesdropping of the traffic to take place. Therefore, managing keys in order to encrypt the data plane would be beneficial to the network.

Furthermore, encryption could be difficult to implement in some end-devices such as light-weight embedded IoT devices [PGC⁺14, Sch14]. A possible solution to this

would be to extract the encryption process from the devices and to offer encryption of the traffic from the network nodes instead.

Finally, SDN could be incorporated into a network of a large company with many departments. All of the departments would share the same infrastructure, where some departments might want their traffic protected against other users. Hence, encryption of the data plane in SDN would allow for specific flows – similar network traffic routed the same way – to be isolated from the rest of the traffic. This would make the traffic inaccessible to others.

1.2 Scope and Objectives

The goal of this thesis is to develop a scheme for management of keys in the Data plane in SDN where the keys are used by WireGuard, a secure network tunnel. A secure OpenFlow channel exists between the controller and each of the nodes, and it is protected by Transport Layer Security (TLS). WireGuard is responsible for key exchange and encryption of the connection between two network nodes.

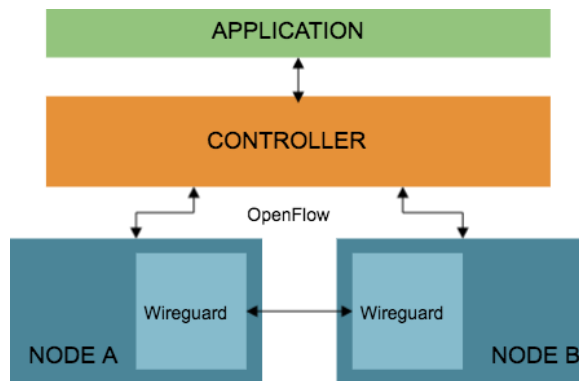


Figure 1.1: Reference model for the scheme to be developed in the thesis, DPKMS.

Figure 1.1 shows the reference model for the scheme to be developed in this thesis, referred to as DPKMS. The scope encompasses communication between the controller and the nodes, in addition to the communication between the nodes. However, all communication with and implementation of the application is beyond the scope of this thesis.

The main objectives for this thesis are as follows:

- Propose a scheme for management of keys in the dataplane of SDN
- Formulate an extension to the OpenFlow protocol in order for the key management scheme to be implemented
- Analyze the proposed scheme according to a set of established requirements

1.3 Methodology

In the initial phase of the work behind this thesis, the research topic was presented as *Securing Dynamic Software Defined Networks*. The topic was extensive and could therefore possibly lead in multiple directions with regards to method and results. Key management in the data plane was chosen as the main focus, as there exists little research on this related to SDN.

An Internet Draft proposing a model for key management in the data plane of SDN was discovered [MLLM16]. This model implemented key management for a protocol suite called IPsec (2.4.3), which will be presented in the section for related work (2.7). It used the control plane protocol NETCONF, and the initial intention was to conduct a proof of concept the model. However, when the model had been thoroughly investigated, the task proved to be too complex.

Consequently, a reference model for the thesis was processed and a decision was made to use OpenFlow (2.2) as the control plane protocol, discussed in section 6.2. No other attempts had been found to include key management in OpenFlow, which would involve extending the protocol. A theoretical rather than a practical approach was chosen due to the novelty and complexity of the task. As a result of this, a thorough study of the technologies and how to schematically do key management in SDN was conducted.

To determine which technologies to focus on, technical assessment was carried out – presented in section 2.6. After testing, WireGuard – a secure network tunnel (2.5.1) – was found to be the most suitable choice for data plane encryption, due to its simplicity which allowed for significant focus on the scheme.

Following the decision to design a scheme for key management, a set of requirements were made. Appropriate requirements were made by examining standards and protocols for key management and were made in order to enable the assessment and analysis of the produced scheme.

Thereafter, the scheme was designed. Processes needed in the scheme were identified by examining the requirements. Difficulties encountered were related to

synchronization. Therefore, a state transition diagram illustrating the states of a node was made. Each process was then made by illustrating message flow in message sequence diagrams. Finally, each of the processes were assessed by using the requirements previously made.

1.4 Novelty

The novelty of this thesis is the design of a scheme for key management through an encrypted OpenFlow channel in SDN. Previous studies – to be presented in section 2.7 – have researched encryption in the data plane in an SDN approach where a central entity in the network manages the keys. However, including key management as a part of the established OpenFlow control plane has not – to the author’s knowledge – previously been realized. This thesis includes both the scheme for key management between the controller and nodes in a network as well as an extension to OpenFlow.

1.5 Outline

Chapter 2 – *Background and Related Work* – will explain the background information needed in order to understand the research field on which the thesis is based. Central technologies such as WireGuard, Open vSwitch and Floodlight are then explained, followed by a presentation of technical assessment made prior to designing the scheme. The chapter closes with work related to the thesis.

Chapter 3 – *Key Management in SDN* – presents preliminaries with regards to the scheme that is designed, DPKMS. Two characteristics distinguishing the key management in SDN from traditional networks are highlighted, together with the assumptions made prior to designing the scheme. Lastly, the requirements to be used for assessing the scheme are stated.

Chapter 4 – *Data Plane Key Management Scheme (DPKMS)* – presents the scheme developed for solving key management in SDN, DPKMS. The chapter starts by going through each entity making up the model and the interfaces connecting each of them. Thereafter, each process in the scheme is designed. Finally, the chapter presents extensions to the OpenFlow protocol.

Chapter 5 – *Extending OpenFlow* – presents the extensions made to the OpenFlow protocol. Firstly, the Experimenter structure in OpenFlow is presented, followed by a section for each of the messages and thereafter the attributes defined in the experimenter structure.

Chapter 6 – *Analysis and Discussion* – presents a discussion and an analysis of the developed scheme. In addition, alternative ways of implementing the scheme are

examined in order to justify the design of DPKMS. Thereafter, limitations to the scheme are presented.

Chapter 7 – *Conclusion* – summarizes the contribution of this thesis and gives directions with regards to future work to extend the thesis.

Chapter 2

Background and Related Work

2.1 Software Defined Networking (SDN)

In essence, the concept of Software Defined Network (SDN) is the decoupling of the control plane and the data plane [Fun12]. The control functions from the control plane are assembled in a central entity, called *controller*, which manages the network. The controller is defined in software and captures state information from the network nodes in order to have an updated view of the network at all times [GB14]. This enables the controller to make routing decisions based on a global view of the network [BEIE15]. Centralizing the controller makes it easier to define and upgrade policies, as well as the abstraction simplifies the network equipment.

The concept is illustrated in figure 2.1, where a controller is connected with a path to each of the nodes in the network. Additionally, the figure shows applications that can be built on top of the controller. These applications have the ability to directly program the network topology in real time. The connections between the controller and the nodes are part of the *control plane*, and the most common protocol for the SDN control plane is OpenFlow, thoroughly explained in section 2.2. *Data plane* traffic passes through the network nodes where the path is orchestrated by the controller.

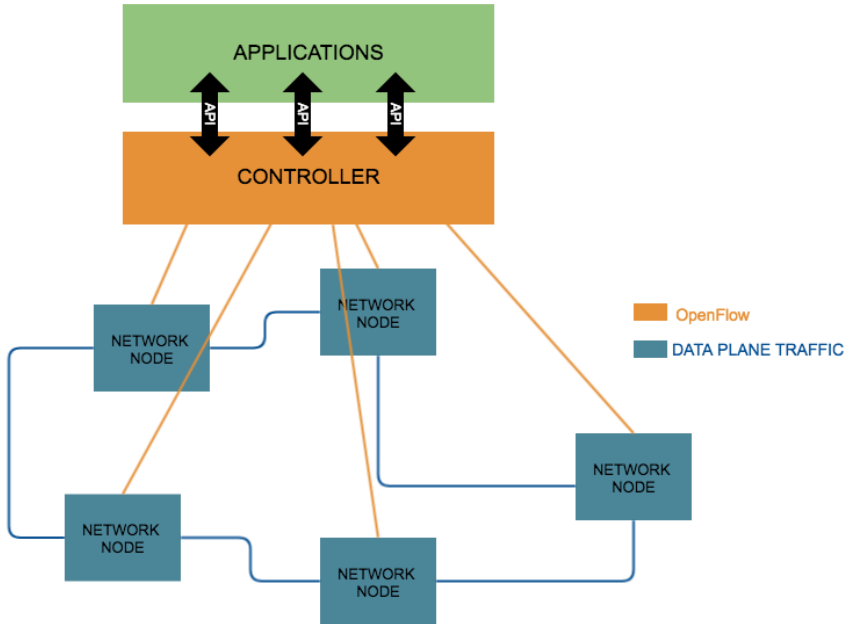


Figure 2.1: The overall architecture of SDN, illustrating the entities in a network with applications, a controller and nodes. The interfaces connecting the different entities are also shown.

Traditionally, network entities consist of a control plane and a data plane implemented in static hardware appliances, as illustrated in 2.2. Packets in the network are routed through these entities by looking up in the routing table which is configured according to a protocol. While the protocols are static, the applications on top of the network are dynamic, making the architecture difficult to scale [CTHN16]. New functionality and small changes to protocols in traditional networks have to be deployed in the hardware which may be a time-consuming task. Traditional networking is unable to respond fast enough to rapid network changes. The architecture was practical when client-server computing was dominant, but as a result of the emergence of dynamic computing and new storage requirements, traditional architecture is unable to maintain the same rate of progress as SDN.

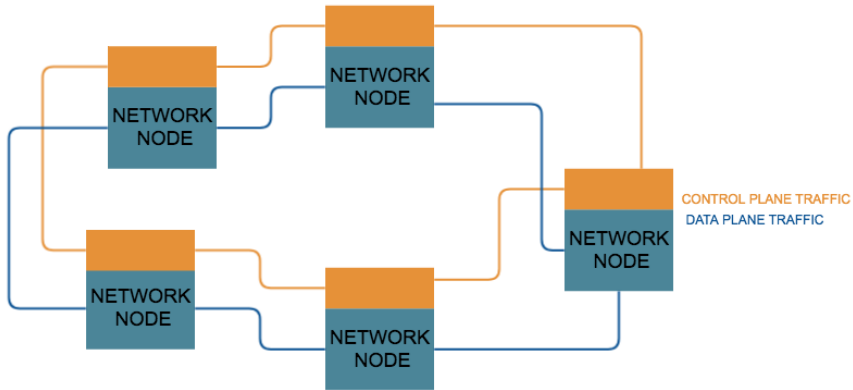


Figure 2.2: The data plane and control plane in traditional networks.

The development of the SDN architecture started when the protocol OpenFlow, discussed in section 2.2, was released in 2009 [Ope15]. In 2011, the architecture gained salience, resulting in more research on the topic being carried out [GB14]. Motivation for the heightened interest included benefits such as cost reductions and openness leading to a higher rate of innovation. Furthermore, the increased volumes of data being processed in data centers has been a driving force for the popularity of the SDN paradigm. However, during the past few years the development rate of SDN technology has stagnated [SDN16]. As stated in *OpenFlow and SDN, State of the Union*, provided by Open Networking Foundation, the lack of interoperability and features yet made for SDN is considered the cause of this stagnation. Nevertheless, one assumes that with the ratification of SDN technology by leading companies within the networking industry, the technology will continue to evolve.

2.2 OpenFlow

OpenFlow is a communication standard for SDN[Ope15]. While other protocols exist for SDN’s control plane, OpenFlow is most commonly used. OpenFlow defines a protocol for sending messages between the controller and the switch. An OpenFlow switch implements the protocol and lets a controller administer the flow tables accordingly.

OpenFlow Messages

OpenFlow messages are sent over Transmission Control Protocol (TCP) which provides reliable message delivery [Ope15]. OpenFlow messages fit 64KB of data in one message, and a Type Length Value (TLV) element can be added to support additional and variable data within a packet. The protocol operates with three types of messages – asynchronous, controller-to-switch, and symmetric – each with multiple subtypes. The asynchronous messages are sent from the switch and can be packets for updating the controller or they can be `PacketOut` messages. `PacketOut` messages are sent by a switch containing a packet that either does not have a match in the flow table or does not match with an action telling the switch to encapsulate the packet and send it to the controller. The controller-to-switch messages are initiated by the controller and concern messages sent from the controller to the switch. These are related to modifications to a flow table, or a `PacketIn` message. A `PacketIn` message is usually a response to a `PacketOut` message which contains the same packet and related actions that the controller wants to perform on it. Symmetric messages are usually messages used for setup or for system checks.

Match field	Priority	Counters	Instructions	Timeouts	Cookie
-------------	----------	----------	--------------	----------	--------

Table 2.1: Flow table entries in an OpenFlow switch.

Switch Architecture

The main components of an OpenFlow switch are a secure channel connecting the switch to the controller, the packet processing pipeline and ports [GB14]. The OpenFlow channel supports TLS-based two-way asymmetrical encryption, but this is not mandatory [Ope15]. The secure channel has the ability to receive packets from the switch to send it to the controller and to let the controller send packets to the switch.

An OpenFlow Switch contains several flow tables that implement the core function of the switch, which is to handle incoming packets [Ope15]. The packet processing pipeline determines how to handle incoming packets. The flow tables contain several

entries for defining a flow – a set of packets that are routed the same way – shown in table 2.1. The first field in a flow table entry – the match field – tells which packets will be routed by this entry. An example of this is matching flows with the same IP-destination address, type of packet or header. If no entries match, the “table-miss” entry decides what to do with the packets. The priority field is distinguishing between packets that match several entries. The packets are matched with the entity with the highest priority and counters are used for tracking statistics in the network. Further, the instruction field decides how the packet is processed. It can either be a set of actions – such as drop, forward, encapsulate or send to the controller – or it can be a pipeline processing instruction for instance when forwarding the packet to the next flow table.

Ports in an OpenFlow Switch can either be a physical port for the device or a virtual port[GB14]. Each port contains queues corresponding to different Quality of Service (QoS) for the packet processing pipeline to choose for outgoing packets.

The Controller’s Role in OpenFlow

A controller that implements the OpenFlow protocol can add packets to the switches using two different modes; proactively and reactively [Ope15]. Reactively adding flow table entries means that packets that do not match any field in the flow tables will be sent to the controller. Here, the controller decides what will be applied to the packet. An example is making a new entry in the flow table. In proactive mode, the controller will transmit the rules before starting, and packets that do not match any fields will be dropped if nothing else is specified in the "table miss" entry. The network can be defined as having multiple controllers working together to achieve a more stable and reliable network. This is achieved through supplementary controllers doing load balancing and being able to take over the control when a controller fails.

State of the art

OpenFlow was, as stated in section 2.1, the driving force behind SDN, ready for early deployment in 2009. Open Networking Foundation (ONF) was established in 2011 by leading operators – for instance by Google, Facebook and Microsoft – to be responsible for the OpenFlow standard [GB14]. As mentioned, a considerable amount of research on the OpenFlow protocol was conducted in the early phase, but the interest has decreased the last couple of years. The challenge of introducing OpenFlow is that both hardware and software needs to guarantee inseparability [SDN16]. However, as the protocol remains the most widespread and acclaimed protocol for SDN, the interest will persist as long as the evolution continues.

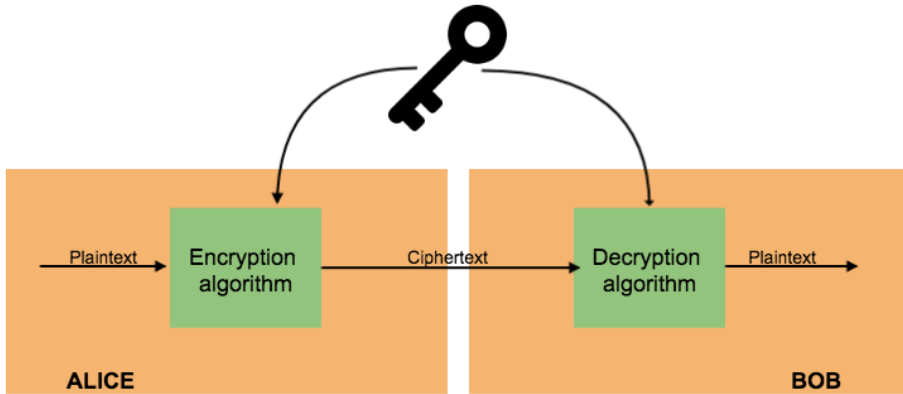


Figure 2.3: Illustration of symmetric key cryptography where two parties are using the same key for encryption and decryption.

2.3 Cryptography

To comprehend how a key management scheme is built up, a basic understanding of cryptography is needed. Consequently, the following section will provide a further explanation of the concepts with regards to cryptographic keys.

The basic concept of encryption is that two parties wish to share a secret, without anyone else being able to understand it. This is done by running it through a cryptographic algorithm, illustrated in figure 2.3. For the other party to be able to extract the secret, the algorithm is reversible if it is exposed to a specific key. The figure illustrates how the plaintext – the secret to be shared – and a key are inserted in a cryptographic algorithm, calculating the ciphertext, the hidden secret. Thereafter, the ciphertext can be supplied in a decryption algorithm together with a key in order to obtain the plaintext.

2.3.1 Cryptographic Keys

From a key-management perspective, there are two types of keys used for ensuring security, namely long-term keys and session keys. Long-term keys are generated to last for a relatively long period in an entity and are commonly not used directly for encryption or decryption. Session keys are often generated using information from the long-term key and are used to encrypt a specific session. For long-term keys, there are two main categories of implementation; either the two parties share

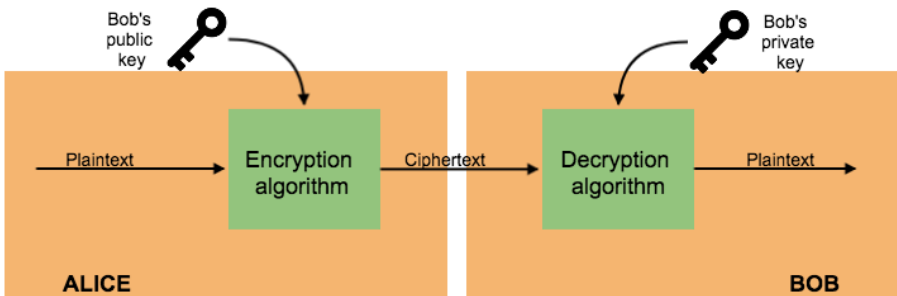


Figure 2.4: The operation of a public key algorithm where a key pair, consisting of a public key and a private key, are used for encryption and decryption respectively.

a *symmetric key* or the algorithm is made so that it has one encryption key and a different decryption key, *asymmetric keys*.

Symmetric Key Cryptography

Symmetric key cryptography is based on the fact that both parties in the communication share the same secret key used for both encryption and decryption, illustrated in figure 2.3. Algorithms for symmetric key cryptography are simple and operate at a high speed, but the distribution of the symmetric key is complex. This is because it needs to be transported securely so that a third party is not able to read the key without permission. Symmetric keys are often used subsequently to establishing a secure channel with asymmetric keys as session keys.

Asymmetric Key Cryptography

Asymmetric key cryptography – also called public-key cryptography – is based on each peer having two keys each, a public key and a private key. Algorithms for asymmetric keys allow for the plaintext and the public key to be used for encryption while the private key together with the ciphertext are used for decryption, shown in figure 2.3.1. Each peer can share its public key so that it is possible for anyone to encrypt a message meant for the peer. Algorithms used for computing asymmetric keys are slower than the ones used for symmetric cryptography, but allows for easier distribution of keys.

2.4 Key Management

Essentially, the purpose of key management is to process cryptographic keys that are used in a system. Key management is based on the phases of the key's lifecycle, defined as creation, distribution, usage, archiving and destruction [DM04]. Properties of key management systems often rely on the design of protocol rather than the cryptographic algorithms used [FM90]. Consequently, it is very important to design secure and correct protocols for key management.

The following section specifies the different phases in a key management scheme. Thereafter, different architectures for key management are discussed in section 2.4.2. Finally, the section presents some key management implementations that were used to draw inspiration for the design of DPKMS.

2.4.1 Phases in Key Management

These phases have been considered to be the main phases in any key management scheme. However, depending on implementation, various others could also be included, such as registering keys when a registration authority is present or making a certificate.

Key Generation Generation of keys is a crucial phase in a key management scheme. Keys are generated with the use of algorithms that process random numbers. For keys to be secure, their randomness needs to be true, at the same time as the algorithms need to have no known weaknesses that can be exploited by attackers. The generation process depends on which type of key needs to be generated [DM04]. Symmetric keys are easy to compute while generation of asymmetric keys is a CPU-intensive task. Key generation is not to be exchanged for key derivation, because when generating a key, the process relies on random number generation. Deriving a key, however, is deterministic, and the inputs to an algorithm are secret but not random.

Key Distribution Key distribution is the task of delivering keying material to the right entities in a network. Distribution technique depends on which type of keys are used, either symmetric or asymmetric, both explained in section 2.3. Asymmetric keys are easy to distribute because the public key can be given to whoever wishes communication, and does not reveal anything crucial about the private decryption key. For symmetric keys, the distribution technique is more difficult, often relying on an already secured channel.

Key Activation Key Activation describes the process where a generated and distributed key is available for use for a cryptographic function in a system.

Key Exchange Key exchange is the process by which two already authenticated entities decide on a key or keypair for the communication between them. There

are several protocols for key exchange, depending on factors such as key type – asymmetric or symmetric – and rekeying interval. Key agreement can be a part of the key exchange phase, where the entities can agree upon the properties for generating a shared key.

Rekeying Rekeying is the process where a session key is changed because the set time – the cryptoperiod – has expired. Each session key has an associated Time To Live (TTL), set by the administrators of the key management. To accomplish Perfect Forward Secrecy, explained in section 3.3, the interval for rekeying needs to be short. There are two main ways of exchanging an old key for a one. Either a new key can be added before withdrawing the old one, or the old one can first be withdrawn before adding the new one.

Key Revocation Key revocation is a routine which takes place when it becomes necessary to remove a key from a system. Revocation can occur due to a compromised key, a deauthorized entity or a compromised cryptographic algorithm. Procedures for key revocation depend on the type of key management system, but a common way of implementing this is by having an updated list of revoked keys.

Storing Keys Also included in key management is how to store keys that are not to be used immediately. In case the entity in which the encryption takes place is stolen, some means of secure storage needs to be arranged. The NIST framework for key management specifies requirements for tamper resistant security modules (TRSM) that are used for secure storage of keys [BSBC13].

Key Destruction Key destruction is the process of deleting all duplicates of a key in a system. Complete destruction of keys is only possible by physically destroying the entity containing the key [DM04]. However, there are other processes where the destruction of devices is not needed, but these processes do not completely destroy the key.

2.4.2 Architectures

The way in which a key management scheme is implemented depends on the specific problem that the system is made to solve. The following section will present various architectures where a third part is present. This is due to the fact that SDN evolves around an architecture where a central entity is present at all times. Hence, architectures involving a third party can be relevant to key management in SDN.

Key Distribution Center

From the definition provided by W. Stallings, a Key Distribution Center (KDC) is used to distribute and generate keys in a system consisting of entities that share a

key with the KDC [Sta05]. An entity needs to contact the KDC for it to issue a shared key between the initiating entity and the one it wants to contact.

Key Translation Center

The operation of a Key Translation Center (KTC) resembles the one of KDC, but does not generate any keys – it is the entities themselves that carry out this operation. The KTC is a trusted entity that shares keys with all parties in the network. An entity wishing to send something to another entity in the network encrypts the message using the key shared with the KTC. The KTC decrypts the message and thereafter encrypts it with the key of the destination, before transmitting it to the rightful destination.

Public Key Authority

For asymmetric keys, a public key authority (PKA) can be kept to maintain a directory of all public keys of entities in the network [Sta05]. The entities know the public key of the authority and send requests to the authority to get the public key of other entities in the network.

Public Key Infrastructure

A Public Key Infrastructure (PKI), like a PKA, provisions asymmetric cryptography [DM04]. The PKI's task is to manage certificates in a system. Certificates is a way of implementing asymmetric cryptography and authentication by combining the public key and identification for the owner of the certificate in a file signed by a Certification Authority (CA). An entity wanting to communicate with the owner of a certificate checks its validity by consulting a CA. Thereafter, the entity uses the public key to encrypt private messages to the owner. The PKI's role in this architecture is to be the infrastructure that distributes the certificates to entities that use them.

A PKI is made up of several entities, the main ones listed below:

- *CAs* generate certificates according to preconditioned policies
- *Registration Authority (RA)s* verify that an entity requesting a certificate is valid
- *Certificate Repositories* store certificates
- *Certificate Status Servers* provide the status of certificates, including a list of revoked certificates

2.4.3 Implementations of Key Management Schemes

When designing DPKMS, schemes built for other purposes needed to be examined. This is because they served as inspiration for the development of the scheme and requirements which will be presented in subsequent chapters. This section will go through some of the most relevant implementations in order to discuss design decisions in section 6.2.

Key Management in Wireless Mobile Networks

In wireless mobile networks such as 3GPP's Long Term Evolution (LTE), managing keys is a complex task where entities are moving and still expect secure communication when the access point changes [HC14, 3GP13]. Even though the network is architecturally very different from SDN, some key concepts can still be examined, such as the emphasis on key separation.

TLS, TLSA and DANE

TLS is a public key protocol that depends on a PKI and certificates for binding keys and names. In general, TLS is used for one-way authentication, where only the server is authenticated. However, two-way authentication is possible [DR08]. A potential weakness to TLS's key management scheme is that it allows a trusted CA to issue certificates for any domain name, and this can be problematic if any CA is compromised.

DNS-Based Authentication of Named Entities (DANE) provides a way to bind public keys to DNS names for TLS x.509 certificates [HS12]. This is done by using DNSSEC [AAL⁺05], a protocol that provides secure communication to a DNS-server by using public-key cryptography, signing the lookups to a DNS-server. DNSSEC is used to store and sign the keys and certificates, while DANE provides the binding of public keys and certificates. This protocol enables domain name holders to issue certificates themselves, without being dependent on a CA.

The DANE and TLSA protocol works in the following way: A client wants to contact a server and looks up the server's name using DNS. The DNS-server then returns the correct IP-address of the server together with its certificate. When the correct server is contacted, and issues a TLS response to the client containing its certificate, the client is able to authenticate the server without contacting a CA [Ken14].

By substituting CAs with DNS-servers, a potential drawback is that the DNS-queries are cached and therefore there is no way of revoking a certificate. If an administrator removes a certificate from the DNS-server, clients that have already cached the certificate will stop using it when the TTL has expired. Thus, setting the

TTL to a very short number will force the client to always choose a certificate which is up to date. This solution will make DNS lookups slower.

IPsec and IKE

It is important to provide a thorough description of the IPsec protocol, seeing as this implementation provided significant inspiration in the development of DPKMS. Additionally, IPsec is seen as an industry standard for cryptography and is widely used for securing Virtual Private Network (VPN).

IPsec is a suite of protocols for securing communication by means of authenticating and encrypting communication between two entities in the IP layer [FK11]. Located below the transport layer, the protocol is transparent to applications. IPsec's main components are Authentication Header (AH), Encapsulating Security Payload (ESP) and Internet Key Exchange (IKE). IKE is the protocol in the IPsec suite responsible for key management and negotiation. Two different protocols for security are implemented in IPsec, namely AH which authenticates the communicating entities, and ESP which combines both authentication and encryption [Sta05].

Security Association (SA) are used in IPsec for defining security measures in a one-way connection between sender and receiver in an IPsec message exchange. Different databases are used in IPsec when the policies in an SA are negotiated. These are the Security Association Database (SAD) – deciding which parameters to use for encryption – and the Security Policy Database (SPD) – for deciding when to use IPsec.

IKE

IKE is IPsec's implementation of key management [KHNE10]. It is a protocol for setting up the SAs and mutually authenticating the communicating entities. Key management in IKE assumes that asymmetric keys are distributed to the communicators. The protocol focuses on the exchange, and the distribution of keys is left to out-of-band management. IKE uses UDP as transport protocol. Two versions exist for IKE, the newest one – IKEv2 – is the one discussed in the following.

The protocol starts with the two exchanges called `IKE_SA_INIT` and `IKE_AUTH`. These are followed by either `CREATE_CHILD_SA` or `INFORMAL`. `IKE_SA_INIT` is the exchange for negotiating security parameters of the SA and sends nonces and values for the Diffie-Hellman (DH) handshake to be carried out. `IKE_AUTH` is an exchange for authentication, so identities are transmitted in order to set up an SA by initiating a DH-key exchange. Authentication can be done in three ways; by digital signatures; public-key encryption; or symmetric key encryption. The SA resulting from this exchange is the basis for the ones created by the next exchange. `CREATE_CHILD_SA`

is an exchange used to establish a session key for protecting data while `INFORMAL` is used for tasks like error reporting or deleting an SA.

2.5 Applied Technologies

The following section is intended to provide an overview of technologies used mainly for experimentation and as a basis when making DPKMS.

2.5.1 WireGuard

WireGuard is a network tunnel that implements security which is meant to replace IPsec [Don17]. It uses public key cryptography with peers identifying each other by a simple scheme consisting of their public key and IP-address. The screenshot in figure 2.5 shows the simplest setup of a WireGuard tunnel. The static public keys are used for the establishment of a symmetric session key. The session key is used to encrypt parts of a session until it is changed. It operates in layer three by encrypting IP packets over User Datagram Protocol (UDP). It is said to use state-of-the-art cryptography with neither cipher nor protocol agility. This is a deliberate choice, as it avoids attacks resulting from the exploitation of broken ciphers.

WireGuard is still in an experimental stage of development, as it was released in June 2016. There are still parts of the software that are not completed, such as cross-platform support. Nevertheless, there is an expectation in the industry that WireGuard can be implemented in the near future in order to replace other tunneling protocols, and in the newly released project from Docker, LinuxKit is going to serve as an incubator for WireGuard [Panil].

Key management in WireGuard is explained in this paragraph. Session keys in WireGuard are established in a key exchange based on the Noise protocol framework [Per17]. However, key distribution is not considered in the standard, and is supposed to be provided by an out-of-band mechanism. The static keys are 32 bytes long Curve25519 points. Curve25519 is considered a state-of-the-art Elliptic Curve Diffie-Hellman function and is used for several applications such as OpenSSH, the Signal Protocol and GnuPG [Ber06]. First, the keys are generated quickly but are CPU-intensive to compute. After this, the keys are used for mutual authentication and establishing symmetric session keys between the communicating entities. The key exchange for the session key is only one Round Trip Time (RTT) and establishes a pair of session keys; one for sending and one for receiving. These keys are replaced after a relatively short time interval in order to achieve perfect forward secrecy. In addition to asymmetric keys, WireGuard has an option to use a pre-shared symmetric key shared with the communicators. This feature exists due to the possibility of quantum cryptography emerging and breaking Curve25519.

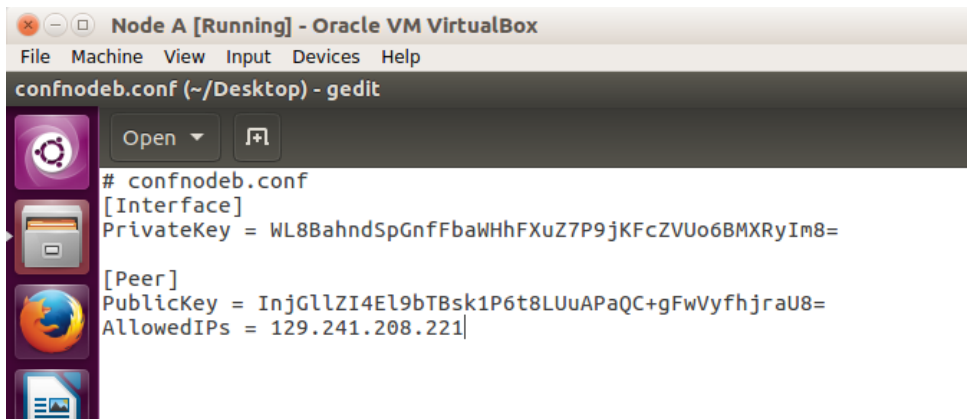


Figure 2.5: Screenshot of the configuration file showing the simple setup of a WireGuard tunnel in one of the peers.

WireGuard is implemented as a virtual interface that is configured with a private key and, optionally, a UDP port that it listens to. The tunnel is implemented in the Linux kernel which implies a smaller codebase and an effectively running program. Encryption and decryption of data is done in-place and can be done in parallel for utilizing all available CPU cores which speeds up the process. Perfect forward secrecy is maintained by always deleting cryptographic operations from memory after use.

The WireGuard API is based on `ioctl(2)` and its command line tool is called `wg`. `wg` lets you do operations on the interface and includes a series of commands to interact with the WireGuard interface.

WireGuard's Architecture is simple and is based upon a concept referred to as *Cryptokey Routing*, which deals with how the peers are authenticated. Each WireGuard interface has, as described earlier, a private key and a port which it listens to. Additionally, the interface has a list of peers that it is able to establish a WireGuard tunnel with, each identified by a public key and a list of allowed IP-addresses. This is illustrated in figure 2.7 showing WireGuard configuration in two nodes.

```

Node B [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
nodeb@nodeb-VirtualBox: ~
nodeb@nodeb-VirtualBox:~$ sudo ip link add dev wg2 type wireguard
nodeb@nodeb-VirtualBox:~$ sudo wg setconf wg2 Desktop/confnodea.conf
nodeb@nodeb-VirtualBox:~$ sudo ip link set up dev wg2
nodeb@nodeb-VirtualBox:~$

```

Figure 2.6: Screenshot of how a new WireGuard interface (`wg2`) is added and thereafter how a link is established.

WireGuard's operation starts by adding an interface, shown in the screenshot in 2.6. In the screenshot, `wg2` is the interface being added in a virtual machine called **Node B**. The configuration file used to set up the interface and one peer, **Node A**, is similar to the configuration file in figure 2.5. Consequently, after the private key and at least one peer has been added, a handshake is performed, shown in figure 2.7. This is done by the initiator sending its public key to the responder, encrypted with the public key of the responder. A counter is included in this message to protect against Denial of Service (DoS) attacks. The responder adds the key to its list of peers and one more handshake message is needed before a symmetric, authenticated session key can be derived. The key exchange is only one RTT. Conclusively, the link is set up and it is possible to view the interface as shown in the screenshot 2.8. Thereafter, the peers can communicate securely with each other through the secure tunnel.

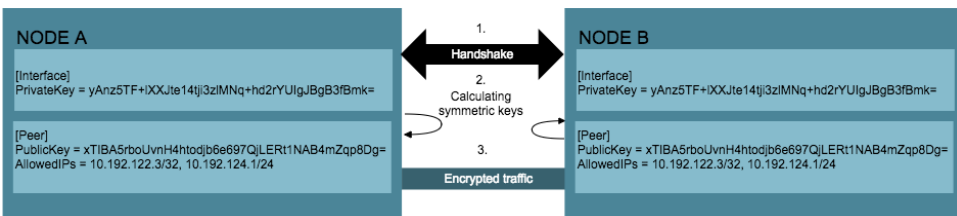
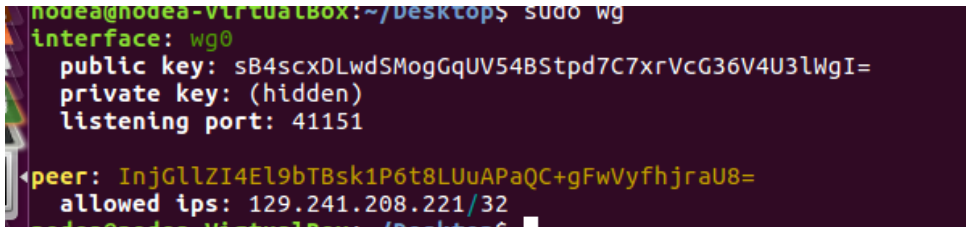


Figure 2.7: The configuration information in the nodes and the steps for setting up an encrypted tunnel through WireGuard.

Limitations with regards to the use of WireGuard are for instance the lack of cipher agility. This can be a problem for peers that are unable to use the supported type of cipher, and therefore the nodes cannot communicate through WireGuard.

Another potential problem with this is that if a vulnerability is discovered in the used cipher, whoever uses WireGuard will be insecure. Nevertheless, this was the reason behind the choice of a rigid protocol; it is simpler for all nodes to update if a vulnerability is discovered. A second limitation to WireGuard, in the context of SDN, is that it only operates at layer three, only forwarding IP packets. SDN-nodes are conceptually able to forward traffic from layer 2-5 and therefore only forwarding IP packets is a restriction to the node. This would either demand the node to encapsulate all packets that will go through WireGuard, or reject non-IP packets to be sent through the WireGuard tunnel. A third limitation is the novelty of WireGuard. The tunnel is not yet fully developed and there still might be a chance that a severe problem occurs, leading to the tunnel losing its reputation.



```

nodea@nodea-VirtualBox:~/Desktop$ sudo wg
interface: wg0
  public key: sB4scxDLwdSMogGqUV54BStpd7C7xrVcG36V4U3lWgI=
  private key: (hidden)
  listening port: 41151

peer: InjGllZI4El9bTBsk1P6t8LUuAPaQC+gFwVyfhjraU8=
  allowed ips: 129.241.208.221/32

```

Figure 2.8: Screenshot showing a WireGuard interface and a configured peer through the use of the `wg` utility.

2.5.2 Open vSwitch

Open vSwitch (OVS) is a virtual OpenFlow switch [PPK⁺15]. The switch is a multilayer open source and multiplatform switch, often used in SDN-research. It was used in this thesis for testing and affirming theories. The virtual switch was chosen because it supports the OpenFlow Experimenter structure and there exist examples for using this in OVS. Furthermore, it was the preferred switch for Steffen Birkeland in his thesis [SF16], as well as it is open source and has many tutorials, making setup simple.

OVS consists of two main components for packet forwarding; `ovs-vswitchd` and the datapath kernel module. `ovs-vswitchd` is a user-space daemon that is connected to a controller through OpenFlow for receiving flow tables. The datapath kernel module is the module responsible for forwarding the packets, with actions from `ovs-vswitchd` on how to handle them. If no action for the specific packet is decided in the datapath kernel module, the packet is sent to `ovs-vswitchd`.

2.5.3 Floodlight

Floodlight is a java-based SDN-controller that supports virtual switches and OpenFlow [Flo17a]. The controller is known to be simple to orchestrate and offers Application Programming Interface (API)s to applications based on REST. The Floodlight controller was used for defining flows to send to specific ports in OVS. The reason for choosing Floodlight was that it was the only controller where there was found examples of use of the OpenFlow Experimenter structure.

2.6 Technical Assessment

To get a better understanding of SDN, technical experiments were conducted using mininet – a network emulator – and Open vSwitch (2.5.2) in Ubuntu. This allowed for a thorough comprehension of how the characteristics of SDN could benefit a key management scheme.

To ascertain the feasibility of implementing DPKMS, it was important to validate a concern before designing a key management scheme for WireGuard. The concern revolved around whether it was possible to send messages through the WireGuard interface connected to an OpenFlow switch. To ascertain this, two already configured WireGuard interfaces in two different virtual machines were connected to switches with Open vSwitch software. To do this, a port in each switch was given a WireGuard interface. Thereafter, the two switches were again connected to a controller, chosen to be the Floodlight (2.5.3) controller. The switches with the WireGuard interfaces were implemented in two virtual machines with Ubuntu, while the controller was run from the native Ubuntu. The goal of this experiment was to find out whether or not the switch was able to distinguish flows to be put into the WireGuard interface.

It was proven possible to send flows through the WireGuard interface when conducting the experiment. This could be seen by inspecting received packets in the packet analyzer Wireshark.

2.7 Related Work

SDN research related to security is in general focused on the security between the nodes and the controller, in the control plane where OpenFlow operates [BEIE15, DFP13, CTHN16]. Nevertheless, there exists other studies examining how the security in the data plane can be enhanced by the use of a software defined scheme, such as the Internet Draft *Software-Defined Networking (SDN)-based IPsec Flow Protection* [MLLM16]. Early in the process of developing the problem description for this thesis, it was considered to conduct a Proof of Concept (PoC) realization of the model proposed here. The Internet Draft proposes a model where the controller

establishes the IPsec SAs and provides protection for flows through the protocol suite IPsec. Their approach to dynamic key management for IKE is through the configuration protocol NETCONF by the use of YANG models [Enn11]. The initial plan of using this model was discarded due to the complexity residing in IPsec and IKE. Instead, a simpler approach through OpenFlow was used. However, the proposal served as great inspiration for DPKMS

Another study examining the security in the data plane of SDN is Steffen Birkeland's thesis from 2016 [SF16]. His thesis – *Software Defined Data Flow Isolation by Virtualization and Cryptographic Key Distribution* – is the reason behind the development of this thesis, as it was initially thought of as an extension to Birkeland's thesis. Birkeland's objective was to *explore how existing open source SDN technology can be used to extend an SDN framework to handle a centralized control of encryption* where he used a manual setup of IPsec to provide encryption to the data traffic. His work was used as a basis when deciding which technologies to use for this thesis.

For key management among network nodes, several schemes are proposed, for instance for the use in Distributed Sensor Networks [EG02, CGPM05] from which inspiration can be derived. However, they are not directly applicable to the key management problem in SDN. Key management itself is a well researched topic, and there are several frameworks [FL93, BBB11] and books [Sta05] presenting the conventional schemes. Nevertheless, none of them are directly related to the problem of key management in a software defined scheme. IKE, the key management protocol for IPsec [KHNE10], has also been studied thoroughly. IPsec and IKE was used as inspiration when composing DPKMS, but the protocol was not used directly due to its complexity which will be discussed in section 6.2.

Chapter 3

Key Management in SDN

This chapter examines preliminary conditions with regards to the key management scheme to be developed, DPKMS. Firstly, the characteristics of SDN are presented. These characteristics are recognized to interfere with the way the scheme is designed. Thereafter, assumptions are stated, followed by requirements made in order to later evaluate the proposed model.

3.1 Characteristics of SDN

SDN is an architecture centered around the concept of central control of a network. This centrality can be both an enhancement and a drawback when it comes to key management in the communicating nodes. There are two main characteristics of SDN that make the process of managing keys in the data plane different to other approaches:

- The network is highly dynamic
- The controller is the initiator of secure communication

The Network is Highly Dynamic

One of the key factors adding to SDN's popularity is that the network architecture offers a change in the logical topology of the network in close to real time. This feature also needs to be true for a scheme where some of the communication channels are secured, and the controller is in charge of distributing the right keys at the same time as maintaining the topology. For a key management scheme, this means that all of the operations need to happen fast, doing as much work as possible before the decision to make or change a secure tunnel is made.

The controller is the initiator of secure communication

As previously discussed, when it comes to key management schemes, and specifically key distribution, the entity starting the communication is conventionally the one initiating key distribution. This way, the entity responsible for issuing and keeping track of the keys does not have to decide when and what to encrypt, as this is entirely up to the communicators.

In SDN, this is radically different. The controller is the entity in charge of issuing and keeping track of the keys, as well as it initiates secure communication. As explained thoroughly in the following chapter, an application on top decides which links to be encrypted. This implies that it is not possible to use the exact architecture of other traditional key management schemes.

3.2 Assumptions

In order to simplify the scope of the thesis, three assumptions were made early in the process. These were made in order to focus on the core process of managing keys for encrypting the data plane in SDN.

- The OpenFlow connection between the controller and the node is already encrypted and mutually authenticated with TLS
- An application on top of the controller determines the paths to be encrypted
- Functionality for WireGuard is built into the network nodes

The OpenFlow connection between the controller and the node is already encrypted and mutually authenticated with TLS

OpenFlow is the protocol between the switch and the controller where the default security mechanism is encryption through TLS. The protocol recommends mutual authentication between the switch and the controller by exchanging certificates. Therefore, an acceptable assumption for this thesis is that the connection is already encrypted and authenticated, allowing for the communication between the controller and switch to be protected.

An application on top of the controller determines the paths to be encrypted

Another assumption is that an application on top of the controller, bound by its northbound interfaces, determines which paths are supposed to be encrypted. This allows for an SDN approach to key management. However, it makes it impossible for end-users to enforce encryption of the data plane.

Functionality for WireGuard is built into the network nodes

This assumption narrows down the scope of the thesis. A WireGuard interface would be included in a port belonging to an Open vSwitch. This assumption is not made because the technology present in today’s network elements, but because it makes it possible to only focus on the key management in SDN, which is the core problem of this thesis.

3.3 Requirements

Requirements that should be fulfilled by DPKMS are listed in this section. Characteristics listed in section 3.1 highlights why key management in SDN is different from implementations in other types of networks. This is why it is hard to directly apply other key management protocols. The key management protocol for IPSec – IKE [KHNE10, FK11] – is used as a reference when formulating the requirements for this particular case of key management, as well as the key management framework developed by NIST *A Framework for Designing Cryptographic Key Management Systems* [BSBC13]. The section characterizes two types of requirements, one related to cryptographic properties that the scheme achieves, and the other related to how the scheme operates.

3.3.1 Cryptographic Requirements

Synchronization Entities in the scheme need to be synchronized so that processes do not start until all nodes are ready. This is an important requirement because unsynchronized nodes can lead to packet loss or worst case the revelation of secret information. In IKE, this is handled by assembling all messages in request and response pairs [KHNE10].

Authenticated Nodes Being able to trust that the nodes in the system are who they claim to be means having authenticated nodes. This is important so that no malicious nodes can access and join the network secured by the key management scheme.

Authenticated Keys This requirement refers to keys truly belonging to the person they are said to belong to. In the key management scheme, this is of concern because a key needs to be bounded to a peer in order to function.

Authenticated Controller The controller needs to be authenticated by the node in order to issue keys for key management which can be achieved by mutual authentication between the nodes and controller.

Perfect Forward Secrecy This is a property of key exchange algorithms where the exposure of the long-term key does not expose any session keys that are

already made [Sta05]. This is important because if an attacker obtains the long-term key, this cannot be used for decrypting past sessions.

Key Integrity Key integrity means that one is sure that the keys are kept the way they were made and not altered along the way. The property is important because if the keys cannot be trusted, a secure connection cannot be constructed.

Key Freshness Freshness of a key is a property enabled by frequently changing the session keys. This property ensures that only a small part of data is encrypted under the same key. If it is leaked, only a small part of the data can be decrypted.

Key Binding Key binding means binding the key to its context, for instance by including metadata in the generation procedure or by computing a digital signature over the key and metadata [BSBC13].

Key provides required level of attack resistance This requirement is set to ensure that the algorithms receive a key that ensures the required level of attack resistance.

3.3.2 Operational Requirements

DPKMS needs to be able to manage 32 bytes keys WireGuard expects keys with a length of 32 bits and therefore the key management system needs to be able to generate and distribute these keys.

DPKMS needs to be able to run over the OpenFlow protocol To make the scheme as simple as possible, messages regarding the key management need to be transmitted over the OpenFlow protocol.

Private keys that are distributed are not exposed This property is made in order to ensure that a third party eavesdropping on traffic cannot get hold of the private keys distributed to the nodes from the controller.

Keys are changed within a bounded time interval Keys should only be valid for a set period of time and a Rekeying procedure should be started when the Cryptoperiod expires. The rekeying procedure should interrupt as little as possible in the normal operation of the scheme. This requirement for rekeying corresponds to the one for IKE [KHNE10].

Compromised keys are revoked In case a key in a node is compromised, the key management scheme needs to have a revocation procedure ensuring that the compromised key cannot be used.

Key revocation happens rapidly As soon as an error is detected, the revocation procedure for a node needs to be initialized. This procedure must interrupt the traffic as little as possible.

All errors are handled Errors occurring in the key management scheme must have a pre-configured way to be processed by the controller, but DoS attacks must at the same time be accounted for.

Secure communication sessions are set up and torn down quickly Because of the dynamic nature of SDN, a quick procedure for setting up and tearing down a secure connection is essential to the scheme.

Reliable message delivery To ensure synchronized nodes, reliable message delivery is needed. In IKE, reliable message delivery is ensured by enforcing a response for each transmitted message.

Packet loss or delay must be handled Packets that are lost or delayed should not have a major negative effect on the system and should be handled in a way where synchronization between the nodes is not affected.

Following generation, the private keys are only known to the node The private keys used in the WireGuard tunnel should only be known to the node using the key for decryption after the controller has generated and distributed it.

Keys are generated based on output from a Random Bit Generator (RBG)

This requirement is a part of the NIST *Recommendation for Cryptographic Key Generation* [BR12] and is important to achieve secure keys.

Session keys are securely exchanged When establishing a shared session key between two nodes, the way that the key is derived is important to ensure secure operation.

Node crash and restart is handled If a node fails, the controller has to handle the communicating peers so that they do not continue to send messages that will be lost.

Chapter 4

Data Plane Key Management Scheme (DPKMS)

Presented in this chapter is the functionality and operation of the key management scheme developed, referred to as Data Plane Key Management Scheme (DPKMS). In chapter 3, assumptions and requirements were set and these are used as the basis for the system demonstrated in this chapter.

The following section is a presentation of the topological design of the model, represented by a reference model in figure 4.1. Subsequently, each of the entities included in the model will be examined followed by an explanation of the interfaces connecting the entities. Finally, the procedures that make up the scheme for key management are addressed, each illustrated with a sequence diagram.

Following is a set of contextual definitions that will allow for a precise description of DPKMS

Node A node is a network entity implementing both the OpenFlow protocol and WireGuard as a secure channel and it is connected to a controller.

Peer A peer is the nodes that the initiating node can contact through a secure WireGuard tunnel.

Initiator The initiator is the entity issuing the message to start a procedure. In DPKMS, the application or the controller has the role as an initiator.

Responder The responder in a message exchange is the entity that the initiator contacts, and the entity receiving the first message.

Peerlist A peerlist is a list of peers that a node is able to communicate with through WireGuard, as it possesses its credentials. The credentials are a public key and an Internet Protocol (IP) address.

Application An application is a program with the ability to communicate with the controller via its northbound interface.

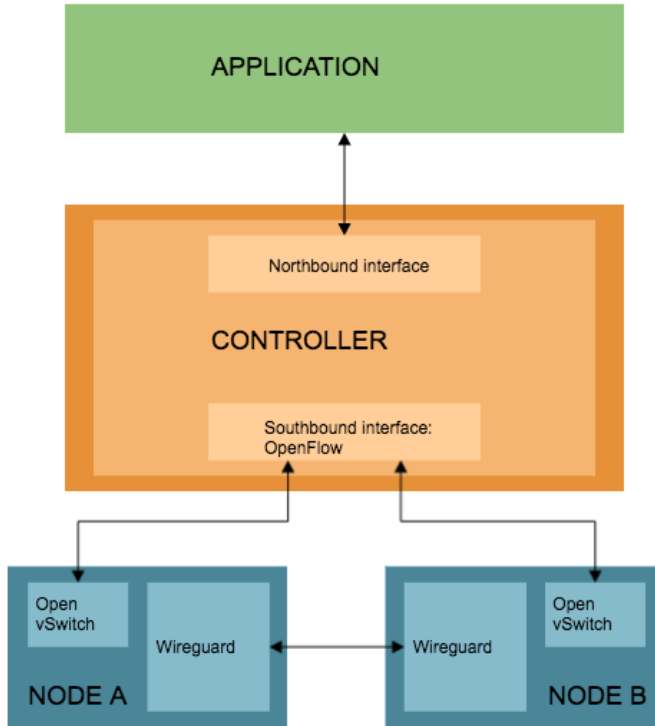


Figure 4.1: High level architecture of the model developed in this thesis.

4.1 Topological Design

Illustrated in figure 4.1 is the reference model of the system where DPKMS is defined. Consisting of only two nodes connected to a controller, the model aims to focus on the essential task of managing keys in the data plane of SDN. This section will go through the elements of the model – the controller, the node and the application – explaining their main functionality.

4.1.1 Controller

Key management processes associated with the key lifecycle should be performed on a secure platform. Secure platforms should have both logical and physical protection, which can be provided by a controller in an SDN-scheme. Logical protection of the controller is provided through the secure channel connecting it to the node, whereas physical protection can be provided by physical isolation of the software.

Responsibilities of the controller with regards to DPKMS are as follows:

- Generating WirGuard keys
- Distributing keys
- Storing public keys
- Initiating encryption
- Communicating with the application and applying its policies and cryptoperiods
- Keeping track of communicating nodes
- Keeping track of which nodes have which private keys
- Enforcing rekeying when the cryptoperiod is reached
- Storing information about which peers have each others' public keys
- Starting the revocation procedure when a node is compromised
- Handling errors that occur in the scheme

The controller will include a database containing information about the peers in the key management scheme. Stored in the controller is at least zero and at most two public keys per node. At most two keys need to be stored in order to be able to conduct rekeying and revocation. Shown in figure 4.2 is the representation of the database tables present in the controller.



Figure 4.2: Controller illustrated with its key management extensions which shows two tables, one displaying the peers communicating, while the other contains all configured peers.

Instead of the private keys identifying the peers – the way WireGuard solves authentication – a unique ID is appended to each peer in the controller. This is because the controller should be able to constitute the switches in rekeying and revocation procedures, where the communicating nodes may at some point have different keys.

The state of the nodes also needs to be tracked in order to ensure that two nodes

are able to make contact through the WireGuard tunnel. A bitmap showing the state of each node is sent as a response to the key management requests. These are received by the controller, enabling it to always have an updated view of the network.

4.1.2 Node

In the model, the entity called *node* is a network element; an Open vSwitch (2.5.2) which implements support for WireGuard. The Open vSwitch functions as an OpenFlow switch, and has a direct secure channel to the controller, as shown in figure 4.1. The node is also assumed to have built-in functionality for the tunneling protocol WireGuard, as stated in section 3.2. Open vSwitch functions like a normal OpenFlow switch, explained in section 2.2, where one of the ports has a WireGuard interface.

Responsibilities of the node with regards to DPKMS are listed below:

- Constructing status messages to send to the controller
- Reporting on errors from WireGuard

In DPKMS, the node is in general responsible for implementing functionality that the controller dictates, as well as being responsible for translating and sending information given by the WireGuard interface, such as status and error messages.

4.1.3 Application

The implementation and realization of a key management application on top of the controller is beyond the scope of this thesis. Nevertheless, it is important to note that the application would be the entity responsible for deciding which paths to protect and how short the Cryptoperiod should be for keys in each secure connection. This is a deliberate choice, enforcing the software defined approach. This is discussed thoroughly in section 4.3.1. The design of the application is also included in the section for future work 7.1.

The responsibilities of the application in DPKMS are as follows:

- Choosing which paths are to be protected through encryption
- Deciding the cryptoperiod for each secure connection
- Storing public keys
- Ending secure communication
- Deciding action when a node is compromised

4.2 Interfaces

Connecting each entity in the system – the controller 4.1.1, the node 4.1.2 and the application 4.1.3 – are interfaces that render information from one subsystem to another. Each entity in the system is defined to hold a state while the interfaces between the entities transport messages that enable transitioning from one state to another. This section gives an overview of the communication between each of the entities in the system with regards to DPKMS.

4.2.1 Application - Controller

The northbound interface of the controller connecting it to an application is beyond the scope of this thesis. However, it may be useful to briefly mention it in order to get a full understanding of the model. Many APIs can be defined for the controller, for instance in REST or SOAP. The application must be designed to communicate with an API that has support for the key management message exchanges that the application needs to take part in.

4.2.2 Controller - OpenFlow Switch

The interface between the controller and the switch is the OpenFlow protocol, defined in *OpenFlow Switch Specification 1.5.0 [Ope15]*, and previously described in section 2.2. The channel is, as stated in section 3.2, already set up securely through TLS. In the realization of DPKMS, OpenFlow needs to be extended, and therefore the interface between the controller and the switch need to support this extension.

Messages sent between the controller and the node are sent in a request/response message pattern. All requests in DPKMS are issued by the controller, and for each request the node sends a reply to ensure synchronization in the scheme. Within the message scope, the responses issued by the switch are `status_wg` messages, described in section 4.3.7. If the controller fails to receive a response from the node, it re-transmits the message within the re-transmission interval specified by TLS. This is because the transmission between the controller and the switch is secured through TLS, providing reliable message delivery.

4.2.3 OpenFlow Switch - WireGuard

The Open vSwitch has implemented the WireGuard interface (section 2.5.1) in a port, and therefore needs to receive and send WireGuard-specific commands in order to operate in DPKMS. This is done by using the OpenFlow Experimenter structure to define new actions.

4.2.4 Node A - Node B

The goal of DPKMS is to set up a secure connection through a WireGuard tunnel between Node A and Node B. WireGuard – thoroughly explained in section 2.5.1 – is a network tunnel in layer three, therefore all communication between the two nodes in the scheme will reside in layer three.

4.3 Detailed Description of the Data Plane Key Management Scheme (DPKMS)

Now that the entities in the system and the interfaces between them have been explained, the interactions in the scheme will be described. Figure 4.3 shows the state transition diagram representing a node in DPKMS. The states and the transitions are made based on the phases making up a key management scheme, explained in section 2.4.1. The transitions will be the basis for the description of the operation of the scheme.

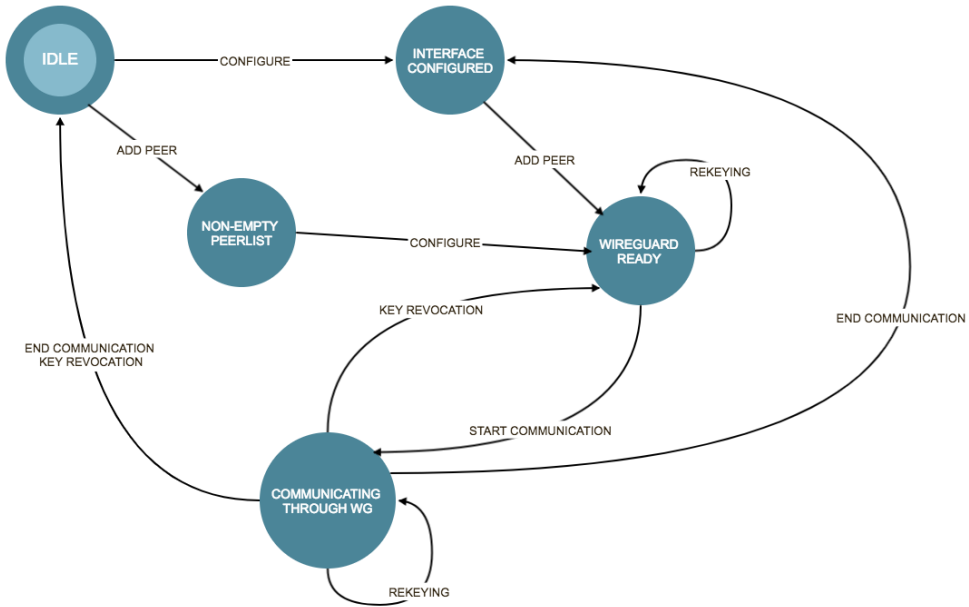


Figure 4.3: State transition diagram explaining states and transitions with regards to the node involved in DPKMS. The circles represent states and the arrows represent the transitions.

The circles represent the possible states held by the node, while the arrows represent the transitions from one state to another. Not all transitions present in the model are displayed in the figure, for instance all state and error messages are left out. This is due to the fact that the state transition diagram aims to show the general operation in a simple manner.

The *idle* state in the figure depicts a node with neither private key nor any peers in the list. To transition from this state, either a configuring procedure needs to be initialized – providing the private key for WireGuard – or the procedure for adding a new peer needs to be started. In the states *interface configured* or *non-empty peerlist*, either the configuring procedure or the procedure for adding a peer needs to be run in order to reach the *WireGuard ready* state. It is only in the *WireGuard ready* state that communication can be started. The rekeying procedure can be run either in the *WireGuard ready* state or when communication goes through WireGuard. Revoking a key can be done in either state, but the figure only shows the *communicating through WG* procedure. Here the two arrows showing the key revocation procedure represent the two modes in which the procedure can be launched. The procedure for ending communication also has two modes, leading to either *idle* state or to the *interface configured* state.

This section will go through each transition of the state transition diagram, and each procedure will be explained with a corresponding message sequence diagram. The message sequence diagram is designed according to the requirements in section 3.3.

4.3.1 Configure WireGuard Interface

The procedure of configuring the WireGuard interface in the node can be said to consist of three main stages. Firstly, a key is generated in the controller. Then the key is included in the controller’s databases. Finally, the correct private key is distributed through the secured channel in OpenFlow. In case a key already exists for the peer, this key needs to be deleted before generating a new one. If the node is present in other node’s peer-lists, the new key needs to be added to the peer’s peer-list by running the add peer procedure explained further in section 4.3.2. The OpenFlow switch is then responsible for sending the info over to the WireGuard interface for configuration. The only parameter that is necessary for a WireGuard interface to work is a 32-bytes long private key, representing a point in Curve25519. Other parameters such as port number, can also be included, but for this thesis, only the private key is used.

Two preliminaries need to be in place before the procedure for configuring WireGuard starts. Firstly, no peers are communicating through WireGuard while the node is being configured. This is ensured by the processes starting this procedure.

Furthermore, it is assumed that the first time the node is configured, an application initiates the procedure and decides the Cryptoperiod specific to the node. The cryptoperiod is then saved by the controller. Figure 4.4 shows the case where an application initiates the procedure.

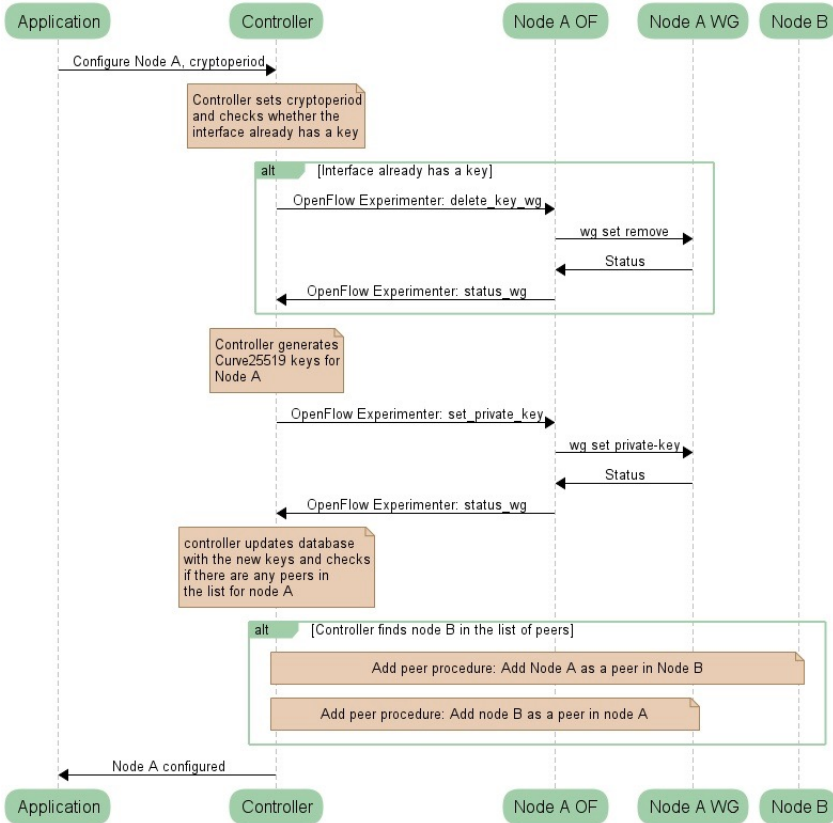


Figure 4.4: Sequence diagram that shows the procedure for configuring a node in the case where an application starts the procedure and specifies the cryptoperiod. *WG* is short for WireGuard and *OF* is short for OpenFlow.

The sequence diagram in 4.4 shows how the different entities in the system communicate when the WireGuard interface is configured. In this example, the application initiates the process of configuring a node by sending a request to the controller, specifying the cryptoperiod. Thereafter, a check is run in the controller to find out if a private key already exists. If this is the case, a `delete_key_wg` message is sent. Subsequently, the controller generates keys, and the private key is sent to the node to be configured in a `set_private_key` message. If the key is successfully

set – if a `status_wg` is sent – the database is updated. In order for the procedure for starting communication (4.3.3) to function, the controller needs to check the database for peers to the configuring node. In figure 4.4, Node B is found in the database as a peer of Node A. Therefore, the *add peer* procedure (4.3.2) is started for both nodes. When the procedure finishes, the controller issues information to the application that Node A is configured.

The configuring procedure is used when a node configures its interface for the first time, but also when a node changes its private key, for example due to revocation 4.3.6 or due to rekeying 4.3.5.

4.3.2 Add a Peer to the Node

Adding a new peer for communication through WireGuard is a simple procedure. The sequence diagram in 4.5 shows how the messages are sent between the different entities in the system. Used both when there are no prior peers in a peer table and when a peer changes its credentials, the procedure needs to support both peer lists with already existing entities and empty peer lists.

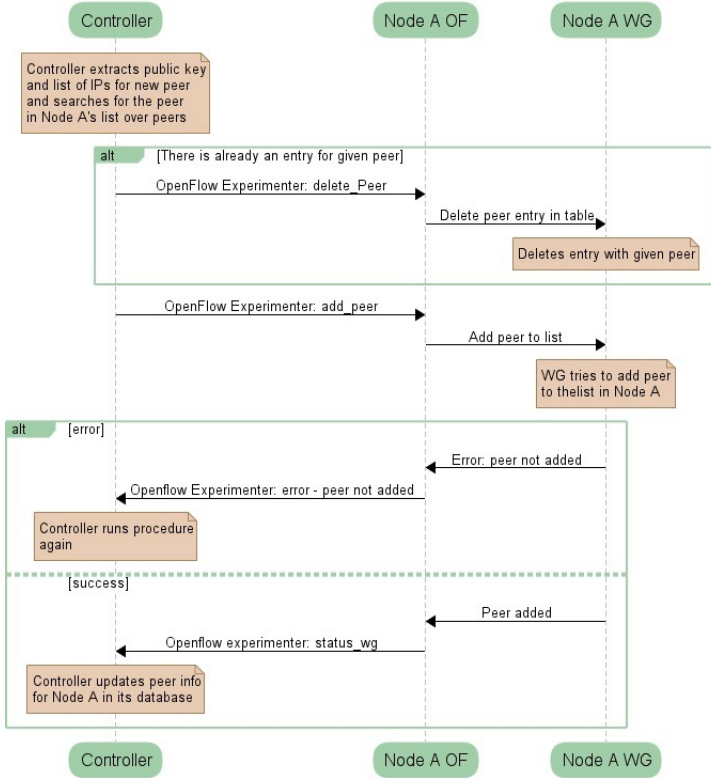


Figure 4.5: Sequence diagram with the general flow in the procedure of adding a peer to the WireGuard interface.

Prior to any messages being sent, the controller checks the information of the node in its database to check if it already has the peer to be added in its list. In case the peer exists, the peer is deleted by issuing the `delete_peer` message, containing the ID of the peer. The reason for deleting the entry is to support rekeying where a peer needs a new key. Subsequently, the `add_peer` message is issued. Following, an action instructs the WireGuard interface to add the peer to the list. Thereafter, in case of success, a `status_wg` message is issued and the controller’s database is updated.

4.3.3 Start Communication

The procedure for starting the communication through a secured WireGuard tunnel can only be initialized if the node has already configured its interface and has added one or more peers. This can be seen in the state transition diagram 4.3. WireGuard allows a node to send packets to a peer that does not have the node in its peer

table. The procedure for starting communication assumes that both of the nodes have the other node in its peer table. This is due to the fact that in SDN, the controller can retain a better overview of the network communication and can issue an `OFPT_TABLE_MOD` to both nodes at the same time by first adding both nodes in both peer tables.

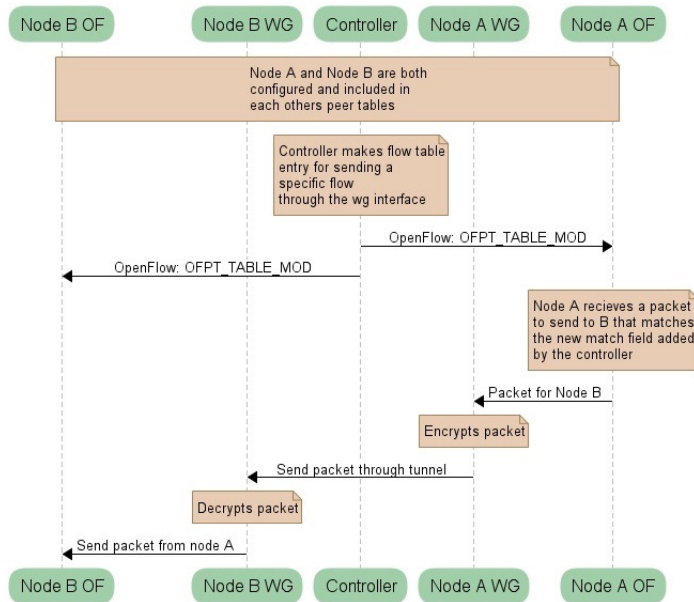


Figure 4.6: Sequence diagram showing the flow in the procedure of starting the communication followed by one packet being sent from Node A to Node B.

The sequence diagram in figure 4.6 shows the procedure for starting the communication. Essentially, what makes up this procedure is the controller providing a new flow table entry and pushing this into each of the switches. The flow table entry is made according to what policy the application on top says, and pushes specific traffic into the port of the WireGuard interface. OpenFlow has a defined message for changing the flow table, namely `OFPT_TABLE_MOD` [Ope15]. This message can be used to include new entries in a flow table. After the flow table has been edited, the switches will start communicating with each other through WireGuard, also shown in the sequence diagram 4.6.

4.3.4 End Communication

In essence, the procedure for ending communication needs to accomplish two things; stop sending packets in the direction of the peer and stop receiving and decrypting

packets from the peer. The communication can be ended in two ways, either by completely terminating all communication with the peer, or by continuing to send packets to the controller, just not through the WireGuard tunnel, but unencrypted through a default port. Ending the communication can be useful in many situations. This is for instance used in the revocation procedure 4.3.6, and in the sequence diagram in figure 4.7, an application issues the request for stopping the communication.

Depending on which mode the communication is ended in, either only one node or both are affected. In case of complete termination of communication, only one node is processed, not allowing any contact with its peer. In case only the WireGuard tunnel is shut down, the process needs to include both of the communicating nodes in order to redirect both their flows through a different port.

The procedure works for two distinct cases, but their realization is similar. If the communication to Node B is supposed to be completely terminated, the controller first emits an `OFPT_TABLE_MOD` telling the node to drop all packets directed to the peer. Thereafter, the peer is deleted with the Experimenter message `delete_peer`. The node will then update the controller on the state of the switch.

If Flows are not supposed to go through the WireGuard tunnel, but can still be directed to the default port without being encrypted, both of the nodes in the communication need to be instructed in order to maintain synchronization. First, an `OFPT_TABLE_MOD` is sent to each of the nodes, telling it to route traffic through the normal port. Thereafter, the peer is removed from the table of peers in WireGuard so that no messages can be sent to the node.

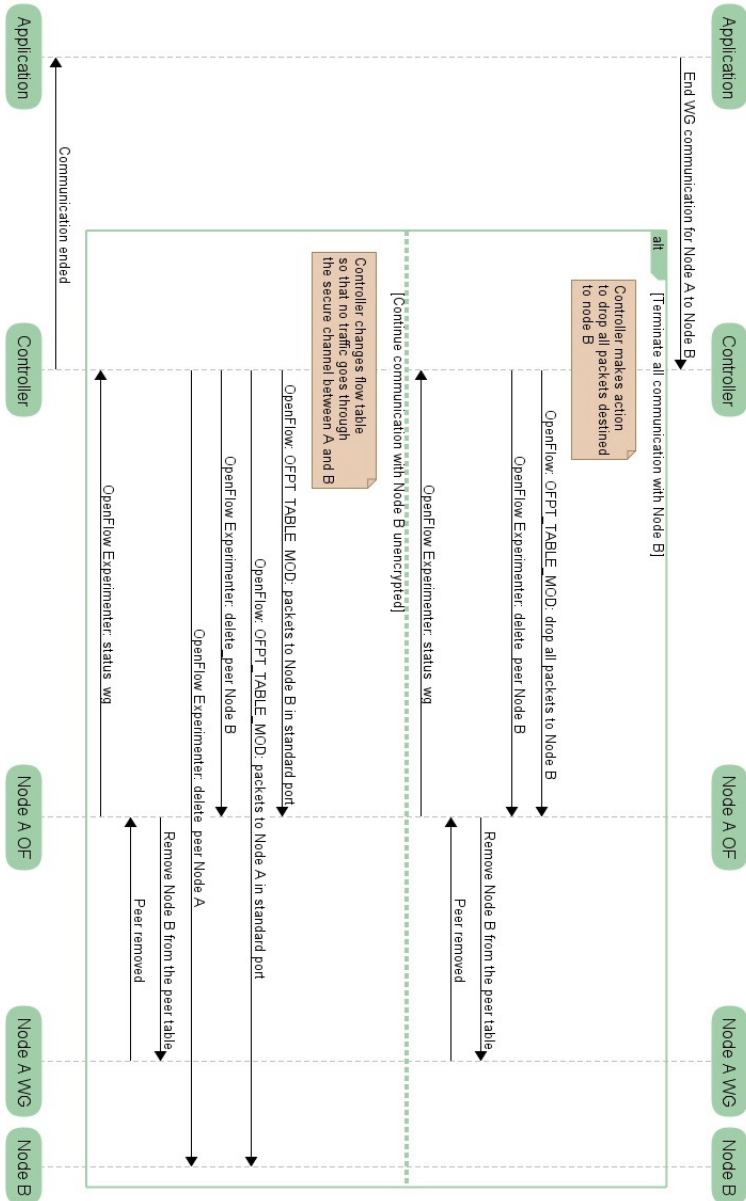


Figure 4.7: Sequence diagram showing the two different ways of ending the secure communication through WireGuard. Either by redirecting flows through the default port and not through WireGuard, or by completely shutting down all communication to the respective peer.

4.3.5 Rekeying

Rekeying is the procedure which is initiated when the Cryptoperiod has expired and a new key needs to be issued. The cryptoperiod for each key belonging to a node is specified in the controller’s database, and is set by an application when the configuring procedure (4.3.1) takes place. A short cryptoperiod could be used for flows where it is critical to have high security, but because changing the keys can be costly, less critical flows can have a longer cryptoperiod. NIST has a publication that contains recommendations for cryptoperiods [BBB11].

WireGuard has the ability to connect several IP addresses to one public key, explained in section 2.5.1, but only one key is allowed to identify the node. This means that for rekeying, one has to interrupt the communication at one time.

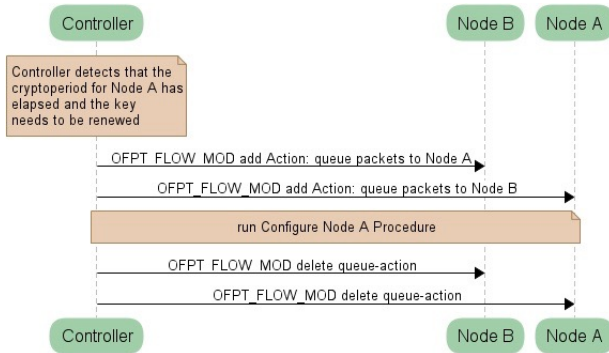


Figure 4.8: Sequence diagram showing the rekeying procedure where the controller initiates the rekeying when the cryptoperiod has elapsed.

Figure 4.3.5 shows the message flow in this procedure. The configuration procedure in section 4.3.1 takes a lot of the processing needed to renew a key. The first thing that happens in the rekeying procedure is that the controller sending out `OFPT_FLOW_MOD` messages. These are sent to nodes communicating with the node in the process of renewing its keys, thereafter to the node itself. The message instructs that packets previously directed to the interface are now queued. Thereafter, the configuration procedure is run for the node renewing its keys. This is possible because the configuration procedure takes rekeying and revocation into account by deleting earlier entries if they ever existed.

4.3.6 Revocation of Keys

As explained in section 2.4.1, revoking a key means removing a key that might have been compromised. The sequence diagram in figure 4.9 shows how the messages

between the entities communicate in case a node is compromised. Detecting that a private key of a node is compromised is out of the scope for a key management scheme, and is therefore done by an entity called *Third party* in the figure.

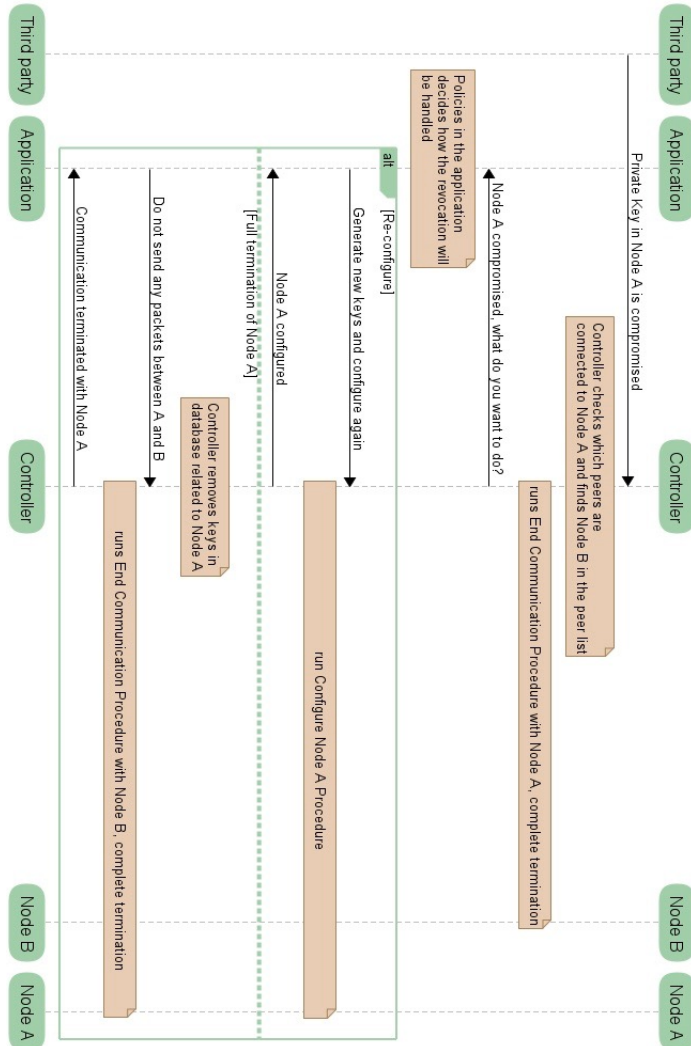


Figure 4.9: Sequence diagram displaying how the messages flow in case a third party – for example an intrusion detection system – instructs the controller to revoke the keys for node A. The application is thereafter able to choose how to handle the revocation.

Upon discovering that a node has been compromised, the controller initiates the procedure for ending the communication where complete termination of communication with the compromised node is chosen. This instructs the nodes communicating with the compromised node to drop all packets directed to it, as specified in section 4.3.4. This approach is inspired by operations in sensor networks discussed in the report *A Key-Management Scheme for Distributed Sensor Networks* [EG02]. Thereafter, the application decides what should be done next in the scheme. If the application decides to reconfigure the switch, the configuring procedure is run again. This procedure already takes into account the fact that keys already exist in the interface to be configured, and that the revoked public key needs to be deleted. In case the application wants to terminate all communication to Node A, nothing is done to the communicating nodes, but Node A is instructed to shut down all communication to its peers.

4.3.7 Status Message and Structure

The `status_wg` message is an asynchronous message sent from the node to the controller in the same way as the `OFPT_PORT_STATUS` message is sent [Ope15]. It is issued when the WireGuard port state is changed and contains a bitmap that represents the state of the switch.

Configured	This field is 0 if the private key is not configured and 1 if WireGuard is configured with a private key.
Peer List	The list of added peers is present here. If no peers are present, the list is empty.
Connection	If the connection field is 0, no peers communicate through WireGuard. If it is set to 1, there is at least one connection made.
Key Revoked	If the revoked-field is set to 1, the private key of the interface is revoked and therefore it will not receive any messages.

Table 4.1: The statuses that can be achieved in the WireGuard interface.

In DPKMS, `status_wg` is used as a response or an acknowledgement for the key management messages both to ensure timely delivery of messages and to guarantee that the controller contains an updated view of the state of the node.

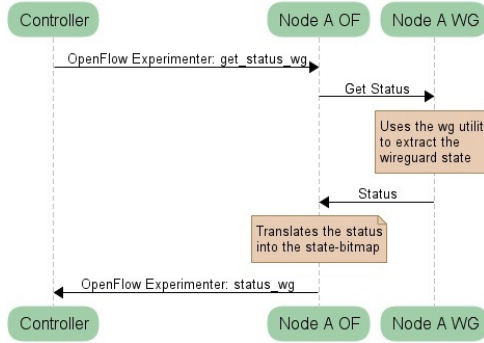


Figure 4.10: Message sequence diagram showing the flow when the controller issues a `get_status_wg` to be informed of the state of the WireGuard port, which is returned to the controller in a bitmap in the `status_wg` message.

The status message, like the `OFPT_PORT_STATUS` message, can be requested by the controller issuing a `get_status_wg`. This message exchange is shown in figure 4.10. The `get_status_wg` message needs to be implemented in the `OFPT_EXPERIMENTER` message because the `OFPT_PORT_STATUS` message has no experimenter field. Table 4.1 displays the information given by the status message.

4.3.8 Error Message and Structure

Error messages in DPKMS are returned if some of the procedures fail, acting as a negative acknowledgement. When an error occurs, no `status_wg` message is sent – the error is sent instead. Each of the messages that are not a part of the standard protocol in OpenFlow needs specific error messages. The OpenFlow protocol handles errors associated with specific messages and they are defined with a code and a type [Ope15]. The following list is a defined set of error codes to be implemented in a error message for the extended key management messages needed in OpenFlow to implement DPKMS.

More errors providing an exact explanation of incidents – such as *invalid peer*, *invalid private key*, *handshake failed*, *unknown peer* – could be implemented in the realization of DPKMS, but only the ones essential to the operation are defined in this thesis.

Unable to set private key	Response in the configuration procedure if a private key is not set when sending the <code>set_private_key</code> message is issued.
Unable to add peer	Response in the procedure for adding a peer if a new peer is not added as a response to the <code>set_peer</code> message.
Unable to remove peer	Error message sent in the procedure for adding a peer and in the procedure for ending communication as a response to a <code>delete_peer</code> message.
Unable to extract status	Response to the <code>get_status_wg</code> message in case the switch is not able to extract a status from the Wire-Guard port.
Unable to delete private key	Response to the <code>delete_key_wg</code> message in case the switch is not able to delete the private key.

Table 4.2: Overview of error messages needed in addition to standard errors issued by the OpenFlow protocol.

4.4 Extensions Needed in OpenFlow

For actual implementation of DPKMS, the OpenFlow protocol needs to be extended. This is due to the OpenFlow protocol lacking support and agility to be able to fulfill the needs of a key management scheme. The subsequent chapter discusses the specific details of how to extend OpenFlow, and therefore this section aims to systematize messages and attributes that are used to realize the operations in DPKMS.

Two tables are defined, table 4.3 that lists all messages that are needed in the OpenFlow protocol to implement DPKMS, and table 4.4 where all attributes needed in the OpenFlow extension are stated.

set_private_key message	Sent from the controller to the node containing the private key of the WireGuard interface used in the procedure for configuring a node.
delete_key_wg message	A message instructing the switch to delete the private key contained in the WireGuard interface.
add_peer message	Issued by the controller and sent to a node to add a WireGuard peer. The message contains both the peer's IP-address and its public key and ID.
delete_peer message	Also issued by the controller and sent to the switch for a peer to be deleted. The message contains the peer's public key and ID for identification.
status_wg message	Sent from the node to the controller and can be used in several cases. It is used as a response and acknowledgement for other messages and is also issued when the state of the WireGuard port changes. The message contains a status attribute with a bitmap representing the state of the node.
get_status_wg message	A request sent by the controller, requesting the status of a switch.
error_wg message	Sent to the controller when there an error occurs in the WireGuard interface and contains error attributes for specifying the type of error.

Table 4.3: Table of all messages needed in the OpenFlow protocol to implement DPKMS together with an explanation of each of the messages.

Key	Consists of either a private or a public key as well as an IP address if the key is a public key in the format needed by WireGuard.
Status	A bitmap indicating the state of the WireGuard port, the states shown in table 4.1.
Port	A specific port supporting WireGuard needs to be defined in the OpenFlow protocol.
Match	OpenFlow matches need to be designed to capture packets in DPKMS designated to the controller.
Action	For instructing of and communication with the WireGuard interface.
Error	Reports on specific problems regarding the scheme. The specific errors are displayed in table 4.2.

Table 4.4: Table describing the attributes that need to be defined in order to maintain the information needed in DPKMS.

Chapter 5

Extending OpenFlow

The last section of the previous chapter, 4.4, discussed what needs to be supplemented in order for DPKMS to be implemented in the OpenFlow protocol. Two tables were defined; one comprising of all messages (4.3), and the other one containing its associated attributes (4.4). The attributes are designed in order to hold the information transported by the messages. The tables hold all messages and attributes needed which do not already exist in the OpenFlow protocol.

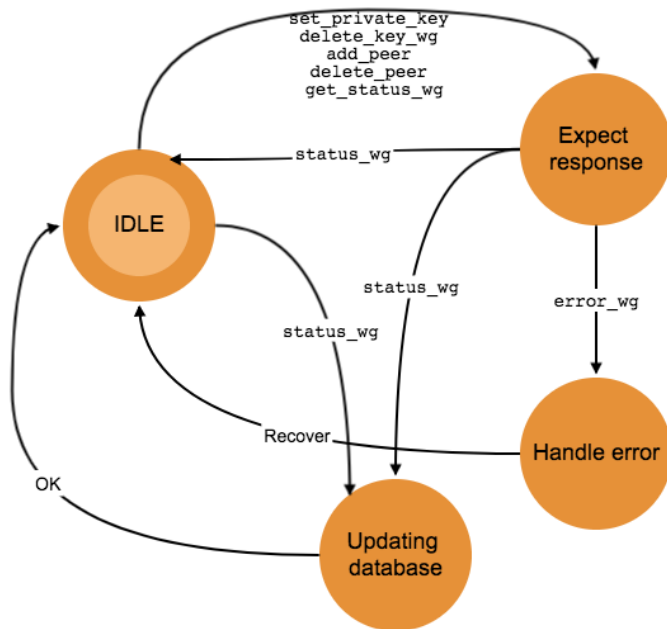


Figure 5.1: The messages extending the OpenFlow protocol for key management, displayed as transitions. The messages are shown together with their associated responses displayed in a state transition diagram. The states illustrate the state of the controller that issues the different messages.

Figure 5.1 summarizes the message flow to be included in the OpenFlow protocol. It displays the responses obtainable when sending the messages in the extension. The states, illustrated as circles, represent the controller. When the controller is in the *idle* state, it can issue the different messages displayed, or receive a `status_wg` message from a node because its state has changed. The *Recover* and *OK* transitions are processes that take place within the controller, and are not specified in DPKMS.

This chapter proposes an extension to the OpenFlow protocol, where the messages and their associated attributes are made. Here, the focus is on how to define the extension by using an OpenFlow structure made to enable vendors to include additional functionality in OpenFlow switches, the *Experimenter* structure. The Experimenter structure is used to provide the basis for a Proof of Concept which can potentially be developed in future work, something which will be discussed in section 7.1.

5.1 OpenFlow Experimenter Structure

OpenFlow Experimenter is a structure developed in order for vendors to define their own functionality in OpenFlow [Smi15]. Experimenter defines a header – shown in table 5.1 – for Experimenter messages, actions, matches, instructions, queues, errors and meters. These give the ability to implement functionality according to needs. The structure is used both for defining new message flows, as well as for specifying supplementary callback functions. Experimenter was first introduced in the OpenFlow switch specification 1.3 and several organizations have defined extensions to OpenFlow through the use of Experimenter [Ope15].

16 bit type	16 bit length
32 bit Experimenter ID	
32 bit exp_type	

Table 5.1: The structure of an Experimenter header.

Type and Length are values that are always contained in headers in the OpenFlow protocol and defines the name of the type together with the length of the packet.

Experimenter ID is used for uniquely defining an extension and is issued by the Open Networking Foundation for public use of the extension. It is 32 bits long and the most significant bit is zero and will be the same for all messages contained within one Experimenter extension.

`exp_type` is used to identify which message the packet is a part of, uniquely identified within an Experimenter ID.

5.1.1 Experimenter in Practice

Implementing Experimenter structures in a project requires that the software in both the controller and the switch is changed [Flo17b]. Not all controllers and switches support Experimenter, and therefore it is not a common structure to use. If a switch does not support an Experimenter extension, all messages are rejected [Ope15]. The virtual switch *Open vSwitch* (2.5.2) and the OpenFlow controller *Floodlight* (2.5.3) both support Experimenter extensions. Only the controller, Floodlight, has provided a tutorial for implementation of Experimenters.

5.1.2 Experimenter Message

To show an example of the Experimenter structure, the Experimenter message is chosen. Experimenter messages are not associated with an OpenFlow message and can be used to manage new attributes and APIs. Below, the structure of the Experimenter message from the OpenFlow Switch Specification is shown. The definition is acquired from the OpenFlow Switch Specification [Ope15].

```
/* Experimenter extension message. */
struct ofp_experimenter_msg {
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter;    /* Experimenter ID:
                               * - MSB 0: low-order bytes are IEEE
                               * - MSB != 0: defined by ONF. */
    uint32_t exp_type;        /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
    uint8_t experimenter_data[0];
};
```

`exp_type` defines the Experimenter type and is a field defined by the Experimenter extension. `experimenter_data` represents the body of the message. The length of the body must fit within an OpenFlow object.

5.2 Extensions to OpenFlow

The extensions to OpenFlow are constructed in relation to findings in section 2.4, and are also based on work done by Tal Mizrahi and Yoram Moses in [MM13]. Their report – *Time-based Updates in OpenFlow: A Proposed Extension to the OpenFlow Protocol* – also employed the Experimenter structure and could therefore provide some guidelines regarding the use of the structure. Their report defined an Experimenter message together with an attribute. Furthermore, NoviFlow inc – a company offering SDN switches – used the Experimenter structure [May17], and this was used as an example when designing the extension to the protocol. NoviFlow has constructed messages, matches, actions, port properties and errors by using the experimenter for features such as UDP, IP and MPLS payload matching and handling.

Messages included in the Experimenter Key Management message scope are identified if they match the following criteria:

1. The type field in the OpenFlow header matches `OFPT_EXPERIMENTER`
2. The experimenter field matches the extension ID used for the Experimenter Key Management

The extensions are proposed using TLV fields which makes it possible to include attributes of variable format and length.

This section defines all the extensions needed and their Experimenter structure, using the same notation as the *OpenFlow Switch Specification v. 1.5* [Ope15]. Following, the attributes are defined and thereafter messages using the attributes are described.

5.2.1 Experimenter Attributes

This section provides definitions of the attributes needed in order to realize DPKMS. Table 4.4 lists all attributes defined in this section.

Key

The Key TLV describes how a WireGuard key of 32 bytes, either public or private, is incorporated into an Experimenter message. Table 5.2 illustrates how the attribute is built.

16 bits type	16 bits length
32 bits flag	
8 x 32 bits key	
32 bits IPv4 address	

Table 5.2: The structure of the key TLV where each field specifies its length.

The struct for the key first defines the type of the struct, thereafter the length. Then a 32 bytes key to be used in WireGuard is specified using the following flags: `PRIVATE_KEY`, `PUBLIC_KEY` and `DELETE_PEER`. The flags are set in order to determine which kind of key is being transmitted. Thereafter, an IP address is added, if and only if the key is a public key.

The flags contained within the key struct indicates whether the key included is a public key, private key or a key to be deleted. In case the `PRIVATE_KEY` flag is set, the attribute contains a private key to configure a WireGuard interface and no IP address. If the key is a public key belonging to one of the peers, the key TLV also contains an IP address for configuring the peer-table. The `DELETE_PEER` flag is set if the key TLV is contained within a delete-peer message for identifying which peer to delete. The flags are represented as a 32 bits long bitmap, where three of the bits are assigned to flags while the others should be zero. If the bitmap results in an undefined message, an error is sent.

```

/* Key TLV Format. */
struct onf_key_tlv {
    uint16_t type;           /* ONF_KEY_TLV_TYPE */
    uint16_t length;
    uint32_t key[8];        /* WireGuard key 256 bits */
    uint32_t key_flag;     /* onf_key_flags defining the key */
    uint32_t ipv4_dst;     /* IPv4 address, can be empty if
                           private key*/
};

/* Key TLV flags. */

enum onf_key_flags {
    PRIVATE_KEY = 1 << 0; /* Indicates that this TLV contains a
                           private key to use for the interface */
    PUBLIC_KEY = 1 << 1; /* Indicates that the TLV contains a
                           peer's identity, a public key and
                           an IP address */
    DELETE_PEER = 1 << 2; /* Indicates that the attribute is used
                           for deleting a peer */
};

```

Status

Contained within the status struct is a bitmap explaining the state of the switch. Three flags are defined; **CONFIGURED**, **CONNECTION** and **REVOKED**. **CONFIGURED** is set if the node is configured with a private key for the WireGuard interface. **CONNECTION** is set if an active connection made to another peer exists. **REVOKED** is set in case the private key is revoked. **peers** are used for presenting the list of peers contained within the WireGuard interface. The **length** field is used to specify the number of peers.

16 bits type	16 bits length
32 bits flags	
11 x 32 bits KEY_TLV	
32 bits IPv4 address	

Table 5.3: The structure of the status TLV for a node containing only one peer in its list of peers

`peer` is a list of undetermined size, seeing as this field can vary depending on the list of peers. As mentioned, the `length` field defines the number of peers contained in the struct.

```

/* Status TLV Format. */
struct onf_status_tlv {
    uint16_t type;           /* ONF_STATUS_TLV_TYPE.*/
    uint16_t length;       /* The length of the message which
                           depends on the number
                           of peers */
    uint32_t status_flag;  /* onf_status_flags */
    onf_key_tlv peers[];  /* WireGuard peers with public key
                           and IP address*/
};

/* Status TLV flags. */
enum onf_status_flags {
    CONFIGURED = 1 << 0; /* Indicates that the node has a private
                           key */
    CONNECTION = 1 << 1; /* Indicates that the node has made a
                           connection to another node through
                           WireGuard */
    REVOKED = 1 << 2;   /* Indicates that the private key is
                           revoked */
};

```

Error

The error attribute is contained within the error message with information regarding which error has occurred in the system. The message contains a bitmap, `onf_error_flags` representing each error obtainable in DPKMS. The length of the error message is always the same.

```

/* Error TLV Format. */
struct onf_error_tlv {
    uint16_t type;           /*ONF_ERROR_TLV_TYPE.*/
    uint16_t length;        /* Error message always has the same
                             length */
    uint32_t error_flag;    /* onf_error_flags */
};

/* Error TLV flags. */
enum onf_error_flags {
    SET_PRIVATE_KEY = 1 << 0;    /* Unable to set private key */
    ADD_PEER = 1 << 1;          /* Unable to add peer */
    REMOVE_PEER = 1 << 2;       /* Unable to remove peer */
    EXTRACT_STATUS = 1 << 3;    /* Unable to extract status */
    DELETE_PRIVATE_KEY = 1 << 4; /* Unable to delete private key */
};

```

Match

Matches are defined in the OpenFlow protocol with a general header containing one or more match fields described in OpenFlow Extensible Match (OXM) format. Matches are added to the flow table in a node using the `OFPT_FLOW_MOD` message.

A OXM field needs to be defined in order to be used in DPKMS, matching the Experimenter structure. The OXM format has its own match class for Experimenter, `OFPMXMC_EXPERIMENTER`. Shown below is the general header for a match field, defined in OpenFlow Switch Specification [Ope15]. Thereafter, a match for the Experimenter structure is defined with the OXM structure. The structure will match all packets within the Experimenter structure defined here, as all those packets share the same Experimenter ID. For future work, more granular matches will need to be made.

```

/* Flow Match Header */
struct ofp_match {
    uint16_t type;          /* ofp_match_type OFPMT_OXM extensible
                           match */
    uint16_t length;       /* Length of ofp_match (excluding
                           padding) */
    uint8_t oxm_fields[0]; /* 0 or more OXM match fields */
    uint8_t pad[];         /* Exactly ((length + 7)/8*8 length)
                           (between 0 and 7) bytes of all-zero
                           bytes */
};

struct oxm_fields {
    uint16_t oxm_class;     /* Match class OFPXM_EXPERIMENTER
                           0xffff */
    uint8_t oxm_length;     /* Length of OXM payload */
    uint32_t experimenter; /* Experimenter ID */
}

```

Action

Actions are the structures making changes to packets and ports. Several actions need to be defined in order to realize DPKMS. Only one of the required actions is defined below, as a revised version of actions made in the NoviFlow's Experimenter extensions [May17].

Actions that need to be defined in order for the scheme to extend the OpenFlow protocol are listed below:

Set private key instructs the switch to deliver the attribute in the `set_private_key` message to insert the private key in the interface. Thereafter, a response is expected to be returned and sent to the controller.

Delete private key instructs the switch to delete the private key in the WireGuard interface. A response is expected to be returned and sent to the controller.

Add peer instructs the switch to add a peer, specified in the attribute of the message. A response is expected and returned to the controller.

Delete peer instructs the switch to delete a peer, specified in the attribute contained within the `delete_peer_wg` message. A response is expected and returned to the controller.

5.2.2 Experimenter Messages

All messages needed in order to realize DPKMS are defined in this section. A list of the messages defined here can be found in table 4.3, and the state transition diagram in figure 5.1 displays all messages with their possible responses. Messages described containing TLV fields are multipart messages, defined in the OpenFlow Switch Specification as *messages used to encode requests or replies that potentially carry a large amount of data and would not always fit in a single OpenFlow message, which is limited to 64KB* [Ope15].

set_private_key message

Contained within the `set_private_key` message is an `onf_key_tlv` attribute with the private key of the node. The message is sent during the configuration procedure in section 4.3.1 and is sent from the controller. Responses to this message are either a `status_wg` message in case of success, or an `error_wg` message in case of failure. Contained within the message is an action to be applied to the packet information when reaching the OpenFlow switch. The message structure is constructed by using the `OF_PACKET_OUT` message from the OpenFlow specification as a model.

```

/* Set private key message */
struct set_private_key_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Set private key message */
    struct ofp_action_header; /* Set private key Experimenter
                               action */
    struct onf_key_tlv; /* The private key */
}

```

delete_key_wg message

Deleting the private key in the node is the responsibility of the `delete_key_wg` message. No attributes are needed in this message, seeing as its only task is to instruct WireGuard to delete the private key in the node. The action contained within the message instructs the switch to delete the private key.

```
/* delete private key message */
struct delete_key_wg_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Delete private key message */
    struct ofp_action_header; /* Experimenter Action
                               instructing the switch to delete
                               the WireGuard key */
}
```

add_peer message

`add_peer` is a message that contains an action and information about a peer, comprised in the `onf_key_tlv` attribute, which contains both the public key and the IP address.

```
/* Add peer message */
struct add_peer_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Add peer message */
    struct ofp_action_header; /* Experimenter Action for adding
                               a peer */
    struct onf_key_tlv; /* The public key and IP address
                        of a peer*/
}
```


delete_peer message

To delete a peer, the `delete_peer` message is sent. The public key of the peer is required in the message and is contained within an `onf_key_tlv` attribute where the `DELETE_PEER` flag is set, together with an action.

```
/* Delete peer message */
struct delete_peer_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Delete peer key message */
    struct ofp_action_header; /* Experimenter Action that
                               deletes a peer from
                               WireGuard*/
    struct onf_key_tlv; /* The public key and IP address
                        of a peer for deleting*/
}
```

status_wg message

The status message is sent from the controller to the switch and contains the `status` attribute, described in the previous section. The message does not contain an action because it is sent from the switch to the controller.

```
/* Status WireGuard message */
struct status_wg_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Status wg message */
    struct onf_status_tlv; /* Status attribute containing
                           state information of
                           WireGuard*/
}
```

get_status_wg message

If the controller requires the state of the WireGuard interface from a switch, a `get_status_wg` message is issued with no attributes, only an action.

```
/* Get status message */
struct get_status_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Get status message */
    struct ofp_action_header; /* Experimenter action to obtain
                               the status of a switch and
                               send a response*/
}
```

error_wg message

In case of an error occurring in WireGuard, an `error_wg` message is issued by the switch and sent to the controller. The message contains an `error` attribute which explains the reason behind the error. Because the message is sent from the node to the controller, it does not contain any actions.

```
/* Error WireGuard message */
struct error_wg_msg{
    struct ofp_header header; /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter; /* Experimenter ID */
    uint32_t exp_type; /* Error WG message */
    struct onf_error_tlv; /* Error attribute containing
                           reason for error in a bitmap
                           */
}
```

Chapter 6

Analysis and Discussion

This chapter presents and discusses an analysis of the scheme that has been developed. Two different sets of requirements – cryptographic and operational – were made in section 3.3. For the cryptographic requirements, the emphasis is on properties and techniques related to cryptography. The operational requirements are set in order to ensure correct operation in relation to the phases in key management presented in section 2.4.1. Further, the various design alternatives are presented and examined in order to justify the selected alternative. The last section aims to highlight limitations as well as summarize the shortcomings found in the previous two sections.

6.1 Requirement Analysis

This section will analyze the scheme – its design shown in chapter 4 – with regards to the requirements defined and explained in section 3.3. Firstly, all cryptography requirements are discussed and thereafter requirements related to the operation of the scheme will be analyzed.

6.1.1 Cryptographic Requirements

Authenticated Nodes

A preliminary assumption for the key management scheme is that a secure channel connecting the controller and the node is already established. This secure channel is established through the OpenFlow protocol, and security is enforced by using TLS. TLS – explained in section 2.4.3 – provides mutual authentication if the controller and the switch have exchanged certificates [Ope15]. As this is an assumption for the model developed, one could assume that the nodes are authenticated.

Authenticated keys

As the controller is responsible for generating and distributing keys, and as this central entity is already trusted, the keys could be said to be authenticated.

Authenticated controller

As stated through the assumptions (3.2), a connection in OpenFlow through TLS already exists between the node and the controller. TLS offers mutual authentication, which means that the controller and the node are both authenticated [Ope15]. Consequently, one could state that the controller is authenticated.

Perfect forward secrecy

Perfect forward secrecy is a property for key exchange. WireGuard (2.5.1) is responsible for key exchange in the model. The secure tunnel initializes a handshake at a set time interval that establishes new symmetric keys for retaining perfect forward secrecy. The handshake is based on the *Noise_IK protocol*, a well established protocol framework which is used in the Signal protocol [Per17]. As previously mentioned, WireGuard has recently been developed, and therefore there is a possibility that some of its security properties do not hold. However, based on today's security analysis of the handshake in WireGuard, Perfect Forward Secrecy is provided [Don17].

Key integrity

Key integrity can be ensured if the entities in the key generation and distribution are trusted and authenticated, and if the links transporting the key are encrypted. In DPKMS, authentication and encrypted distribution of the key is ensured and therefore integrity of the keys can be guaranteed.

Key freshness

Freshness of a key is a property of short-term session keys. WireGuard is the tunnel responsible for the key exchange process, which generates session keys. Therefore WireGuard is also responsible for ensuring key freshness. This is done through the algorithm for generating session keys which always uses fresh keying material.

Key binding

In WireGuard, a peer is identified by its public key together with an IP address which is already distributed to nodes. Therefore, it can be said that the key is bound to the IP address. Keys managed in the scheme are only used for the WireGuard connection and a new session key is generated for each connection. Nevertheless, the long-term keys in the system are used for all connections through the tunnel, and are not bound to any other metadata except for the IP address.

Key provides required level of attack resistance

Keys used in the scheme are 32 bytes long Curve25519 points. To derive the private key from the public key, breaking the Curve25519 function is known to be more expensive than brute-forcing a typical 128-bit secret-key cipher [Ber06]. Therefore, it can be claimed that the key is in fact strong enough.

6.1.2 Operational Requirements**DPKMS needs to be able to manage 32 bytes keys**

The attribute defined for keys, `onf_key_tlv` (5.2.1) takes keys that are 32 bytes. Hence, DPKMS fulfills this requirement. However, an increase in key length in the WireGuard protocol would imply a change to the extension. However, the protocol is defined in TLV format, which implies that the length of the message could possibly increase.

DPKMS needs to be able to run over the OpenFlow protocol

As shown in chapter 5, DPKMS fulfills this requirement. The chapter presents the extension made to the OpenFlow protocol in order to include the messages in the secure channel established between the controller and the node. Nevertheless, further development is required in order to include key management in the OpenFlow protocol. This is because both the software in the node and in the controller need to be modified in order to manage the new messages.

Private keys that are distributed are not exposed

In DPKMS, the private keys are generated in the controller and thereafter distributed. The keys are sent to the designated node through an OpenFlow channel secured by TLS, presented in section 2.2. This implies that the key is encrypted and sent between the controller and the node, and will consequently not be exposed.

A potential means of exposing the keys is in the interface connecting the OpenFlow switch and WireGuard. In DPKMS, private keys are sent unencrypted inside the node and to the interface. This can be a potential threat to the system if an attacker were to get access to traffic within a node. However, this would only be a threat if a potential adversary had full control over the node, which should be detected by an Intrusion Detection System (IDS).

Keys are changed within a bounded time interval

A cryptoperiod is set by the controller during the configuration procedure of a node, shown in section 4.3.5. Thereafter, the controller assures that the rekeying procedure

is run whenever the cryptoperiod runs out. This way, the keys can be said to be changed within a bounded time interval.

Compromised keys are revoked

Keys that are reported to the controller as being compromised are revoked according to the revocation procedure in section 4.3.6. Firstly, all packets directed to the node containing the compromised node are dropped. It is left up to the application to decide how the revocation procedure is carried out and it is given two options. By deleting the public key in the controller and the peers, both of the options ensure that the key is not used again. Therefore, DPKMS revokes compromised keys.

Key revocation happens rapidly

DPKMS ensures that messages to the node containing a compromised key are dropped by sending a `ofp_table_mod` message to the connected nodes. *OFLOPS: An open framework for OpenFlow switch evaluation* [RSU⁺12] measured the performance of modifying the flow table in an Open vSwitch. The performance was measured to be 10 flow-mods per second, independently of the size of the network. After a compromised node is detected, approximately 100 milliseconds can be seen as an acceptable duration for the key to become idle.

All errors are handled

Errors directly related to the messages in DPKMS, for instance *unable to set private key* or *unable to add peer*, are stated in section 4.3.8. However, the way in which the controller responds to all of the errors is not specified, and therefore this requirement is not fulfilled.

Secure communication sessions are set up and torn down quickly

When the nodes and the peerlists are configured, the *start communication* procedure needs to be initialized. This procedure consists of editing the flow tables in each node using `ofp_table_mod`. As mentioned in section 6.1.2, this can be done in approximately 100 milliseconds. In order to end secure communication, the flow tables are edited in the same amount of time. Consequently, the duration of setting up and tearing down is acceptably short.

Reliable message delivery

Reliable message delivery is ensured by the OpenFlow protocol which runs over TCP and TLS [Ope15]. In addition, DPKMS is implemented in a request/response model. All requests issued by the controller are responded to with a `status_wg` message from the node. This ensures that the controller knows that the message has reached

its designated node. Furthermore, the controller is updated with the state of the switch, ensuring synchronization of the key management. Hence, DPKMS ensures reliable message delivery.

Packet loss or delay must be handled

Reliable message delivery is ensured within DPKMS, as discussed in the previous section (6.1.2). Therefore, lost packets are restored in the case of loss or delay with the mechanisms from TCP, ensuring synchronization within the system. Therefore, by extending the OpenFlow protocol, the requirement is fulfilled.

Following generation, the private keys are only known to the node

Key generation is realized by the controller which distributes the private keys to its designated nodes. Distribution takes place after generation in the secure channel connecting the controller and the node. Subsequently to distribution, the controller deletes the private keys. Hence, the private keys only exist in the nodes after generation.

Keys are generated based on output from a RBG

Keys are generated by the Curve25519 function [Ber06]. The function requires a random number as an input. The controller generating the keys is defined in software. Therefore, the controller is able to generate true random numbers, meaning that DPKMS satisfies the requirement.

Session keys are securely exchanged

WireGuard is responsible for establishing session keys between the entities. Session keys are established using the Noise protocol, described in section 2.5.1. The Noise protocol framework is widely used in applications for key exchange, such as WhatsApp [Per17]. Therefore, one can say that through the use of WireGuard, session keys are securely exchanged within DPKMS.

Node crash and restart is handled

DPKMS does not have any procedure for handling node crash and restart. However, in SDN, the controller is responsible for detecting and handling node failures. In case of node crash, the routes going through the failed node would be recalculated. If the node were part of secure communications, the packets directed to it through WireGuard would be lost.

In case of a node restart, where information in the node remained intact, it would be possible to continue sending packets through the WireGuard tunnel. No packets

would be lost if queued while restarting the node. On the contrary, if the node were to be configured again, the packets directed to it through WireGuard would be lost because the private key would no longer be in the system. The configuring procedure would be run for the node, giving it new keys and setting up its peers.

6.2 Design Decisions

Before constructing DPKMS, some significant decisions had to be made regarding the structure of the entities and how they were meant to communicate. This section will introduce the alternatives that were considered, before the selected alternative is presented.

6.2.1 Tunnel Between the Nodes

The goal of DPKMS is to set up a secure connection between two nodes. Therefore, a decision with high significance was the choice of channel between the nodes. This choice was important because the operation of the scheme would depend greatly on which parameters were demanded by the connection between the nodes. Two solutions were considered; the *IPsec protocol suite* and the network tunnel *WireGuard* – both located at layer three, encrypting IP packets.

IPsec with IKE – discussed in section 2.4.3 – was initially seen as the most prominent protocol suite for handling this task, with its associated key management protocol, IKE. IPsec is a well-researched protocol and provides strong security [FK11]. However, it is criticized for being overly complex, for instance because it supports several modes where problems with broken cipher algorithms still in use, can occur [Sta05].

WireGuard – described in section 2.5.1 – is proposed as a replacement for IPsec. The tunnel's channel is simple and its design is made to be understandable and clear. A deliberate choice is the lack of cipher agility. Limitations to WireGuard has already been discussed in 2.5.1, but the most prominent limitation is that the protocol is still in the development phase, which has presented two problems for this thesis. Firstly, the lack of sustainable proof for the validity of the security properties, and secondly because there exists little documentation for users of the protocol.

Nonetheless, WireGuard was chosen to do the encryption between the nodes, as was considered less complex than implementing IPsec with IKE in SDN. WireGuard solves key establishment, but the other parts of key management are up to DPKMS.

6.2.2 Storage of Keys

Deciding which entity was responsible for storage of keys had great implications for the design of DPKMS. Storage of keys relied on the key generation process chosen – discussed in the next section – but mainly three alternatives were possible, *either Controller does not store any keys, Controller stores public keys or Controller stores both public and private keys*. The SDN architecture ensures that the controller acts as a trusted third party for both of the nodes, and therefore the focus of the following alternatives for key storage will be on the controller.

Controller does not store any keys The architecture employed would be the one of a Key Translation Center (KTC) – explained in section 2.4.2 – where the controller would act as the KTC, with a secure OpenFlow channel for transportation of keys connected to each node. An advantage to this alternative is that the scheme is simple for the controller and therefore ensures that private keys do not need to be distributed. This means that the private keys are only contained within the nodes. A drawback to this solution is that it demands processing capacity for generating keys by the nodes. Additionally, the solution would imply slower key distribution because the public key needs to be transported first to the controller, then to the node requesting it. Another factor making this architecture unsuitable for DPKMS is the fact that the controller would be unable to directly initiate the secure connection.

Controller stores public keys This solution would make the controller similar to a Public Key Authority – explained in section 2.4.2 – where the controller maintains a dynamic directory for all public keys of the nodes connected to it. The keys can then either be generated in the node or in the controller. In this case, key distribution would be faster than if the controller did not store any keys, seeing as there would only be one step between the controller and the node receiving the public key. The fact that only the public key is saved also makes the scheme more secure in the way that only the nodes in need of the keys hold them. If the controller generates the keys, a drawback is that the private keys need to be transported and are therefore increasingly exposed. If keys were to be generated by the nodes, a drawback would be that the generation would take longer seeing as this also includes transporting the public key to the controller.

Controller stores both public and private keys In this case, the controller would be able to generate keys and thereafter distribute both private and public key information associated to each node. This solution does not hold considering that the private key would exist in two instances in the system. The private key being present in the controller is not directly useful to the controller and therefore it should not have the functionality to decrypt secure sessions between nodes.

The second alternative – *Controller stores public keys* – was chosen for DPKMS,

and the controller is therefore able to initiate secure connections effectively. Furthermore, it ensures that the database in the controller being revealed to an adversary does not risk the security of the encrypted sessions between the nodes. A possibility considered early in the development of the scheme was the use of an external DNS-server with TLSA and DANE for storing and distributing keys, explained in section 2.4.3. A drawback for a key management scheme in SDN using TLSA and DANE would have been the revocation procedure, which cannot be made as fast and dynamic as required in SDN. Hence, this option was discarded.

6.2.3 Key Generation

In a key management scheme, the keys need to be generated by some kind of process aggregating random keys. Because WireGuard was chosen as the secure network tunnel, the keys to be generated needed to be 32 byte long asymmetrical keys – described in section 2.3.1 – of the Curve25519 algorithm. Compared to other Elliptic Curve Diffie Hellman (ECDH) algorithms, the Curve25519 is a fast working algorithm but is still considered to be CPU-intensive [Ber06]. The alternatives when it comes to key generation is either *generating keys in the controller* before distribution, or the *nodes generating keys* and thereafter distributing the public key to the controller. The controller storing the public key of each peer was discussed in the previous section (6.2.2).

Controller generates keys The controller already takes care of CPU-intensive tasks like determining traffic paths, so generating keys would not add much to the workload and it would be able to process it fast. It would take longer for the transportation from the controller to the WireGuard interface than if the node had generated the key, but the increased transportation speed would possibly have been balanced out with the increase in generation speed. Furthermore, the keys could be distributed in parallel to the peers.

If the node generates keys, the private keys would not need any transportation after generation which would imply higher configuration speed. Parallelism could be achieved by the interface being configured at the same time as sending keys to the controller for it to manage. As already mentioned, the generation of keys is a CPU-intensive task, and this could delay the normal operation of the switch. After receiving the key from the node, the controller would have to include it in its databases and thereafter distribute it to peers wishing to communicate with the node through WireGuard.

In DPKMS the controller was put to the task of generating keys. This choice was made in order to maintain a software defined approach to the problem which focuses on making switches simpler. Also supporting this decision is the fact that one of the keys would have to be delivered to the controller in any case, as discussed in last

section, 6.2.2. Seeing as the controller was chosen as the entity to generate the keys, one can assume that the keys are also distributed by this entity.

6.2.4 Which Entity Initiates Encryption

Initiating encryption, in other words deciding which links are to be encrypted and thereafter starting the key management, can be carried out in several ways. The alternatives considered for DPKMS were: *having an application on top of the controller decide*, *implementing an opportunistic encryption scheme* or *making the data plane entities responsible for the initiation*.

Application on top of the controller decides An application, for instance governed by network administrators, can be the entity responsible for deciding which paths to be secured through a WireGuard tunnel. Provided with an overview of the network – containing nodes and the paths connecting them – it would have the ability to either make decisions according to policies and algorithms, or manually. The application would communicate with the controller through the northbound interface, giving it instructions for channels to encrypt. This alternative would be a centralized way of solving the problem, giving the controller all authority.

Opportunistic encryption Opportunistic encryption is a method for deciding what data to encrypt by trying to encrypt all data and, if encrypting fails, transmitting the data unencrypted [Ken14]. If an opportunistic key management scheme were to be implemented, all flows would be directed into the WireGuard interface. WireGuard would then either accept the packet and send it through a secure channel, or reject the packet if some of the parameters for establishing a secure channel – private keys or peer identification – were not present. In case of rejection, the controller would be asked to provide the missing keys and in this case the flow would be sent through a WireGuard tunnel. If the controller does not provide any keys for WireGuard, the packets are sent through another port without encryption. This approach would require that the peerlists be configured before sending packets, meaning that the key management would be realized at activation of the node.

Data plane nodes decide This approach is the most common way of solving the problem described. Section 2.4.2 presents several key management architectures where the main concept is that the endpoints in the communication start encryption. For this option, the entities in the communication path to be encrypted would request the keys of the opposite part and would have to manage the keys themselves.

The approach chosen for DPKMS was to have an application on top of the controller decide. Initially, opportunistic encryption was considered, but was eventually found unsuitable for SDN. This is because the existence of numerous flows that were not to be encrypted could possibly result in numerous calls to the controller.

Data plane entities deciding in SDN would not be a suitable choice either, as this would be a more distributed solution that would not utilize the SDN capabilities. However, an application would gain from the overview of the network provided by the controller, utilizing its benefits. Furthermore, the application could provide more specific regulation for encryption, depending on the needs of network administrators. An example would be applying encryption to specific flows, for instance all VoIP traffic.

6.2.5 Choice of Control plane Protocol

The control plane – connecting the nodes and the controller – needed to be defined by a protocol. Only two alternatives were considered when making DPKMS. Birkeland’s project for his thesis – reviewed in section 2.7 – used an SSH connection for the control plane. However, for future work he suggested to make his solution to key distribution through OpenFlow. Hence, an *extension to the OpenFlow protocol* was considered in DPKMS. Another protocol was explored, namely *NETCONF*, because the model in the internet draft [MLLM16] – discussed in section 2.7 – was considered as a possible base for this thesis.

Using NETCONF with YANG The protocol NETCONF with its data modelling language YANG was considered because this is used by Marin-Lopez et. al., who are currently working on a key management solution for SDN – discussed in section 2.7. NETCONF is a network configuration protocol based on XML and realized through Remote Procedure Call (RPC)s [Enn11]. At the time being the team behind the Internet Draft *(SDN)-based IPsec Flow Protection* is working on a key distribution scheme for IPsec using NETCONF and YANG [MLLM16]. A drawback of implementing a separate channel for key management using these protocols is that it can overwhelm a controller. Furthermore, the benefits of the already established secure channel are left unused, as well as few controllers and no switches for SDN support NETCONF and YANG. This implies that software for both the controller and the node need to be written from scratch.

Extending OpenFlow OpenFlow defines a way of extending the protocol in a structure called *Experimenter*, reviewed in chapter 5. One of the greatest advantages of extending OpenFlow is that it can ensure secure connection between the controller and the node, as this is already supported though the protocol. Another advantage is that there is no need for additional end-point signalling for devices implementing the model. A constraint for extending OpenFlow is that a change in the protocol means a change to the software of both the node and the controller.

Extending OpenFlow was chosen as the most suitable alternative for DPKMS. The reason for this was firstly the fact that OpenFlow utilizes the benefits of the already established secure channel between the node and the controller. Furthermore,

as the author didn't have any experience with NETCONF and YANG, it would be time-consuming to learn how to develop a protocol in the language. Also, as already mentioned, new software both for the controller and for the switch would have needed to be developed.

6.2.6 Revocation of Keys

Revocation of a key, described in section 2.4.1, is an important procedure in key management in order to ensure that no attackers can access the network traffic if they manage to compromise a key. There are several ways of implementing solutions to this problem, and two were considered for DPKMS, namely a *controller removing a node from the network* and the *controller reconfiguring a compromised node*.

Controller removes node from network If an error is detected and keys from a node need to be revoked, the node can immediately be left out of the network topology. This can be achieved through the controller removing it by deleting flow table entries that direct packets to the compromised node. This procedure will affect all nodes communicating with the compromised node, changing their flow tables.

Controller reconfigures compromised node A configuration procedure is run if a node is compromised. In this case, packets encrypted with the old key that were found to be compromised, are not sent by the peers and instead dropped. Thereafter, the node is given a new key.

The solution chosen for revocation was to combine both of the options. As the scheme shows in section 4.3.6, the choice of method is left to the application. First, when a compromised node is detected, the nodes connected to it drops all packets to the switch. Thereafter, the application is informed of the compromization, and is provided with two options; removing the node from the network or reconfiguring the WireGuard interface. This solution was chosen because, depending on the importance of securing a path, different measures must be taken.

6.2.7 Key Distribution

Key distribution, explained in section 2.4.1, is the phase of key management where the generated keys are to be distributed in order for them to be used for encryption or decryption. This section discusses at what point the keys are to be distributed, either *reactively* or *proactively*.

Reactive Distribution Reactive key distribution implies that the keys are distributed when they are requested in the management scheme. When initializing encrypted communication, the keys are generated and thereafter distributed to the designated nodes.

Proactive Distribution Proactive key distribution means that the controller populates the tables in the nodes before encrypting communication through the WireGuard interface.

In DPKMS there is a possibility of using both options. The choice is left up to the application initiating the encrypted communication, with the ability to add peers before starting the communication, as well as dynamically when the peers are communicating. Because of the SDN architecture, it is important to have the possibility of reactively changing the network topology.

6.2.8 How to be informed of the state of the node

The state of the node is important for the controller to keep an updated image of the network topology. Included in the state is information related to keys and communication in WireGuard. Two ways were found for containing an updated state in the controller, either by the *node sending information when the state changes* or by the *controller regularly asking for the status*.

Node sends status message when the state changes Whenever the state of the node changes, the node sends a `status_wg` message to the controller to inform it of the new state. In order for this to be achievable, debug-mode in WireGuard needs to be turned on. Otherwise the tunnel will not give any information regarding state to the interface. This alternative is demanding for the node, however, it can result in fewer messages being sent to the controller.

Controller regularly asks for node status If the controller were to ask for the state of the switch, this could be done regularly within certain periods of time. This makes it possible to keep the default mode in the WireGuard interface, not forcing it to send all information. This option could result in more information being processed. If the state rarely changes, multiple redundant messages would be sent. Another drawback to this solution is that it would not receive state information in real time.

To inform the controller of the state of each node, a combination of each of the options is used, the method used is up to the implementation of the controller. The controller has the ability to ask for state information from the node, issuing a `get_status_wg` message. Additionally, the `state_wg` is used as a reply to all messages issued by the controller in order to ensure synchronization and to verify that instructions sent to the switch are actually applied.

6.3 Known Limitations

Based on the analysis made previously in this chapter, the following section aims to present the limitations and shortcomings of DPKMS.

6.3.1 WireGuard is in an Experimental Stage of Development

WireGuard was released in June 2016 and is still in an experimental stage of development; a stable release 1.0 has not yet been published. Hence, there might be vulnerabilities to the technology that has not yet been discovered, possibly making it insecure. Moreover, there are still parts of WireGuard that have not yet been developed, including cross-platform implementation. As for now, the tunnel has been developed only for the Linux kernel. If any vulnerabilities were to be found, or if the development were to be terminated, the value of the scheme developed in this thesis would be limited.

6.3.2 One Key Pair is Used as Long-term Keys for All Communication

Each of the nodes is only issued one key pair to use for the secure communication through WireGuard. Hence, a limitation to the scheme is that the security of all WireGuard communication rests on only one pair of keys. A solution would be to incorporate several WireGuard interfaces in one node, each configured with a distinct key.

However, if more keys were to be introduced in the system, metadata would have to be bound to the keying material to distinguish keys used in one node. This could be done, for instance, by digitally signing the key and the metadata. This solution would indicate a different implementation of the database in the controller, described in section 4.1.1. Additionally, changes to the protocol extension would have to be made, especially with regards to the key attribute, defined in section 5.2.1.

6.3.3 The Extension Only Supports One Key Size

WireGuard does not allow for any agility in the protocol, and all its keys have a size of 32 bytes. This is why only this key size is implemented in the extension to OpenFlow. This boundary limits DPKMS and could be a potential problem if it were to be used for other protocols or if WireGuard were to change its key size.

6.3.4 The Scheme Does Not Specify Error Handling in the Controller

Error messages are discussed in section 4.3.8, where their use in DPKMS is explained. However, the way in which the controller handles the error messages returned from the switch is not specified. Hence, to propose a proof of concept, error handling in the controller needs to be resolved.

6.3.5 The Scheme Only Includes Two Nodes

The reference model for the scheme contains one central controller connected to two nodes. This limits the use of the scheme, as networks often consist of more than two nodes. However, all of the processes allow for more than two nodes. If DPKMS were to be extended to include additional nodes, it would need to be reviewed.

6.3.6 WireGuard Only Encrypts Layer Three Data

Mentioned in section 2.5.1, WireGuard only encrypts IP packets, residing at layer three, and SDN nodes can potentially transmit all types of data from layer 2 to 5. This limitation would force DPKMS to encapsulate non-IP packets before transmitting them to the tunnel and then decapsulating when received at the tunnel endpoint. This proposal would lead to additional processing in the node. Another way this limitation could be approached is by implementing policies that do not allow flows to direct non-IP packets through the secure tunnel. However, this would limit the use of the secure tunnels.

6.3.7 Private Keys are Transported Unencrypted Within a Node

If an adversary assumed control over a node, it would be able to eavesdrop on the communication within the node. In this case, the adversary would be able to obtain the private key of the WireGuard interface and thereafter be able to decrypt packets sent to the node. This is because it is not possible to send packets containing keys encrypted into the WireGuard tunnel, as this also requires a key. Therefore, this is a potential limitation to the security of DPKMS. However, an intrusion detection system should be in place to prevent adversaries from accessing the network nodes.

6.3.8 At One Point the Controller Possesses Private Keys for the Nodes

After generating the key pair consisting of a public and a private key, the controller possesses the private key of the node before it distributes it to its designated node. The attack surface for attackers that want to obtain the key is thereafter larger, making the system less secure. A way to avoid this would be to have the nodes

generate their own key pair. However, as discussed in section 6.2, this would affect the duration of the configuration procedure.

6.3.9 Queuing of Packets Can Cause Delay and Loss

The procedure for rekeying, described in section 4.3.5, depends on queuing packets while waiting for the configuration procedure to run. Queuing will ensure that the right operation is performed, but might cause packet delay or loss which will influence the QoS.

Chapter 7

Conclusion

In this thesis, a key management scheme for encrypting the data plane in SDN – DPKMS – has been proposed. Automating key management in SDN is seen as a necessity due to the highly dynamic nature of the architecture. As no previous work was found on including key management abilities in the OpenFlow protocol, an extension of the protocol has been formulated. The protocol extension for including key management has been designed with the OpenFlow Experimenter structure containing messages and associated attributes. WireGuard was chosen as tunneling protocol between the nodes in order to ensure encryption and key establishment. It was found to encompass all functions needed in addition to bringing simplicity to the scheme.

An analysis of DPKMS was carried out, and this highlighted the benefits of utilizing the secured OpenFlow channel for key management in the data plane. Several requirements were met by employing properties of the channel connecting the node and the controller.

Contributions made in this thesis presents the possibility of utilizing benefits from the SDN paradigm in the scope of key management. If the scheme were to be implemented and proven to be feasible, it could lead to simpler means of applying encryption in SDN. This would be useful seeing as a scheme for key management needs to be automatic in order to encrypt highly dynamic software defined networks.

7.1 Future Work

Proof of Concept

This thesis proposes a scheme for key management in SDN and an extension to the OpenFlow protocol. It would be possible to implement the scheme using these suggested technologies; *Open vSwitch* (2.5.2), *Floodlight* (2.5.3) and *WireGuard* (2.5.1). The OpenFlow Experimenter structure is used for defining extensions to OpenFlow, and is intended to be incorporated in the protocol by editing the controller and the switch software. Some Experimenter attributes are not fully developed, such as match fields and actions. These needs to be granularly defined depending on the software used for the realization.

In section 2.6, it was proven to be possible to include a WireGuard interface in a port in Open vSwitch. However, the difficulty of an implementation relies on the ability to instruct WireGuard from the Open vSwitch, as well as the ability to capture responses from WireGuard. Nevertheless, it would be possible to extract information from implementation of other ports in Open vSwitch, such as IPsec and GRE. Thus, if communication with WireGuard through Open vSwitch can be achieved, the development of a proof of concept of the scheme would be feasible.

A proof of concept would be essential in order to determine whether the scheme has practical potential. As discussed in motivation section (1.1), DPKMS has many plausible applications and could be valuable in the deployment of SDN in larger networks communicating through insecure channels.

Designing the Application

As described in chapter 3, DPKMS relies on input from an application. The application communicated with the controller through its northbound interface. A user interface would have to be constructed, enabling the user to have an overview of the network nodes and to select the paths to be encrypted. In correspondence with the user's actions, the application makes calls to the controller through the defined APIs.

Included in the work would be the making of the interface between the controller and the application. This interface would include specified instructions that the application could conduct, such as setting the cryptoperiod, selecting the type of revocation or adding a peer. If the Floodlight controller were to be used, the interface would be made using REST.

References

- [3GP13] 3GPP TS 33.401. 3GPP System Architecture Evolution (SAE); Security Security Architecture. *Technical Specification*, 12(13), 2013.
- [AAL⁺05] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Dns security introduction and requirements. RFC 4033, RFC Editor, March 2005. <http://www.rfc-editor.org/rfc/rfc4033.txt>.
- [BBB11] Elaine Barker, William Barker, and William Burr. Recommendation for Key Management – Part 1 : General (Revision 3). *Nist Special Publication*, pages 1–142, May 2011.
- [BEIE15] Jaouad Benabbou, Khalid Elbaamrani, Noureddine Idboufker, and Raja Ellassali. Software-defined networks, security aspects analysis. In *Information Assurance and Security (IAS), 2015 11th International Conference on*, pages 79–84. IEEE, 2015.
- [Ber06] Daniel J. Bernstein. *Curve25519: New Diffie-Hellman Speed Records*, pages 207–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [BQCM⁺16] August Betzler, Ferran Quer, Daniel Camps-Mur, Ilker Demirkol, and Eduard Garcia-Villegas. On the benefits of wireless SDN in networks of constrained edge devices. In *EUCNC 2016 - European Conference on Networks and Communications*, pages 37–41, 2016.
- [BR12] Elaine B Barker and Allen Roginsky. Recommendation for Cryptographic Key Generation. *NIST Special Publication 800-133*, pages 1–26, 2012.
- [BSBC13] Elaine B Barker, Miles Smid, Dennis Branstad, and Santosh Chokhani. A Framework for Designing Cryptographic Key Management Systems. *NIST Special Publication 800-130*, pages 1–120, 2013.
- [CGMP12] Salvatore Costanzo, Laura Galluccio, Giacomo Morabito, and Sergio Palazzo. Software defined wireless networks: Unbridling SDNs. In *Proceedings - European Workshop on Software Defined Networks, EWSDN 2012*, pages 1–6, 2012.
- [CGPM05] Haowen Chan, Virgil D. Gligor, Adrian Perrig, and Gautam Muralidharan. On the distribution and revocation of cryptographic keys in sensor networks. *IEEE Transactions on Dependable and Secure Computing*, 2(3):233–247, 2005.

- [CTHN16] Niketa Chellani, Prateek Tejpal, Prashant Hari, and Vishal Neeralike. Enhancing security in openflow, 2016. Capstone Research Project Proposal.
- [DFP13] Diego Kreutz, Fernando M. V. Ramos, and Paulo Verissimo. Towards Secure and Dependable Software-Dened Networks. *HotSDN*, pages 55–60, 2013.
- [DM04] Alex W Dent and Chris J Mitchell. *User’s Guide To Cryptography And Standards (Artech House Computer Security)*. Artech House, Inc., 2004.
- [Don17] Jason Donenfeld. Wireguard: Next generation kernel network tunnel. *Proceedings of the Network and Distributed System Security Symposium*, March 2017.
- [DR08] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. <http://www.rfc-editor.org/rfc/rfc5246.txt>.
- [EG02] Laurent Eschenauer and Virgil D Gligor. A key-management scheme for distributed sensor networks. *Proceedings of the 9th ACM conference on Computer and communications security*, pages 41–47, 2002.
- [Enn11] R Enns. NETCONF configuration protocol. *Network*, RFC 4741:1–95, 2011.
- [FK11] Sheila Frankel and Suresh Krishnan. RFC6071: IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. *Request for Comments, IETF*, pages 1–63, 2011.
- [FL93] Walter Fumy and Peter Landrock. Principles of Key Management. *IEEE Journal on Selected Areas in Communications*, 11(5):785–793, 1993.
- [Flo17a] Project Floodlight. Floodlight openflow controller -project floodlight. <http://www.projectfloodlight.org/floodlight/>, 2017. (Accessed on 05/12/2017).
- [Flo17b] Project Floodlight. How to add an openflow experimenter extension - floodlight controller. <https://floodlight.atlassian.net/wiki/display/floodlightcontroller/How+to+Add+an+OpenFlow+Experimenter+Extension>, December 2017. (Accessed on 06/04/2017).
- [FM90] Walter Fumy and Michael Munzert. A modular approach to key distribution. In *Conference on the Theory and Application of Cryptography*, pages 274–283. Springer, 1990.
- [Fun12] Open Networking Foundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.
- [GB14] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Morgan Kaufmann, 2014.
- [HC14] Chan Kyu Han and Hyoung Kee Choi. Security analysis of handover key management in 4G LTE/SAE networks. *IEEE Transactions on Mobile Computing*, 13(2):457–468, 2014.

- [HS12] P. Hoffman and J. Schlyter. The dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa. RFC 6698, RFC Editor, August 2012. <http://www.rfc-editor.org/rfc/rfc6698.txt>.
- [Ken14] Stephen Kent. Opportunistic security as a countermeasure to pervasive monitoring. Internet-Draft draft-kent-opportunistic-security-01, IETF Secretariat, April 2014. <http://www.ietf.org/internet-drafts/draft-kent-opportunistic-security-01.txt>.
- [KHNE10] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. RFC 5996: Internet Key Exchange Protocol Version 2 (IKEv2), 2010.
- [May17] Luc Mayrand. *PD-0006 – NoviFlow Experimenter Extensions*. NoviFlow, Inc., January 2017.
- [MLLM16] R. Marin-Lopez and G. Lopez-Millan. Software-defined networking (sdn)-based ipsec flow protection. *Internet Engineering Steering Group (IESG)*, pages 1–22, July 2016.
- [MM13] Tal Mizrahi and Yoram Moses. Time-based updates in openflow: A proposed extension to the openflow protocol. *technical report, CCIT# 835*, 2013.
- [Ope15] Open Networking, Foundation. *OpenFlow Switch Specification*, April 2015. Version 1.5.1.
- [Panil] Suzanne Panoplos. Docker advances the software containerization movement with two new collaborative projects. <http://www.prnewswire.com/news-releases/docker-advances-the-software-containerization-movement-with-two-new-collaborative-projects-300440892.html>, 2017 April. (Accessed on 05/01/2017).
- [Per17] Trevor Perrin. The noise protocol framework, May 2017.
- [PGC⁺14] Mark Patton, Eric Gross, Ryan Chinn, Samantha Forbis, Leon Walker, and Hsinchun Chen. Uninvited connections: A study of vulnerable devices on the internet of things (IoT). In *Proceedings - 2014 IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014*, pages 232–235, 2014.
- [PPK⁺15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.
- [RSU⁺12] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7192 LNCS, pages 85–95, 2012.
- [Sch14] Bruce Schneier. Essays: The internet of things is wildly insecure—and often unpatchable - schneier on security. https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html, January 2014. (Accessed on 05/07/2017).

- [SDN16] SDNCentral. Special report: Openflow and sdn - state of the union. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/special-reports/Special-Report-OpenFlow-and-SDN-State-of-the-Union-B.pdf>, December 2016. (Accessed on 04/05/2017).
- [SF16] Birkeland Steffen Fredrik. Software defined data flow isolation by virtualization and cryptographic key distribution. Master's thesis, Norwegian University of Science and Technology, June 2016.
- [Smi15] S. Smiler. *OpenFlow cookbook : over 110 recipes to design and develop your own OpenFlow switch and OpenFlow controller*. Packt Publishing, Birmingham, UK, 2015.
- [Sta05] William Stallings. *Cryptography and Network Security: Principles and Practices*. Pearson Publishing, 2005.